• CYPRESS •

SEMICONDUCTOR

APPLICATIONS HANDBOOK

APPLICATIONS HANDBOOK

Cypress Semiconductor, 3901 North First St., San Jose, CA 95134 (408) 943-2600
Telex: 821032 CYPRESS SNJ UD, TWX: 910 997 0753, FAX: (408) 943–2741

Published August 1991

Preface

About This Book

This *Applications Handbook* is a learning tool for using Cypress devices. The application notes included here range from general product overview articles, such as "Understanding Dual-Port RAMs," to specific design examples.

The general overviews describe product-family characteristics and explain some of the products' capabilities. These application notes appear at the beginning of this *Handbook*.

Next appear application examples that show how to use specific Cypress devices in the context of real designs. The application examples are organized by product type (e.g., SRAMs or EPLDs). Within each product type examples are arranged by product number, using the product that is the article's primary focus.

Although your specific application might not appear explicitly in an application note, the design examples can still be useful to you. If the design example is similar to your application, you might be able to adapt the hardware or software to your design easily. Many of the application notes provide PLD software code for design tools from a variety of vendors, so that you can copy the code and use it as a skeleton for your own PLD designs. Even if none of the examples relate directly to your design, they can stimulate new ideas by showing features or applications that might not have occurred to you. The information can also significantly reduce the learning curve normally associated with unfamiliar ICs.

Most of the designs described in this *Handbook* are based on actual circuits produced either by Cypress or by one of our customers. Application notes that discuss specific designs indicate whether the designs have been simulated and/or built and completely debugged.

If you have questions about any Cypress product, please contact your local Field Applications Engineer at the nearest direct sales office. A list of Cypress sales offices, representatives, and distributors is included at the back of this Handbook. For continuous on-line information about Cypress products, you can connect to the Cypress Bulletin Board at (408) 943-2954.

About Cypress Semiconductor

Since its incorporation in 1982, Cypress has successfully addressed diverse, high-performance niche markets by creating technologically sophisticated products, using innovative packaging, and emphasizing quality. Cypress is a complete semiconductor manufacturer, performing its own process development, circuit design, wafer fabrication, assembly, and test. Its core CMOS and BiCMOS processes lead the industry with 0.8-micron design rules. Cypress ships over 200 products in seven product areas: SRAMs, PROMs, PLDs, logic devices, SPARC microprocessors and peripherals, multichip modules, and high-speed BiCMOS PLD and memory devices. Cypress is an international company, with headquarters in San Jose, California and fabrication facilities in San Jose; Round Rock, Texas; and Bloomington, Minnesota. The company has started up five subsidiaries that are funded by Cypress but run as independent businesses, including Cypress Semiconductor (Texas) Inc., Aspen Semiconductor Corporation, Multichip Technology Incorporated, Ross Technology Inc., and Cypress Semiconductor (Minnesota) Inc.

Contents

Section Contents

CYPRESS SEMICONDUCTOR

Systems Design Considerations When Using Cypress CMOS Circuits

This application note describes some factors to consider when designing new systems using Cypress high-performance CMOS integrated circuits or when using Cypress products to replace either bipolar or NMOS circuits in existing systems. The two major areas of concern are device input sensitivity and transmission line effects due to impedance mismatching between the source and load.

To achieve maximum performance when using Cypress CMOS ICs, pay attention to the placement of the components on the printed circuit board (PCB); the routing of the metal traces that interconnect the components; the layout and decoupling of the power distribution system on the PCB; and perhaps most important of all, the impedance matching of some traces between the source and the loads. The latter traces must, under certain conditions, be analyzed as transmission lines. The most critical traces are those of clocks, write strobes on SRAMs and FIFOs, output enables, and chip enables.

Replacing Bipolar or NMOS ICs

Cypress CMOS ICs are designed to replace both bipolar ICs and NMOS products and to achieve equal or better performance at one-third (or less) the power of the components they replace.

When high-performance Cypress CMOS circuits replace either bipolar or NMOS circuits in existing sockets, be aware of conditions in the existing system that could cause the Cypress ICs to behave in unexpected ways. These conditions fall into two general categories: device input sensitivity and sensitivity to reflected voltages.

Input Sensitivity

High-performance products, by definition, require less energy at their inputs to change state than low- or medium-performance products.

Unlike a bipolar transistor, which is a current-sensing device, a MOS transistor is a voltage-sensing device. In fact, a MOS circuit design parameter called K′ is analogous to the gm of a vacuum tube and is inversely proportional to the gate oxide thickness.

Thin gate oxides, which are required to achieve the desired performance, result in highly sensitive inputs. These inputs require very little energy at or above the device input-voltage threshold (approximately 1.5V at 25°C) to be detected. CMOS products might detect high-frequency signals to which bipolar devices would not respond.

MOS transistors also have extremely high input impedances (5 to 10 MΩ), which make these transistors' gate inputs analogous to the input of a high-gain amplifier or an RF antenna. In contrast, because bipolar ICs have input impedances of 1000Ω or less, these devices require much more energy to change state than do MOS ICs. In fact, a typical Cypress IC requires less that 10 picojoules of energy to change state. Thus, when Cypress CMOS ICs replace bipolar or NMOS ICs in existing systems, the CMOS ICs might respond to pulses of energy in the system that are not detected by the bipolar or NMOS products.

Reflected Voltages

Cypress CMOS ICs have very high input impedances and — to achieve TTL compatibility and drive capacitive loads — low output impedances. The impedance mismatch due to low-impedance outputs driving high-impedance inputs might cause unwanted voltage reflections and ringing, under certain conditions. This behavior could result in less-than-optimum system operation.

When the impedance mismatch is very large, a nearly equal and opposite negative pulse reflects back from the load to the source when the line's electrical length (PCB trace) is greater than

$$l = \frac{t_r}{2T_{pd}}$$

where t_R is the rise time of the signal at the source, and T_{pd} is the one-way propagation delay of the line per unit length.

The input clamping diodes in bipolar IC families (e.g., TTL, LS, ALS, FAST, FACT) are inherent in the

fabrication process. The P substrate is usually grounded and N-wells are used for the NPN transistors and P-type resistors. The wells are reverse biased by connecting them to the V_{cc} supply. As a result, a PN junction diode is formed between every input pin (cathode or N material) and the substrate (anode or P material). A negative voltage at an input pin due to either lead inductance or a voltage reflection forward biases the diode, which turns on and clamps the input pin to a V_f below ground (approximately -0.8V).

Historically, as circuit performance improved, the output rise and fall times of the bipolar circuits decreased to the point where voltage reflections began to occur even for short traces when an impedance mismatch existed between the line and the load. Most users, however, were unaware of these reflections because the reflections were suppressed by the diodes' clamping action.

Conventional CMOS processing results in PN junction diodes, which adversely affect the ESD (electrostatic discharge) protection circuitry at each input pin and cause an increased susceptibility to latch-up. In addition, when the input pin is negative enough to forward bias the input clamping diodes, electrons are injected into the substrate. When a sufficient number of electrons are injected, the resulting current can disturb internal nodes, causing soft errors at the system level.

To eliminate this problem, all Cypress CMOS products use a substrate bias generator. The substrate is maintained at a negative 3V potential, so the substrate diodes cannot be forward biased unless the voltage at the input pin becomes a diode drop more negative than -3V. (See *Figure 5* in "CMOS PAL Basics" for a schematic of the input protection circuits used on all Cypress CMOS products.) To the systems designer, this translates to approximately five times (3.8V divided by 0.8V = 4.75) the negative undershoot safety margin for Cypress CMOS integrated circuits versus those that do not use a bias generator.

Voltage reflections should be eliminated by using impedance matching techniques and passive components that dissipate excess energy before it can cause soft errors. Crosstalk should be reduced to acceptable levels by careful PCB layout and attention to details.

Crosstalk

The rise and fall times of the waveforms generated by Cypress CMOS circuit outputs are 2 to 4 ns between levels of 0.4 and 4V. The fast transition times and the large voltage swings could cause capacitive and inductive coupling (crosstalk) between signals if insufficient attention is paid to PCB layout.

You can reduce crosstalk by avoiding running PCB traces parallel to each other. If this is not possible, run ground traces between signal traces.

In synchronous systems, the worst time for the crosstalk to occur is during the clock edge that samples the data. In most systems, it is sufficient to isolate the clock, chip select, output enable, and write and read control lines from each other and from data and address lines so that the signals do not cause coupling to each other or to the data lines.

It is standard practice to use ground or power planes between signal layers on multi-layered PCBs to reduce crosstalk. The capacitance of these isolation planes increases the propagation delay of the signals on the signal layers, but this drawback is more than compensated for by the isolation the planes provide.

The Theory of Transmission Lines

A connection (trace) on a PCB should be considered as a transmission line if the wavelength of the applied frequency is short compared to the line length. If the wavelength of the applied frequency is long compared to the length of the line, you can use conventional circuit analysis.

In practice, transmission lines on PCBs are designed to be as nearly lossless as possible. This simplifies the mathematics required for their analysis, compared to a lossy (resistive) line.

Ideally, all signals between ICs travel over constant-impedance transmission lines that are terminated in their characteristic impedances at the load. In practice, this ideal situation is seldom achieved for a variety of reasons.

Perhaps the most basic reason is that the characteristic impedances of all real transmission lines are not constants, but present different impedances depending upon the frequency of the applied signal. For "classical" transmission lines driven by a single-frequency signal source, the characteristic impedance is "more constant" than when the transmission line is driven by a square wave or a pulse.

According to Fourier series expansion, a square wave consists of an infinite set of discrete frequency components — the fundamental plus odd harmonics of decreasing amplitude. When the square wave propagates down a transmission line, the higher frequencies are attenuated more than the lower frequencies. Due to dispersion, the different frequencies do not travel at the same speed.

Dispersion indicates the dependence of phase velocity upon the applied frequency (*Reference 1* pg. 192). The result is that the square wave or pulse is distorted when the frequency components are added together at the load.

A second reason why practical transmission lines are not ideal is that they frequently have multiple loads. You can distribute the loads along the line at regular or irregular intervals or lump them together as close as practical at the end of the line. The signal-line reflections and ringing caused by impedance mismatches, non-uniform transmission line impedances, inductive leads, and non-ideal resistors could compromise the dynamic system noise margins and cause inadvertent switching.

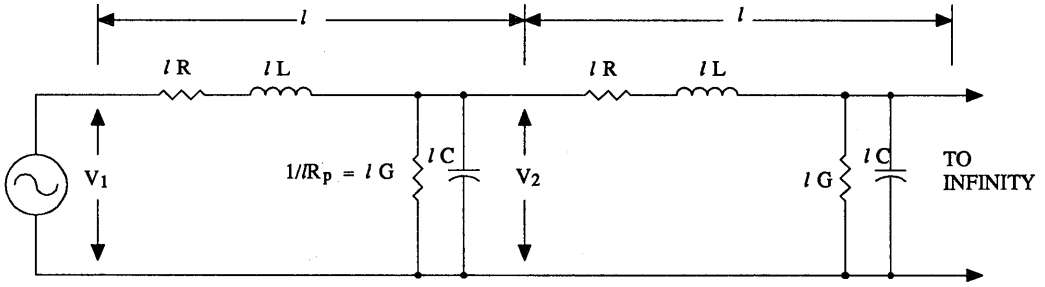


Figure 1. Transmission Line Model

One system design objective is to analyze the critical signal paths and design the interconnections such that adequate system noise margins are maintained. There will always be signal overshoot and undershoot. The objective is to accurately predict these effects, determine acceptable limits, and keep the undershoot and overshoot within the limits.

The Ideal Transmission Line

An equivalent circuit for a transmission line appears in *Figure 1*. The circuit consists of subsections of series resistance (R) and inductance (L) and parallel capacitance (C) and shunt admittance (G) or parallel resistance, R_p. For clarity and consistency, these parameters are defined per unit length. Multiply the values of R, L, C, and R_p by the length of the subsection, l, to find the total value. The line is assumed to be infinitely long.

If the line of *Figure 1* is assumed to be lossless (R = 0, R_p = infinity) *Figure 1* reduces to *Figure 2*. A small series resistance has little effect upon the line's characteristic impedance. In practice and by design, the series resistance is quite small. For 1-ounce (0.0015-inch-thick), 1-mil-wide (0.010-inch) copper traces on G-10 glass epoxy PCBs, the trace resistance is between 0.5 and 0.3Ω per foot. 2-ounce copper has a resistance 50 percent lower than that of 1-ounce copper.

Input or Characteristic Impedance

To calculate the characteristic impedance (also called AC impedance or surge impedance) looking into terminals a-b of the circuit in *Figure2*, use the following procedure.

Let Z_1 be the input impedance looking into terminals a-b, with Z_2 for terminals c-d, Z_3 for terminals e-f, etc. Z_1 is the series impedance of the first inductor (IL) in series with the parallel combination of Z_2 and the impedance of the capacitor (IC).

From AC theory:

$X_L = j\omega lL$

where X_L is the inductive reactance.

$$X_C = \frac{1}{j\omega lC}$$

where X_C is the capacitive reactance.

Then

$$Z_1 = X_L + \frac{Z_2 X_C}{Z_2 + X_C} \qquad \textit{Eq. 1}$$

If the line is reasonably long, $Z_1 = Z_2 = Z_3$. Substituting $Z_1 = Z_2$ into Equation 1 yields

$$Z_1 = X_L + \frac{Z_1 X_C}{Z_1 + X_C}$$

or,

$$Z_1^2 - Z_1 X_L - X_C X_L = 0 \qquad \textit{Eq. 2}$$

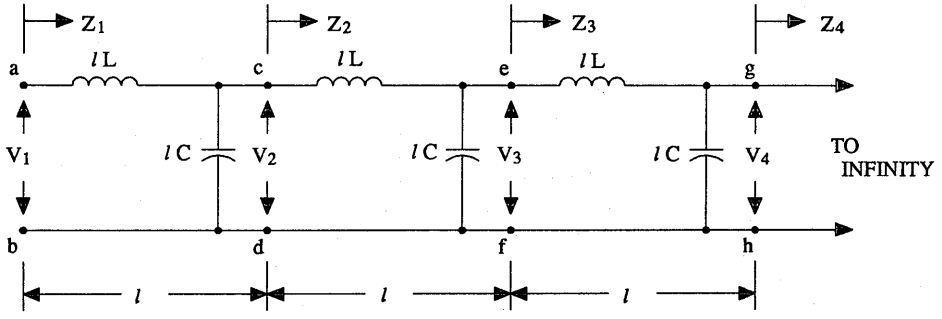


Figure 2. Ideal Transmission Line Model

Substituting the expressions for X_C and X_L yields

$$Z_1^2 - j\omega L = \frac{L}{C} \qquad Eq.\ 3$$

Equation 3 contains a complex component that is frequency dependent. You can eliminate the complex component by allowing 1 to become very small and by recognizing that the ratio L/C is constant and independent of l or ω:

$$Z_1 = \sqrt{L/C} \qquad Eq.\ 4$$

The AC input impedance of a purely reactive, uniform, lossless line is a resistance. This is true for AC or DC excitation.

Propagation Velocity and Delay

The propagation velocity (or phase velocity) of a sinusoid traveling on an ideal line (*Reference 1* pg. 33) is

$$\alpha = \frac{1}{\sqrt{LC}}$$

The propagation delay for a lossless line is the reciprocal of the propagation velocity:

$$T_{pd} = \sqrt{LC} \qquad Eq.\ 5$$
$$= Z_1 C$$

where L and C are once again the intrinsic line inductance and capacitance per unit length.

Adding additional stubs or loads to the line (*Reference 2* pg. 129) increases the propagation delay by the factor

$$\sqrt{1 + C_D/C}$$

where C_D is the load capacitance.

Therefore, the propagation delay of a loaded line, T_{pdL}, is

$$T_{pdL} = T_{pd}\sqrt{1 + C_D/C} \qquad Eq.\ 6$$

This application note shows later that a transmission line's unloaded or intrinsic propagation delay is proportional to the square root of the dielectric constant of the medium surrounding or adjacent to the line. Propagation delay is not a function of the line's geometry.

The characteristic impedance of a capacitively loaded line decreases by the same factor that the propagation delay increases:

$$Z_1' = \frac{Z_1}{\sqrt{1 + C_D/C}} \qquad Eq.\ 7$$

Note that the capacitance per unit length must be multiplied by the line length, l, to calculate an equivalent lumped capacitance.

The Condition for Voltage Reflection

It is relatively straightforward to obtain a closed-form solution for a transmission line's maximum allowable length, which, if exceeded, might cause a voltage reflection. If the line is not terminated in its characteristic impedance, a reflection is guaranteed to occur. The reflection's amplitude depends on the amount of impedance mismatch between the line and the load and whether the rise time of the signal at the source equals or is greater (slower) than two times the propagation delay of the line.

The condition for a voltage reflection to occur is

$$L \geq \frac{t_r}{2T_{pdL}} \qquad Eq.\ 8$$

Solving for the loaded propagation delay yields

$$T_{pdL} = \frac{t_r}{2L} \qquad Eq.\ 9$$

However, the actual physical length of the line is

$$l = \frac{t_r}{2T_{pd}} \qquad Eq.\ 10$$

The intrinsic capacitance of the line from Equation 5 is

$$C_o = \frac{T_{pd}}{Z_o} \qquad Eq.\ 11$$

It is standard practice to use C_o to designate the intrinsic line capacitance, L_o the intrinsic line self inductance, and Z_o the intrinsic line characteristic impedance.

Substituting the expressions from Equations 9, 10, and 11 into Equation 6 gives the relationship for the line length at which voltage reflections might occur. Two conditions must be present for voltage reflections to occur: The line must be long, and there must be an impedance mismatch between the line and the load.

$$\frac{t_r}{2L} = T_{pd}\sqrt{1 + \frac{C_D}{\frac{t_r}{T_{pd}} \times \frac{T_{pd}}{Z_o}}} \qquad Eq.\ 12$$

Solving Equation 12 for the line length, l, yields

$$L = \frac{t_r}{2T_{pd}} \frac{1}{\sqrt{1 + \frac{C_D Z_o}{t_r}}} \qquad Eq.\ 13$$

Equation 13 is very useful to the system designer. It is generic and applies to all products irrespective of circuit type, logic family, or voltage levels. The equation allows you to estimate when a line requires termination, using variables you can easily determine.

When driving a distributed or non-lumped load, the signal's rise time depends on the source — not the load, as you might expect. The intrinsic, or unloaded, line propagation delay per unit length is a function of the dielectric constant and can be easily calculated. The intrinsic line characteristic impedance is a function of the dielectric constant and the PCB's physical construction or geometry and can also be calculated. Finally, you can estimate the equivalent (lumped) load capacitance by adding up the number of loads (device inputs) being driven and multiplying by 10 pF. For I/O pins, use 15 pF per pin.

Signal Transition Times

The standard Cypress 0.8μ (L drawn) CMOS process yields output buffers whose signals transition approximately 4V in 2 ns, or, have a slew rate of 2V per

nanosecond. The rise time/fall time is 2 ns. Products fabricated using the Cypress BiCMOS process have the same rise times.

The Cypress ECL process yields products with 500-ps output signal rise times and fall times, or slew rates of 1V/0.5 ns = 2V per nanosecond. Internal signal slew rates are 10V per nanosecond, but only for short (usually less than 500 mV) voltage excursions. Thus, high-frequency noise is generated on chip, which you can eliminate by using 100- to 500-pF ceramic or mica filter capacitors between V_{cc} and ground.

The values in *Table 1* come from using Equation 13 to calculate the line length at which voltage reflections might occur. The calculations assume a 50Ω intrinsic line characteristic impedance and that the PCB is multilayer, using stripline construction on G-10 glass epoxy material (dielectric constant of 5). These conditions result in an unloaded line propagation delay of 2.27 ns per foot.

Table 1 reveals that decreasing the source rise time from 2 to 0.5 ns (a factor of 4) decreases the line length at which a voltage reflection might occur by a factor of 5 (4.73 divided by 0.93 = 5.09) for the same load (10 pF) and intrinsic propagation delay (2.27 ns/ft.). A second observation is that for signals with rise times of 0.5 ns, you should terminate all lines.

Reflection Coefficients

Another attribute of the ideal transmission line is reflection coefficients, which are not actually line characteristics. The line is treated as a circuit component, and reflection coefficients are defined that measure the impedance mismatches between the line and its source and the line and its load. The reason for defining and presenting the reflection coefficients becomes apparent later when it is shown that if the impedance mismatch is sufficiently large, either a negative or positive voltage reflects back from the load to the source, where the voltage either adds to or subtracts from the original signal. A mismatch between the source and line impedance might also cause a voltage reflection, which in turn reflects back to the load. Therefore, two reflection coefficients are defined.

Table 1. Line Length at which a Voltage Reflection Occurs

t_r (ns)	C_D (pF)	L (inches)
2	10	4.73
2	20	4.32
2	40	3.74
2	80	3.05
1	10	2.16
1	20	1.87
1	40	1.53
1	80	1.18
0.5	10	0.93
0.5	20	0.76
0.5	40	0.59
0.5	80	0.44

For classical transmission lines driven by a single frequency source, the impedance mismatches cause standing waves. When pulses are transmitted and the source's output impedance changes depending upon whether a Low-to-High or a High-to-Low transition occurs, the analysis is complicated further.

You can use classical transmission line analysis — where pulses are represented by complex variables with exponentials — to calculate the voltages at the source and the load after several back and forth reflections. However, these complex equations tend to obscure what is physically happening.

Energy Considerations

Now consider the effects of driving the ideal transmission line with digital pulses and analyze the behavior of the line under various driving and loading conditions. The first task is to define the load and source reflection coefficients.

Figure 3 shows the circuit to be analyzed. The ideal transmission line of length l is driven by a digital source of internal resistance R_S and loaded with a resistive load R_L. The characteristic impedance of the line appears as a pure resistance,

$Z_o = \sqrt{L/C}$

to any excitation.

The ideal case is when $R_S = Z_o = R_L$. The maximum energy transfer from source to load occurs under this condition, and no reflections occur. Half the energy is dissipated in the source resistance, R_S, and the other half is dissipated in the load resistance, R_L (the line is lossless).

If the load resistor is larger than the line's characteristic impedance, extra energy is available at the load and is reflected back to the source. This is called the underdamped condition, because the load under-uses the energy available. If the load resistor is smaller than the line impedance, the load attempts to dissipate more energy than is available. Because this is not possible, a reflection occurs that signals the source to send more energy. This is called the overdamped condition. Both the underdamped and overdamped cases cause negative traveling waves, which cause standing waves if the excitation is sinusoidal. The condition $Z_o = R_L$ is called critically damped.

The safest termination condition, from a systems design viewpoint, is the slightly overdamped condition, because no energy is reflected back to the source.

Line Voltage For a Step Function

To determine the line voltage for a step function excitation, you apply a step function to the ideal line

and analyze the behavior of the line under various loading conditions. The step function response is important because any pulse can be represented by the superposition of a positive step function and a negative step function, delayed in time with respect to each other. By proper superposition, you can predict the response of any line and load to any width pulse. The principle of superposition applies to all linear systems.

According to theory, the rise time of the signal driven by the source is not affected by the characteristics of the line. This has been substantiated in practice by using a special coaxially constructed reed relay that delivers a pulse of 18A into 50Ω with a rise time of 0.070 ns (*Reference 1* pg. 162).

The equation representing the voltage waveform going down the line (*Figure 3*) as a function of distance and time is

$$V_L(X, t) = V_A(t)\ U(t - X\ t_{pd}) \text{ for } t < T_O \qquad \textit{Eq. 14}$$

$$V_A(t) = V_S(t)\left(\frac{Z_o}{Z_o + R_S}\right) \qquad \textit{Eq. 15}$$

where

V_A = the voltage at point A
X = the voltage at a point X on the line
l = the total line length
t_{pd} = the propagation delay of the line in nanoseconds per foot
T_o = l t_{pd}, or the one-way line propagation delay
U(t) = a unit step function occurring at x = 0
$V_s(t)$ = the source voltage

When the incident voltage reaches the end of the line, a reflected voltage, V_L', occurs if R_L does not equal Z_o. The reflection coefficient at the load, ρ_L, can be obtained by applying Ohm's Law.

The voltage at the load is $V_L + V_L'$, which must be equal to $(I_L + I_L')R_L$. But

$$I_L = \frac{V_L}{Z_o}$$

and

$$I_L' = -\frac{V_L'}{Z_o}$$

(The minus sign is due to I_L being negative; i.e., I_L is opposite to the current due to V_L.) Therefore,

$$V_B = V_L + V_L' = \left(\frac{V_L}{Z_o} - \frac{V_L'}{Z_o}\right) R_L \qquad \textit{Eq. 16}$$

By definition:

$$\rho_L = \frac{\textit{reflected voltage}}{\textit{incident voltage}} = \frac{V_L'}{V_L}$$

Solving for V_L'/V_L in Equation 16 and substituting in the equation for ρ_L yields

$$\rho_L = \frac{R_L - Z_o}{R_L + Z_o} \qquad \textit{Eq. 17}$$

The reflection coefficient at the source is

$$\rho_S = \frac{R_S - Z_o}{R_S + Z_o} \qquad \textit{Eq. 18}$$

Rearranging Equation 16 yields

$$V_B = V_L + V_L' = \left(1 + \frac{V_L'}{V_L}\right) V_L \qquad \textit{Eq. 19}$$

$$= (1 + \rho_L)\, V_L$$

Equation 19 describes the voltage at the load (V_B) as the sum of an incident voltage (V_L) and a reflected voltage (ρ_L Vl) at time t = T_o. When $R_L = Z_o$, no voltage is reflected. When $R_L < Z_o$, the reflection coefficient at the load is negative; thus, the reflected voltage subtracts from the incident voltage, giving the load voltage. When $R_L > Z_o$, the reflection coefficient is positive; thus, the reflected voltage adds to the incident voltage, again giving the load voltage.

Note that the reflected voltage at the load has been defined as positive when traveling toward the source. This means that the corresponding current is negative, subtracting from the current driven by the source.

This piecewise analysis is cumbersome and can be tedious. However, it does provide an insight into what is physically happening and demonstrates that a complex problem can be solved by dividing it into a series of simpler problems. Also, eliminating the exponentials — which provide phase information in the classical transmission line equations — simplifies the mathematics. To use the piecewise method, you must do careful bookkeeping to combine the reflections at the proper time. This is quite straightforward, because a pulse travels with a constant velocity along an ideal or low-loss line, and the time delay between reflected pulses can be predicted.

The rules to keep in mind are that at any location and time the voltage or the current is the algebraic sum of the waves traveling in both directions. For example, two voltage waves of the same polarity and equal amplitudes, traveling in opposite directions, at a given location and time add together to yield a voltage of twice the amplitude of one wave. The same reasoning applies to all points of termination and discontinuities on the line. The total voltage or current is the algebraic sum of all the incident and reflected waves. Polarities must be observed. A positive voltage reflection results in a negative current reflection and vice versa.

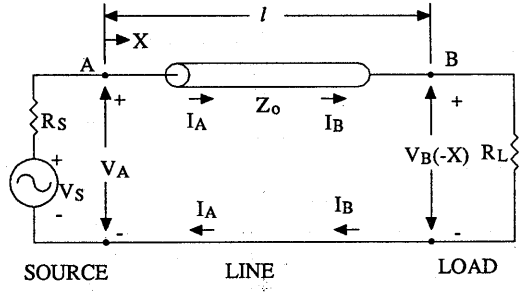


Figure 3. Ideal Transmission Line Loaded and Driven

Step Function Response of the Ideal Line

Before examining reflections at the source due to mismatches between the source and line impedances, consider the behavior of the ideal line with various loads when driven by a step function. The circuit for analysis appears in *Figure 3*. *Figure 4* shows the voltage and current waveforms at point A (line input) and point B (the load) for various loads. (These values are drawn from *Reference 1* pg. 158 - 159.) Note that $R_S = Z_O$ and that V_A at t = 0 equals $V_S/2$. This means that no impedance mismatch exists between the source and the line; thus, there is no reflection from the source at t = $2T_O$. T_O is the one-way propagation delay of the line.

The time-domain response of the reactive loads are obtained by applying a step function to the LaPlace transform of the load, then taking the inverse transform.

Note that the reflection coefficient at the load is not the total reflection coefficient (a complex number) but represents only the real part of the load. The piecewise method eliminates the complex ($j\omega t$) terms by performing the bookkeeping involving the phase relationships, which the complex terms account for in classical transmission line analysis.

Note that for the open-circuit condition in *Figure 4b*, Z_L = infinity, so that $\rho_L = +1$. The voltage is reflected from the load to the source (at amplitude $V_O = V_S/2$). Thus, at time = 2 T_O, the reflected voltage adds to the original voltage, $V_O = V_S/2$, to give a value of $2V_O = V_S$. While the voltage wave is traveling down to and back from the load, a current of

$$I_o = \frac{V_o}{Z_o} = \frac{V_S}{2} Z_o$$

exists. This current charges up the distributed line capacitance to the value V_S, then the current stops.

The waveforms at the source and load for the series RC termination shown in *Figure4g* are of particular interest because this network dissipates no DC power; you can use this network to terminate a transmission line in its characteristic impedance at the input to a Cypress IC. *Figure 4h* represents the equivalent circuit of a Cypress IC's input. Combining both networks models a Cypress IC driven by a transmission line terminated in the line's characteristic impedance, when the values of R and C are properly chosen.

Reflections Due to Discontinuities

Figure 5 illustrates three types of common discontinuities found on transmission lines. Any change in the characteristic impedance of the line due to construction, connectors, loads, etc., causes a discontinuity, which causes a reflection that directs some energy back to the source. The amount of energy reflected back is determined by the discontinuity's reflection coefficient. Because discontinuities are usually small by design, most of the energy is transmitted to the load.

In general, a discontinuity has series inductance, shunt capacitance, and series resistance. An example is a via from a signal plane through a ground plane to a second signal plane in a multilayer PCB or module. IC sockets and other connectors can also cause discontinuities.

Ideal Transmission Line's Pulse Response

Consider next the behavior of the ideal transmission line when driven by a pulse whose width is short compared to the line's electrical length — when the pulse width is less than the line's one-way propagation delay time, T_O.

Figure 6 shows another series of response waveforms for the circuit in *Figure 3*, this time for a pulse instead of a step (drawn from *Reference 1* pg. 160 - 161). Note that $R_S = Z_O$ and that V_A at t = 0 equals $V_S/2$. This means that there is no impedance mismatch between the source and the line; thus, there is no reflection from the source at t = $2T_O$.

Finite Rise Time Effects

Now consider the effects of step functions with finite rise times driving the ideal transmission line. During the rise time of a pulse, half the energy in the static electric field is converted into a traveling magnetic field and half remains as a static electric field to charge the line.

If the rise time is sufficiently short, the voltage at the load changes in discrete steps. The amplitude of the steps depends on the impedance mismatch, and the width of the steps depends on the line's two-way propagation delay.

As the rise time and/or the line gets shorter (smaller T_O), the result converges to the familiar RC time constant, where C is the static capacitance. All devices should be treated as transmission lines for transient analysis when an ideal step function is applied. However, as the rise time becomes longer and/or the traces shorter, the transmission line analysis reduces to conventional AC circuit analysis.

Reflections From Small Discontinuities

Figure 7 shows a pulse with a linear rise time and rounded edges driving the transmission line of *Figures 5a* and *5b*. The expressions for V_r are derived on pages 171 and 172 of *Reference 1*. The reflection caused by the small series inductance is useful for calculating the value of the inductor, L', but little else.

The reflection caused by the small shunt capacitor is more interesting. If this capacitor is sufficiently large, it can cause a device connected to the transmission line to see a logic Zero instead of a logic One.

The Effect of Rise Time on Waveforms

Next, consider the ideal line terminated in a resistance less than its characteristic impedance and driven by a step function with a linear rise time. The stimulus, the circuit, and the response appear in *Figures 8a, b* and *c*, respectively. Once again, note that because the source

$V_A = V_S/2,\ I_O = V_O/Z_O,\ T_O = \ell\sqrt{LC},\ \rho L = (R_L - Z_O)/(R_L + Z_O)$

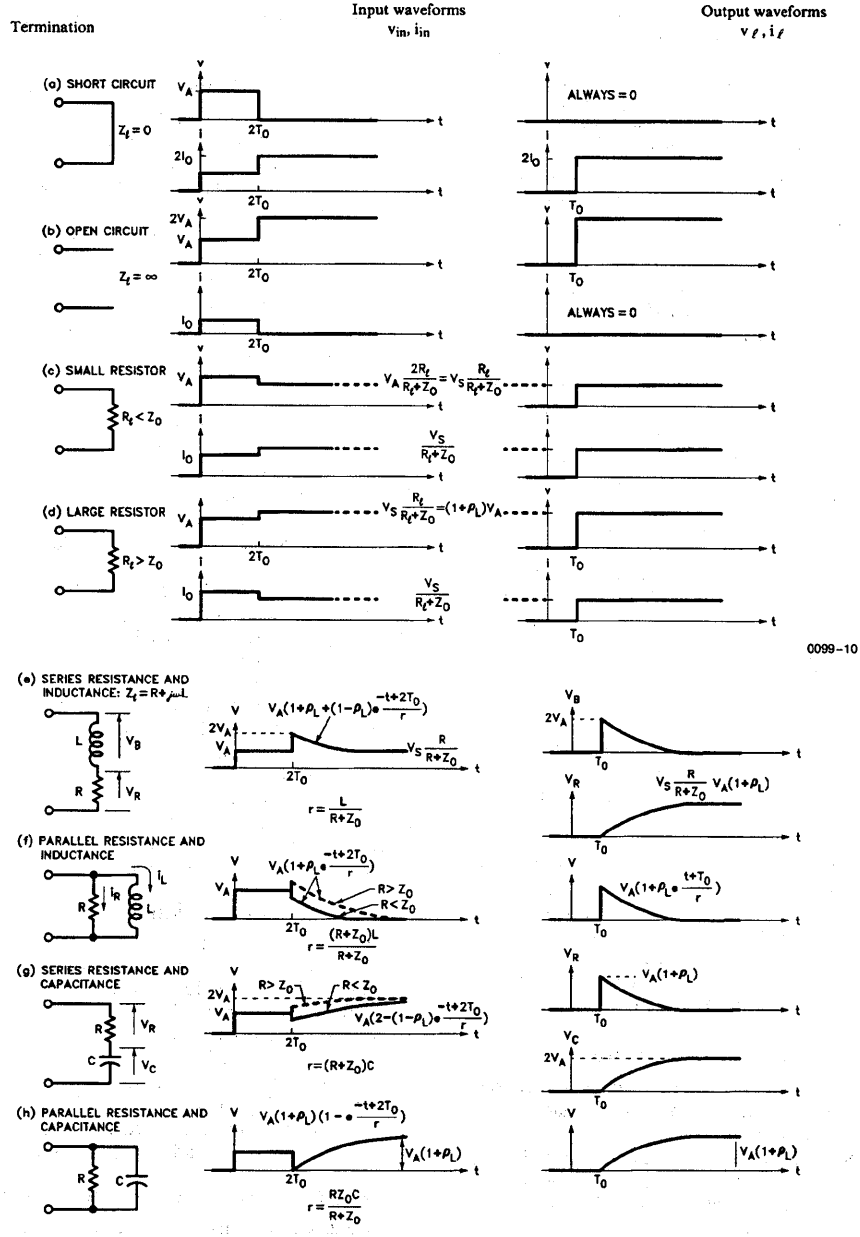


Figure 4. Step Function Response of Figure 3 for Various Terminations

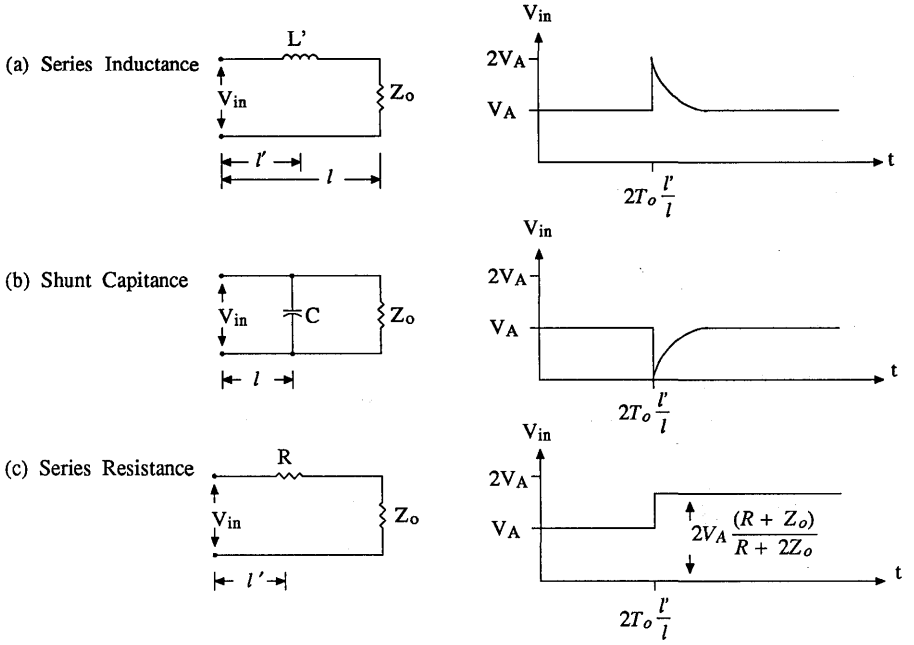


Figure 5. Reflections from Discontinuities with an Applied Step Function

resistance equals the line characteristic impedance, there are no reflections from the source.

The resulting waveforms are similar to those of *Figure 4c* when modified as shown in *Figure 8c*. The waveform's final value must be the same as before (*Figure 4c*).

The resultant wave at the line input (V_{in}) is easily obtained by superposition of the applied wave and the reflected wave at the proper time. In *Figure 8*, because the step function's rise time is less than the line's two-way propagation delay, the input wave reaches its final value, $V_S/2$. At $t = 2T_o$, the reflected wave arrives back at the source and subtracts from the applied step function (the load reflection coefficient is negative). *Figure 9* illustrates waveforms for two relationships between the step function rise time and the propagation delay.

Multiple Reflections

Now consider the case of an ideal transmission line with multiple reflections caused by improper terminations at both ends of the line. The circuit and waveforms appear in *Figure 10*. The reflection coefficients at the source and the load are both negative — the source resistance and the load resistance are both less than the line characteristic impedance.

When the switch is initially closed, a step function of amplitude

$$V_o = V_{in} = \frac{V_S Z_o}{R_S + Z_o}$$

appears on the line and travels toward the load. After a one-way propagation delay time, T_o, the wave reflects back with an amplitude of $\rho_L V_o$.

This first reflected wave than travels back to the source, and at time $t = 2T_o$, the wave reaches the input end of the line. At this time, the first reflection at the source occurs, and a wave of amplitude $\rho_S (\rho_L V_o)$ reflects back to the load. At time $t = 3T_o$, this wave again reflects from the load back to the source with amplitude

$$\rho_L \rho_S (\rho_L V_o) = \rho_S \rho_L^2 V_o$$

This back and forth reflection process continues until the amplitudes of the reflections become so small that they cannot be observed. Then, the circuit is said to be in a quiescent state.

Effective Time Constant

Voltage reflections in small increments and of short durations approximate an exponential function, as indicated by the dashed line in *Figure 10b*. The smaller and narrower the steps become, the more closely the waveform approaches an exponential curve.

The mathematical derivation is presented on pages 178 and 179 of *Reference 1*. The time constant is

$$K = -\frac{2T_o}{1 - \rho_S \rho_L} \qquad \text{Eq. 20}$$

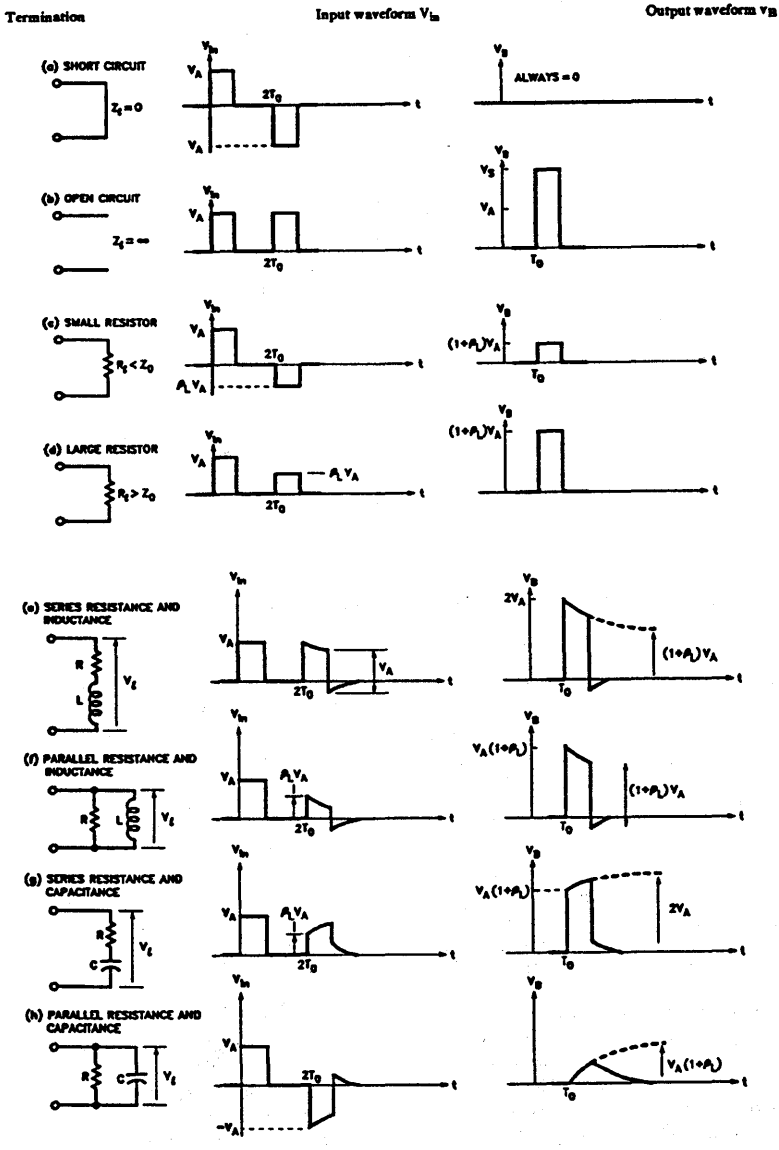


Figure 6. Pulse Response of *Figure 3* for Various Terminations

$V_A = V_S/2$, $I_O = VO/Z_O$, $T_O = l\sqrt{LC}$, $\rho_L = \dfrac{(R_L - Z_o)}{(R_L + Z_o)}$

Thus, the resultant voltage waveform at the load can be approximated by

$$V(t) = V_o e\left(\frac{t}{K}\right) \qquad \textit{Eq. 21}$$

For Equation 21 to be accurate, ρ_L and ρ_S must be reasonably large (approaching ± 1) so that the incremental steps are small. Because the product $\rho_S\rho_L$ is a positive number, less than one, the time constant is a negative number, which indicates that the exponential decreases with time. This is usually the case in transient circuits.

Both reflection coefficients must also have the same sign to yield a continually decreasing or increasing waveform. Opposite signs give oscillatory behavior that cannot be represented by an exponential function.

From Transmission Line to Circuit Analysis

When a transmission line is terminated in its characteristic impedance, the line behaves like a resistor. It usually does not matter if you use transmission line or circuit analysis, provided that you take the propagation delays into account.

Consider the case of a short-circuited transmission line driven by a step function with a source impedance unequal to the characteristic line impedance. The general case is shown in *Figure 10a*. For $R_L = 0$ the reflection coefficients are

$$\rho_S = \frac{Z_S - Z_o}{Z_S + Z_o} \qquad \rho_L = -1$$

The approximate time constant is

$$-k = \frac{2T_o}{1 - \rho_S\rho_L} = \frac{2T_o}{1 + \rho_S} = \frac{T_o(Z_S + Z_o)}{Z_S}$$

$$\text{or} \quad -k = T_o + \frac{T_o Z_o}{Z_S} \qquad \textit{Eq. 22}$$

Recall that

$T_o = l\sqrt{LC}$

(one-way delay) and

$Z_o = \sqrt{L/C}$

where l is the physical length of the line, and L and C are the per-unit-length parameters. Substituting these variables into Equation 22 yields

$$-k = T_o + l\frac{L}{Z_S}$$

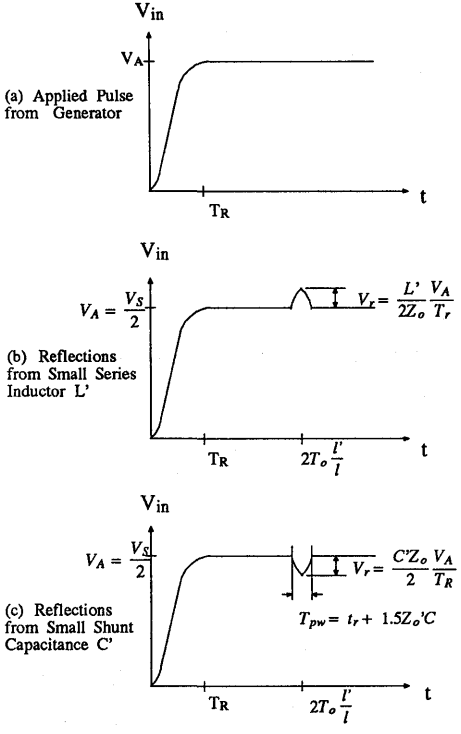


Figure 7. Reflections From Small Discontinuities with a Finite Rise Time Pulse

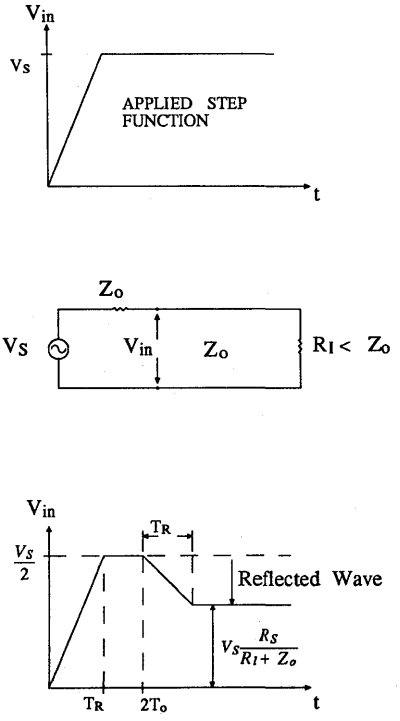


Figure 8. Effect of Rise Time on Response of Mismatched Line with $R_I < Z_o$

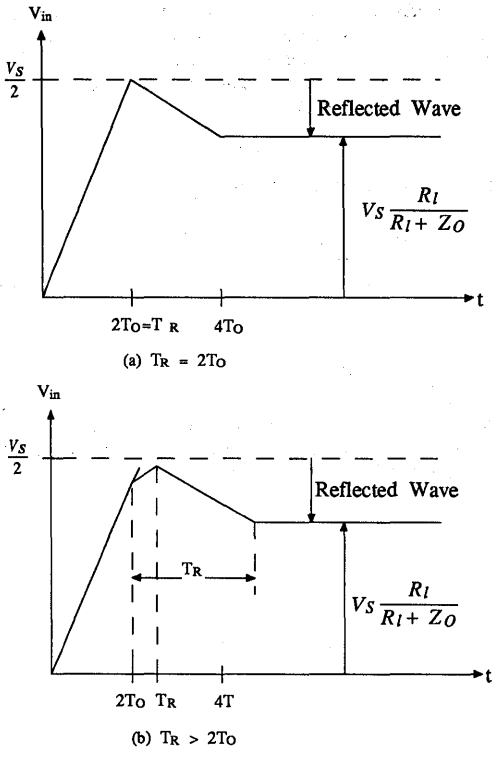


Figure 9. Effects of Rise Time on Response for $R_l < Z_o$

It is necessary to have Z_s smaller than Z_o. Thus, the reflection coefficients have the same sign to give exponential behavior. Opposite signs give oscillatory behavior.

If $Z_s < Z_o$, the exponential approximation becomes more accurate. If Z_s is very small compared to Z_o, then T_o is negligible compared to l L/Z_o, so that Equation 22 reduces to

$$k = -l\frac{L}{Z_S}$$

But l L is the total loop inductance, and Z_s is the circuit's total series impedance. The time constant is then

$$k = \frac{L'}{R_S}$$

This is the same time constant you would obtain by a circuit analysis approach if you considered the line a series combination of L′ and R_s. By open-circuiting the line and performing a similar analysis, it can be shown that an RC time constant results.

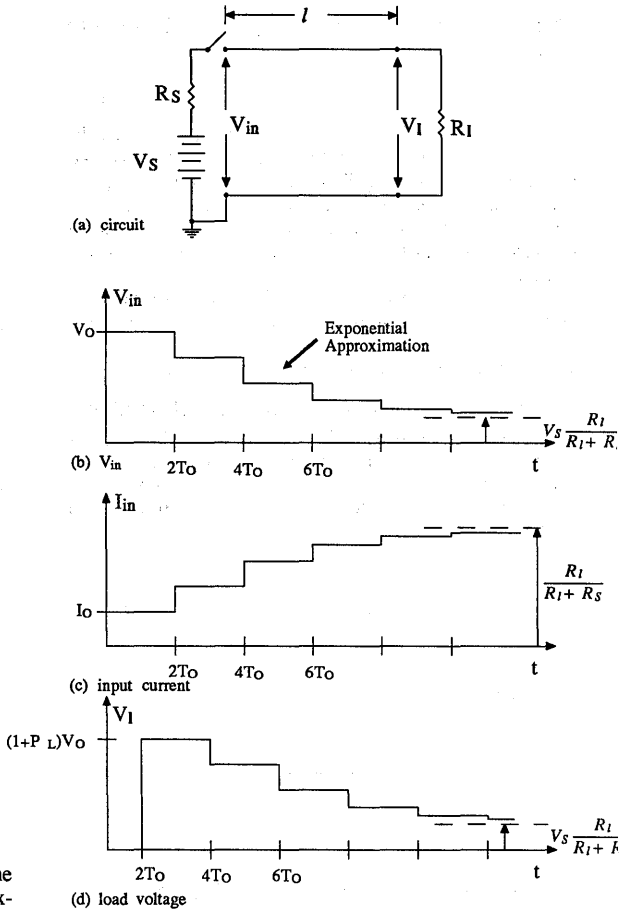


Figure 10. Step Function Applied to Line Mismatched on Both Ends; Shown for Negative Values of ρ_S and ρ_L

Types of Transmission Lines

The types of transmission lines include

- Coaxial cable
- Twisted pair
- Wire over ground
- Microstrip lines
- Strip lines

Coaxial Cable

Coaxial cable offers many advantages for distributing high-frequency signals. The well-defined and uniform characteristic impedance permits easy matching. The cable's ground shield reduces crosstalk, and the low attenuation at high frequencies make the cable ideal for transmitting the fast rise- and fall-time signals

generated by Cypress CMOS ICs. However, because of its high cost, coaxial cable is usually restricted to applications that permit no alternatives. These applications usually involve clock distribution systems on PCBs or backplanes.

Because coaxial cable is not easily handled by automated assembly techniques, its application requires human assemblers. This requirement further increases costs.

Coaxial cables have characteristic impedances of 50, 75, 93, or 150Ω. These values are the most common, although special cables can be made with other impedances.

Coaxial cable's propagation delay is very low. You can compute it using the formula

$$T_{pd} = 1.017\sqrt{e_r} \quad (ns/ft) \qquad Eq.\ 23$$

where e_r is the relative dielectric constant and depends upon the dielectric material used. For solid Teflon and polyethylene, the dielectric constant is 2.3. The propagation delay is 1.54 ns per foot. For maximum propagation velocity, you can use coaxial cables with dielectric Styrofoam or polystyrene beads in air. Many of these cables have high characteristic impedances and are slowed considerably when capacitively loaded.

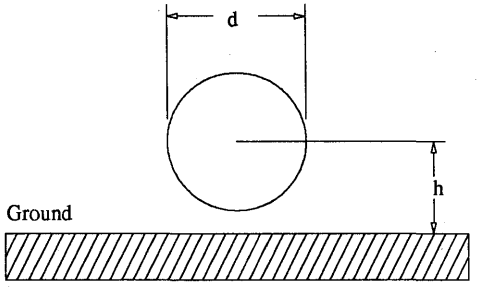
Twisted Pair

You can make twisted pairs from standard wire (AWG 24 - 28), twisted about 30 turns per foot. The typical characteristic impedance is 110Ω.

Because the propagation delay is directly proportional to the characteristic impedance (Equation 5), the propagation delay is approximately twice that of coaxial cable. Twisted pairs are used for backplane wiring, sometimes for driving differential receivers, and for breadboarding.

Wire Over Ground

Figure11 shows a wire over ground. This configuration is used for breadboarding and backplane wiring.



$$Z_o = \frac{60}{\sqrt{e_r}} \ln\left(\frac{4h}{d}\right)$$

Figure 11. Wire Over Ground

The characteristic impedance is approximately 120Ω. This value can vary as much as ± 40 percent, depending upon the distance from the ground plane, the proximity of other wires, and the configuration of the ground.

Microstrip Lines

A microstrip line (*Figure 12*) is a strip conductor (signal line) on a PCB separated from a ground plane by a dielectric. If the line's thickness, width, and distance from the ground plane are controlled, the line's characteristic impedance can be predicted with a tolerance of ± 5 percent.

The formula given in *Figure 12* has proven to be very accurate for width-to-height ratios between 0.1:1 and 3.0:1 and for dielectric constants between 1 and 15.

The inductance per foot for microstrip lines is

$$L = (Z_o)^2 C_o \qquad Eq.\ 24$$

where Z_o is the characteristic impedance and C_o is the capacitance per foot.

The propagation delay of a microstrip line is

$$T_{pd} = 1.017\sqrt{0.45\,e_r + 0.67} \quad (ns/ft) \qquad Eq.\ 25$$

Note that the propagation delay depends only upon the dielectric constant and is not a function of the line width or spacing. For G-10 fiberglass epoxy PCBs (dielectric constant of 5), the propagation delay is 1.74 ns per foot.

Strip Line

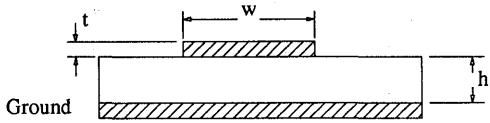
A strip line consists of a copper strip centered in a dielectric between two conducting planes (*Figure13*). If the line's thickness, width, dielectric constant, and distance between ground planes are all controlled, the tolerance of the characteristic impedance is within ± 5 percent. The equation given in *Figure13* is accurate for $W/(b - t) < 0.35$ and $t/b < 0.25$.

The inductance per foot is given by the formula

$$L = (Z_o)^2 C_o$$

The propagation delay of the line is given by the formula

$$T_{pd} = 1.017\sqrt{e_r} \quad (ns/ft) \qquad Eq.\ 26$$



$$Z_o = \frac{87}{\sqrt{e_r + 1.41}} \ln\left(\frac{5.98h}{0.8w + t}\right)$$

Figure 12. Microstrip Line

For G-10 fiberglass epoxy boards, the propagation delay is 2.27 ns per foot. The propagation delay is not a function of line width or spacing.

Modern PCBs

Most PCBs employ microstrip, stripline, or some combination of the two. Microstrip construction on a double-sided board with power and ground nets can suffice for low- to medium-performance, and low-density PCBs.

For high-performance, high-density PCBs, stripline construction is preferred. Power planes isolate signal layers from each other and provide higher-quality power and grounds than those of a two-layer board. Manufacturing quality control assures that the metalization is of uniform thickness and that the layers are properly laminated, thus ensuring uniform, predictable electrical characteristics.

When to Terminate Transmission Lines

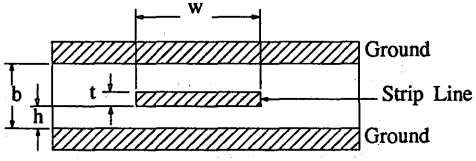
Transmission lines should be terminated when they are long. From the preceding analysis, it should be apparent that

$$Long\,Line > \frac{T_r}{2\,T_{pdL}}$$

where T_{pdL} is the loaded propagation delay of the line per unit length. For Cypress CMOS and BiCMOS products, the rise time, T_r, is typically 2 ns.

For stripline construction (multilayer PCBs), the line length at which voltage reflections occur has been shown to vary from 4.73 inches for a 10-pF load to 3.05 inches for an 80-pF load (see Equation 13 and *Table 1*).

Not all lines exceeding these lengths need to be terminated. Terminations are usually required on control lines (such as clock inputs, write and read strobe lines on SRAMs and FIFOs) and chip select or output-enable lines on RAMs, PROMs, and PLDs. Address lines and data lines on RAMs and PROMS usually have time to settle because they are normally not the highest-frequency lines in a system. However, if very heavily loaded, address and data bus lines might require terminations.



$$Z_O = \frac{60}{\sqrt{e_r}} \ln\left(\frac{4b}{0.67\pi w\left(0.8 + \frac{t}{w}\right)}\right)$$

Figure 13. Strip Line Construction

Line Termination Strategies

There are two general strategies for transmission line termination:

- Match the load impedance to the line impedance
- Match the source impedance to the line impedance

In other words, if either the load reflection coefficient or the source reflection coefficient can be made to equal zero, reflections are eliminated. From a systems design viewpoint, strategy 1 is preferred. Eliminating the reflection at the load (i.e., dissipating the excess energy) before the energy travels back to the source causes less noise, electromagnetic interference (EMI), and radio frequency interference (RFI).

Multiple Loads, Buses, and Nodes

In the case where multiple loads are connected to a transmission line, only one termination circuit is required. The termination should be located at the load that is electrically the greatest distance from the source. This is usually the load that is the greatest physical distance from the source. A point-to-point or daisy chain connection of loads is preferred.

Bidirectional buses should be terminated at each end with a circuit whose impedance equals the intrinsic, characteristic line impedance. The reason is that each transmitting device sees the characteristic impedance of the line when the device is transmitting.

Consider next a line that has three bidirectional nodes: one on each end and one in the middle. The middle node, when driving the line, sees an impedance equal to $Z_o/2$, because the node is looking into two lines in parallel with each other. The end nodes, however, see an impedance of Z_o. In this case, as in a backplane, each end of the line should be terminated in an impedance equal to $Z_o/2$.

Types of Terminations

There are three basic types of terminations: series damping, pull-up/pull-down, and parallel AC terminations. Each has its advantages and disadvantages.

Except for series damping, the termination network should be attached to the input (load) that is electrically the greatest distance from the source. Component leads should be as short as possible to prevent reflections due to lead inductance.

Series Damping

Series damping is accomplished by inserting a small resistor (typically 10 to 75Ω) in series with the transmission line, as close to the source as possible (*Figure 14*). Series damping is a special case of damping in which the series resistor value plus the circuit output impedance equals the transmission line impedance. The strategy is to prevent the wave reflected back from the load from reflecting back from the source. This is done by making the source reflection coefficient equal to zero.

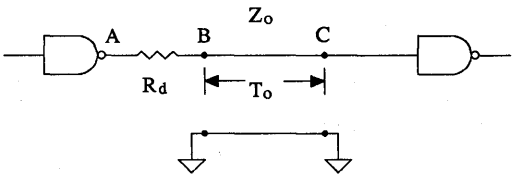


Figure 14. Series Damping Termination

The channel resistance (On resistance) of the pull-down device for Cypress ICs is 10 to 20Ω, depending upon the current-sinking requirements. Thus, subtract this value from the series damping resistor, R_d.

$$Z_o = R_S + R_d \qquad Eq.\ 27$$

A disadvantage of the series damping technique, as illustrated in *Figure 15*, is that during the two-way propagation delay time of the signal edges, the voltage at the input to the line is halfway between the logic levels, due to the voltage divider action of R_d. The "half voltage" propagates down the line to the load and then back from the load to the source. This means that no inputs can be attached along the line, because they would respond incorrectly during this time. However, you can attach any number of devices to the load end of the line because all the reflections are absorbed at the source. If two or more transmission lines must be driven in parallel, the value of the series damping resistor does not change.

The advantages of series termination are

- Requires only one resistor per line
- Consumes little power
- Permits incident wave switching at the load after a T_o propagation delay
- Provides current limiting when driving highly capacitive loads; the current limiting also helps avoid ground bounce

The disadvantages of series termination are

- Degrades rise time at the load due to increased RC time constant
- Should not be used with distributed loads

The low input current required by Cypress CMOS ICs results in essentially no DC power dissipation. The only AC power required is to charge and discharge the parasitic capacitances.

Pull-Up/Pull-Down Termination

The pull-up/pull-down resistor termination shown in *Figure 16* is included for historical reasons and for the sake of completeness. For TTL driving long cables, such as ribbon cables, the values R_1 = 220Ω and R_2 = 330Ω are recommended by several bus interface standards. If the cable is disconnected, the voltage at point B is 3V, which is well above the 2V minimum High TTL specification. Because most control signals are active Low, a disconnected cable results in the unasserted state.

The maximum value of R_1 is determined by the maximum acceptable signal rise time, which is a function of the charging RC time constant. The minimum value of R_1 is determined by the amount of current the driver can sink. The value of R_2 is chosen such that a logic High is maintained when the cable is disconnected and the equivalent Thevenin resistance is

$$R_T = \frac{R_1 R_2}{R_1 + R_2}$$

The value of R_1 and R_2 in parallel is slightly less than the cable's characteristic impedance. Ribbon cables with characteristic impedances of 150Ω are typical.

If both resistors are used, DC power is dissipated all the time. If only a pull-down resistor (R_2) is used,

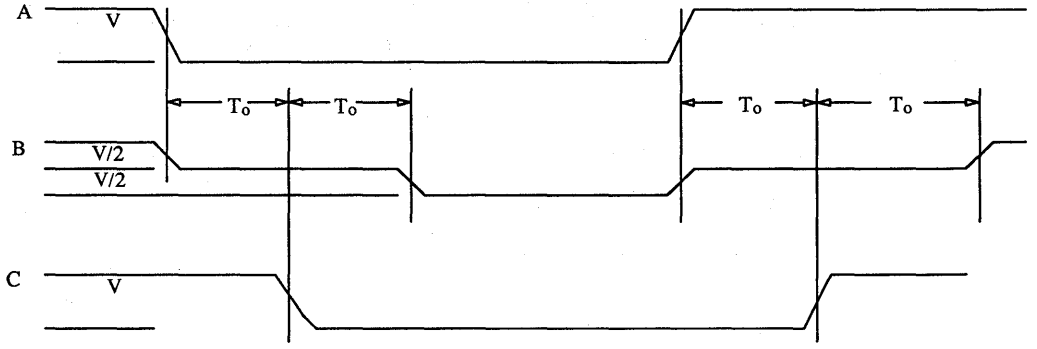


Figure 15. Series Damping Timing

DC power is dissipated when the input is in the logic High state. Conversely, if only a pull-up resistor (R_1) is used, power is dissipated when the input is in the Low state. Due to these power dissipations, this termination is not recommended.

If an unterminated control signal on a PCB is suspected of causing a problem, a resistor whose value is slightly less than the characteristic impedance of the line (e.g., 47Ω) can be connected between the input pin and ground. Be sure that the driver can source sufficient current to develop a TTL High voltage level (2.0V) across the resistor.

In special cases where inputs should be either pulled up (High) for logic reasons or because of very slow rise and fall times, you can use a pull-up resistor to V_{CC} in conjunction with the terminating network shown in *Figure 17*. DC power is dissipated when the source is Low.

Parallel AC Termination

Figure 17 illustrates the recommended general-purpose termination. It does not have the disadvantage of the half-voltage levels of series damping terminations, and it causes no DC power dissipation. You can attach loads anywhere along the line, and they see a full voltage swing.

The disadvantage is that a parallel AC termination requires two components, versus the one-component series damping termination.

Commercially Available RC Networks

A variety of combinations of R and C values are available as series RC networks in SIP packages from at least two sources.

Bourns calls these networks the Series 701 and 702 RC Termination Networks. You can obtain data sheets by calling the factory in Logan, Utah (801-750-7200) or a local sales office.

Thin Film Technology also refers to the networks as RC Termination Networks. You can obtain data sheets by calling the factory in North Mankato, Minnesota at 507-635-8445.

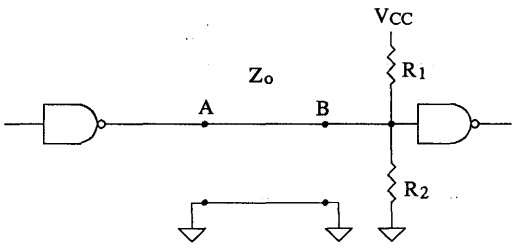


Figure 16. Pullup/Pulldown

Low-Pass Filter Analysis

The parallel AC termination has another advantage: It acts as a low-pass filter for short pulses. You can verify this by analyzing the response of the circuit illustrated in *Figure 18* to a positive and a negative step function. The positive step function is generated by moving the switch from position 2 to position 1. The negative step function is generated by moving the switch from position 1 to position 2. The response of the circuit to a pulse is the superposition of the two separate responses. The input impedance of the Cypress circuits connected to the termination network are so large that they can be ignored for this analysis.

Classic circuit analysis is usually assumes an ideal source ($R_1 = R_2 = 0$). In real-world digital circuits, the source output impedance is not only non-zero, but also varies depending upon whether the output is changing from Low to High or vice versa.

For Cypress ICs, $100\Omega > R_1 > 50\Omega$ and $20\Omega > R_2 > 10\Omega$, depending upon speed and output current-sinking requirements.

Positive Step Function Response

The initial voltage on the capacitor is zero. At t = 0, the switch is moved from position 2 to position 1. At t = 0+, the capacitor appears as a short circuit, and the voltage V is applied through R_1 to charge the load (R_3C). The voltage across the capacitor $V_c(t)$, is

$$V_c(t) = V\left(1 - e^{\left[\frac{-t}{(R1+R3)C}\right]}\right) \qquad \textit{Eq. 28}$$

In theory, the voltage across the capacitor reaches V when t equals infinity. In practice, the voltage reaches 98 percent of V after 3.9 RC time constants. You can verify this by setting $V_c(t)/V = 0.98$ in Equation 28 and solving for t.

Negative Step Function Response

The capacitor is charged to approximately V. At t = 0, the switch is moved from position 1 to position 2, and the capacitor is discharged. The voltage across the capacitor, $V_c(t)$ is

$$V_c(t) = Ve^{\left[\frac{-t}{(R2+R3)C}\right]} \qquad \textit{Eq. 29}$$

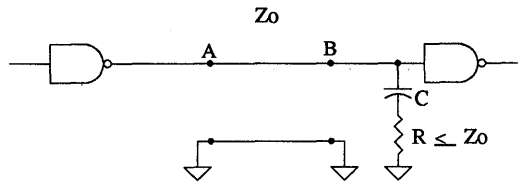


Figure 17. Parallel AC Termination

The voltage decays to 2 percent of its original value in 3.9 RC time constants. You can verify this by setting $V_c(t)/V = 0.02$ in Equation 29 and solving for t.

The Ideal Case

Consider the ideal case, where $R_1 = R_2 = 0$. Let $R_3 = R$ in Equations 28 and 29. If a positive pulse of width T is applied to the modified circuit of *Figure 18*, the pulse disappears if 4RC > T.

Because the discharging time constant is the same as the charging time constant for the ideal case, a negative-going pulse of width T also disappears if 4RC > T. That is, if the applied signal is normally High and goes Low, as does the write strobe on an SRAM, the termination filters out all negative glitches less than 4 RC time constants in width.

The maximum frequency that the circuit passes is

$$F(max.) = \frac{1}{2T} \qquad Eq.\ 30$$

This is true because the charging and discharging time constants are equal for the ideal case.

Capacitance for the Ideal Case

The value of the capacitor, C, must be chosen to satisfy two conflicting requirements. First, the capacitor should be large enough to either absorb or supply the energy contained or removed when positive-going or negative-going glitches occur. Second, the capacitor should be small enough to avoid either delaying the signal beyond some design limit or slowing the signal rise and fall times to more than 5 ns.

A third consideration is the impedance caused by the capacitor's capacitive reactance, X_C. The digital waveforms applied to the AC termination can be expressed as a Fourier Series, so that they can be manipulated mathematically. However, because these signals are not periodic in the classical meaning of the word, it is not clear that the AC steady-state analysis model of X_C applies here.

In most applications, the degradation of the signal's rise and fall times beyond 5 ns determines the maximum value of the capacitor. The procedure is to calculate the rise time between the 10- and 90-percent amplitude levels, equate this rise time to 5 ns, and solve for C in terms of R:

$$V(t) = V\left(1 - e^{\left[\frac{-t}{RC}\right]}\right)$$

for t yields

$$t = RC \ln\left[\frac{1}{1 - \frac{V(t)}{V}}\right] \qquad Eq.\ 31$$

For $\frac{V(t)}{V} = 0.1$, $t = 0.10RC$.

For $\frac{V(t)}{V} = 0.9$, $t = 2.3RC$.

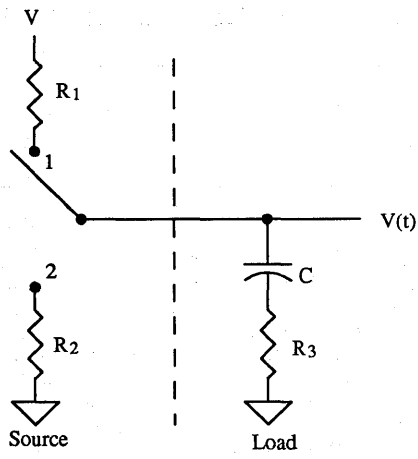


Figure 18. Lumped Load; AC Termination

The time for the signal to transition from 10 to 90 percent of its final value is then T = 2.2 RC. Solving for C yields

$$C = \frac{T}{2.2R} \qquad Eq.\ 32$$

For T = 5 ns, *Table 2* can be constructed. This table indicates that 50Ω transmission lines on PCBs that are terminated with RC networks should use a 47Ω resistor and a capacitor of 48 pF max; 47 pF is a standard value. This network eliminates glitches of 9 ns or less. The table's second column applies to wirewrapping construction, which is not recommended for systems operating at frequencies over 10 MHz. An exception is if the system consists of less than six MSI or SSI ICs.

The Real World

To go from the ideal to the real world, calculate the values of R_1 and R_2 from the curves on the data sheet of the device driving the line. R_1 is the slope of the output source current vs. output voltage between 2 and 4V. R_2 is the slope of the output sink current vs output voltage between 0 and 0.8V.

Add the value of R_1 to 47Ω and calculate C, using Equation 32. Then check to see that the RC charging time constant does not violate some minimum positive pulse-width specification for the line. If so, reduce C.

Add the value of R_2 to 47Ω and calculate C. Then check to see if the discharging RC time constant violates some minimum pulse-width specification for the line. If so, reduce C.

Schottky Diode Termination

In some cases it can be expedient to use Schottky diodes or fast-switching silicon diodes to terminate

Table 2. Termination Values for an Ideal Case

	PCB	Wirewrapped
Z_o (Ω)	50	120
R (Ω)	47	110
C (max., pF)	48	20
RC (ns)	2.25	2.2
4RC (ns)	9	8.8

lines. The diode switching time must be at least as fast as the signal rise time. Where line impedances are not well defined, as in breadboards and backplanes, the use of diode terminations is convenient and can save time.

A typical diode termination appears in *Figure 19*. The Schottky diode's low forward voltage, V_f (typically 0.3 to 0.45V), clamps the input signal to a V_f below ground (lower diode) and V_{cc} + V_f (upper diode). This significantly reduces signal undershoot and overshoot. Some applications might not require both diodes.

The advantages of diode terminations are:

- Impedance matched lines are not required.
- The diodes replace terminating resistors or RC terminations.
- The diodes' clamping action reduces overshoot and undershoot.
- Although diodes cost more than resistors, the total cost of layout might be less because a precise, controlled transmission-line environment is not required.
- If ringing is discovered to be a problem during system debug, the diodes can be easily added.

As with resister or RC terminations, the leads should be as short as possible to avoid ringing due to lead inductance.

A few of the types of Schottky diodes commercially available are

- 1N4148 (switching diode)
- 1N5711
- MBD101, MBD102 (Motorola)
- SN74S1050/52/56 (TI, single-diode arrays)
- SN74S1051/53 (TI, double-diode arrays)

Unterminated Line Example

The following example is presented to illustrate the procedure for calculating the waveforms when a Cypress PLD generates the write strobe for four Cypress FIFOs. The PLD is a PAL C 16L8 device and the FIFOs are CY7C429s.

The equivalent circuit appears in *Figure20* and the unmodified driving waveform in *Figure21*. The rise and fall times are 2 ns. The length of the stripline trace on the PCB is 8 inches and the intrinsic characteristic line impedance is 50Ω. The voltage waveforms at the source (point A) and the load (point B) must be calculated as functions of time. Stripline construction is used for this

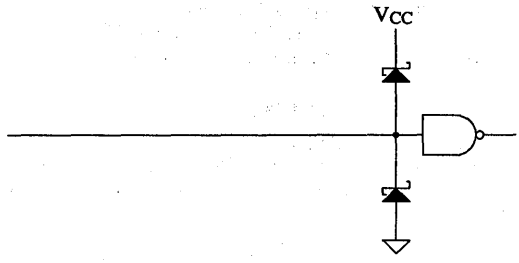


Figure 19. Schottky Diode Termination

example because in most modern high-performance digital systems, the PCBs have multiple layers.

The equivalent On channel resistance of the PLD pull-up device, 62Ω, is calculated using the output source current versus voltage graph, over the region of interest (2 to 4V), from the PAL C 20 series data sheet. The equivalent resistance of the pull-down device, 11Ω, is calculated in a similar manner, using the output sink current versus output voltage graph, over the region of interest (0.4 to 2V), also on the data sheet.

The equivalent input circuit for the FIFO is constructed by approximating the input and stray capacitance with a 10-pF capacitor and the input resistance with a 5-MΩ resistor. The input leakage current for all Cypress products is specified as a maximum of ± 10 μA, which guarantees a minimum of 500 KΩ at V_{in} = 5V. Typical leakage current is 10 pA.

Because the PLD is driving four FIFOs in parallel, the equivalent lumped capacitance is 4 X 10 pF = 40 pF, and the equivalent lumped resistance is 5,000,000/4 = 1.25 MΩ.

The next step is to calculate the propagation delay and the loaded characteristic impedance of the line. The unloaded propagation delay of the line is calculated using Equation 26 with a dielectric constant of 5:

$$T_{pd} = 2.27\ ns/ft$$

To calculate the loaded line propagation delay, the intrinsic capacitance must first be calculated using Equation 5.

$$T_{pd} = Z_o C_o$$

where Z_o is the intrinsic characteristic impedance, and C_o is the intrinsic capacitance.

$$C_o = \frac{T_{pd}}{Z_o} = \frac{2.27\ ns/ft}{50} = 45.4\ pF/ft.$$

Because the line is loaded with 40 pF, Equation 6 is used to compute the loaded propagation delay of the line.

$$T_{pdL} = T_{pd}\sqrt{1 + C_D/C_o}$$

$$T_{pdL} = 2.27\ ns/ft \sqrt{1 + \frac{40\ pF}{45.4\ pF/ft \times \frac{8\ in.}{12\ in./ft}}}$$

$$T_{pdL} = 3.46\ ns/ft.$$

Note that the capacitance per unit length must be multiplied by the line length to arrive at an equivalent lumped capacitance.

The intrinsic line impedance is reduced by the same factor by which the propagation delay is increased (1.524; see Equation 7):

$$Z_o' = \frac{50\Omega}{1.524} = 32.8\Omega$$

Initial Conditions

At time t = 0, the circuit shown in *Figure20* is in a quiescent state. The voltage at points A and B must be the same. By inspection:

$$V_A = V_B = (V_{CC} - Vf)\left(\frac{R_L}{R_S + R_L}\right)$$

$$= (5 - 1)\left(\frac{5 \times 10^6}{28 + 5 \times 10^6}\right) = 4V$$

At t = 0, the driving waveform changes from 4V to approximately 0V with a fall time of 2 ns. This is shown in *Figure20* by the switch arm moving from position 1 to position 2.

The wave propagates to the load at the rate of 3.46 ns per foot and arrives there

$$T_o = 3.46\,ns/ft \times \frac{8\,in.}{12\,in./ft} = 2.3\,ns$$

later, as illustrated in *Figure22b*.

Because the reflection coefficient at the load is ρL = 1, an early equal and opposite polarity waveform is propagated back to the source from the load. The reflection arrives at t = $2T_o$ = 4.6 ns (*Figure 22a*). Note that the fall time is preserved.

The reflection coefficient at the source is:

$$\rho_S = \frac{R_S - Z_o'}{R_S + Z_o'} = \frac{11 - 32.8}{11 + 32.8} = -0.498$$

To simplify the calculations that follow, consider -0.5 to be the Low-level source reflection coefficient.

The magnitude of the reflected voltage at the source is then

$V_{S1} = -4V \times (-0.5) = 2V.$

This wave propagates from the source to the load and arrives at t = $3T_o$. The wave adds to the 0V signal. The rise time is preserved, and thus the time required for the signal to go from 0 to 2V is

$$t_r = \frac{2V \times 2\,ns}{4V} = 1\,ns.$$

The signal at the load thus reaches the 2V level at time

$t = 3T_o + 1\,ns = 7.9\,ns.$

and remains at that level until the next reflection occurs at

$t = 5T_o$

The wave that arrives at the load at $3T_o$ reflects back to the source and arrives at

$t = 4T_o = 9.2\,ns.$

The 2V level adds to the -4V level, for a total of -2V. The rise time is preserved, so that this level is reached at

$t = 4T_o + 1\,ns = 10.2\,ns.$

and maintained until the next reflection occurs at

$t = 6T_o.$

The 2V wave that arrives at the source at t = $4T_o$ reflects back to the load and arrives at t = $5T_o$. The portion that is reflected back to the load is

$V_{S2} = 2 \times (-0.5) = -1V.$

This value subtracts from the 2V level to give 2 - 1 = 1V. Because the fall time is preserved, the time required for the signal to go from 2 to 1V is

$$t_f = \frac{1V \times 2\,ns}{4V} = 0.5\,ns.$$

The 1V level is thus reached at time

$t = 5T_o + 0.5\,ns = 12\,ns.$

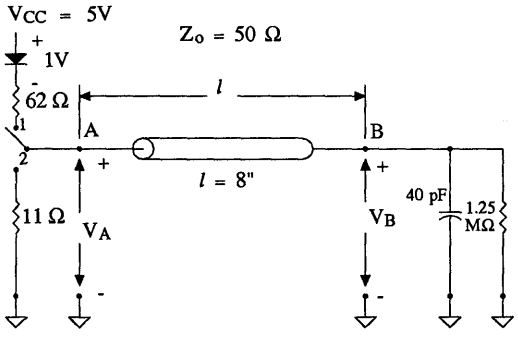


Figure 20. Equivalent Circuit for Cypress PAL Driving

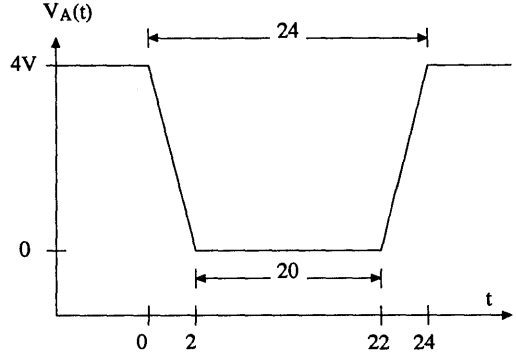


Figure 21. VA(t), Unmodified

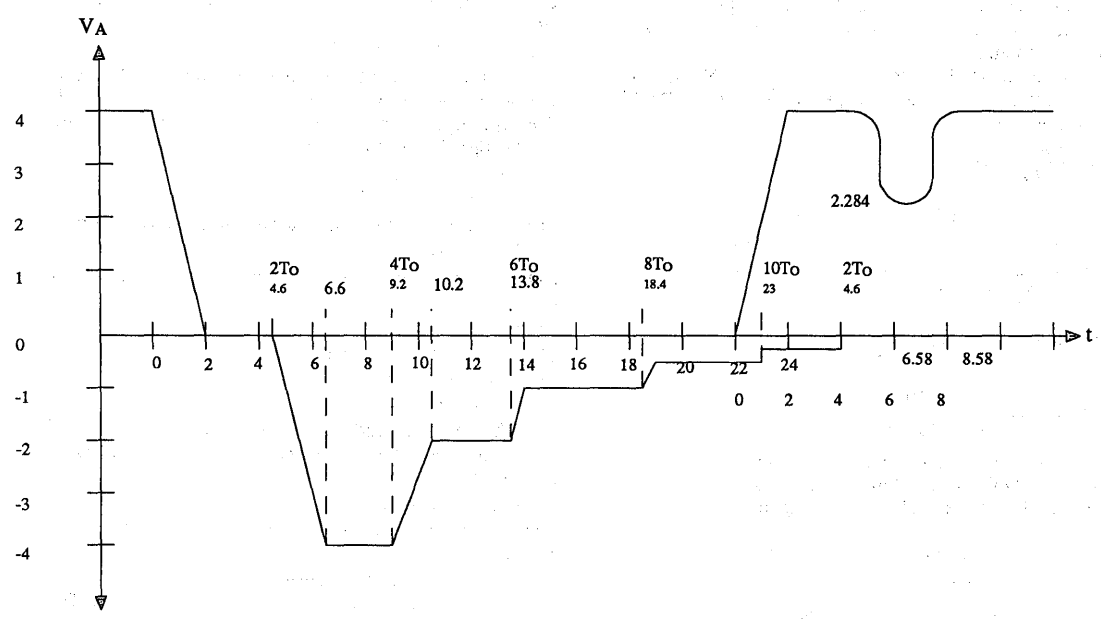


Figure 22(a). Unterminated Line Example; VA(t)

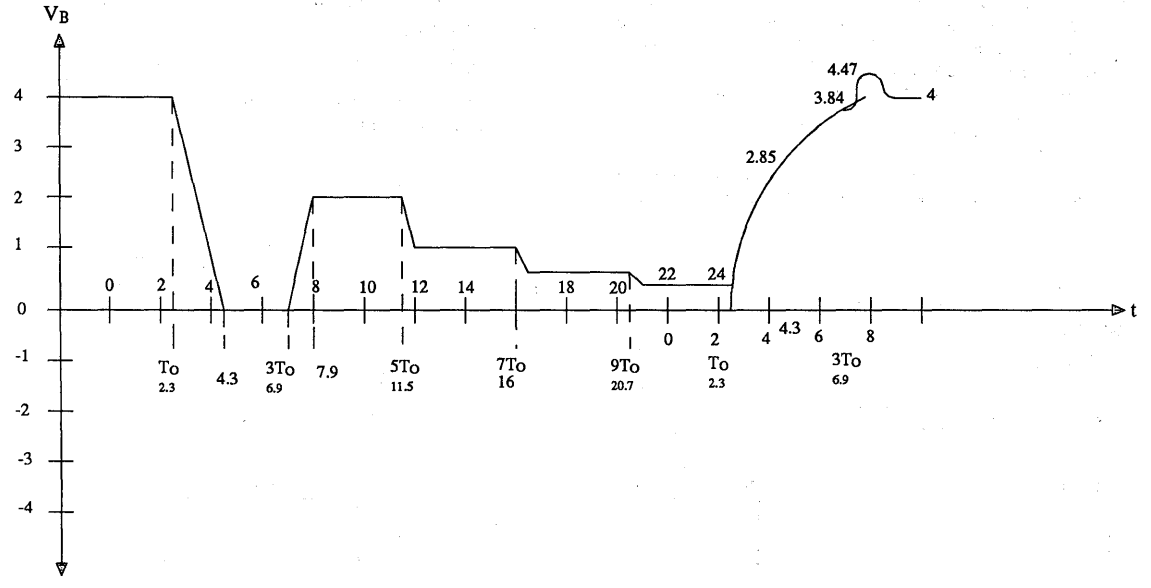


Figure 22(b). Unterminated Line Example; VB(t)

At t = $6T_o$, the 1V wave arrives back at the source, where it subtracts from the - 2V level to give -1V. The rise time is

$t_r = 1 \times 0.5\,ns/V = 0.5\,ns.$

The signal at the source reaches the 1V level at

$t = 6T_o + 0.5 = 14.3\,ns.$

The 1V wave that arrives at the source at t = $6T_o$ is reflected back to the load and arrives at t = $7T_o$. The portion that is reflected back is

$V_{S3} = 1 \times (-0.5) = -0.5V.$

This value subtracts from the 1V level to give 0.5V. The fall time is 0.25 ns. The 0.5V level remains until the next reflection reaches the load at

$t = 9T_o$

At t = $8T_o$ the 0.5V wave that reflects from the load at t = $7T_o$ arrives back at the source, where it subtracts from the - 1V level to give - 0.5V. The rise time is 0.25 ns. The portion that reflects back to the load is

$V_{S4} = 0.5 \times (-0.5) = -0.25V.$

The -0.25V signal arrives at the load at t = $10T_o$ = 23 ns and subtracts from the 0.5V signal to give 0.25V.

This process continues until the voltages at points A and B decay to approximately 0V.

Observations

The positive reflection coefficient at the load and the negative reflection coefficient at the source result in an oscillatory behavior that eventually decays to acceptable levels. The voltage at point A reaches -1V after 6 T_o delays and the voltage at point B reaches 0.5V after 7 T_o delays.

The reflection at the load that causes the voltage to equal the TTL minimum One level (2V) at T = $3T_o$ causes a problem. The actual input voltage threshold level is 1.5V for TTL-compatible devices that do not exhibit hysteresis.

The voltage at the load falls from 4 to 0V in 2 ns, beginning at t = T_o. Because T_o = 2.3 ns, the voltage reaches zero at

$2.3\,ns + 2\,ns = 4.3\,ns.$

The 1.5V level occurs at

$$4.3\,ns - \frac{2\,ns}{4V} \times 1.5V = 3.55\,ns.$$

The rising edge begins at

$t = 3T_o = 6.9\,ns.$

The 1.5V level occurs at

$$6.9\,ns + \frac{2\,ns}{4V} \times 1.5 = 7.65\,ns.$$

The time difference (7.65 - 3.55 = 4.1 ns) is long enough for the FIFO to interpret the signal as a Low.

Next, consider the width of the positive pulse that begins at the load at t = $3T_o$. Because the rise time is preserved, the signal takes 1 ns to reach 2V, or 0.75 ns to reach 1.5V. The signal begins to fall at t = $5T_o$, reaching 1.5V at

$t = 5T_o + 0.25\,ns = 11.75\,ns.$

The difference (11.75 - 7.65) is 4.1 ns, which is wide enough for the FIFO to interpret as a second clock. To eliminate this pulse, the line must be terminated.

Strobe Shortening Considerations

In this example the width of the negative strobe is 22 to 24 ns. If a CY7C429-20 FIFO is used, the write (or read) strobe must not be shorter than 20 ns. Even if the FIFO does not recognize the 4.1-ns negative pulse, the shortening of the write strobe by $5T_o$ = 11.5 ns is sufficient to violate the minimum negative-pulse-width specification.

This strobe-shortening phenomenon might also occur on other active-Low control lines such as output enables and chip selects. Clock lines must also be analyzed for this problem; in general, these lines should be terminated.

The Rising Edge of the Write Strobe

Now consider an analysis of the write strobe's rising edge to assure that the reflections associated with this edge do not cause multiple clocks or false triggering of the FIFO. At t = 22 ns, the rising edge of the write strobe begins, which is the equivalent of closing the switch in *Figure20* in the 1 position. For this analysis, it is convenient to start the time scale over at zero, as appears in *Figures22a* and *22b*.

If the forcing function were a step function, the equations of *Figure 4h* would apply. The time constant in the equation is

$$T = \frac{RZ_o'C_e}{R + Z_o'} \qquad Eq.\ 33$$

Because

$R > Z_o', T = Z_o'C_e$

where Z_o' = 32.8Ω, and C_e = 45.4 pF.

This is the equivalent of saying that you can ignore the 1.25-MΩ device input resistance for transient circuit analysis. Substituting Z_o' and C_e into the preceding equation yields a time constant of T = 1.489 ns.

Writing the equation for the voltages for the circuit of *Figure20* yields

$$V_B(t) = iZ_o' + \frac{1}{C_e}\int_o^t i\,dt \qquad Eq.\ 34$$

Also,

$V_B(t) = K_tU(t) - K(t - Tl)\,U(t - Tl).$ *Eq. 35*

where K_t is the rising edge of the write strobe (K = 2V/ns) applied at t = 0 using a unit step function, U(t); and -K(t - T1) represents an equal but opposite waveform applied at t = T1 (after the rise time) using a unit step function, U(t - T1).

Equating the expressions and taking the LaPlace transforms of both sides yields

$$\frac{K}{s^2} - \frac{Ke^{-Tls}}{s^2} = Z_o'I(s) + \frac{I(s)}{C_es} = \left(Z_o' + \frac{1}{C_{es}}\right)I(s) \qquad Eq.\ 36$$

However,

$$V_B(t) = \frac{1}{C_e}\int_0^t i\,dt, \text{ or, } V_B(s) = \frac{I(s)}{C_e s}$$

Therefore,

$$\frac{K}{s^2} - \frac{Ke^{-Tls}}{s^2} = \left(Z_o' + \frac{1}{C_e s}\right) C_e s V_B(s). \qquad \text{Eq. 37}$$

Solving for $V_B(s)$ yields

$$V_B(s) = \frac{\frac{K}{s^2}\left(1 - \varepsilon^{-Tls}\right)}{C_e s\left(Z_o' + \frac{1}{C_e s}\right)} \qquad \text{Eq. 38}$$

which is equivalent to

$$\frac{\frac{K}{Z_o' C_e}\left(1 - e^{-Tls}\right)}{s^2\left(s + \frac{1}{Z_o' C_e}\right)} \qquad \text{Eq. 39}$$

Taking the inverse LaPlace transform yields

$$V_B(t) = \left[KZ_o' C_e\left(e^{-\frac{t}{Z_o' C_e}} - 1\right) + K_t\right] U(t)$$

$$- \left[KZ_o' C_e\left(e^{\left[\frac{-(t-Tl)}{Z_o' C_e}\right]} - 1\right) + K(t - T1)\right] U(t - T1) \qquad \text{Eq. 40}$$

The first term in Equation 40 applies from time zero up to and including T_1, and the second term applies after T_1:

$$V_B(t) = \frac{KZ_o' C_e}{T1}\left(e^{\left[\frac{-t}{Z_o' C_e}\right]} - 1\right) + \frac{K}{T1}(t) \qquad \text{Eq. 41}$$

for $t \leq T1$

$$V_B(t) = \frac{KZ_o' C_e}{T1}\left(1 - e^{\left[\frac{T1}{Z_o' C_e}\right]}\right) e^{\left[\frac{-t}{Z_o' C_e}\right]} + K1 \qquad \text{Eq. 42}$$

for $t > T1$

where K_1 is the final value, which is 4V.

Substituting the correct values for t = T_1 = 2 ns yields

$$V_B(t = T1) = \frac{2 \times 32.8 \times 45.4 \times 10^{-12}}{2 \times 10^{-9}}(e^{-1.489} - 1)$$

$$+ \frac{2V}{ns} \times 2\,ns$$

$$= -1.15 + 4 = 2.85V.$$

If the forcing function is a step function, the equation is

$$V_B(t) = 4V\left(1 - e^{\left[\frac{-t}{Z_o' C_e}\right]}\right) \qquad \text{Eq. 43}$$

at t = 2 ns, V_B = 3V, which is more than the 2.85V calculated using Equation 41.

At t = 22 ns + T_o, the voltage waveform begins to build up at the load and continues to build until the first reflection from the source occurs at t = $3T_o$.

Equation 42 is used to calculate the voltage at the load at t = $2T_o$, because 1 T_o is used for propagation delay time:

$$V_B(t = 2T_o) =$$

$$\frac{-2V \times 32.8 \times 45.4 \times 10^{-12}}{2 \times 10^{-9}}(1 - e^{-1.489})(e^{-2}) + 4$$

$$= -1.489\,(0.774)\,(0.1353) + 4$$

$$= -1.559 + 4 = 3.84V.$$

The voltage at the load remains at this value until the first reflection from the source reaches the load at t = $3T_o$.

Meanwhile, at t = T_o, the wave at the load reflects back to the source and arrives at t = $2T_o$. The wave subtracts from the 4V level at the source, as illustrated in *Figure6c*. The amplitude of the droop is given by

$$V_r = \frac{C' Z_o'}{2}\frac{V_o}{T_r} \qquad \text{Eq. 44}$$

for $R_S = Z_o$.

If R_S does not equal Z_o', Equation 44 must be modified. Instead of $V_o/2$, the voltage is

$$V_o\left(\frac{R_S}{R_S + Z_o'}\right)$$

so that Equation 44 becomes

$$V_r = \frac{C' Z_o' V_o}{T_r}\left(\frac{R_S}{R_S + Z_o'}\right) \qquad \text{Eq. 45}$$

where C' = 40 pF, Z_o' = 32.8Ω, R_S = 62Ω, T_R = 2 ns, and V_o = 4V. Substituting these values into Equation 45 yields

$V_r = 1.716V$.

Because 4V - 1.716 = 2.284, the voltage does not drop below the minimum TTL V_{IH} level of 2V, but the voltage does come close.

The reflection coefficient at the source is

$$\rho_s = \frac{R_S - Z_o'}{R_S + Z_o'}$$

where R_S = 62Ω, Z_o' = 32.8Ω, ρ_s = 0.308.

The amount of voltage reflected from the source back to the load is then

$V_{S1} = 1.716 \times 0.308 = 0.53V$.

The 40-pF capacitor reduces the rise time of the waveform at the load. The reflection at the source caused by the load capacitor is insufficient to reduce the 4V level to less than the TTL One level (2V).

The reflection coefficient at the source is small enough so that the energy reflected back to the load is insufficient to cause a problem.

References

1. Matick, Richard E. *Transmission Lines for Digital and Communications Networks*. McGraw Hill, 1969.

2. Blood, Jr., William R. *MECL System Design Handbook*. Motorola Inc., 1983.

CYPRESS SEMICONDUCTOR

Power Characteristics of Cypress Products

This application note presents and analyzes the power dissipation characteristics of Cypress products. The knowledge and tools presented here will help you manage power when using Cypress CMOS products.

Design Philosophy

The design philosophy for all Cypress products is to achieve superior performance at reasonable power dissipation levels. The CMOS technology, circuit design techniques, architecture, and topology are carefully combined to optimize the speed/power ratio.

Power Dissipation Sources

Power is dissipated both inside and outside ICs. The internal and external power have a quiescent (or DC) component and a frequency-dependent component. The relative magnitudes of each depend upon the circuit design objectives.

In circuits designed to minimize power dissipation at low to moderate performance, the frequency-dependent component is signifigantly greater than the DC component. In the high-performance circuits designed and manufactured by Cypress, the frequency-dependent power component is much lower than the DC component. This is because a large percentage of the internal power is dissipated in linear circuits such as sense amplifiers, bias generators, and voltage/current references, which are required for high performance.

Frequency-Dependent Power

CMOS circuits inherently dissipate significantly less power than either bipolar or NMOS circuits. The ideal CMOS circuit has no direct current path between V_{CC} and V_{SS}. In circuits using other technologies, such paths exist, and DC power is dissipated while the device is in a static state.

The principal component of power dissipation in a power-optimized CMOS circuit is the transient power required to charge and discharge the capacitances associated with the inputs, outputs, and internal nodes. This component is commonly called $C V^2$ power and is directly porportional to the operating frequency, f.

The charge, Q, stored in a capacitor, C, that is charged to a voltage, V, is given by the equation:

$$Q = C V \qquad \textit{Eq. 1}$$

Dividing both sides of Equation 1 by the time required to charge and discharge the capacitor (one period, or T) yields:

$$\frac{Q}{T} = \frac{C V}{T} \qquad \textit{Eq. 2}$$

By definition, current (I) is the charge per unit time and

$$f = \frac{1}{T}$$

Therefore,

$$I = C V f \qquad \textit{Eq. 3}$$

The power ($P = V I$) required to charge and discharge the capacitor is obtained by multiplying both sides of Equation 3 by V:

$$P = V I = C V^2 f \qquad \textit{Eq. 4}$$

It is standard practice to assume that the capacitor is charged to the supply voltage (V_{CC}), so that

$$P = V_{cc} I = C V_{cc}^2 f \qquad \textit{Eq. 5}$$

The total power consumption for CMOS systems depends upon the operating frequency, the number of inputs and outputs, the total load capacitance, the internal equivalent (device) capacitance, and the static (quiescent) or standby power consumption. In equation form:

$$P_d = [C_{INT} F_{INT} + C_{load} F_{load}] V_{cc}^2 + I_{cc} V_{cc} \qquad \textit{Eq. 6}$$

The first four quantities are frequency dependent, and the last is not. This same equation can be used to describe the power dissipation of every IC in the system. The total power dissipation is then the algebraic sum of the individual components.

The relative magnitudes of the various terms in the equation are device dependent. Note that Equation 6

Table 1. Types of Input Buffers

BUFFERTYPE	ICC (MAX. IN mA)
A	1.3
B	0.8
C	0.6

must be modified if all of the internal nodes or all of the outputs are not switching at the same frequency.

Transient Power

Cypress devices incorporate N-well CMOS inverters that can affect the devices' transient power consumption. In an ideal N-well CMOS inverter, the P-channel pull-up transistor and the N-channel pull-down transistor (which are in series with each other between V_{CC} and V_{SS}) are never on at the same time. Thus, there is no direct current path between V_{CC} and ground, and the quiescent power is very nearly zero.

In the real world, when the input signal makes the transition through the linear region (i.e., between logic levels) both the n-channel and p-channel transistors are partially turned On. This creates a low-impedance path between V_{CC} and V_{SS} whose resistance equals the sum of the n- and p-channel resistances.

DC or Static Power

In addition to conventional gates, Cypress devices contain sense amplifiers; input and output buffers; and bias and reference generators that all dissipate power. RAMs and FIFOs also have memory cells that dissipate standby power whether the IC is selected or not. PROM and PAL products have EPROM memory cells that do not dissipate as much standby power as a RAM cell.

Power-Down Options

Five Cypress static RAMs offer a power-down option that enables you to reduce the devices' power dissipation by approximately an order of magnitude when they are not accessed. The power-down technique disables or turns off the input buffers and sense amplifiers.

Power Dissipation Model

The rest of this application note presents power dissipation models for various Cypress CMOS products as well as information on each product's typical and worst-case power dissipation. The information is presented as functions of frequency, V_{CC}, and temperature.

A general-purpose power dissipation model for all Cypress ICs appears in *Figure1*.

To obtain power dissipation data on an IC, you must isolate the three components of power dissipation included in Equation 6 by controlling the IC's inputs. The standby current (I_{CC}) is measured with the inputs to the IC at 0.4V or less. Under this condition, the input buffers and unloaded output buffers draw only DC leakage currents. All other direct currents derive from the substrate bias generator, sense amplifiers, other internal voltage or current references, and NMOS memory circuits.

At V_{in} = 1.5V, the input buffers draw maximum I_{CC}. To find the total input buffer I_{CC} current, you measure the total current and subtract the quiescent current. You can then calculate the current per input buffer by dividing the total input-buffer current by the number of input buffers.

Input Buffers

Cypress products use three different types of input buffers. For purposes of illustration, they are referred to as types A, B, and C. *Table 1* lists the buffer types used in various products.

Figure2 shows schematics and input characteristics for the three types of buffers. A circle on a transistor's gate means that the transistor is a P-channel device.

As *Figure2* shows, the input buffers draw essentially zero I_{CC} when V_{in} is 0.4V or less. This is also true when V_{in} is 4V or more, except for type A. In other words, if the inputs are driven rail to rail, the B and C input buffers dissipate power only during the input signal transitions.

Core and Output Buffers

The standby power dissipation of an IC's core derives from the substrate bias generator, reference generators, sense amplifers, and polyload RAM cells or EPROM cells. This current is measured with V_{in} = 0V, so that the input buffers draw no current. Under these conditions, the output buffers draw only leakage current and dissipate essentially no power.

Programming either PROMs or PALs stores charge on the floating gate of an NMOS transistor, which increases the transistor's threshold voltage. This

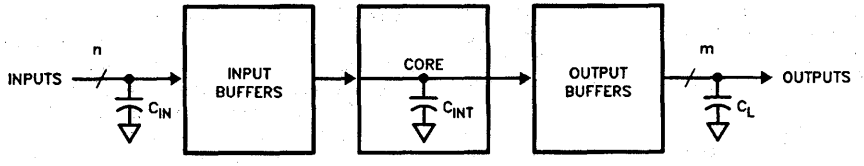


Figure 1. Power Dissipation Model

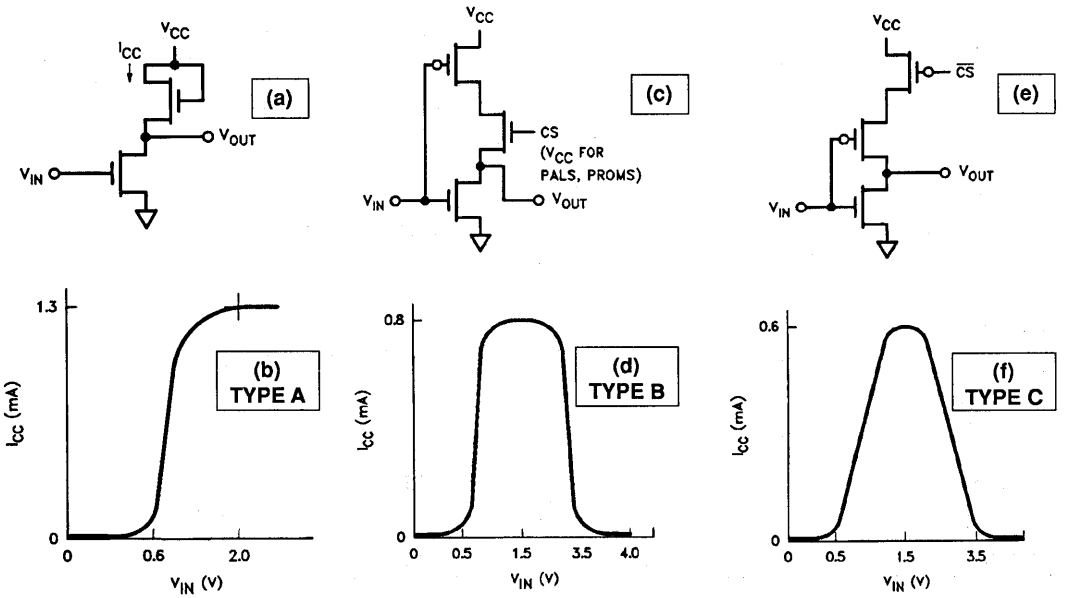


Figure 2. Three Buffer Types

higher threshold prevents the transistor from turning on during normal operation; unprogrammed transistors do turn on. Therefore, unprogrammed PALs and PROMs draw more current and dissipate more power than programmed devices.

The output buffers on Cypress products have n-channel pull-up devices that cause the output voltage level to reach

$V_{OH} = V_{CC} - V_T = 5V - 1V = 4V$

The capacitance of the output buffers, including stray capacitance, is typically 10 pF. If

$C_L = 10$ pF, $V_{OH} \approx 4V$

Again, using Equation 3,

$I_{CC}(f) = 40 \times 10^{-12} f$

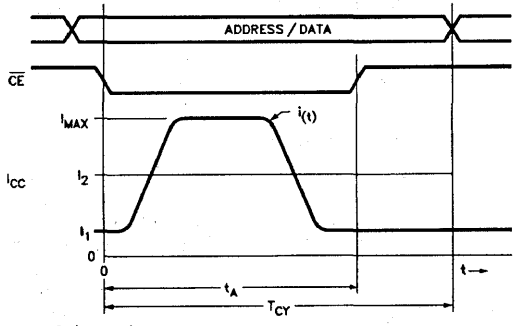
for the output buffers.

Current Measurement

Figure 3 illustrates the instantaneous current drawn by a Cypress RAM. The instantaneous power is calculated by multiplying this current times the constant supply voltage, V_{CC}. Most of the power is dissipated during the access time. This is also true for PROMs and PALs.

The current measurement unit in an automatic tester integrates the instantaneous current over the measurement cycle and arrives at an equivalent average current. In other words, the average current, I_2, during time T_{CY} equals the area between the instantaneous current, i(t), and the X axis during T_{CY}. Thus, because the "current pulse" is effectively spread over a longer time when the frequency is decreased, the average current is proportionately lower.

Note that the preceding calculations have not accounted for any DC loads. You must calculate these separately.



I_1 = Quiescent I_{CC}
I_2 = Average I_{CC}
i(t) = Instantaneous I_{CC}

Figure 3. RAM I_{CC}

Table 2. Static RAMs

Part No.	Buffer Type	No. Inputs	No. Outputs	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max) (mA)
CY7C122/123	A	16	4	24	50	90
CY7C128	B	14	8	27	59	120
CY7C147	B	15	1	34	28	90
CY7C148/149	B	12	1	32	45	90
CY7C150	B	18	4	20	44	90
CY7C161/162	B	22	4	300	13	70
CY7C164	B	20	4	300	13	70
CY7C166	B	21	4	300	13	70
CY7C167	C	17	1	75	25	70
CY7C168/169	C	18	4	75	50	70
CY7C170	B	18	4	50	33	90
CY7C171/172	B	18	4	100	27	70
CY7C185/186	B	25	8	330	13	100
CY7C187	B	19	1	150	7	100
CY7C189/190	B	10	4	21	32	90

Product Characteristic Tables

Tables 2 through *5* allow you to calculate the current requirements for Cypress products. C_{INT} is the equivalent device internal capacitance, $I_{CC(Q)}$ is the quiescent or DC current, and $I_{CC(MAX)}$ is the maximum I_{CC} (as specified on the data sheet) for the commercial operating temperature range. Conditions are V_{CC} = 5V and T_A = 25°C.

Note that for the 16L8, 16R8, 16R6, and 16R4 PALs, the number of inputs and outputs is user configurable. All the PALs use type B buffers.

SRAM Calculation Example

To illustrate how to use *Tables 2* through *5*, consider an example of estimating the typical I_{CC} for the CY7C169-35 RAM at room temperature (T_A = 25°C) and V_{CC}. Assume the duty cycle is 100 percent at the specified acces time. The procedure shown here calculates the typical and worst-case I_{CC} with all inputs and outputs changing and with output loading of 10 pF.

From the RAM product characteristic table:

Number of inputs = 18

Number of outputs = 4

C_{INT} = 75 pF

$I_{CC(Q)}$ = 50 mA

Because the input buffers on the CY7C169 are type C, the average current is 0.3 mA. If the input-signal-level transitions are 4V and the transition times are 2V/ns, the transition time is

$$T_t = \frac{4V}{2V/ns} = 2\ ns$$

The duty cycle is then

$2\ ns/35\ ns = 0.057$

Each input buffer thus draws

$0.3\ mA \times 0.057 = 0.0171\ mA$

If all inputs change, the total transient input buffer current is

$18 \times 0.0171 = 0.31\ mA$

To calculate the CVf input buffer current:

I = CVf

C_{IN} = 5 pF

I = 0.57 mA

V = 4V

f = 1/35 ns

TOTAL = 18 × 0.57 = 10.28 mA

To calculate the internal CVf current:

I = CVf

C_{INT} = 75 pF

I = 10.71 mA

V = 5V

f = 1/35 ns

To calculate the output CVf current:

I = CVf

C_{OUT} = 10 pF

I = 1.15 mA

V = 4V

Table 3. PROMs

Part No.	Buffer Type	No. Inputs [1]	No. Outputs	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max) (mA)
CY7C225	B	12	8	32	35	90
CY7C235	B	13	8	35	35	90
CY7C245	B	13	8	35	50	90
CY7C251	C	18	8	43	9.5	100
CY7C254	C	18	8	43	35	100
CY7C261/3/4	C	14	8	60	45	100
CY7C268	C	19	1/8	60	60	100
CY7C269	C	17	1/8	60	60	100
CY7C281/282	B	14	8	35	35	100
CY7C291/292	B	14	8	35	50	100

[1]/Bidirectional pins

f = 1/35 ns
TOTAL = 4 × 1.15 = 4.6 mA

The quiescent current is 50 mA. The total current at T_{CY} = 35 ns is:

Input Transient	0.31	mA
Input CVf	10.28	mA
Internal CVf	10.71	mA
Output CVf	4.6	mA
Quiescent	50	mA
Total I_{CC}	75.9	mA (all inputs/outputs changing)

Note that the worst-case transient current is 25.9 mA. If half the inputs and outputs change, the worst-case transient current decreases to 12.95 mA, which gives a total current of 63 mA (typical I_{CC}).

Note also that the input CVf current the the output CVf current have the same values for a bipolar device.

Worst-, Worst-, Worst-Case I_{CC}

Now consider a procedure for estimating I_{CC} for worst-case V_{CC} and low temperature, in addition to all inputs and outputs changing. Then you can compare the result with the I_{CC} specified on the data sheet.

I_{CC} is greater at high V_{CC}, which is 5.5V, or 1.1 times the nominal 5V V_{CC}. Because the increase in I_{CC} due to the lower temperature is 3 percent, the total increase is 13 percent. These factors apply to the internal CVf current (10.71 mA), the output CVf current (4.6 mA), and the quiescent current (50 mA), which together total 65.31 mA.

Table 4. PALs

Part No.	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max) (mA)
PALC16L8/R8/R6/R4	40	25	45
PLDC20G10	50	30	55
PALC22V10	50	40	80
PLDCY7C330	300	42	120

Total I_{CC} = Input Transient I_{CC}
+ Input CVfI_{CC}
+ [Internal CVf + Output CVf + I_{CC}(Q)] × 1.13
I_{CC} = 0.31 + 10.28 + [65.31] × 1.13 = 84.4 mA

This value is approximately 94 percent of the 90 mA specified on the data sheet. Note, however, that the data sheet I_{CC} maximum does not include the output CVf current.

I_{CC}-Versus-Frequency Characteristic

The I_{CC}-versus-frequency curves for all Cypress products have the same basic shape, which is illustrated by the PAL 16R8 curve in *Figure4*. The current remains essentially constant at the quiescent I_{CC} value until the frequency increases to the point where the capacitances begin to cause appreciable currents. The location of this point depends upon the input, internal, and output capacitances; the number of inputs and outputs; the rate at which the inputs and outputs change; and the voltage levels the inputs and outputs are switched be-

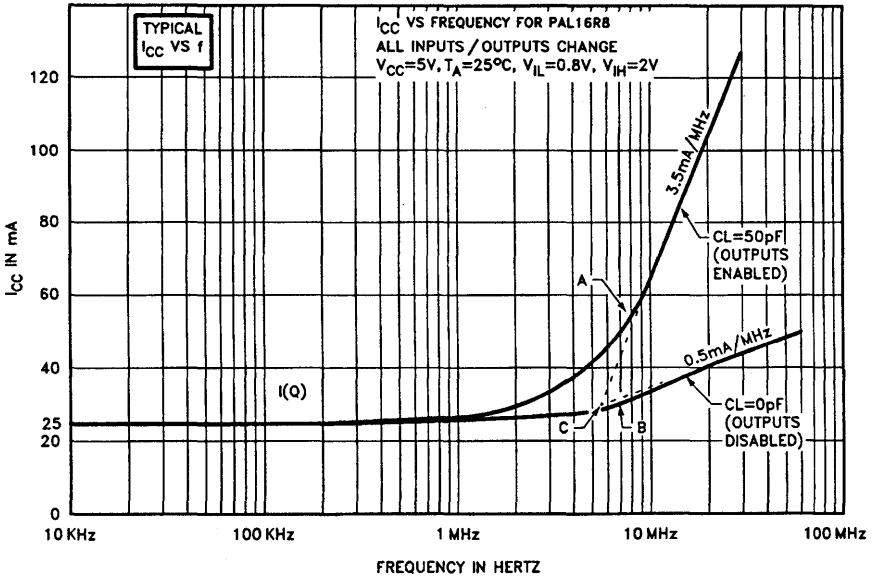


Figure 4. Typical I_{CC} vs f

Table 5. Logic Products

Part No.	Buffer Type	No. Inputs	No. Outputs [1]	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max) (mA)
CY7C401	B	6	6	53	30	75
CY7C402	B	7	7	53	30	75
CY7C403	B	7	6	53	30	75
CY7C404	B	8	7	53	30	75
CY7C408	B	11	12	100	42	135
CY7C409	B	11	13	100	42	135
CY7C428/9	C	14	12	190	18	80
CY7C510	C	24	19/16	60	30	100
CY7C516	C	28	16/16	60	30	100
CY7C517	C	28	16/16	60	30	100
CY3341	B	6	6	53	30	45
CY7C601	C	25	19/64	950	89	600
CY7C901	C	24	10/4	160	25	80
CY7C909	C	21	5	80	25	55
CY7C910	C	22	16	150	2.6	70
CY7C911	C	13	5	80	25	55
CY7C9101	C	36	22/4	70	30	60
CY7C9116	C	22	1/20	1000	35	150
CY7C9117	C	38	1/4	1000	35	150

[1]/Bidirectional pins

tween. For Cypress products, this point is in the 1- to 10-MHz range.

The PAL 16R8 devices that were tested to optain the data for the curve were exercised such that all inputs and outputs changed every cycle. Curve A shows the total I_{CC} for a 50-pF load on each of the eight outputs. Curve B shows the total I_{CC} when the outputs are disabled. The B curve results from the input and the internal capacitances. In most applications, the actual operation of the device falls somewhere between the A and B curves.

You can extrapolate the A and B curves backwards until they intersect the quiescent current, which occurs at point C in *Figure 4*. Point C is approximately 5.6 MHz. This gives you an easy-to-use formula for calculating I_{CC}. For frequencies less than 5.6 MHz:

$I_{CC} = I_{CC}(Q) = 25$ mA

For frequencies greater than 5.6 MHz:

$I_{CC} = I_{CC}(Q) + 3.5$ mA/MHz (all outputs changing)

or

$I_{CC} = I_{CC}(Q) + 0.5$ mA/MHz (no outputs changing)

CYPRESS SEMICONDUCTOR

Tips for High-Speed Logic Design

This application note provides tips and makes substantive suggestions for designing high-speed logic circuits that operate reliably. The tips and suggestions are organized under the headings:

Noise Considerations
Clock Distribution
Buses and Memories
Care and Feeding of PLDs
PCB Effects
Metastability and Crosstalk

As electronic system clock rates reach ever higher, logic designers who were engineering 10-MHz, 100-ns-cycle-time systems are finding themselves working with systems running at speeds upwards of 20 MHz, with 50-ns cycle times. These designers are discovering that adequate techniques for work at 10 MHz are no longer appropriate at 20 MHz and beyond. At 10 MHz, you can utilize sluggish and relatively well-behaved LS TTL logic with its leisurely set up and hold parameters; long propagation delays; forgiving output enable and disable times; and high-output current-drive capacity.

As an alternative, designers turned to faster bipolar logic families, but found that power dissipation rose proportionally. To save power and enhance reliability, today's designers are changing to CMOS components. Designers are happy to find that CMOS can deliver the speed they require at the low power levels they desire.

In the quiescent state, CMOS logic (AC/ACT/FCT) draws three to five orders of magnitude less power than bipolar logic (LS/ALS/AS). At 1 MHz, CMOS logic dissipates about 0.1 mW per gate, while LS TTL logic dissipates about 2.0 mW per gate. CMOS technology has truly rewritten the speed/power rules set forth in the bipolar era.

Plenty of challenges still face the high-speed logic designer, however. For example, high-performance logic families are sensitive to system noise and generate noise themselves. As a result of the effort to make these devices as fast as possible, they often have anemic output drive capacity. Clock distribution becomes much more of an issue at high frequencies because skew and slow rise times degrade operating margins. As bus cycles tighten, it becomes increasingly difficult to avoid bus clashes (multiple devices driving a bus). Very fast SRAMs and FIFOs require read and write pulse widths that are very difficult to synthesize using synchronous logic; hence the appearance of self-timed memory devices. PLDs have become ubiquitous in modern board-level designs, but high-speed designers must carefully consider PLDs' relatively long propagation delays and slow switching speeds.

You can no longer think of printed circuit boards as an ideal electrical interconnect. In the high-speed realm, you must account for the effects of distributed capacitance, inductance, and propagation delay on the PCB. To mitigate the effects of ringing, resistive termination of critical signals becomes a practical necessity above 20 MHz. In the days of old, it wasn't appropriate to factor loading into propagation delays. Today, conservative designers account for loading when calculating worst-case prop delays and worst-case signal skew. Heavy capacitive bypassing and low-inductance decoupling is essential to minimize switching noise above 20 MHz.

Metastability, a phenomenon not widely appreciated until recently, is a critical issue in high-frequency systems. It is essential to be able to resolve asynchronous events quickly and reliably in high-performance designs. Finally, crosstalk is a substantial concern with high-slew-rate and noise-sensitive CMOS logic.

Noise Considerations

High-speed CMOS logic tends to be noisier than LS TTL for two reasons: CMOS voltage swings are rail-torail, and small-geometry, dual-layer-metal CMOS technology makes possible faster edge rates (2V per ns and faster).

The classic ground-bounce noise situation arises when several outputs of a CMOS logic device switch from High to Low. The simultaneous switching causes a relatively large sink current from the load capacitance to flow to ground through the device package inductance. The potential developed momentarily across this

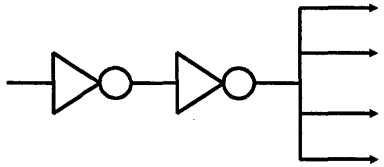


Figure 1. Maintaining Duty Cycle Symmetry

inductance equals the product of the package inductance and the sink current's rate of change. This ground-bounce voltage spikes the Low state held on the quiescent outputs. The spike can often exceed the input Low-level maximum voltage (0.8V), causing the downstream logic device to switch erroneously. Both the chip ground reference and the chip V_{cc} reference are spiked, but because more energy is switched through the ground-lead inductance, it is much more common to see a problem in a quiescent Low-state output.

Here are some procedures that minimize ground and V_{cc} bounce noise:

1. Pursue any steps that reduce the parasitic inductance between the package and ground and V_{cc}. These steps includes using a PCB with ground and V_{cc} planes or, at the very least, power distribution elements. Avoid use of sockets, but do use low-inductance decoupling and bypass capacitors. On critical parts, use a standard ceramic decoupling capacitor (0.01 to 0.1 μF) along with a high-frequency filtering capacitor (approximately 470 pF). The Rogers Corp. Micro/Q 1000 Series high-frequency, low-inductance caps are optimal for this purpose. Surface-mount packages have lower package inductance than DIP packages. So-called rotated-die devices with center V_{cc} and ground pins also have lower inductance.

2. Whenever possible, design synchronous circuits. The ground bounce produced by an octal register, for instance, is triggered by the clock. If the register feeds another registered device, then the noisy output has until a set-up time before the next clock to settle. When you must drive an asynchronous signal with an octal driver, use an output pin close to the package ground pin. The output pin next to the V_{cc} pin can have as much as 50% more ground-bounce noise than the output pin next to the ground pin.

3. Use various techniques to slow switching or transition edge rates and, therefore, the sink and source currents' rate of change. This can be accomplished with series damping resistors or by increasing the inductance or capacitance between the driving device's output pin and the receiving device's input pin. PCB traces exhibit parasitic ground-path capacitance and inductance that depend on trace length and topology; these factors are thus difficult to predict. The most common technique is to use series damping resistors in the 25 to 35Ω range; 33Ω is a standard value. Series resistors also limit signal overshoot and undershoot.

4. Try to avoid running control signals through a device that drives data and address lines. When using a 10-output PLD such as a 22V10 in an 8-bit bus-oriented application, for instance, you might be tempted to use the extra two outputs for control signals. If the eight data lines switch simultaneously, however, the control lines will probably be disturbed. Using devices that feature input hysteresis adds to the noise margin. Input hysteresis can typically provide 200 mV of additional noise immunity.

Note that mixing logic families can compromise noise immunity margins. For comparison purposes, the margin for a specific logic family is the magnitude difference between the family's guaranteed input threshold and the guaranteed output voltage for the High and Low states:

$$\text{Noise immunity} = \frac{V_{il} - V_{ol}}{V_{ih} - V_{oh}}$$

When possible, use a logic family that can drive 50Ω (commercial) transmission lines directly. This specification is characteristic of devices that can switch sufficient current to guarantee so-called incident-wave switching. Switching that occurs on the incident wave is faster than having to wait for the reflected wave.

In addition to causing false triggering of downstream sequential logic and glitches in downstream combinatorial logic, ground-bounce noise can also cause registers in the bounced device to "forget" their stored state. This is due to the momentary disturbance in the chip's ground and V_{cc} reference. The switching of multiple outputs can also skew the device's propagation delay by approximately 200 ps per switched output. With an octal or 10-bit device, this 1 to 2 ns additional delay should be included in worst-case timing analyses.

Clock Distribution

Adequate clock distribution is essential for 20-MHz and faster systems because skew can eat up precious nanoseconds and because high-speed logic devices are sensitive to clock waveform distortion and slow rise times.

All physical devices exhibit an edge-dependent propagation delay asymmetry; the Low-to-High edge propagates more quickly than the High-to-Low edge, or vice versa. For example, the clock-to-Q propagation delay for a Signetics 74F74 ranges from 3.8 to 6.8 ns Low to High, and 4.4 to 8.0 ns High to Low. The data sheet for the Texas Instruments 74AS1000 NAND driver specifies a 1-to-4-ns range for both Low-to-High and High-to-Low edges, but any specific physical device shows some asymmetry.

It is possible to maintain duty-cycle symmetry in a buffered-clock distribution network by cascading two inverting drivers. The two drivers must both be in the same package, as shown in *Figure 1*1. Because the two

drivers are in the same package, their prop delay characteristics track, and the High-to-Low and Low-to-High differential delays tend to cancel.

Limit the fanout from a clock buffer to eight to 15 devices. Fanout calculations must account for both AC and DC loading. The AC characteristics for logic components are specified at 50 pF of load capacitance and occasionally at 300 pF of load capacitance. Propagation delays and output-enable times increase by approximately 1 ns per each 50 pF of additional load capacitance. The input capacitance of bipolar logic families is higher (approximately 10 pF) than that of CMOS (approximately 5 pF). If the sum of the capacitance being driven exceeds 50 pF, derate the driver's AC characteristics appropriately.

Input current is the important DC electrical characteristic for loading purposes. The driving device must be able to sink the sum of the Low-level input currents to which it is connected (I_{ol} at V_{ol}). The driving device must also be able to source the sum of the High-level input currents to which it is connected (I_{oh} at V_{oh}).

The Low-level input current for bipolar logic families ranges from -400 to -100 µA, while the Low-level input current for modern CMOS logic families ranges from -5 to -1 µA. The High-level input current for bipolar logic families ranges from 50 to 20 µA, while the High-level input current for modern CMOS logic families ranges from 5 to 1 µA.

Because the I_{ol} at V_{ol} for bus drivers is often as high as 48 mA, and the I_{oh} at V_{oh} is often as high as -24 mA, input current loading is seldom an issue, except when driving a parallel (resistor) terminated load. For example, a 220Ω pull-up resistor requires about 22 mA worst case (V_{ol} = 0V, V_{cc} = 5V), and a 330Ω pull-down resistor requires about 15 mA worst case (V_{oh} = 5V, Gnd = 0V). Consider using an AC termination scheme if this additional current cannot be tolerated.

If a single buffer cannot safely supply a sufficient clock fanout, use parallel drivers (*Figure 22*. When distributing a clock signal, attempt to load each of the parallel lines equally. Unequal loading increases the skew between lines.

Buses and Memories

When you design buses in high-performance systems, it is important to consider the effects of AC and DC loading. The input and output capacitance of CMOS SRAMs, PROMs, and DRAMs ranges from 5 to 7 pF. This capacitance can become a concern with large memory arrays.

Be especially careful when using SRAM modules, which might have high input and output capacitances due to the multiple devices connected to each signal line. Because the signals that drive large memory arrays (such as the address, RAS, CAS, and data lines) tend to have long PCB traces, it is common practice to series-terminate these lines to minimize ringing, undershoot, and overshoot.

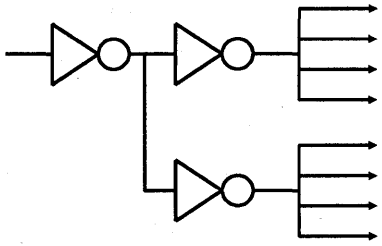


Figure 2. Parallel Clock Drivers

The input load or leakage currents for CMOS SRAMs, PROMs, and DRAMs is approximately 10 µA, sink and source. When you use high-output-current bus drivers (24 mA I_{ol} or greater), DC loading is rarely an issue.

As system cycle times shorten, it becomes more difficult to avoid bus clash situations. Bus clash or bus contention occurs on a shared bus when one three-state device finishes its output-enable time before a second device finishes its output-disable time. For a short period of time, both devices drive the bus. Because the output stages of memories and logic components can typically withstand at least 20 mA of current, the excess current does not shorten the devices' useful lives. Bus clash does cause large positive and negative current changes in the device V_{cc} and ground paths, however. The demand for current induces V_{cc} and ground bounce noise just like the simultaneous switching situation previously discussed. Thus, avoid more than 5 ns of overlap in the worst-case output enable and output disable times.

You can use CMOS components' low input current to advantage on buses when hold time is deficient. For example, consider a CMOS memory connected to a CMOS octal register. The memory is read, the /OE (or the /CE) deasserted, and the data clocked into the register. Ordinarily, the data should be clocked into the register before /OE is deasserted because the memory's worst-case output-disable time could be very short. When the memory is read in this case, however, the distributed capacitance presented by the register inputs, the PCB trace, and the memory's own outputs is charged. Because the memory's output leakage current and the register's input current are very low (5 to 10 µA), this distributed capacitance remains charged for some time. In effect, the data is held long enough to make up for the deficient timing.

High-speed SRAMs and FIFOs have timing requirements that are often difficult to meet using synchronous circuits. In such situations, there are asynchronous alternatives to consider. You can use the delay lines supplied by various manufacturers by combinatorially gating the output taps to synthesize the required signal. Delay lines are typically calibrated by

comparing the input's rising edge to the various delayed outputs' rising edges; the delay times for the falling edges are less accurate. If a decoded signal uses falling edges, make sure that the design can tolerate a few nanoseconds of inaccuracy.

The Engineered Components Company makes a family of pulse-generator modules (PGMs), which issue a precise pulse when presented with a positive-going edge. The company offers standard PGMs, fast-recovery PGMs that have a higher maximum repetition rate, and delayed PGMs, which wait for a specified period before issuing the pulse. Both delay lines and PGMs have propagation delays that range from 5 to 10 ns.

Care and Feeding of PLDs

Programmable Logic Devices (PLDs) are exceedingly useful for designing high-performance systems, but their characteristics and shortcomings must be well understood. The set-up time for most registered PLDs is usually just less than the propagation delay. This is because the signal to be latched must propagate through the AND array as well as the OR/XOR gate before reaching the flip-flop, while the clock is connected directly from the pin to the flip-flop. Accordingly, the hold time for this type of PLD is 0 ns minimum worst case and several nanoseconds negative, typically. This negative hold time implies that the PLD samples the state of the inputs as they existed several nanoseconds before the clock's rising edge. You can take advantage of this phenomenon when the device feeding the PLD is hold-time deficient with respect to the PLD clock.

PLD outputs usually do not have the drive capacity of standard logic. When you use a PLD to generate a critical signal, such as a FIFO-read or shift-out pulse, buffer the signal with a fast, hard-driving gate. Bear in mind, too, that identical equations implemented in the same PLD can exhibit different propagation delays due to different on-chip path lengths. PLD propagation delays are especially dependent on capacitive loading.

PCB Effects

The most conservative way to handle PCB signal distortion effects is to consider every substrate interconnect as a transmission line. In practice, this approach only works when the unloaded signal transition time approaches the round-trip substrate propagation delay.

For ordinary PCB materials (G-10 fiberglass epoxy), the round-trip propagation delay is approximately 0.3 ns per inch. Therefore, for 3-ns transition times, you should consider any PCB trace longer than 10 inches as a transmission line.

A transmission line presents a characteristic impedance and has distributed inductance and capacitance. You can minimize ringing on a transmission line by closely matching the output impedance of the driving device to the line's characteristic impedance. According to the microstrip model, for a 10-mil-wide, 1-oz. copper line 1.5 mils thick over a ground plane separated by a dielectric of G-10 fiberglass epoxy 62.5 mils thick, the theoretical unloaded characteristic impedance is approximately 130Ω. In reality, PCB trace characteristic impedances can range from 50 to 200Ω. Capacitive loading reduces the characteristic impedance, increases the delay, and slows the rise time on a transmission line.

Table 1. Pull-Up and Pull-Down Values

RESISTOR VALUES	THEVENIN EQUIVALENT
220Ω PULL UP 330Ω PULL DOWN	132Ω
330Ω PULL UP 470Ω PULL DOWN	194Ω

The conventional method for reducing reflections on transmission lines is with some form of termination, the most common being the so-called Thevenin type. This termination consists of a pull-up resistor to V_{cc} and a pull-down resistor to ground. The goal is to match the two resistors' Thevenin equivalent to the trace's characteristic impedance.

Table 1 lists common values for the pull-up and pull-down resistors. Both of the termination pairs shown in the table pull the line to a logic High of approximately 3V when the driver is disabled. Place the termination resistors as close as possible to the receiver. Keep in mind that many CMOS logic components have input and output clamp diodes to help damp overshoot and undershoot.

Metastability

The output of a latch or flip-flop can go into an undefined or metastable state (neither High or Low) when the set-up time or hold time for the device is violated. The metastable condition typically occurs when an asynchronous signal is being synchronized. It occurs in all process technologies and is impossible to completely eliminate.

The two important metastability parameters to consider in design work are the mean time between failures (MTBF) at maximum operating frequency and the average or typical resolution or settling time, T_{sw}. The latter is the time the device takes to resolve from a metastable state to a stable state. These parameters and/or the equations for deriving them should be available from a device's manufacturer.

Metastability performance is proportional to a technology's V_{ih}-to-V_{il} slew time. High-speed CMOS registers such as those found in Cypress PLDs have very fast slew times and typical settling times that range from 100 to 600 ps, depending on the device type.

By double-latching asynchronous inputs, you can dramatically increase a system's MTBF and reduce the probability of a metastable event causing system mal-

functions. When determining the length of time to delay before clocking the second register, multiply the published typical settling time by two or three to create an extra margin of protection.

Crosstalk

Crosstalk is the undesirable coupling of a transition on an active line (talker) onto an inactive line (listener). The crosstalk amplitude is proportional to the talker edge rates, the physical proximity between signal lines, and the distance over which the two lines are parallel or adjacent.

Crosstalk results from two important physical causes: mutual impedance and velocity differences. Mutual impedance is due to the mutual inductance and capacitance between adjacent signal lines and is a transformer-like effect. Velocity differences arise when a signal propagates along a conductor that is in contact with two materials of differing dielectric constants, such as fiberglass epoxy and air in PCBs. The wave propagating at the copper-to-epoxy interface travels slower than the wave propagating at the copper-to-air interface. A pulse develops whose duration is twice the difference in the arrival times of the two waves; thus, the magnitude of the disturbance increases when the length of the parallel or adjacent traces increases.

Due to CMOS's fast edge rates, crosstalk is a legitimate concern. You can take the following steps to reduce forward and reverse crosstalk:

1. Maximize the distance between traces, and minimize the distance over which traces are parallel or adjacent. When possible, make the signals on adjacent PCB layers perpendicular. Use the power and ground layers as shields between the signal layers. On two-layer PCBs, run ground lines between adjacent, parallel signal lines.

2. Make every other conductor a ground line when using flat ribbon cable. Protect critical signals such as clock lines with a dedicated ground strip on PCBs or with a ground twisted pair on backplanes.

3. Use Thevenin termination of a line to its characteristic impedance to reduce crosstalk amplitude by 50 percent.

CYPRESS SEMICONDUCTOR

Protection, Decoupling, and Filtering of Cypress CMOS Circuits

This application note explains how to protect your ICs with a low-cost zener diode and why it is good insurance against inadvertent voltage transients. Also explained is the reason why decoupling and high-frequency-filtering capacitors are required. A method is provided for determining the capacitors' values.

Zener Diode Protection

Linear power supplies can cause large voltage transients. When caused by the collapse of a magnetic field, the transient is negative. When the supply is turned on, the resulting transient is positive.

Some commercially available laboratory bench supplies behave the same way. When they turn on, they can over-shoot several volts. When they turn off, lead inductance can cause a negative transient voltage at the V_{CC} pin. If sufficient energy is available, internal gate oxides can break down, either destroying or weakening the IC such that it might fail later.

You can avoid this problem by adding a 20¢ zener diode (also called a voltage-regulator diode) between V_{CC} and ground. Connect the diode's cathode to V_{CC} and the anode to ground (*Figure 1*). A 400-mW, 6.2V 1N525 or equivalent is recommended. You can also use the 1N753, a 500-mW, 6.2V zener diode.

If a voltage greater than the zener voltage (6.2V) occurs on V_{CC}, the diode breaks down, clamping the voltage to 6.2V and shunting the current to ground (*Figure 2*). The diode can be destroyed if the current multiplied by the zener voltage exceeds the diode's power rating. Because zener diodes *always* fail shorted, they cause the power supply to "crowbar" and thus protect the ICs.

A negative voltage on the V_{CC} line puts a forward bias on the diode. This turns on the diode, which clamps the voltage to approximately -0.8V. If the negative voltage times the current exceeds the diode's power rating, the diode fails shorted, as in the reversed-bias case, and protects the ICs.

High-Frequency Filtering

In addition to the protection offered by zener diodes, decoupling and high-frequency-filtering capacitors are required on high-performance CMOS circuits. To use these capacitors effectively, you must understand why they are required.

To realize the fast rise and fall times that Cypress CMOS integrated circuits are capable of achieving, the power-distribution system must be able to supply the instantaneous current required when the device outputs switch from Low to High. The energy converted to current is stored as charge on the local decoupling capacitors . They decouple or isolate the circuit from the power-distribution system. It is standard practice to use one decoupling capacitor for each IC that drives a transmission line and one capacitor for every three devices that do not.

The PCB trace inductance plus the IC lead inductance can "current-starve" the output circuits, causing

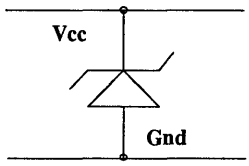


Figure 1. Zener Diode Connection

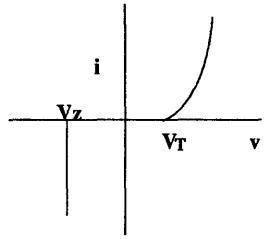


Figure 2. Zener Diode Characteristic

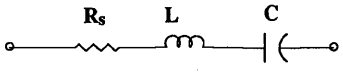


Figure 3. Simplified Capacitor Equivalent Circuit

rise-time degradation. Remember that the current through an inductor cannot change instantaneously. Therefore, you must minimize any series inductance, including the lead inductance of the decoupling capacitor s.

Decoupling-Capacitor Calculations

To determine the value of the decoupling capacitor, you must estimate the instantaneous current required when all the outputs of an IC switch from Low to High, assuming a reasonable droop of the voltage on the capacitor. The charge stored on the local decoupling capacitor is

$Q = CV$

Differentiating yields

$$i(t) = \frac{dQ}{dt} = C\frac{dV}{dt} \qquad \textit{Eq. 1}$$

The characteristic impedance of a typical transmission line is 50Ω. Lines with a heavy capacitive load have a lower characteristic impedances.

Next, assume that the IC is a nine-output FIFO, such as the CY7C429. The outputs reach

$V_{cc} - V_t = 5V - 1V = 4V$

Each output thus requires 4V/50Ω = 80 mA. Because the FIFO has nine outputs, it requires a total of 720 mA during the rise times of the outputs.

Solving *Equation 1* for C yields

$$C = i\frac{dt}{dv} \qquad \textit{Eq. 2}$$

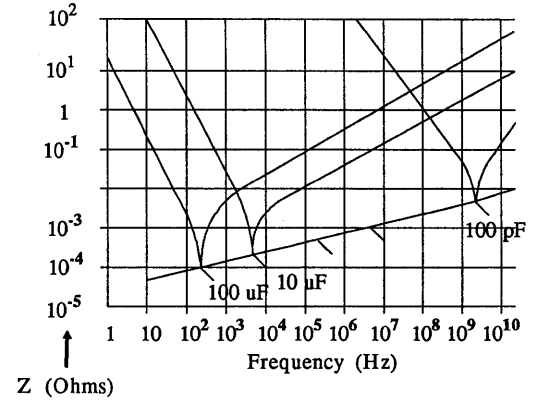


Figure 4. Capacitor Impedance Versus Frequency

The last step is to assume a reasonable, tolerable droop in the capacitor voltage. Assume dV = 100 mV. Additionally, the signal rise and fall times are 2 ns. Substituting these values in Equation 2 yields

$$C = \frac{720\times 10^{-3}\times 2\times 10^{-9}}{100\times 10^{-3}}$$

$$= 14.4\times 10^{-9}$$

$$= 0.0144\,\mu F$$

It is standard practice to use 0.01 to 0.1-µF decoupling capacitors. A 0.1-µF capacitor can supply 5A under the conditions assumed in the preceding calculations. Another way to look at the situation is that a 0.1-µF capacitor supplies 720 mA of instantaneous current in 2 ns with only 14.4 mV of voltage droop across the capacitor.

Decoupling capacitors for high-speed Cypress CMOS circuits should be of the high-K ceramic type with a low effective series resistance (ESR). Capacitors using 5 ZU dielectric are a good choice.

High-Frequency Filter Capacitors

The 0.1 to 0.01-µF decoupling capacitors usually do not provide high-frequency decoupling or filtering. These capacitors do not behave like capacitors at high frequencies because their series resonance frequency is not high enough. This is primarily because of lead inductance in their construction, which is a result of the capacitor's relatively large value.

For high-frequency filter analysis, you can use the simplified equivalent circuit of a capacitor shown in *Figure 3*. R_S is the effective series resistance (ESR), L is the effective series inductance (ESL), and C is the capacitance.

The impedance of the simplified equivalent circuit is:

$$Z_c = R_s + j\omega L + \frac{1}{j\omega C} \qquad \textit{Eq. 3}$$

$$Z_c = R_s + j\left[\omega L - \frac{1}{\omega C}\right] \qquad \textit{Eq. 4}$$

The magnitude of the impedance is

$$Z_c = \sqrt{R_s^2 + \left[\omega L - \frac{1}{\omega C}\right]^2} \qquad \textit{Eq. 5}$$

At the series resonant frequency:

$$\omega L = \frac{1}{\omega C}$$

or,

$$\omega = \frac{1}{\sqrt{LC}}$$

At the resonant frequency, $Z_c = R_S$, which is the minimum impedance.

Figure 4 shows how the impedance varies with frequency. The series resistance usually increases as the

capacitance decreases. Also as the capacitance decreases, the inductance typically decreases, which means that the resonant frequency increases. This is usually due to the capacitor's physical construction. Note that a surface-mounted capacitor's lead inductance is at least an order of magnitude less than that of an axial-lead capacitor.

The next step in high-frequency filter analysis is to determine a typical system's expected high-frequency components. Begin by assuming that the circuit is driven by a series of digital pulses with finite rise and fall times, then perform a Fourier transform on the series to determine their frequency components.

Fourier Transform of a Periodic Pulse

Figure 5 illustrates a periodic pulse of amplitude A, period T, rise and fall times of t_r, and pulse width of T_p, as measured between the 50-percent-amplitude points.

The approximate frequency-domain transform appears in *Figure 6*. The amplitude of the frequency-domain voltage is a function of the signal's amplitude and duty cycle in the time domain. The fundamental frequency, F_0, is related to the pulse train's period. The first harmonic, F_1, is of equal energy and is a function of the pulse width. The second harmonic, F_2, contains half the energy of F_0 and is a function of the pulse rise time.

The rise and fall times of Cypress's CMOS and BiCMOS circuits are 2 ns, by design. If a Cypress PLD is driving the write- or read-strobe inputs of a CY7C429-20 FIFO at the maximum frequency of 33.3 MHz (T = 30 ns) with a 10-ns/30-ns-duty-cycle signal (T_p = 10 ns), the following signal frequencies are generated:

$$F_0 = \frac{1}{\pi T} = \frac{1}{3.1416 \times 30 \times 10^{-9}} = 10.61\ MHz$$

$$F_1 = \frac{1}{\pi T_p} = \frac{1}{3.1416 \times 10 \times 10^{-9}} = 31.83\ MHz$$

$$F_2 = \frac{1}{\pi t_r} = \frac{1}{3.1416 \times 2 \times 10^{-9}} = 159.15\ MHz$$

Within the IC, signal rise and fall times can be as fast as 300 ps (picoseconds), which means that F2 = 1.061 GHz (1,061 MHz). In some ICs short timing pulses are generated internally, but they are usually longer than the 300-ps rise time, so the preceding F2 is the highest harmonic present.

Because the IC's data outputs can normally change no faster than those of the inputs, the outputs do not generate additional higher-frequency harmonics.

Parallel the Filter Capacitors

You cannot find a capacitor whose three series resonant frequencies correspond to F_0, F_1, and F_2. Instead, select three separate capacitors with the appropriate resonant frequencies and connect them in parallel between V_{cc} and ground, as close to the IC as possible. The capacitors act as a bandpass filter, shunting the unwanted, high-frequency signals to ground. The sum of the capacitors' values should be greater than or equal to the capacitance value given by Equation 2. The total high-frequency filtering capacitance is usually between 100 and 500 pF.

Low-Frequency Filter Capacitors

A solid tantalum capacitor of 10 µF is recommended for every 50 to 100 ICs to reduce power-supply ripple. Place this capacitor as close as physically possible to where the V_{cc} and ground enter the PCB or module.

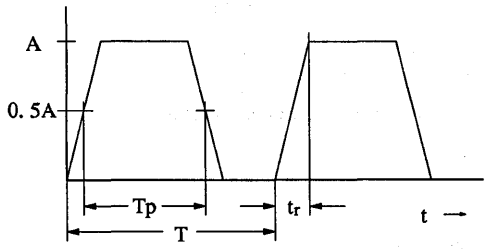


Figure 5. Periodic Pulse Waveform

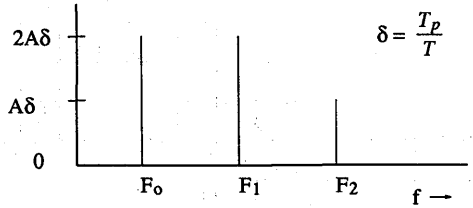


Figure 6. Fourier Transform of Periodic Pulse

Section Contents

CYPRESS SEMICONDUCTOR

Choosing Packages in High-Density Module Designs

This application note describes the various packages in which high-density memory modules are available and reviews some of the application areas where specific packages find use. Module outline drawings accompany the text.

You can use high-density memory modules in place of multiple monolithic ICs to minimize space, achieve better performance, and obtain single-device solutions. These modules are now available in a variety of package styles, each of which satisfies different needs in high-performance systems. *Table 1* summarizes the characteristics of the different package types.

There are two general module types. The first type uses plastic-encapsulated ICs mounted on an epoxy-fiberglass substrate. The monolithic ICs on the modules can be mounted in SOIC, VSOP, or SOJ packages, which are small-outline parts with either gull-wing or J-bend leads. The second module type offers hermetically sealed LCC (leadless chip carrier) ICs mounted on ceramic substrates.

Modules built on epoxy-fiberglass substrates offer economic advantages over modules with ceramic substrates. In general, however, ceramic substrates can accommodate more components than epoxy-fiberglass substrates. Further, when assembled using military-grade components, ceramic modules can be used in military applications. For all applications, the ceramic-substrate devices have better thermal characteristics than non-ceramic types.

SIPs

The single in-line package, or SIP, is a vertically mounted module with a single row of pins along one edge for through-hole mounting. The pins are on a 100-mil pitch. Note in *Figure 1* that the footprint of this plastic package is only 0.66 square inches.

SIPs are typically used in low-pin-count applications and are often used where high component density is required. These modules' vertical orientation and accommodation of components on both sides can increase component density by a factor of four or more over designs that use monolithics. In addition to meeting space constraints, this higher density can also improve memory system performance by reducing path lengths from chip to chip.

Another chief source of appeal for the SIP module is fast, easy access to state-of-the-art package technology. That is, a design's main circuit board can be implemented in conventional, high-yield, through-hole technology, while the system overall achieves superior component

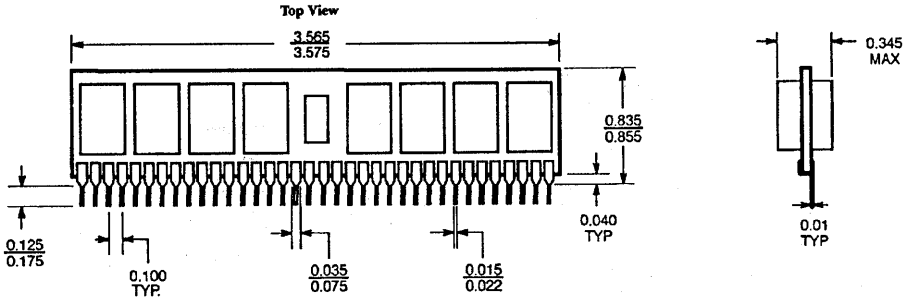


Figure 1. SIP

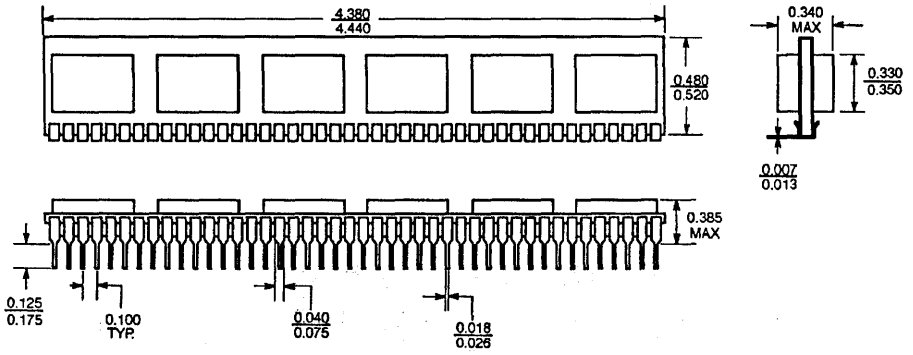


Figure 2. Flat SIP

density and high performance by employing fully-tested modules whose fine-pitch, surface-mount components are mounted on a multilayer, tight-tolerance substrate.

Flat SIPs

Flat SIPs are virtually identical to SIPs, except that their single rows of pins have a 90° bend (*Figure 2*). Therefore, flat SIPs lie close and parallel to the board on which they are mounted. Flat SIPs' advantage is their low profile; they are typically used where component height above the main board is constrained. Flat SIPs range in height from 0.300 to 0.38 inch.

ZIPs

ZIP modules are similar to SIPs. However, the ZIP module has pins on 100-mil centers along both sides of

Table 1. Module Package Characteristics

Package Type	Typical Pin Count		Typical Height (in.)		Mil	Advantages	Disadvantages	Board Space (sq. in.)	
	Min	Max	Min	Max				FR4	Cer
SIP	24	50	0.5	0.9	N	Vertical orientation; FR4 or ceramic	Limited pin count	1.2	0.9
FSIP	24	50	0.2	0.4	N	Very low profile; mechanical stability; FR4 or ceramic	Lower density due to horizontal orientation	2.7	2.4
ZIP	24	100	0.5	0.9	N	Vertical orientation; JEDEC standard pinouts; pinout compatible with SIMM		1.2	N/A
SIMM	24	100	0.5	0.9	N	Vertical orientation; socket mounting; pinout compatible with ZIP		1.2	N/A
VDIP	36	104	0.5	0.95	Y	Vertical orientation		1.2	0.9
DIP	24	60	0.17	0.37	Y	Low profile; excellent mechanical ruggedness	Horizontal	2.9	2.9
QUIP	48	200			Y	Low profile; excellent mechanical ruggedness; increased number of pins	Horizontal	2.9	2.9
QFP	68	144			Y	Surface mount; low profile; excellent mechanical ruggedness; large number of pins in small area	Surface mount technology required; horizontal; components on one side only	3.1	3.1
PGA	68	144			Y	Large number of pins in through-hole technology; low profile; excellent mechanical ruggedness	Multilayer boards; horizontal; components on one side only	2.9	2.9

Notes: Mil entries indicate whether a hermetic, military version is available
Board space is the mother board area that the package occupies when the module carries eight to 28 components

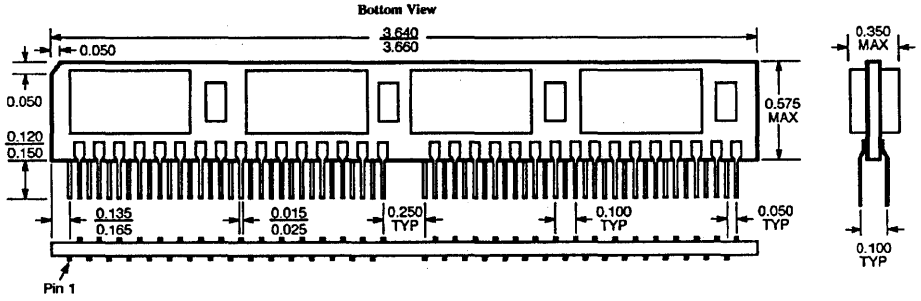


Figure 3. ZIP

the substrate (*Figure 3*). Pins on alternate sides are staggered by 50 mils. The dual row of staggered pins allows a higher connection density than the SIP, while maintaining 100-mil spacing between adjacent pins. The staggering provides additional separation for the lead vias and supports between-lead traces. At the same time, pin count is doubled over that of SIPs.

Many ZIP modules have a vertical dimension of 0.500 inch maximum. This low profile makes them candidates for VME systems, where there is a maximum allowable component height.

SIMMs

Single in-line memory modules, or SIMMs, are also similar to SIPs, except that SIMMs have no pins for through-hole mounting (*Figure 4*). Instead, the module's bottom edge effectively acts as an edge connector, which is part of the substrate material.

Contacts directly opposite each other are connected together. Some SIMMs have contacts on 100-mil centers; others have 50-mil centers.

The typical application for SIMM modules requires socket-mounted components, either for repair or for upgrades in the field. Some SIMM sockets hold the SIMM at an angle, which reduces the height of the module on the board.

Some module devices are available in both ZIP and SIMM packages with the same form factor. The pin out is such that the footprint of some SIMM sockets matches the footprint of corresponding ZIP modules. This allows system prototypes to use socketed SIMMs, and production systems to use through-hole-soldered ZIPs, with no change in the motherboard.

Some SIMMs and matching ZIPs have presence-detect pins, whose unique combination of no-connects or grounds can be used by external logic to identify the module's memory capacity. Thus, the system can determine the amount of memory present without user input.

DIPs

DIP modules have identical footprints and similar form factors to standard IC DIPs. The modules are typically taller than the DIP packages used for monolithics. Components are mounted on both the top and bottom of the substrate.

Generally, these modules are used in anticipation of monolithic devices that will someday fit the same footprint. DIP modules allow engineers to design-in monolithic devices that do not yet exist by employing the modules to meet immediate production needs. Practically, even after monolithic devices become available, the modules generally continue to find utility while initial

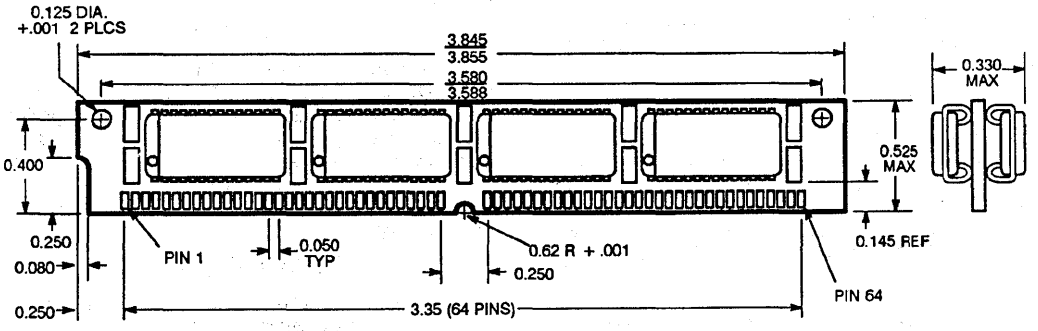


Figure 4. SIMM

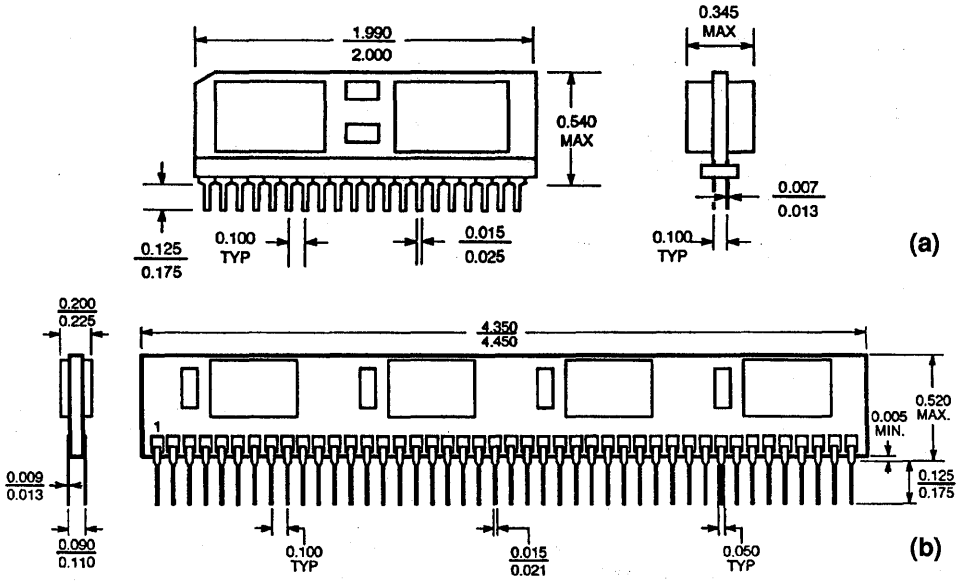


Figure 5. VDIP (a) and HVDIP (b)

production ramp-up of the monolithic devices keeps supplies short.

VDIPs

VDIP modules typically have the largest pin out of any modules. Similar to ZIPs, VDIPs are vertically mounted modules with plastic-encapsulated components and epoxy-encapsulated chips (*Figure 5a*).

VDIP modules have pins along both sides of the substrate, with the pins on alternate sides aligned. Spacing along each row and across the module is 100 mils. The dual row of pins allows a higher connection density than SIPs, while maintaining 100-mil minimum spacing between adjacent pins.

Like ZIPs, VDIPs are useful in high-pin-count devices, where the host board is designed to normal through-hole design rules. VDIPs help retain the density advantages of vertical packages, while providing a low profile.

Ceramic Modules

For harsher environments, several types of modules are available with ceramic substrates and side-brazed leads. These modules sometimes have sealed metal lids to protect directly-mounted IC chips or utilize hermetically sealed LCC-packaged ICs. Four hermetic packaging styles are available: HVDIPs, HDIPs, PGAs, and QFPs.

HVDIPs

Hermetic vertical DIPs (HVDIPs) are vertically mounting ceramic modules with pins along both edges for through-hole mounting (*Figure 5b*). Components are hermetically encapsulated. Used in both low- and high-pin-count applications, they are especially attractive when high component density is required on the main board.

As with the plastic VDIP, pins on opposite sides of the module are aligned, and spacing in both directions is 100 mils.

HDIPs

Hermetic DIP (HDIP) modules have ceramic substrates with the same pin arrangements and footprints as standard IC DIPs (*Figure 6*). Hermetic components are mounted on both sides of the substrate. Hermetic DIP modules range in size from 24-pin devices with 300-mil widths to 60-pin, 600-mil devices to 900-mil special modules.

The QUIP

The quad in-line package (QUIP, *Figure 7*) is similar to the DIP except that the QUIP has a dual row of pins along the package edge. In-row and row-to-row spacing is 100 mils, with pins in adjacent rows aligned directly across from one another. The QUIP is a low-profile package with excellent mechanical ruggedness and the added advantage over DIPs of higher pin density for the same package length.

PGAs and QFPs

Pin grid arrays (PGAs, *Figure 7*) and quad flat packs (QFPs *Figure 8*) are ceramic-substrate packages similar to those used for monolithic devices, except that the

modules' cavities house more than one die. Each die is individually bonded to pads. The customized substrate provides the die-to-die interconnect and the connection to the I/O pins.

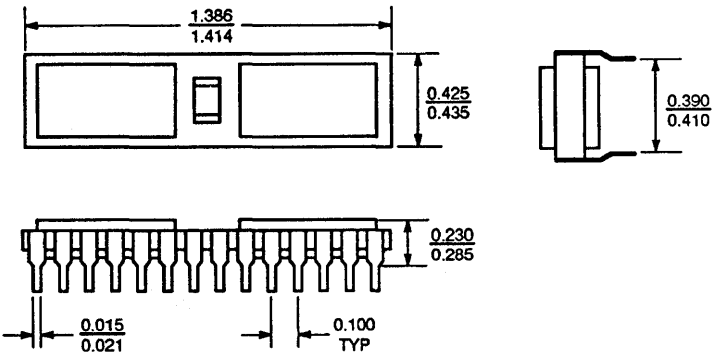


Figure 6. HDIP

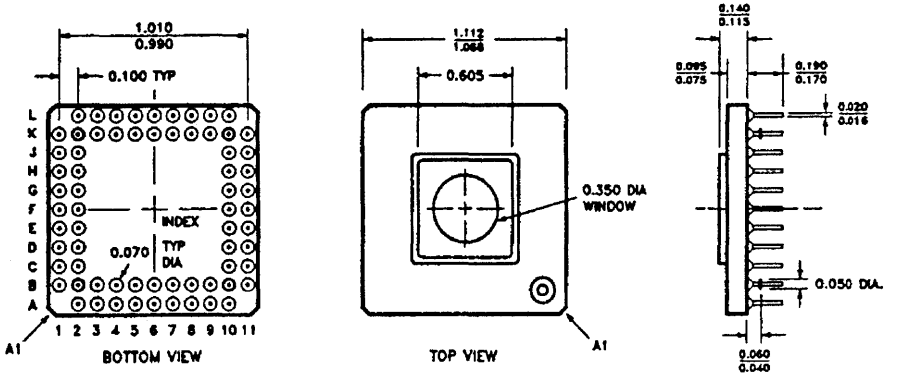


Figure 7. PGA Module

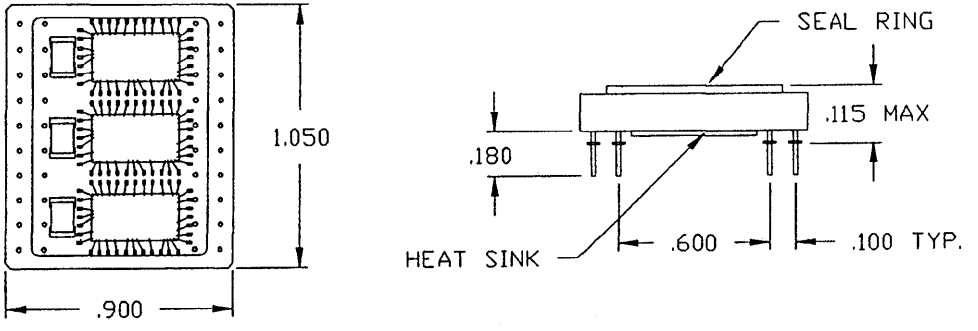


Figure 8. QUIP

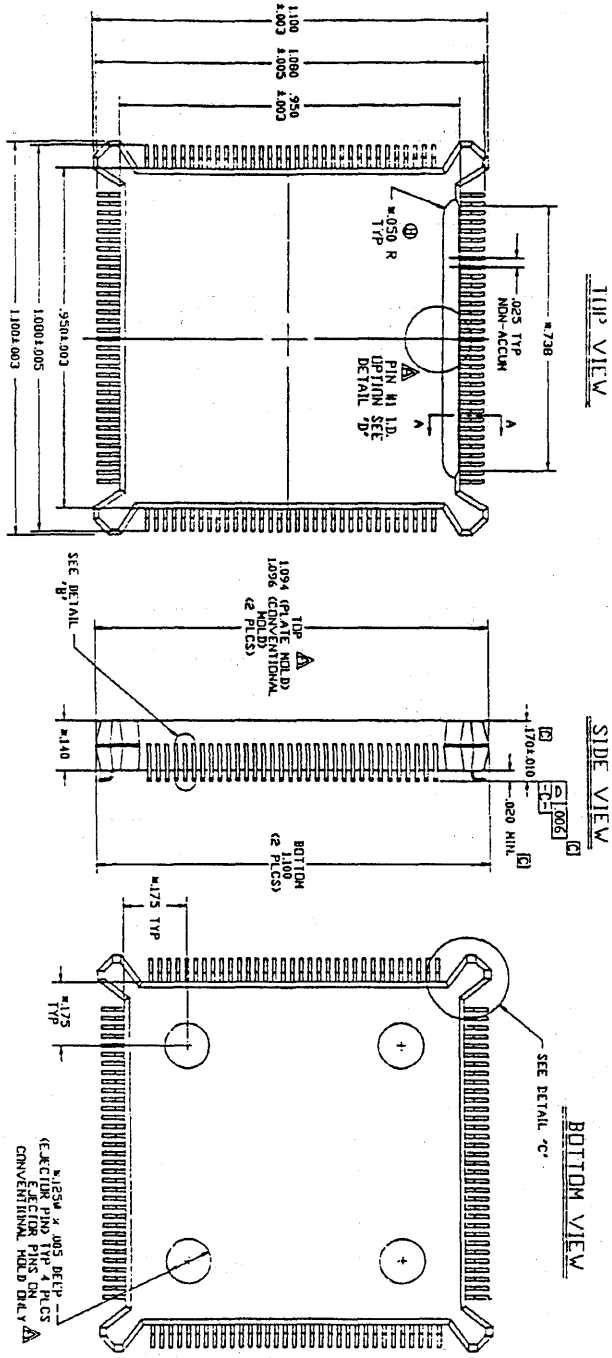

TOP VIEW
SIDE VIEW
BOTTOM VIEW
SEE DETAIL 'B'
SEE DETAIL 'C'
PIN #1 I.D. (OPTION SEE DETAIL 'D')
.025 TYP NON-ACCUM
.020 MIN
.175 TYP
.175 TYP
BOTTOM 1.100 (2 PLCS)
TOP (PLATE MOLD) 1.094 1.096 (CONVENTIONAL MOLD) (2 PLCS)
.950±.003
1.000±.005
1.100±.003
.950 ±.003
1.000 ±.005
1.100 ±.003

Figure 9. QFP

CYPRESS SEMICONDUCTOR

The Multichip Family of Universal JEDEC ZIP/SIMM Modules

This application note describes three Cypress memory modules, their special features, and how to use the modules as universal memory building blocks. The three modules are the CYM1821, CYM1831, and CYM1841, which provide 512K, 2M, or 8M of static RAM.

The CYM1821, CYM1831, and CYM1841 provide the versatility to design many different systems with the same memory modules. The pin out and footprints allow you to use the same module in 8-, 16-, or 32-bit systems in ZIP or SIMM form factors using a single board layout.

History

The JEDEC Solid State Products Engineering Council approved four series of SRAM ZIP/SIMM module pin outs for balloting[1] in December, 1987. The 64-pin module included definitions for 4 x 16K x 8, 4 x 64K x 8, and 4 x 256K x 8 generations.

The JEDEC definition established the industry standard for the mechanical specifications and pin outs of the three generations of modules (*Figure 1*). The CYM1821, CYM1831, and CYM1841 follow the JEDEC definition.

Variable Width

The JEDEC pin-out definition includes four chip-enable pins that each control a byte-wide block of memory (*Figure 2*):

- CS_1, on pin 32, enables I/O_0 through I/O_7
- CS_2, on pin 31, enables I/O_8 through I/O_{15}
- CS_3, on pin 34, enables I/O_{16} through I/O_{23}
- CS_4, on pin 33, enables I/O_{24} through I/O_{31}

You can use each generation as a x32, x16, or x8 memory block by driving the chip enables as address pins and connecting the I/O pins in parallel. This scheme allows the memory configurations shown in *Table 1*.

Variable Depth

The three modules provide additional flexibility in memory depth. All three are 64-pin modules with compatible pin outs. The CYM1821 has four no-connect pins: 29, 30, 35, and 36. Pins 29 and 30 are address pins on the CYM1831, and pins 35 and 36 are still no-connects. On the CYM1841, pins 35 and 36 are address pins. This allows the modules to function as memory options in a design.

The module family's variable depth is enhanced by the inclusion of two presence-detect pins: PD_0 and PD_1 on pins 2 and 3, respectively. These pins provide unique logic conditions for a system to automatically sense the amount of memory present, which permits the system to adapt automatically to the module that is plugged in. The presence-dectect pins are either tied to ground on the module or left open, according to the information in *Table 2*.

Figure 3 shows a simple circuit that decodes the presence-detect pins and generates depth-indicator status signals.

Layout Considerations

The three modules are available in either ZIP or SIMM form factors. Additional versatility is included in

Table 1. Memory Configurations

	Word Width		
	x8	x16	x32
CYM1821	64Kx8	32Kx16	16x32
CYM1831	256Kx8	128x16	64Kx32
CYM1821	1Mx8	512Kx16	256Kx32

Table 2. Presence-Detect Pins

	PD1	PD0
No Module	OPEN	OPEN
CYM1821	OPEN	GND
CYM1831	GND	OPEN
CYM1841	GND	GND

the module footprint to allow ZIP or SIMM modules to fit into the same board layout. The ZIP pins are arranged in the same hole pattern as a SIMM socket. If the board layout fits a SIMM socket, such as the AMP 821825-1, a ZIP plugs right in. This capability is useful for board prototyping and testing various memory depths in the same socket.

Reference

JEDEC Solid State Products Engineering Council, Committee Letter Ballot JC-42.3-88-9, 16 January 1988.

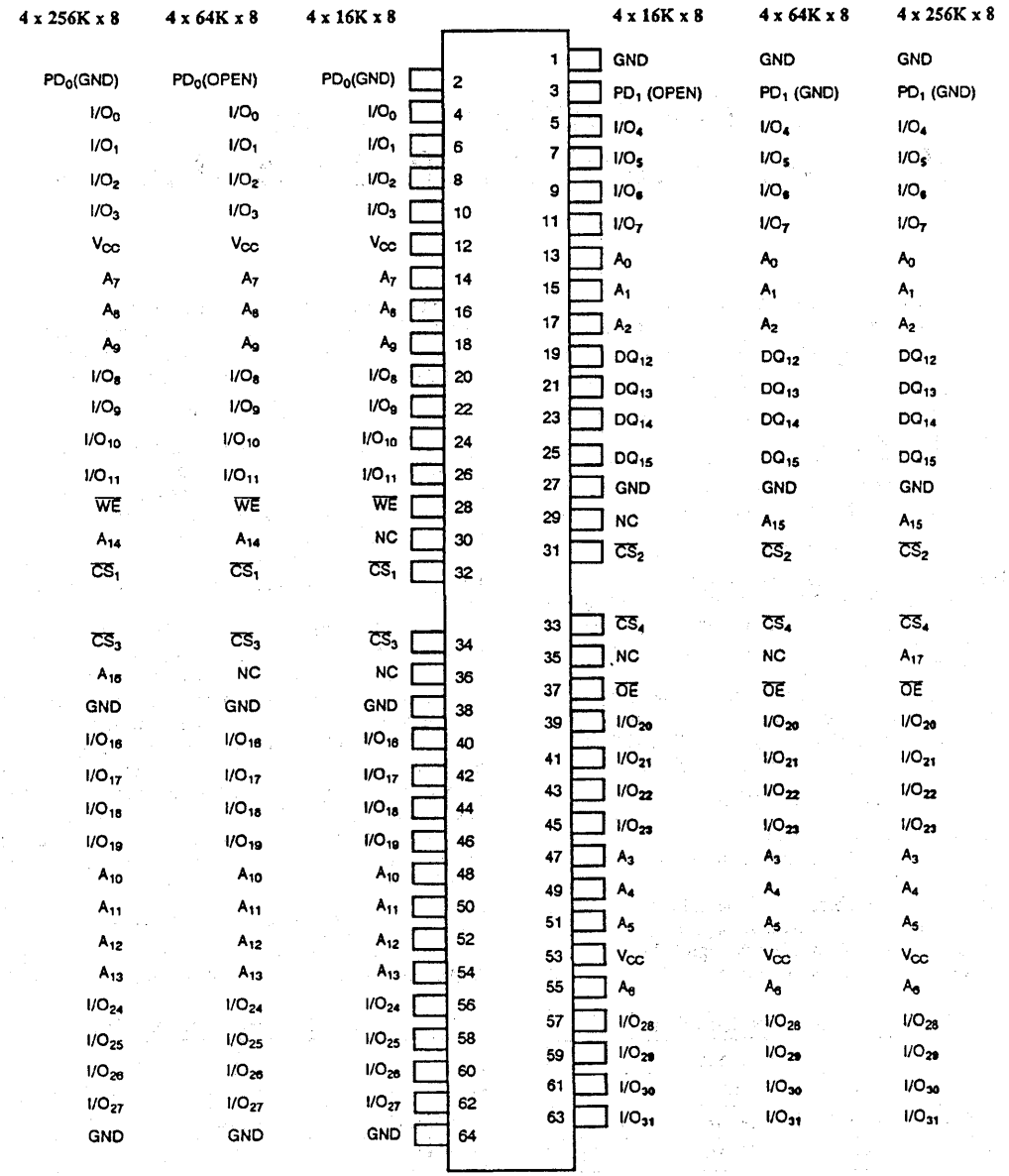


Figure 1. 64-Pin SRAM Module Pinout

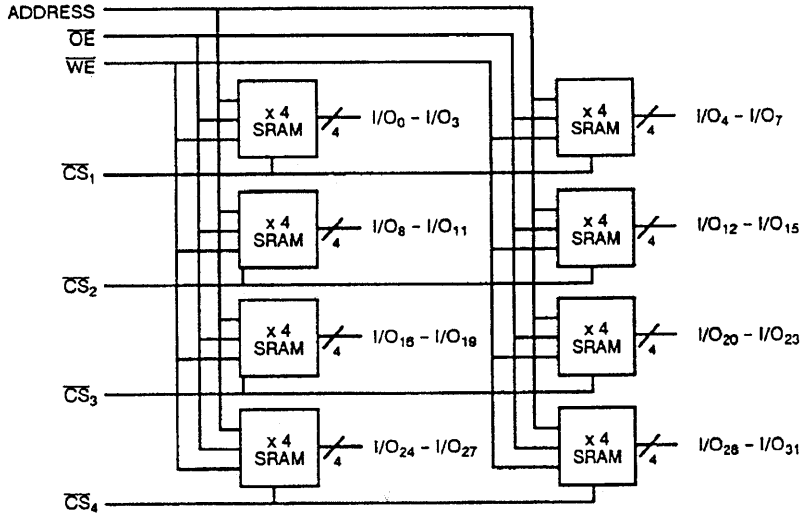


Figure 2. 64-Pin SRAM Module Block Diagram

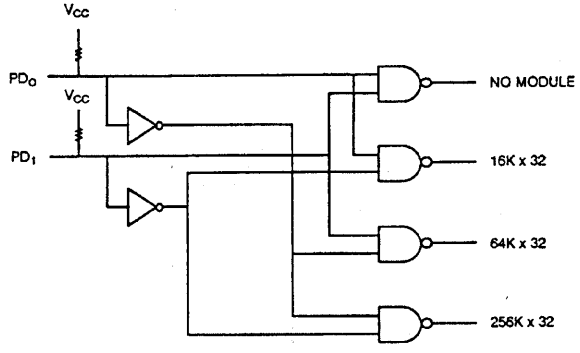


Figure 3. Depth Indicator Circuit

Section Contents

CYPRESS SEMICONDUCTOR

Noise Considerations in High-Speed Logic Systems

This application note explains why ECL is a lower-noise logic family than TTL or CMOS logic, both internally at the circuit level and externally at the system level. Also presented are the implications of ECL for your design needs.

In state-of-the-art logic system design, clock frequencies of 50 MHz and beyond are not uncommon and give rise to many noise problems that were not significant in the past. Due to the nature of TTL/CMOS logic, operating at these faster clock rates is inherently noisier and requires high-power output drivers with their associated ground-bounce problems.

Fortunately, ECL solves these problems. It is built for speed and is available in a low-power BiCMOS process technology. Since ECL was designed for high-speed applications in 1962, a number of design iterations have improved ECL devices. Consequently, it is the premier high-speed logic family.

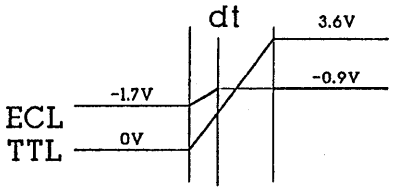
ECL's Internal Advantages

Internally, ECL steers current and compares input signals to a voltage level instead of switching transistors on and off over a wide voltage excursion, as do other logic families. ECL's small voltage swings and low-current switching in signal paths minimize crosstalk and noise generation (*Figures 1* and 2). ECL generates less noise switching logic levels due to the smaller dV/dt in the I = CdV/dt equation, where C is the coupling capacitance between signal paths, I is crosstalk current, dt is the rise/fall time, and dV is logic swing.

Additionally, built-in temperature and voltage compensation provides constant noise immunity in 100K devices, so that noise margins are flat. In these devices, temperature compensation is designed into the DC input thresholds by voltage regulation. A correction factor designed into the current source, along with added circuitry between the output transistors' bases, make 100K ECL's output voltage levels insensitive to temperature. These corrections rely on opposing positive or negative temperature-tracking-coefficient circuits. In both 100K and 10KH ECL devices, voltage compensation is done by regulating an internal reference voltage, supplying a constant current source, and making both functions independent of supply voltage. These compensations result in a 3x improvement over TTL noise immunity.

Additional anti-noise features include differential pairs, which prevent large current spikes when switching logic states, provide clean power supplies, and reduce ground bounce. Differential paths also cancel internal parasitic charging currents.

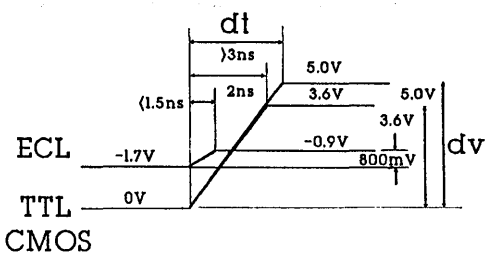
Finally, ECL's more constant power dissipation — independent of operating frequency — keeps power-supply surges to a minimum. Supply current drain is governed by the constant-current sources that provide operating current for the differential switches and level-shifting networks. Thus, ECL's current drain remains the same regardless of the state of the switches. The high ratio of ECL noise immunity to internally generated noise also contributes significantly to reliable system operation.



$$I = C \frac{dV}{dt}$$

Crosstalk current I is less for ECL than TTL due to smaller dV and dt

Figure 1. Effects of Rise and Fall Time



SLEW RATE = dV/dt

RISE/FALL TIME = dt
LOGIC SWING = dV
Therefore, time is saved because the logic swings are smaller and rise/fall time faster

Figure 2. Effects of Slew Rate

ECL in the System Environment

In the ECL system environment, low-impedance open-emitter outputs and high current capacity allow you to use board-level transmission-line techniques that reduce reflections and decrease roll-off of high-speed rise and fall times. To understand these system-level advantages, consider that voltage-mode circuits have a High-state output impedance between 50 and 150Ω and exhibit an output-stepped characteristic. They first reach 50 percent of the final value, then later reach the final value, which can be 3.5V and above.

In contrast, ECL output impedances are less than 10Ω and ensure a full-valued signal into transmission lines. The signal only needs to be 800 mV. Outputs are also capable of supplying 50 mA, which is required to drive passive terminations. Because ECL gives you the built-in ability to drive controlled-impedance PCB traces, you can make tradeoffs among power dissipation, speed considerations, and PCB trace width.

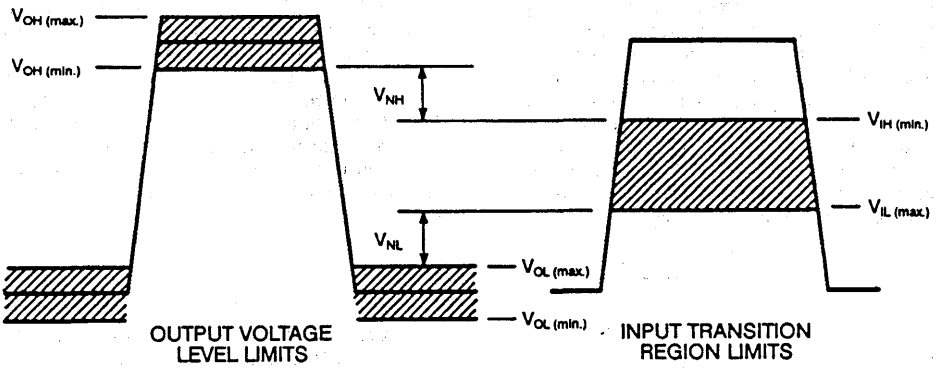
Some ECL devices have skew-free differential or complementary outputs for common-mode noise rejection at the receiving end of either board traces or twisted-pair wire. As mentioned earlier, ECL's smaller logic transitions lower crosstalk between board-level signal traces, as well as at the IC level.

The smaller transitions also prevent the emitter-follower outputs from generating large current spikes when switching logic states, unlike TTL totem-pole outputs. TTL current spikes are also related to I = CdV/dt. For ECL, C is the capacitive load. TTL ground bounce results from the current spikes and the inductance (L) between the board and the device's pins and bond wires. The bounce voltage (V) equals -Ldi/dt, which can be severe (see the *Reference*) and can cause the chip's ground to rise. Because ECL's emitter followers provide superior output current and the lower capacitance of characteristic-impedance transmission lines, ECL solves the problem of power-supply droop and spikes when a large number of transistors change state.

Logic Family of Choice

The factors described here make ECL the logic family of choice when designing systems at 50 MHz or greater clock/data rates. As a percentage of total logic swing, ECL provides superior noise margin in the system environment compared to both TTL and CMOS logic.

In a typical TTL/CMOS system, board-level noise can be 800 mV or higher due to ground bounce and other switching noise. The *Reference* explains this effect for both CMOS and TTL and includes actual measurements



Note: V_{NH} and V_{NL} are the High- and Low-level device noise margins. Because ECL system noise is much lower than TTL system noise, the smaller ECL device noise margins are still better than the TTL margins.

Figure 3. Identifying Specification Limits on Input and Output Voltage Levels

Table 1. System Noise Generated by Logic

Parameter	TTL	CMOS	100K	10K
HIGH "1" (V_{NH}) (mV)	400	400	140	145
LOW "0" (V_{NL}) (mV)	400	400	145	175
Typ. System Noise (mV)	800	800	20	20
Logic Swing	3.5V	5V	900 mV	
Percent Noise[1]	22.8%	16%	2.2%	

[1]Percent Noise = (system noise/logic swing) x 100

of ACL (CMOS) devices. In an ECL system, the noise is closer to 20 mV. As shown in *Table 1* and *Figure 3*, ECL's overall system or board-level noise is at a much lower percentage of total signal swing than in TTL/CMOS systems. ECL is therefore less susceptible to logic errors at high speeds (*Figures 4* and *5*).

ECL's full temperature and voltage compensation results in relatively constant signal levels and thresholds; improved noise margins over chip-to-chip temperature and voltage variations; and a tighter AC window in the system environment. Another benefit of reduced noise generation is the improvement in electromagnetic interference (EMI) and radio-frequency interference (RFI) in ECL systems versus TTL and CMOS.

Reference

"EDN's advanced CMOS logic ground-bounce tests," David Shear, *EDN*, March 2, 1989.

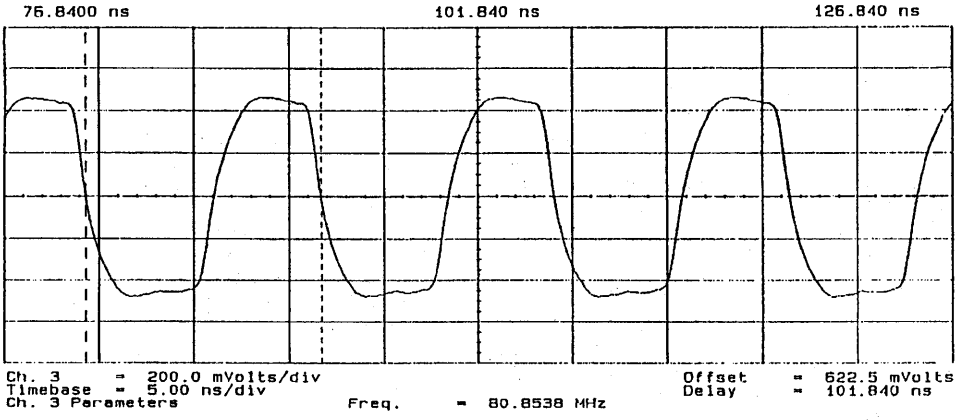


Figure 4. ECL Signal

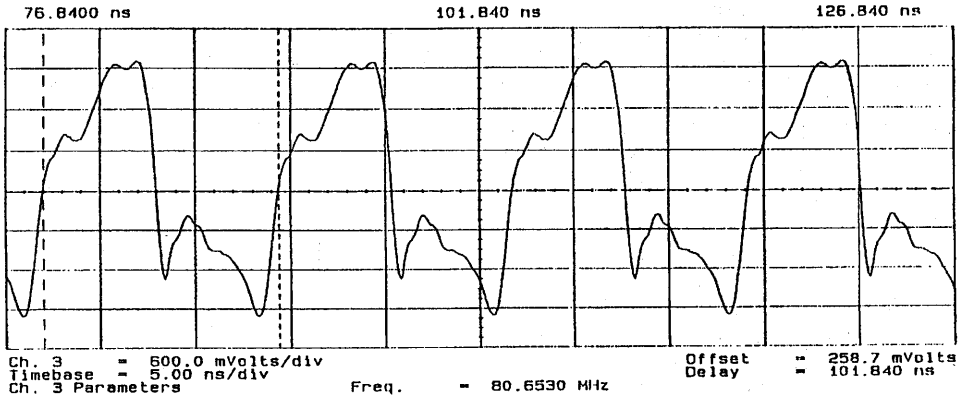


Figure 5. TTL Signal

CYPRESS SEMICONDUCTOR

Using ECL in Single +5V TTL Systems

The advent of very high speed, low-power, ECL-compatible BiCMOS SRAMs and PLDs is causing an evolution in high-performance systems. ECL's inherent speed and noise improvement is well documented, but questions and misconceptions concerning the devices might occur. These questions stem from the fundamental problems of mixing CMOS logic and bipolar ECL circuits on the same die and from interfacing ECL devices in single +5V supply CMOS/TTL systems.

Chip-Level Considerations

At the chip level, it is possible to integrate both ECL and CMOS logic with negligible noise coupling. This compatibility is mainly due to the absence of noisy high-drive output devices between the device's CMOS sections and the ECL I/Os.

The combined ECL/CMOS chips exhibit very low interconnect capacitance between devices on-chip, and the drive requirements are minimal. The devices generate less noise than occurs between devices at the board level. The noise magnitude on the chip V_{EE} line equals approximately 20 mV worst case, in contrast to 800 mV of noise in typical high-speed, board-level CMOS/TTL designs.

Further, the unique configuration that Cypress Semiconductor employs to connect the device ECL circuit ground (V_{CC}) and ECL output ground (V_{CCA}) reduces noise coupling between the internal CMOS circuitry and the ECL output drivers. Because the devices have a low overall noise level and employ internal supply decoupling, both the ECL and CMOS sections of Cypress devices run successfully on the same power pin.

Board-Level Considerations

At the board level, using ECL-I/O-type devices in single +5V TTL systems is possible with off-the-shelf level translators. These translators are made specifically to run standard ECL devices in a pseudo-ECL logic mode, with switching levels pulled up to the range between the +5V supply and ground.

ECL normally uses a -5.2V supply for 10K- and 10KH-compatible devices or a -4.8V supply for 100K-compatible devices. Pulling ECL circuits and memories up to a single positive 5V level instead of using the normal supply does not change any performance or absolute-value logic levels so long as all the ECL device V_{CC} pins are tied to +5V, and the device V_{EE} pins are tied to ground.

The translators have separate supply pins and either separate or common ground pins for the circuit's ECL and TTL portions. This feature isolates the noisy TTL supplies from the ECL section, which runs at much faster speeds and with tighter noise margins.

ECL-TTL-ECL Translation

The Brooktree Bt501 (10KH ECL compatible) and Bt502 (100K compatible) octal transceivers and translators perform bidirectional ECL/TTL transfers. These devices offer the option of supplying +5V only to the circuit's ECL portion (*Figure 1*). This arrangement makes it possible to design the system with only one power source and simplifies the task of adding ECL circuitry to a TTL board.

You can isolate the ECL section from the TTL section in much the same way you isolate analog and digital sections on a mixed-signal board. To isolate the TTL-generated noise from the ECL +5V supply lines, you must maintain separate ECL and TTL power lines; you can

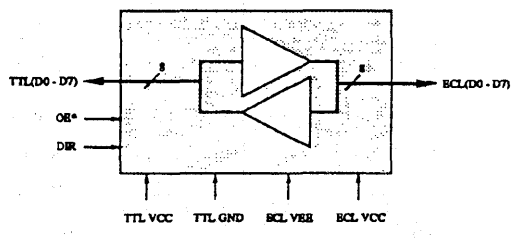


Figure 1. Bt501/502 TTL/ECL Transceiver

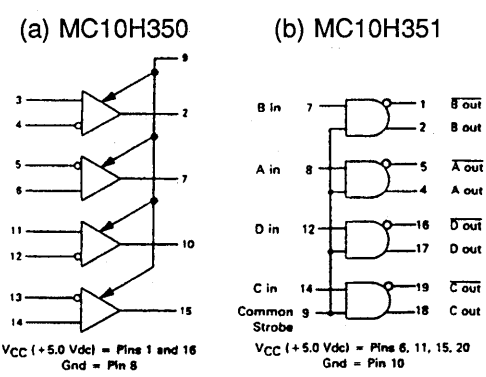


Figure 2. ECL to TTL (a), TTL/NMOS to ECL (b)

have common or separate ground planes. Employ normal power-supply decoupling for ECL devices.

The Brooktree devices have the advantage of providing eight sets of transceivers for both translation directions in one IC package. A disadvantage is speed. The Bt501/502 devices have maximum propagation delays of 7 ns when translating from TTL to ECL and 11 ns in the other direction. In some applications this might be too slow.

For a faster set of translators that run on single 5V supplies, try the Motorola MC10H350 and MC10H351 (*Figure 2*). The MC10H350 only translates in the ECL-to-TTL direction, but it is faster than the Brooktree parts, with a 5-ns maximum propagation delay, and includes differential inputs. The MC10H351 is the TTL-to-ECL converter, offering a maximum delay of 2.1 ns. These devices have separate ECL and TTL supply pins, but have common ground-pin connections.

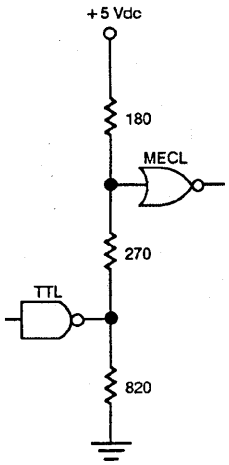


Figure 3. TTL to ECL

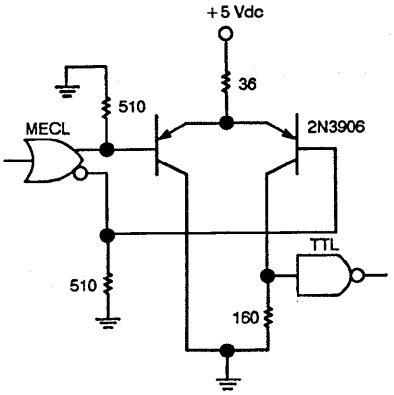


Figure 4. ECL to TTL

Another method of translation is to use all discrete components or a combination of discrete and integrated products. The purely discrete approach speeds up the translation but introduces the risk that noise from the TTL-to-ECL sections might feed through the power and ground connections. You also have to consider the lack of temperature and/or voltage compensation, which affects noise margins.

For translating TTL signals to ECL, use a simple voltage divider network, whose primary purpose is to reduce the TTL levels to ECL-level logic transitions (*Figure 3*). In the other direction, a high-speed PNP transistor increases the logic swing to accommodate TTL-logic-level transitions (*Figure 4*).

A faster approach appears in *Figure 5*, where a differential pair consisting of two PNP transistors takes advantage of the ECL differential outputs. The choice of transistors greatly influences the propagation delay through these translators. Motorola manufactures some

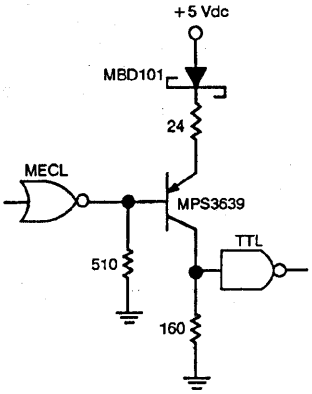


Figure 5. ECL to TTL

very fast RF-type PNP transistors and matched PNP pairs that can serve well in the circuits shown.

CY10E383/101E383 Full-Duplex Translator

For the ultimate in speed and flexibility, the Cypress CY10E383/CY101E383 is a new-generation, full-duplex, TTL-to-ECL and ECL-to-TTL logic-level translator designed for high-performance systems (*Figure 6*). The CY10E383/CY101E383 has many features to satisfy a variety of applications.

In the past, level translators suffered from having an insufficient number of channels or supply options. This caused skew and noise problems that made the use of high-speed ECL logic levels in TTL systems highly undesirable. The CY10E383/CY101E383 contains ten independent TTL-to-ECL translators and ten independent ECL-to-TTL translators for high-speed, bidirectional, full-duplex data-transmission, mixed-logic, and bus applications. The CY10E383/CY101E383 is especially well suited to driving ECL backplanes between TTL system boards.

The translator is implemented with differential ECL I/O to provide balanced, low-noise operation over controlled-impedance buses between TTL and/or ECL subsystems. The part features a delay of only 2 ns max from TTL to ECL and 3 ns max from ECL to TTL, with minimum skew between channels.

The CY10E383/CY101E383 comes equipped with internal 2-KΩ pull-down resistors tied to V_{EE} (ECL supply) to decrease the number of external components. For system testing purposes or for driving light differential loads, the pull-down resistors are the only termination, thus eliminating up to 20 external resistors. You can also use standard ECL terminations with the internal pull-down resistors and still adhere to standard 10K/10KH and 100K logic levels. Additionally, the translator contains an ECL V_{BB} reference voltage output, which you can use to tie half of the ECL inputs for single-ended operation.

The device is designed with ample ground pins to reduce bounce and has separate ECL and TTL power/ground pins to reduce noise coupling between logic families. The CY10E383/CY101E383 can operate in single or dual supply configurations while maintaining absolute 10K/10KH and 100K level swings to be used with either TTL-type (+5V) or ECL-type (-5.2V) supplies or both.

The translators are offered in standard 10K/10KH (10E) and 100K (101E, 100K levels with up to -5.2V power supply) ECL-compatible versions. The TTL I/O is

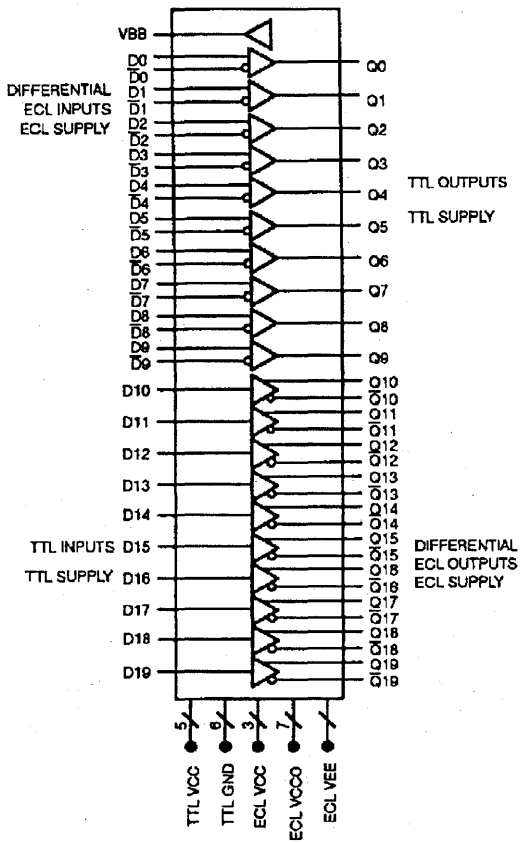


Figure 6. CY10E383 Full-Duplex TTL/ECL/TTL Translator

fully TTL compatible. The CY10E383/CY101E383 is packaged in an 84-pin, surface-mountable PLCC.

Reference

Blood, William R., Jr., "Motorola MECL System Design Handbook," (Motorola Semiconductor Products Inc., Fourth Edition, 1988.)

CYPRESS SEMICONDUCTOR

BiCMOS TTL and ECL SRAMs Improve High-Performance Systems

A new BiCMOS process based on clean-slate approaches to implementing ECL or TTL logic with bipolar, BiCMOS, and CMOS transistors in single devices is revolutionizing the speed/density characteristics of SRAMs. Historically, BiCMOS technologies were developed as either CMOS speed enhancers or bipolar power misers. The resulting BiCMOS processes were patches on either CMOS or bipolar process flows, and performance for the complementary bipolar or MOSFET components was sub-optimal.

In contrast, Cypress's STAR M2 is a third-generation, 0.8µ BiCMOS technology in which the baseline process is BiCMOS. In the STAR process, nonvolatile elements such as polysilicon loads and TiW fuses are easily incorporated into the baseline process. This results in high-density SRAMs, high-speed PLDs, and high-density EPROMs/PLDs.

Figure 1 shows a simplified cross section of the STAR M2 BiCMOS process. This 18-mask, double-poly, double-metal technology utilizes a thin epitaxial layer to achieve NPN F_t greater than 10 GHz and CMOS latch-up immunity. The MOSFETs both use lightly doped drains for high performance and reliability.

In contrast to the architectures of SRAMs made using first-generation BiCMOS processes, STAR's polysilicon bipolar emitter is the same poly used for MOS gates. This enhances NPN performance and decouples the NPN from the poly load module used for 4T SRAM cells. By utilizing this poly load resistor, STAR allows for an 85-square-micron memory cell. This third-generation process beats second-generation BiCMOS technologies in terms of product performance, density, and manufacturability.

SRAMs are a key technology driver, and BiCMOS fills the gap between the power-hungry pure bipolar ECL and the very high density, medium-speed CMOS. To indicate the performance of the Cypress process, *Table 1* summarizes gate delays as a function of logic family and fanout.

Table 1. Gate Delays

Gate	T_{pd} (ps), Fanout = 1	ps/Fanout
CMOS	110	70
BiCMOS	240	12
ECL	95	30*
*ECL delay varies with the square root of fanout		

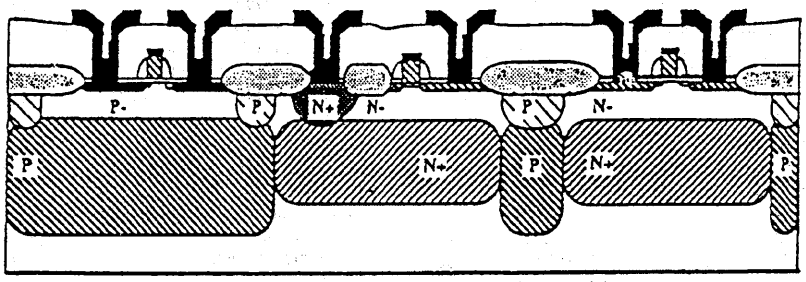


Figure 1. STAR M2 BiCMOS Process, Simplified

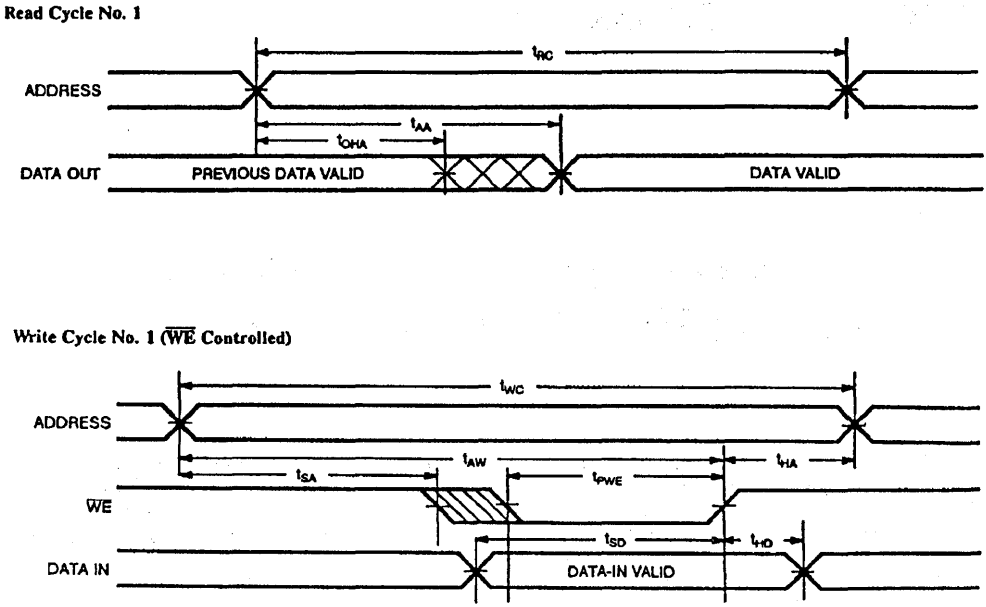


Figure 2. Balanced Read/Write Cycle Timing Diagrams

This performance comparison shows that you can use the gates according to functional capabilities. ECL gates are fastest. They use the most power, but because they can drive 50Ω loads and have low offset voltages, they are usually used for I/O or other high-drive paths. CMOS gates' internal propagation delays are short for small-load logic functions, but their outputs are less suited for driving heavy loads. However, the CMOS gates' small size make them attractive for memory arrays. BiCMOS gates are used where medium loading has an effect on speed, such as for internal logic between I/O and the memory array.

Cypress STAR combines bipolar ECL I/O with bipolar, BiCMOS, and CMOS internal functions. This helps parts such as Cypress's industry-standard CY10E474/CY100E474 1K x 4 BiCMOS SRAMs draw 275 mA, while exhibiting world-class access times of 3.5 ns (285 MHz). The STAR process makes possible a low-power version (190 mA) exhibiting 5-ns access times.

STAR also combines CMOS I/O with the bipolar, BiCMOS, and CMOS internal functions for TTL compatibility and faster access. For example, a BiCMOS TTL implementation of an industry-standard SRAM such as a 64K common-I/O device has an access time of 8 ns and runs on 130 mA max., 40 mA standby. The majority of the power is consumed driving the outputs at fast switching times. At 40 MHz, the part runs at 100 mA. (These specifications are for Cypress's CY7B166 TTL-compatible devices.)

TTL BiCMOS

In a Cypress TTL BiCMOS SRAM (for a basic layout, see *Figure 1* in "A New Generation of BiCMOS TTL SRAMs"), CMOS provides a small cell size, which in turn gives high yields and lower cost than pure bipolar. Consequently, the memory array consists of pure CMOS for low power and high density.

For the sake of good frequency response, low offset voltage, and high g_m (I_{out}/V_{in}), the sense amp that reads the memory cell is all bipolar.

Special attention has been given to the read/write paths to achieve an overall balanced read and write cycle. *Figure 2* shows timing diagrams of the balanced read/write cycle for a Cypress 8-, 10-, or 12-ns TTL BiCMOS SRAM. The access time on a read cycle (t_{RC}) is the same duration as the write cycle (t_{WC}), which consists primarily of write pulse width. Such a balanced cycle reduces overall design complexity by maintaining a 50-percent duty cycle for the system timing clocks.

Because the bit-line paths that the sense amp drives are highly capacitive, the high-drive current and small ECL-type voltage swings provided by bipolar NPN transistors is essential for fast access times. Further, the low-offset feature of a bipolar differential-pair amplifier makes it possible to resolve small memory-cell output voltages quickly.

The write amplifier, on the other hand, is BiCMOS, which can use supply-voltage levels to write the memory

locations. Additionally, BiCMOS's fanout characteristics enhances write cycle timing. Other internal logic gates with highly capacitive nodes, such as output buffers, decoders, and wordline and write drivers are all BiCMOS. Control functions with small fanouts are CMOS.

The high internal speeds obtained by mixing bipolar, BiCMOS, and CMOS transistors allows the outputs to switch more slowly to achieve a given access time. This reduces noise and ground bounce, making the BiCMOS SRAMs easier to use.

The I/O architecture for TTL BiCMOS appears in *Figure 3a*. I/O circuitry for the TTL devices is CMOS for compatibility with existing products. However, bipolar transistors are employed as diodes in conjunction with the CMOS output transistors to keep output swings within traditional TTL levels, instead of the full 5V swing typical of CMOS. This further reduces ground bounce.

On the input side in *Figure 3a*, the CMOS devices are labeled M2 and M4. Input clamping diodes are included to serve as ESD protection devices and to meet MIL STD-883C Method 3015 static discharge voltages of 2001V. The inputs meet standard CMOS specifications.

The output bipolar transistors Q2 and Q3 drop two V_{be} levels (1.6V) to reduce the High-level output swing. One device is tied base-to-collector as a diode and the other is the High-level drive transistor. This arrangement limits the voltage swing to TTL-type levels, which increases speed and limits noise by decreasing the total dV/dT rate of change in the output swing. The Low-level pull-down resistor is M18, an N-type MOSFET.

ECL BiCMOS

In ECL BiCMOS SRAMs, a larger percentage of pure bipolar circuitry is used than in TTL BiCMOS, but CMOS and BiCMOS are still used extensively. ECL I/O is shown in *Figure 3b*. The inputs consist of an NPN transistor used as an emitter follower (Q1) tied to one side of a differential pair (Q2 and Q3). The output is an open emitter NPN transistor that can be tied to an external pull-down resistor to drive transmission lines.

It is not difficult to integrate both ECL and CMOS logic on a chip with negligible noise coupling. This is mainly due to the absence of noisy high-drive output devices between the device's CMOS sections and the

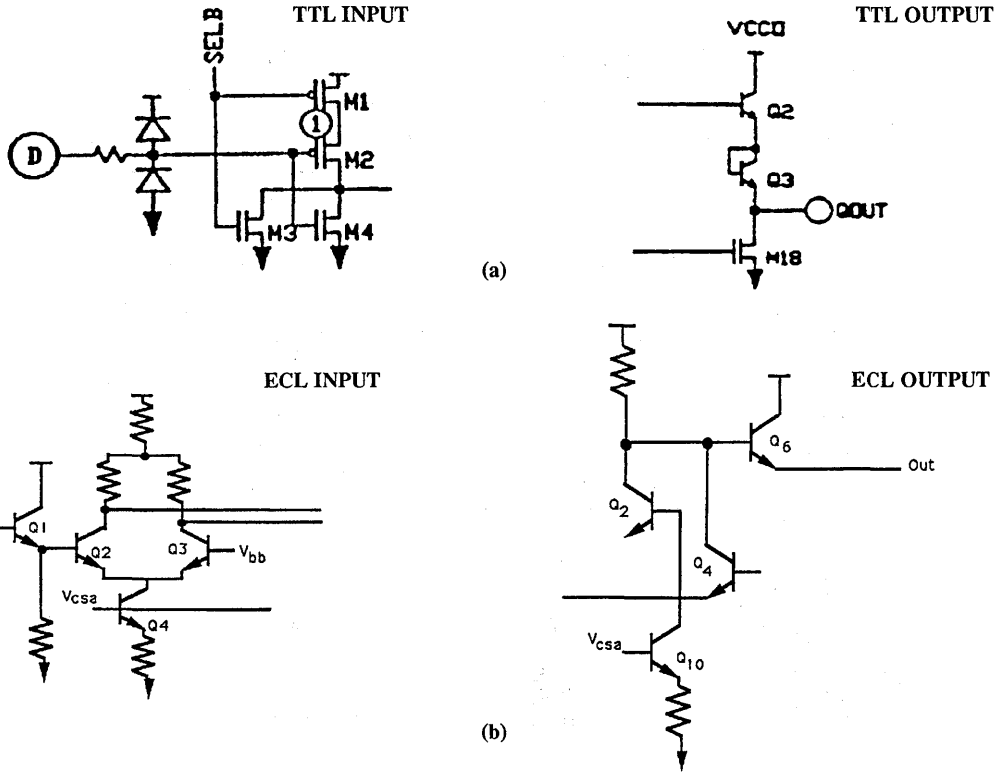


Figure 3. I/O Architectures

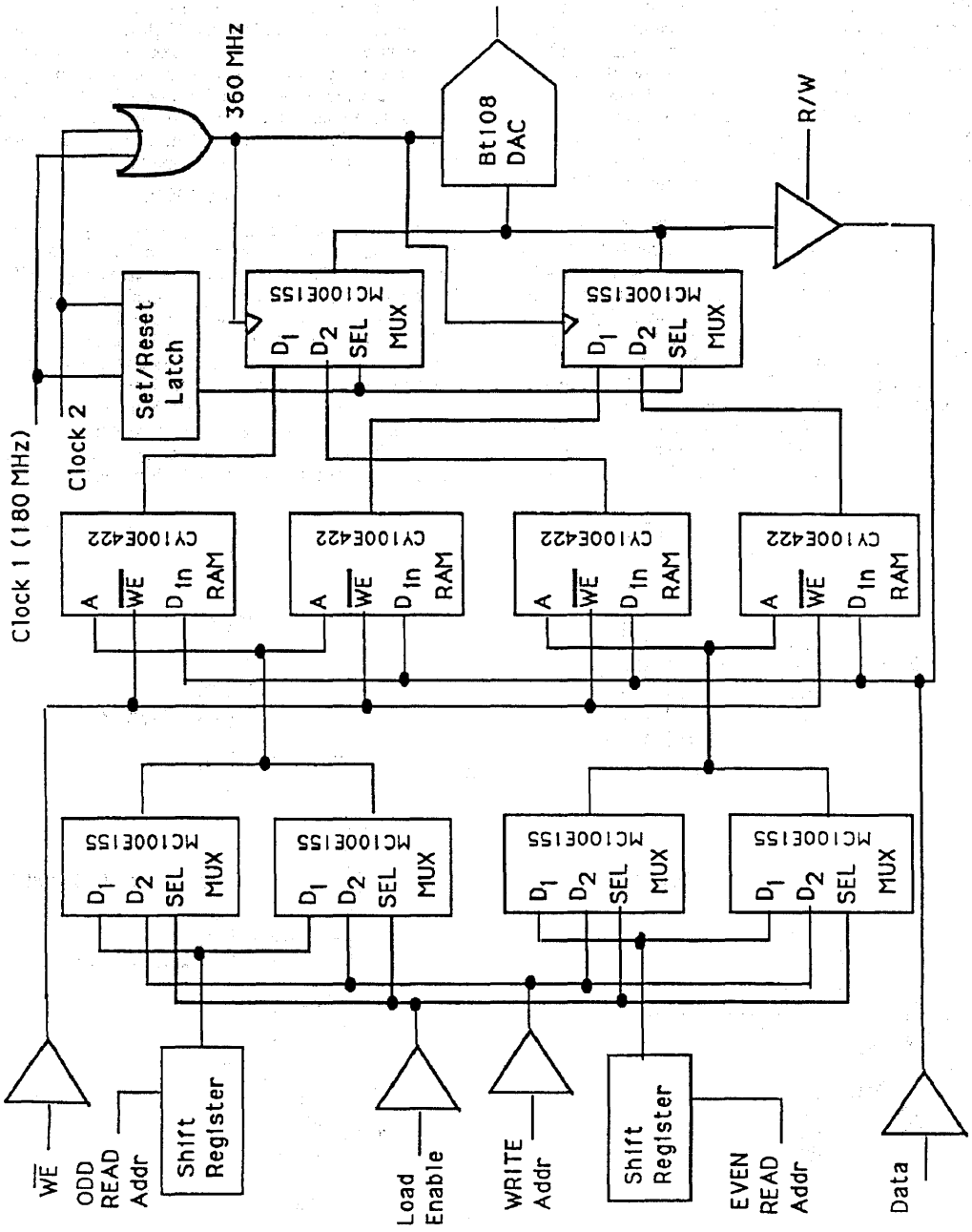


Figure 4. Raster-Graphics Video System

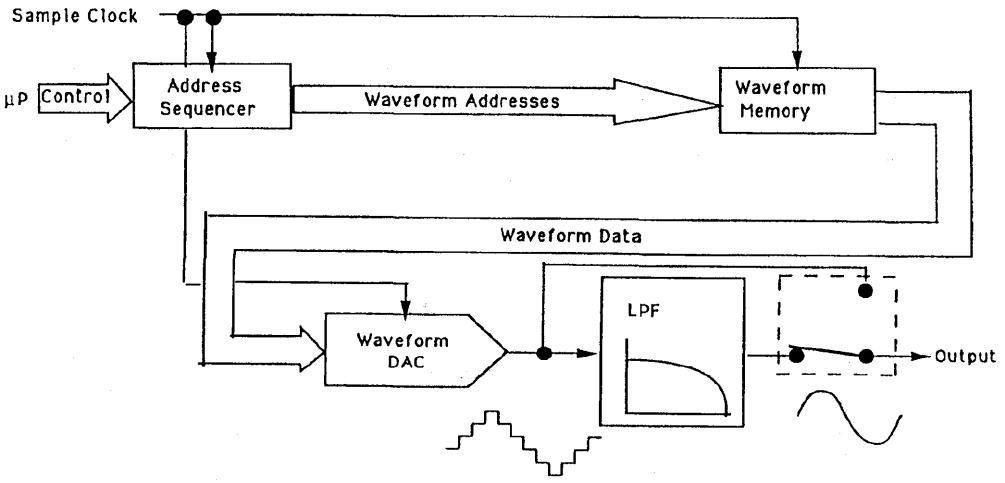


Figure 5. Waveform Synthesis System

ECL I/Os. Interconnect capacitance between devices on the chip is very low, and drive requirements are minimal. Consequently, noise is not generated at the high levels encountered between discrete devices installed on a board. The noise magnitude on the chip V_{ee} line is approximately 20 mV worst case, rather than the 800 mV encountered in typical high-speed, board-level CMOS/TTL designs. With a low overall noise level and internal supply decoupling, both the ECL and CMOS sections of Cypress devices run successfully on the same power pin.

Cypress employs a unique configuration to connect the device ECL circuit ground (V_{cc}) and ECL output ground (V_{cca}). This configuration further reduces noise coupling between the internal CMOS circuitry and the ECL output drivers. The configuration also inhibits output oscillation in response to slow or noisy input signals.

BiCMOS ECL and TTL SRAM Applications

Applications for ECL and TTL SRAMs include graphics and image processing, waveform generation via direct digital synthesis (DDS), and fast µP systems.

In video graphics, ECL memory stores color image information. In waveform generation and DDS, ECL memory stores digital representations of analog waveforms before they are fed to a digital-to-analog converter (DAC).

In a typical raster-graphics video system (*Figure 4*), 3.5-ns CY100E422 ECL SRAMs are used as color look-up tables (LUTs) to drive a Brooktree BT108 video DAC. The SRAMs are interleaved to achieve the necessary speed and to supply the 8-bit words required for 3D solids shading. Motorola MC100E155s, which have clock-to-output delays of 1 ns, are used as 2:1 mux latches.

In operation, read and write addresses and data are fed to the SRAMs from the octal 2:1 multiplexer/latches, and the color pixel data from the memories is sent to the DAC. This path is one of three in which the DAC drives the intensity of the display's red, green, or blue (RGB) electron gun drivers. This system's 360-MHz speeds are sufficient to drive 2K x 2K displays.

The waveform synthesis system in *Figure 5* can be controlled by either a microprocessor or a numerically controlled oscillator (NCO). Another part of the system writes waveform data to memory. Then the processor commands an address sequencer, whose output controls the memory, and the data read out is fed to the DAC, which outputs an analog waveform. This type of fast digital waveform synthesis finds many applications in satellite communications and video and test equipment.

The 8-, 10-, and 12-ns speeds of the TTL 16K x 4 CY7B166 SRAM have improved the throughput of such systems. The system could also use ECL BiCMOS for increased speed, but the resolution of available high-speed ECL DACs is not as high as available TTL DACs.

For analog-to-digital applications, ECL and TTL SRAMs are used with high-speed flash A/D converters. Some of converters have ECL outputs, whose clock rates range from 20 MHz to 1 GHz. Other converters have TTL outputs as fast as 25 MHz. In applications such as HDTV, phased-array radar, digital oscilloscopes, and single-event digitizers, the SRAMs create high-speed specialty memories such as self-timed SRAM, pipelined SRAM, and interleaved SRAM.

Further applications for ECL and TTL SRAMs are found in high-performance workstations, file servers, and high-end embedded controllers. *Figure 6* shows an example based on Mips Computer's 100K ECL version of a

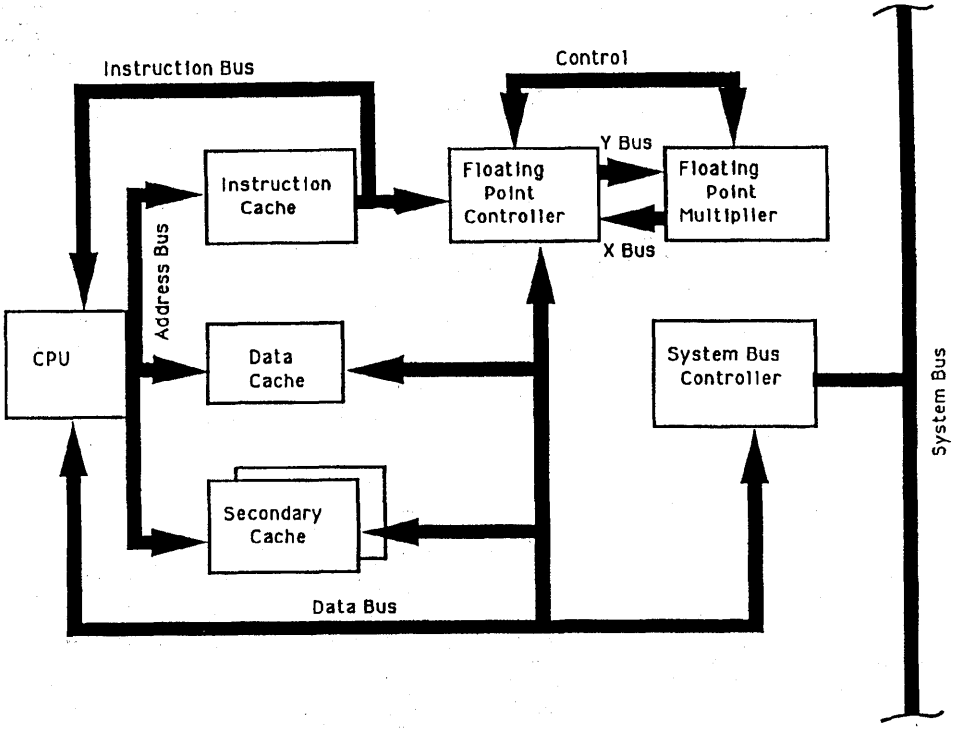


Figure 6. RISC System

commercial RISC microprocessor, which has a clock frequency of 67 MHz and is rated in a general-purpose application mix at 55 MIPS. The cache and TAG blocks are implemented using ECL BiCMOS SRAMs.

The system uses standard memories to provide two levels of data cache. The primary caches include 64 Kbytes of storage for instructions and 16 Kbytes for data, using the fastest 64-Kbit, 8-ns, CY100E494 SRAMs. Cache control is part of the integer unit. With primary 8-ns caching, the R6000 CPU can fetch both an instruction and a data word every cycle, instead of having to wait several cycles for main memory to keep pace. The slower 512-KByte secondary cache is made up of 20-ns devices.

A general-purpose cache-TAG implementation using standard ECL memories (*Figure 7*) uses two Cypress 1K x 4 CY100E474 ECL SRAMs (3.5 ns max access time). Two Motorola MC100E107 quint XOR/XNOR gates (800 ps max prop delay D to F) perform the compare function. The speed of the SRAMs and logic correspond to a 4.5 ns address to match comparison time. Note that the outputs of ECL PLDs or logic are wire ORed to save one additional component.

Alternatively, one CY100E302 (16P4) ECL PLD could be programmed to implement the 8-bit compare function in approximately 3 ns and save board space. Other memory sizes (e.g., 16K x 4) could be used to increase depth, and word width could be optimized by cascading devices.

Figure 8 shows the critical path for a TTL 80386-based cache system with a two-phase clock. The path consists of a DRAM controller implemented in a gate array, address generation configured in PLDs, cache SRAM, and cache TAG. *Table 2* shows how the speed of cache tag and cache RAM affects path speeds.

Table 2. Path Speeds for 80386 Cache

	Bus Cycle Time (MHz)		
Device	**33**	**40**	**50**
Gate array	17.5	15.0	12.0
TTL PLDs	7.5	7.0	5.0
Cache RAM enable	15.0	12.0	10.0
Cache TAG	20.0	15.0	13.0
Total	60.0	50.0	40.0
+ 2-phase clock	30.0	25.0	20.0

As bus cycle times decrease because of faster µP clocks, the speed of the cache TAG and cache RAM become very important in achieving path speeds. For today's TTL µPs, BiCMOS TTL SRAM implementations reduce access times to meet cycle time requirements. An example is shown in using the Cypress CY7B160 16K X 4 TTL BiCMOS SRAM. It is an 8-, 10-, and 12-ns device with internal decoding to enable easy memory expansion without sacrificing speed. *Figure 9* shows how to use four devices to create an 8-, 10-, or 12-ns 64K X 4 memory and a 32K X 8 memory. Additional configurations are also easily implemented using no external decoding.

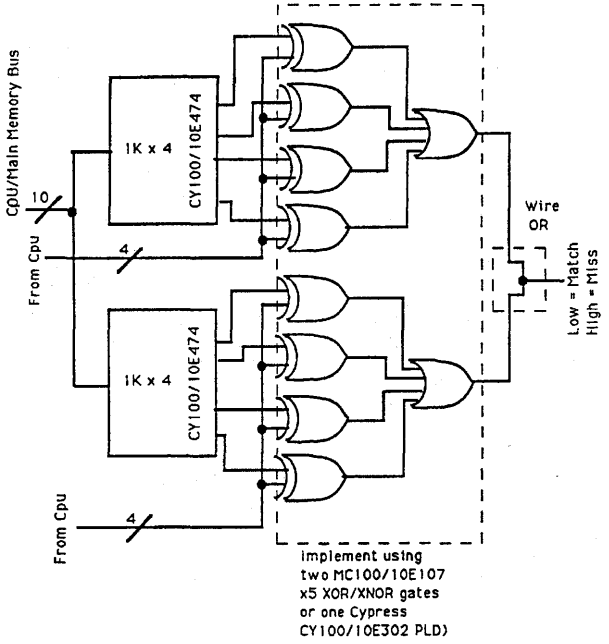


Figure 7. General-Purpose Cache-TAG Implementation

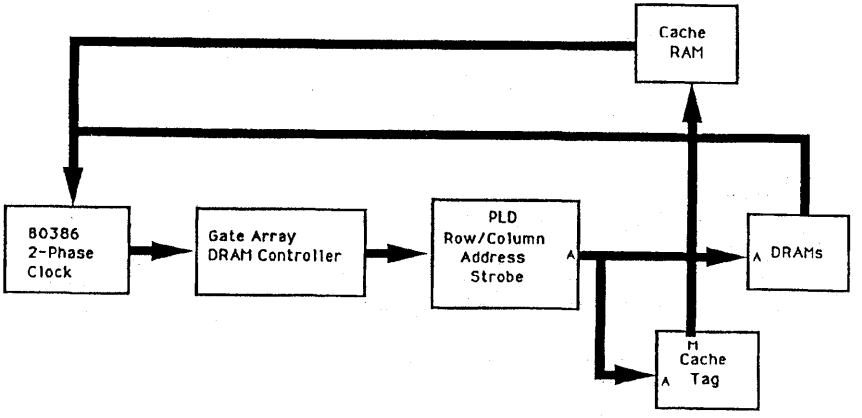


Figure 8. 80386 Cache System Critical Path

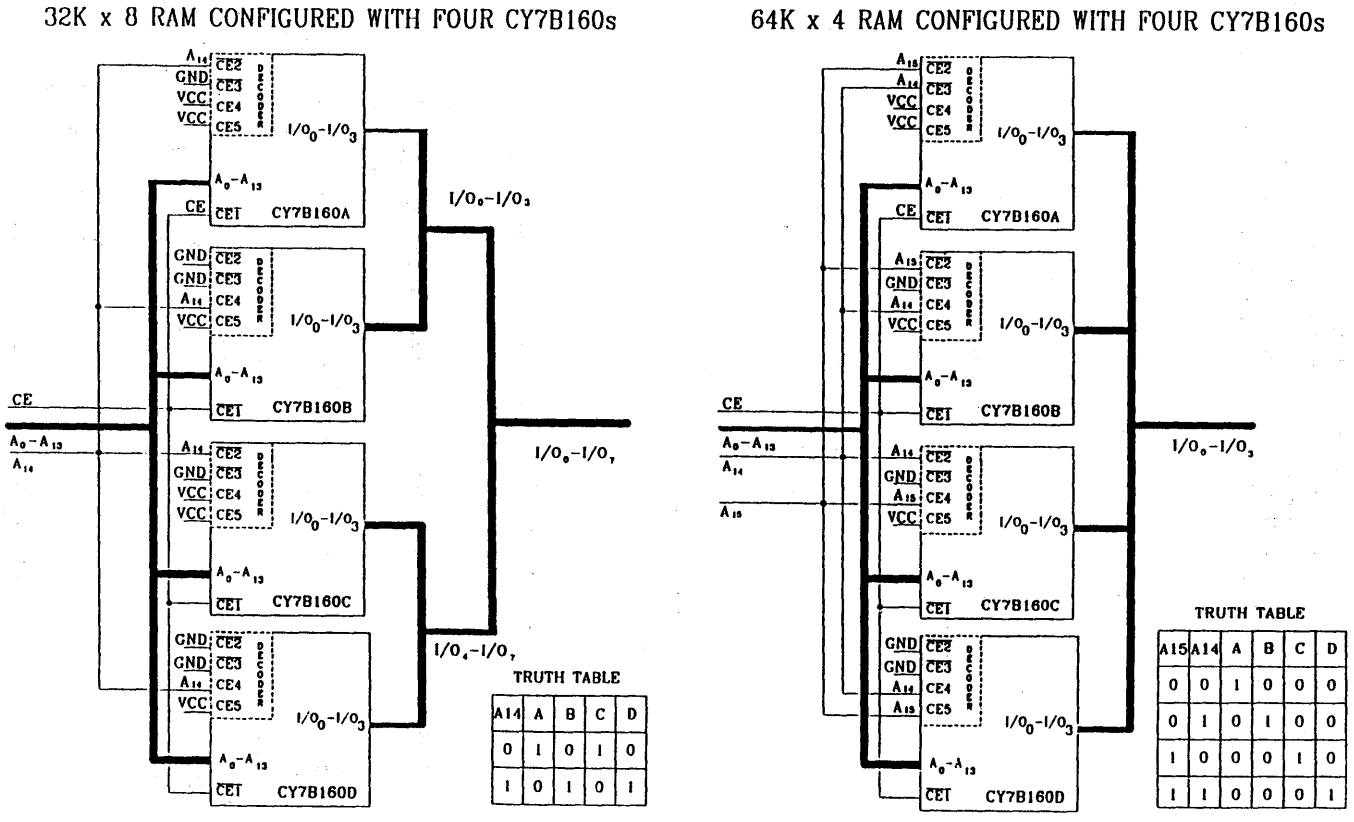


Figure 9. Example Memory Configurations

CYPRESS SEMICONDUCTOR

PLCC and CLCC Packaging for High-Speed Parts

The semiconductor industry is constantly searching for package options that enhance the capabilities of high-performance devices. For fast device performance with minimal ground bounce, electrical characteristics must include low inductance and capacitance from external pin to die bond-wire pad. A package should also furnish good thermal characteristics for reliability over extended temperature ranges.

Other major properties sought after are low cost, as well as standardized outline/pin configurations for compatibility, ease of manufacturing, and handling throughput. The package must also work with surface mount technology and have a small footprint to save board space.

The package that best meets all these requirements is the PLCC (plastic leaded chip carrier). In the past, utilization of PLCCs was not practical for high-power, bipolar devices. However, the advent of low-power bipolar and BiCMOS ECL-compatible SRAMs and PLDs now provides the opportunity for high-volume usage. As manufacturers switch from bipolar to BiCMOS, the lower power dissipation of high-density ECL SRAMs and complex PLDs promise to give PLCC packages a bright future. For military applications and extended temperature environments or for devices with higher power dissipation, you can substitute the CLCC (ceramic leaded chip carrier).

The PLCC has many desirable qualities:

- Suitable for surface mounting with J-type leads
- Small footprint to save board space
- Low inductance and capacitance for high speed with little ground-bounce
- Good thermal characteristics for reliability over temperature range
- Ease of manufacturing and handling for production throughput
- Low cost compared to CERDIP, flatpack, LCC
- Standard package outline and pin-configuration compatibility

The PLCC's J-type surface-mount leads have the advantage over gull-wing leads, which are susceptible to fatigue. J leads also enhance handling ease in test and burn-in fixtures. The PLCC's 1-pF capacitance compares favorably with the 3 and 6 pF for plastic DIPs and CERDIPs, and inductance is equally impressive: 2 nH versus 6 and 11 nH for plastic DIP and CERDIP. Unlike flatpacks, PLCCs are available in standard tooling. PLCCs come in a variety of pin configurations, from 18 to over 200 pins, versus a maximum of 40 pins for plastic DIPs.

The Ceramic Leaded Chip Carrier

For high-temperature environments and high-power devices, you can make use of the ceramic leaded chip carrier (CLCC, Y package), which can also be surface mounted. The Y package has the same footprint and J leads as the PLCC (*Figure 1*) and works well for the faster PLDs and SRAMs.

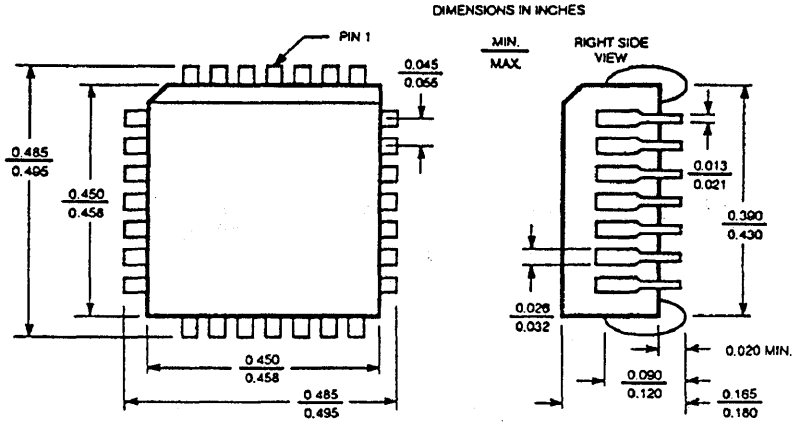
If you do not know system temperature in the early stages of a design, you can substitute the Y package for the PLCC and vice versa, so long as the device's die junction temperature does not exceed 150°C. The Y package is slightly more expensive than the PLCC, but with a thermal resistance from junction to ambient (Θ_{JA}) of 35°C/W at 500 LFPM, the Y package can dissipate heat more efficiently.

Reliability

Cypress's bipolar and BiCMOS products in PLCC and CLCC packages go through extensive burn-in and testing at elevated temperature to guarantee package integrity. Cypress strongly recommends 500-LFPM system forced air flow but guarantees reliability in systems with or without the flow if the ambient air does not cause the junction temperature (T_J) to exceed 150°C.

The PLCC's Θ_{JA} is approximately 45°C/W. The SRAMs have power dissipation that ranges from 780 mW max for the CY100E422L-5 up to 1097 mW max for the CY10E474L-5. This dissipation results in junction temperature rises from 35 to 49°C. The 16P4-type PLD (CY100E302L-6) has a temperature rise of 39°C, and the

28-Lead Plastic Leaded Chip Carrier J64



28-Pin Ceramic Leaded Chip Carrier Y64

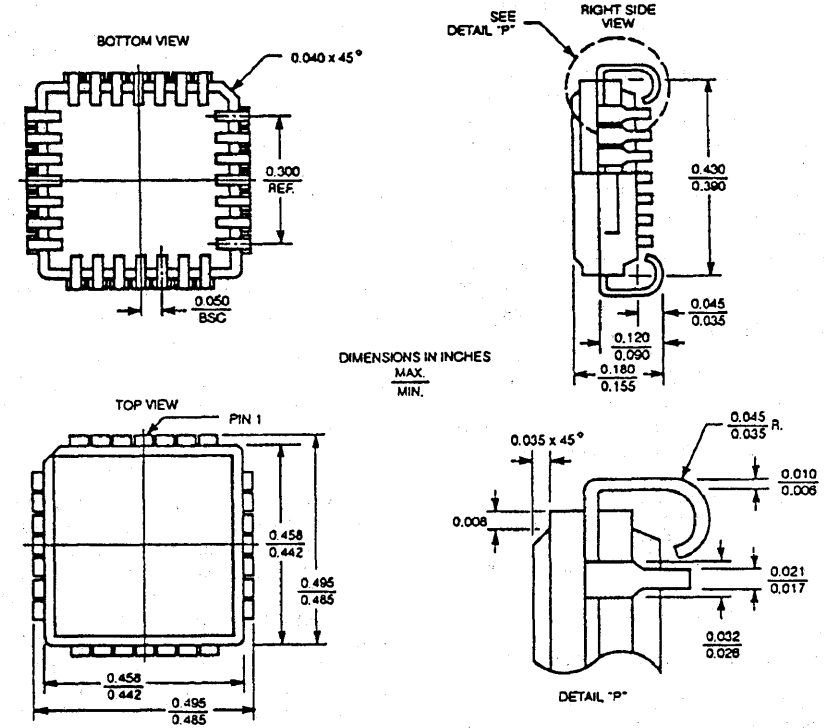


Figure 1. Diagrams of 28-Lead Chip Carriers

16P8-type PLD (CY10E301L-6) has a temperature rise of 47°C. The CLCC package's Θ_{JA} equals 35°C/W for temperature rises of up to 55°C (CY10E474-3).

Finding Chip-Level Junction Temperature

The following relationship determines chip-level junction temperature for the PLCC package:

$T_J = \Delta T + T_A$

where

$\Delta T = P_D \times \Theta_{JA}$

and

$\Theta_{JA} = \Theta_{JC} + \Theta_{CS} + \Theta_{SA}$

To calculate worst case junction temperature (T_J) use maximum supply V_{EE} and I_{EE} for power dissipation and maximum T_A for the temperature range of interest. For the 10K/10KH CY10E301L in a PLCC, for example, device I_{EE} = 170 mA max and V_{EE} = 5.46V max for P_D = 928 mW. Add 15 mW per output for a total output P_D = 120 mW. Therefore, the total P_D = 1048 mW.

For a PLCC, Θ_{JA} = 45°C/W at 500 LFPM, and Θ_{JA} = 64°C/W for still air.

For a CLCC, Θ_{JA} = 35°C/W at 500 LFPM, and Θ_{JA} = 54°C/W for still air.

Because

T_J = total $P_D \times \Theta_{JA} + T_A$

and

T_A = 75°C worst-case commercial temperature range, for the PLCC:

T_J = (1.048 W)(45°C/W) + 75°C = 122°C at 500 LFPM

T_J = (1.048 W)(64°C/W) + 75°C = 142°C in still air

This calculation is for absolute worst-case data sheet conditions. The burn-in temperature used by Cypress (T_J) is much higher than the device will ever see in a system. Note that *most systems will not run at worst case due to guard-banding*. For this reason, use V_{EENOM} = 5.2V or 4.5V and I_{EENOM} = (I_{EEMAX})(85%) for nominal-condition calculations.

Real-World Values

Obviously, most systems do not operate at the worst-case conditions. Therefore, *Figures 2* through *5* show graphs over different operating conditions to determine failures in time (FITs) and mean time between failure (MTBF) for a typical system or in a worst-case scenario.

The graphs are based on a linear method of interpreting the failures observed at burn-in and indicate the long-term reliability of Cypress devices. You can use the graphs to determine MTBF and FITs for any Cypress device in any package after calculating the appropriate ΔT.

The X-axis on the graphs indicates junction temperature. These values are determined by adding the ΔT to ambient temperature, as described earlier. As an example, *Figures 2* and *3* note the following critical points for a CY10E301L ECL PLD under three different operating conditions:

- Point A — 10K/10KH typical data sheet conditions: 25°C ambient, nominal V_{EE} and I_{EE}, 50Ω loads, 500 LFPM air flow, T_J = 64°C, FITs = 7, MTBF = 18,000 yrs.
- Point B — 10K/10KH typical operating conditions: 55°C ambient, nominal V_{EE} and I_{EE}, 50Ω loads, 500 LFPM air flow, T_J = 94°C, FITs = 45, MTBF = 2800 yrs.
- Point C — 10K/KH absolute worst-case conditions: 75°C ambient, 5.46 V max and 170 mA max, 50Ω loads, 500 LFPM air flow, T_J = 122°C, FITs = 225, MTBF = 525 yrs.

The activation energy used for the MTBF and FITs information is 0.7 eV. This is an average number for die-surface-related defects, such as metal and oxide pinholes, etc., but is very conservative for silicon defects or mechanical interfaces to packages. The number is usually 1.0 eV. A small change here results in a significant change in MTBF or FITs. A change to 0.8 eV equates to a 33% reduction in FITs rate or a 50% increase in MTBF.

The Packages of Choice

The PLCC and CLCC are accepted as the packages of choice by many manufacturers of high-speed devices. Motorola Semiconductor uses the PLCC as the only package for the company's very high speed ECL$_{INPS}$ ECL logic family, which stands for "ECL in picoseconds" and is pronounced "eclipse." This family has set-up times and propagation delays in the sub-nanosecond range, with power dissipation of over 1W. Fully compatible with Cypress SRAMs and PLDs, the ECL$_{INPS}$ family includes many 10K, 10KH, and 100K standard logic gates, building blocks, and transceivers.

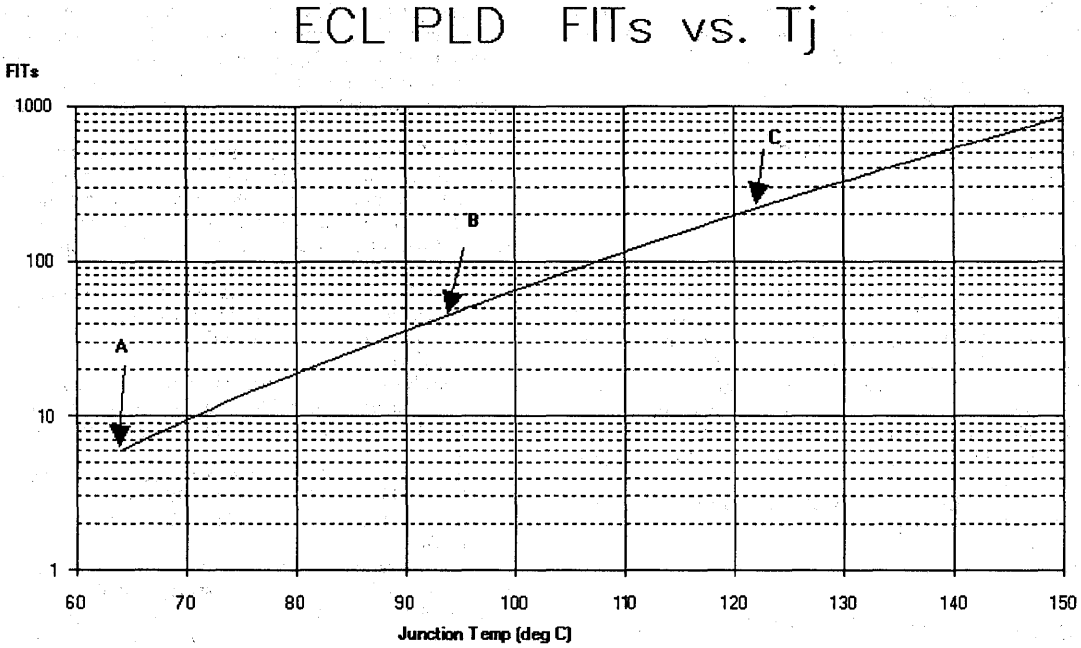


Figure 2. Failures in Time vs Junction Temperature

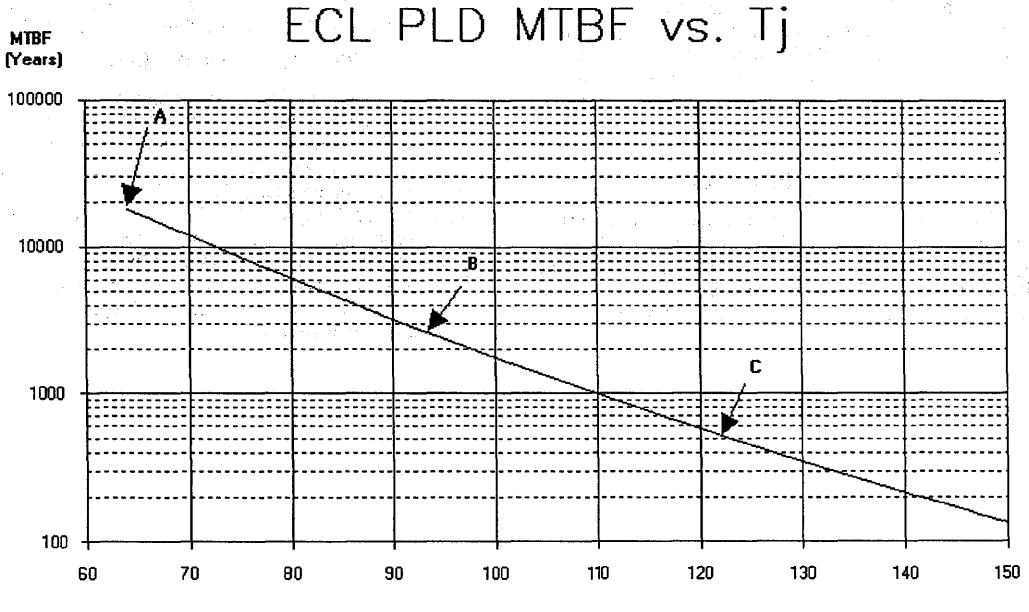


Figure 3. Mean Time Between Failures vs Junction Temp.

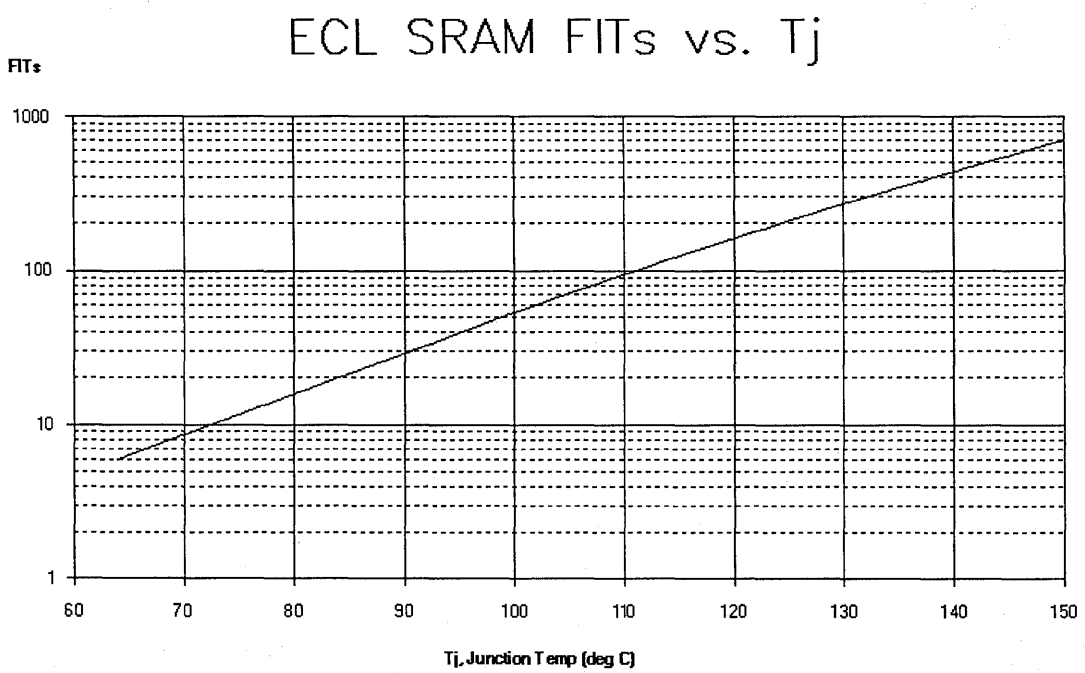


Figure 4. Failures in Time vs Junction Temperature

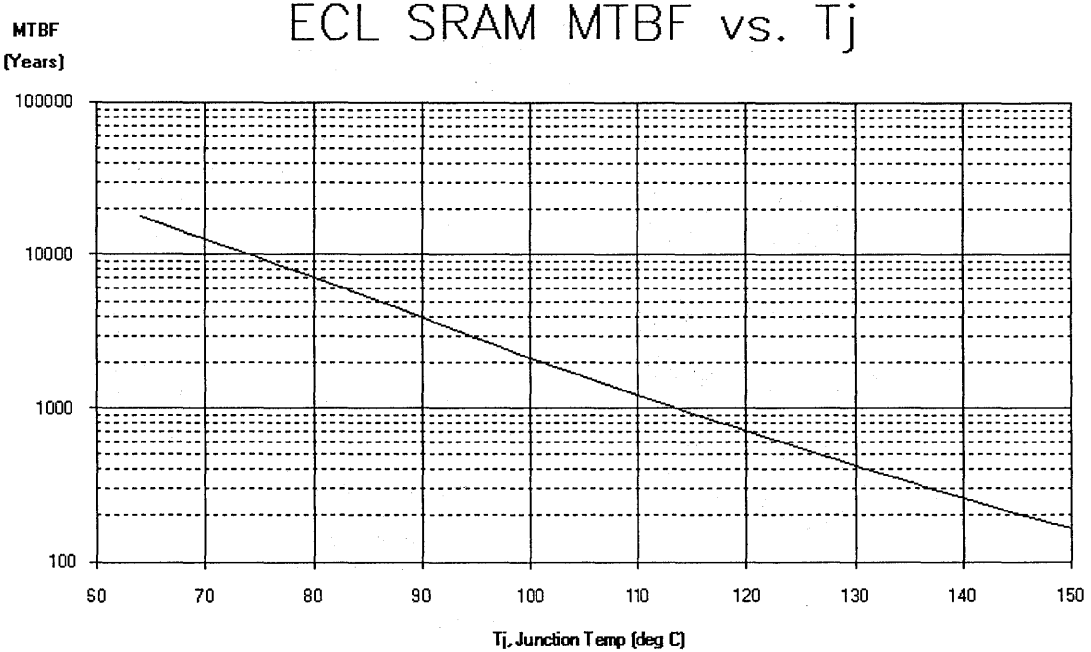


Figure 5. Mean Time Between Failure vs Junction Temp.

CYPRESS SEMICONDUCTOR

A New Generation of BiCMOS High-Speed TTL SRAMs

This application note profiles the Cypress CY7B166 family of TTL-I/O 64K SRAMs, which are ushering-in a new era of high-performance memory devices. These are the world's fastest BiCMOS RAMs, with address access times as low as 8 ns. Arranged in 16Kx4 and 8Kx8 architectures, the devices are functionally equivalent to their industry-standard, TTL-compatible, CMOS counterparts; there is no difference in I/O logic-level min/max specifications. In addition, on-chip features provide superior ground-bounce characteristics and faster propagation delays than is possible with rail-to-rail output swings.

BiCMOS Technology

BiCMOS technology employs CMOS inputs for compatibility with existing products and bipolar on-chip bus interconnects and sense amplifiers to speed the internal access timing. The resulting throughput improvement allows more time for the outputs to slew load capacitance.

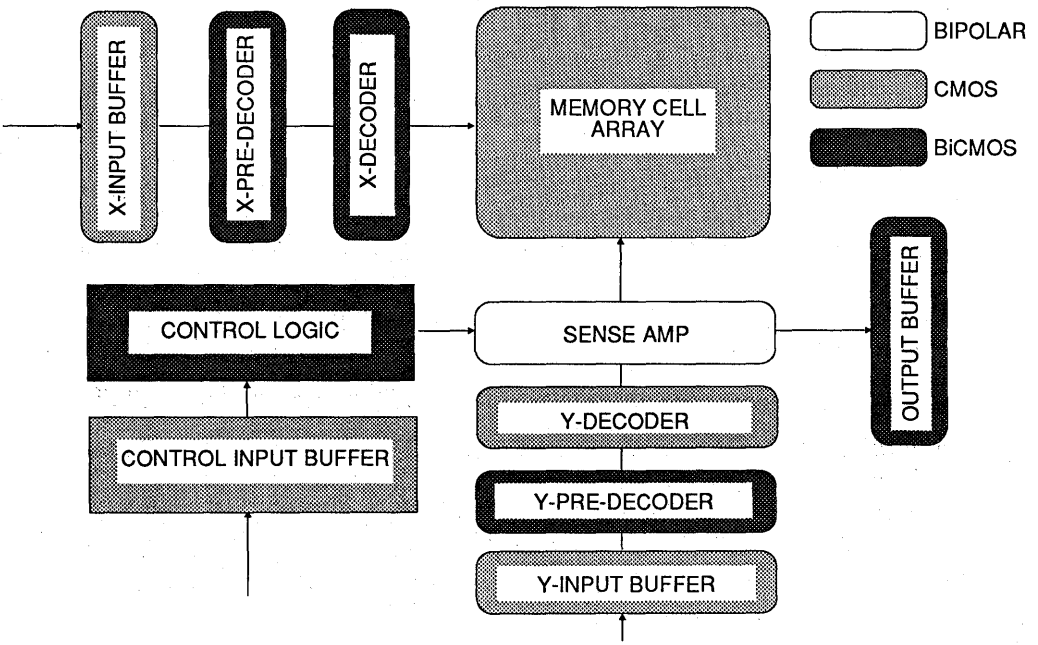


Figure 1. 64K TTL SRAM Circuit Technology

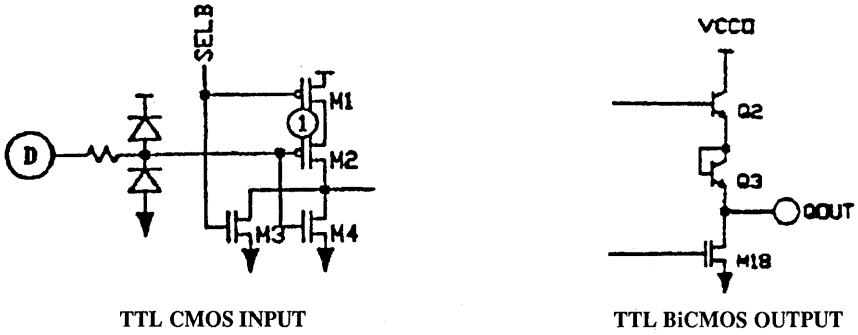


Figure 2. CY7C166C BiCMOS Family I/O Architecture

Further, BiCMOS uses both CMOS and bipolar transistors on the outputs to optimize drive capability.

Figure 1 shows how the parts of the memories are partitioned by technology. On the outputs, two bipolar transistors drop two V_{be} levels (approximately 1.6 V) to reduce the High-level output swing. One device is tied base to collector as a diode, and the other is the High-level drive transistor. Both transistors cause the output to conform to standard TTL logic levels (not CMOS rail-to-rail). This output structure appears in *Figure 2*. The diode is the bipolar transistor Q3, and Q2 is the High-level drive transistor. M18 is an output Low-level pull-down MOSFET (n-type). Keeping the output from swinging to the power supply rail saves time when changing states, as shown in *Figure 3*.

Figure 2 also shows the SRAM's input structure. The CMOS devices are M2 and M4. The input structure includes bipolar-type input clamping diodes, which act as ESD protection devices and meet MIL-STD-883C Method 3015 static discharge voltages of 2001V. The inputs adhere to standard CMOS specifications. The outputs include the same diodes and are an improvement over CMOS-type diodes. The diodes also clamp transmission-line reflections in mis-matched board traces.

Compatibility and Improvements

To reduce ground-bounce noise problems associated with full-swing, high-speed CMOS devices—and TTL parts to a lesser degree—the CY7B166 SRAMs include an internal supply-bypass capacitor between the power supply pin and the ground pin. In parallel with this capacitor, an inductor of equal value to package lead inductance cuts in half the overall inductance associated with output-swing ground bounce. Both the capacitor and inductor decreases the magnitude of the bounce on the output-logic swing's falling edge.

In conclusion, to illustrate BiCMOS compatibility and improvements, *Figure 4* shows I/O waveforms for BiCMOS and CMOS devices. These waveforms show that no compatibility problems arise when substituting BiCMOS-type TTL devices for CMOS parts in a new or existing TTL-I/O system. On the contrary, upgrading from a CMOS 64K TTL-I/O SRAM to Cypress' BiCMOS device family increases speed and noise immunity and decreases noise generation, for an overall system improvement.

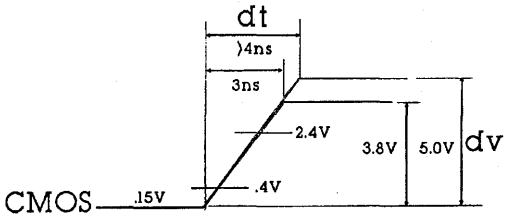
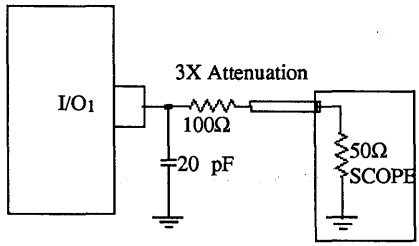


Figure 3. Speed Increase from Reduced Logic Swing



TEST SETUP

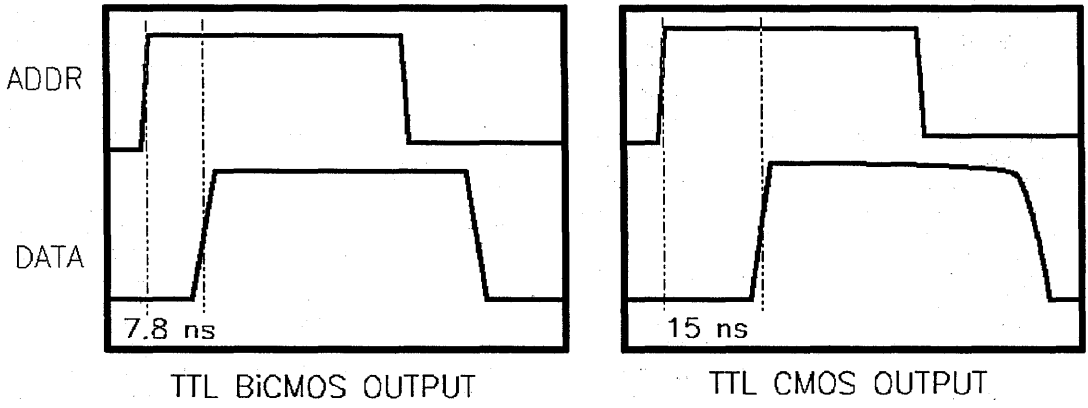


Figure 4. CY7B166 BiCMOS Output vs 64K TTL CMOS Output

CYPRESS SEMICONDUCTOR

Access Time vs Load Capacitance for High-Speed BiCMOS TTL SRAMs

This application note provides a technique for analyzing a system's load capacitance and shows how to determine access time (T_{aa}) degradation. You can also determine other output-related specifications such as t_{DOE} using this method.

The BiCMOS process has made available a new generation of 8-, 10-, and 12-ns TTL-I/O-compatible SRAMs. In the past, the most significant speed barrier in SRAMs was the propagation delay through the device. This delay is now becoming quite small. Consequently, the time the device takes to slew the output load capacitance is a substantial portion of overall delay and must be understood to determine optimum system timing. The techniques presented here can thus help you maximize your system's throughput.

Although many TTL and CMOS components are specified for 30- to 50-pF drive requirements, the actual characteristics of modern high-speed systems are quite different. In a system environment using good transmission lines and termination techniques, the drive requirement depends on the characteristics and length of the transmission lines, the number of succeeding device packages, and where devices are physically distributed along the line. For testing purposes, however, you can approximate the effective capacitance seen at the output of high-speed SRAMs as a lumped capacitance connected directly to the output. This lumped value is from 10 to 30 pF in most board-level systems.

The graph in *Figure 1* represents the additional access time requirements for various lumped-output-

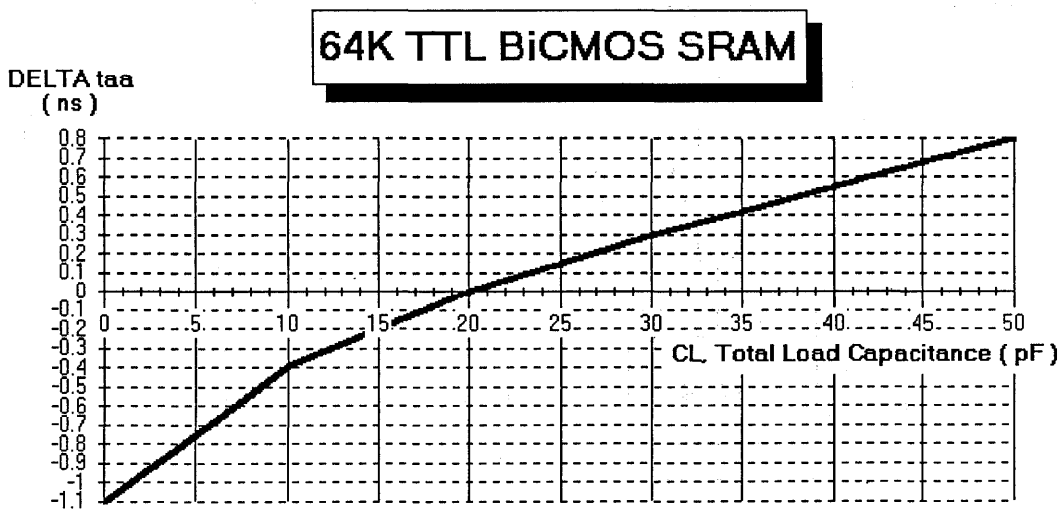


Figure 1. Normalized DELTA T_{aa} vs Load Capacitance

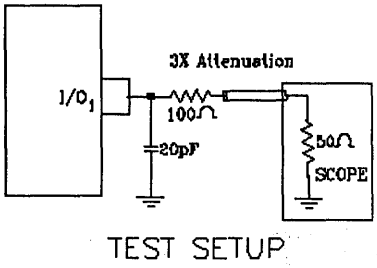


Figure 2. Test Load to Determine T_{aa} vs CL

capacitance values. This graph applies to the CY7B160, CY7B161, CY7B162, CY7B164, CY7B166, CY7B185, and CY7B186. Data is shown for the falling edge only because this edge is effected most by load capacitance. The graph is normalized to 20 pF and can be used for all speed grades. For the -8 devices with no capacitive load, for example, the T_{aa} is 6.9; at 20 pF it is 8 ns; and at 50 pF it is 8.8 ns.

The setup used to get the values in *Figure 1* approximates 50Ω terminated transmission lines and is shown in *Figure 2*. To avoid loading the output, a 100Ω resistor is put in series with the termination resistor. This adds a 3X attenuation factor but does not alter the results.

Using the following measurement techniques and a reasonable number of device loads, you can derive any system's characteristic capacitance. This allows load adjustment to optimize time degradation to keep address access to a minimum. You can also use this technique to determine other specifications that depend on output rise and fall time, such as t_{DOE}.

Measuring Load Capacitance

Now that the capacitance's effect on the device speed is known, the 20-pF approximation can be used to determine T_{aa}. This requires a method for measuring the system load capacitance. A simple method is to use time-domain reflectometry (TDR), which determines capacitance on a transmission line by measuring the pulse reflection the capacitance causes.

The TDR test system (*Figure 3*) consists of a fast pulse generator and oscilloscope with 50Ω terminated inputs. The oscilloscope's channel A measures the reflected voltages, and channel B measures the setup of rise time, logic swing, and pulse width. A single device or a critical

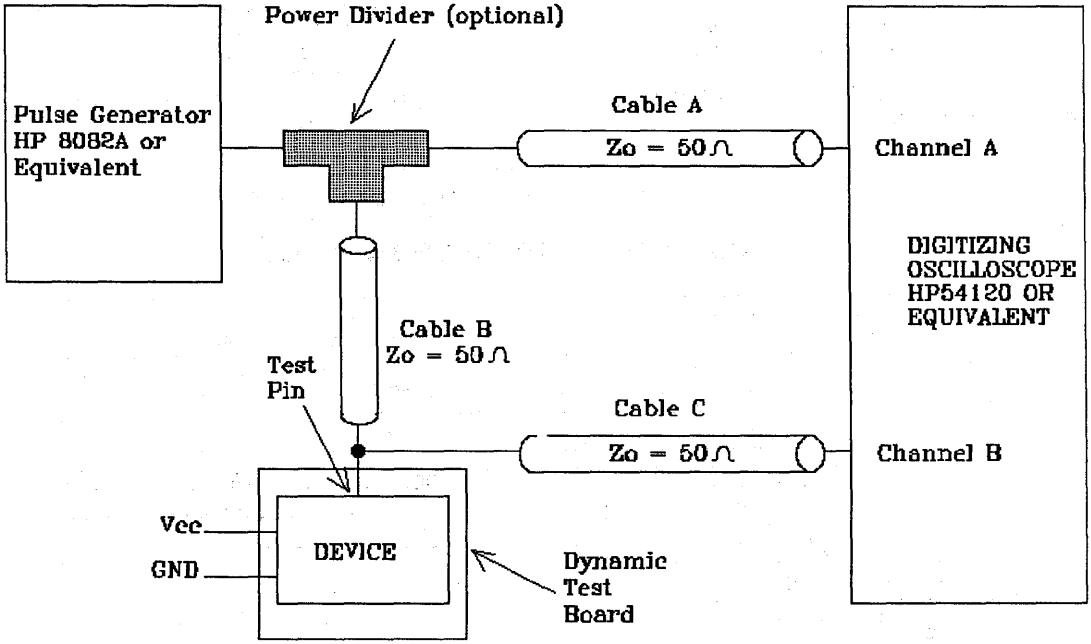


Figure 3. Test Setup for TDR Capacitance Measurement

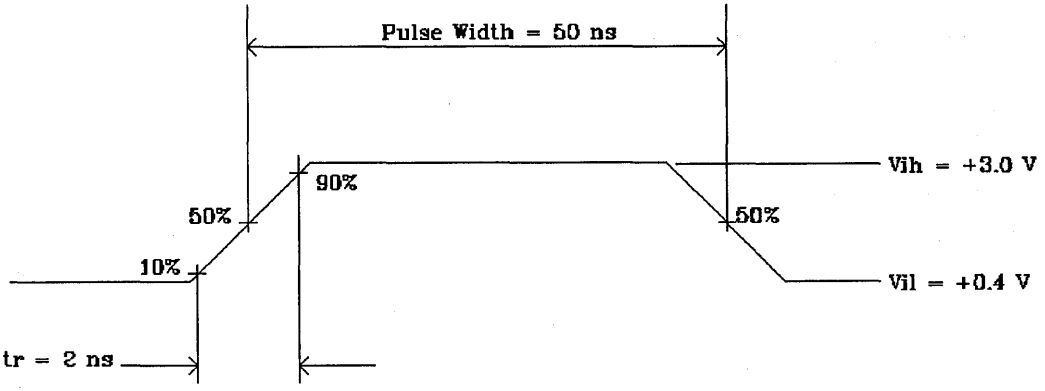


Figure 4. Setup Pulse on Channel B (nothing in the path)

path with various loads can be measured to determine dynamic capacitive loading.

Note that although the length of cables A and C is not critical to the measurement, the time it takes the pulse to traverse cable B must be much greater than the pulse's maximum rise time. This ensures that reflections are measured after the pulse has stabilized and not during a transition. Also note that the test point is any input or output to a PCB transmission line or device and that outputs must be forced to a Low state to be measured.

Figure 4 shows the setup pulse on channel B with no device or board in the path. The setup waveform corresponds to the SRAM's output characteristics. *Figure 5* shows an example reflection, indicating the ΔV reflected voltage measurement and position on cable B.

You can determine capacitance values from the test data using the following equation:

$$C_D = \frac{4(t_r)(\Delta V)}{Z_o V_1} \qquad \textit{Eq. 1}$$

where ΔV = Maximum reflected voltage at channel A, t_r = Rise time of incident pulse, Z_o = 50Ω, V_1 = Logic swing of incident pulse

The equation includes the 2X attenuation factor introduced by the test circuit.

Measuring Capacitance Values Exactly

The line capacitance along with the load capacitance found using Equation 1 determine the total capacitance and time delay added to the access time. Two ways to determine the additional delay are to calculate the extra time and add the result to the no-load access time or calculate the load capacitance and use the graph in *Figure 1*.

These approximations for total capacitance are adequate in most situations, but you can also measure actual line and load capacitance using a high-frequency LCR meter. Usually this equipment is unavailable and/or expensive due to the frequency range needed to get an accurate measurement.

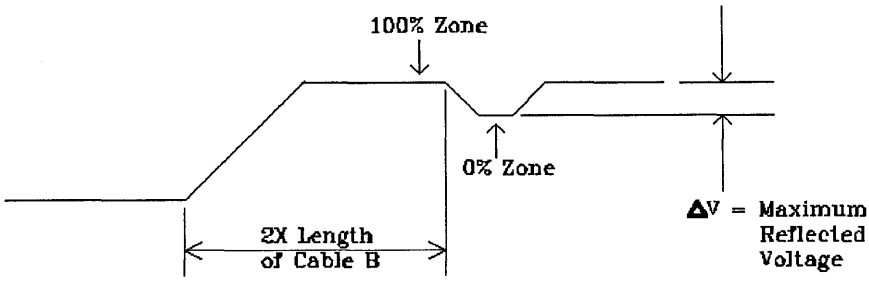


Figure 5. DELTA V Reflected Voltage Measurement

Another approach is to determine the transmission line's capacitance per foot by analyzing the line's characteristics based on the type of line and board construction. A typical 50Ω microstrip line has approximately 35 pF/ft. (3 pF/in.) based on the equation:

$$C_0 = \frac{1.017 \times 10^{-9}(0.45e_r + 0.67)^{(.5)}}{Z_0} \text{ pF/ft.} \qquad \textit{Eq. 2}$$

where Z_0 = 50Ω and e_r = 3 for G-10 fiberglass-epoxy PCBs

The distributed load and line capacitance interact for an overall transmission-line propagation delay equivalent to:

$$t_{pd} = 1.017(e_r)^{(0.5)}\left[1 + \frac{C_D}{C_0}\right]^{(0.5)} \text{ ns/ft.} \qquad \textit{Eq. 3}$$

where C_D = Distributed capacitance

This line length and load-dependent delay can be added to the no-load (0 pF) access time from *Figure 1* to derive system timing. For example, for a 3-in. microstrip transmission line (C_0 = 35 pF/ft.) with a 12-ns device driving one load (5 pF), the total delay is:

t_{aa} @20 pF − 1.1 ns = 12 ns − 1.1ns
= 10.9ns
= No–load access time

$$t_{pd} = 1.017(3)^{(0.5)}\left[1 + \frac{5}{35}\right]^{(0.5)} \text{ ns/ft.}$$

$$= 1.88 \text{ ns/ft} \times \frac{3 \text{ in.}}{12 \text{ in.}}$$

$$= 0.47 \text{ ns}$$

$t_{aa\ total}$ = 0.47 ns + 10.9 ns
= 11.4 ns

Another way to determine delay is from the overall load capacitance, including line and distributed load:

$$C_{total} = C_0\left(1 + \frac{C_D}{C_0}\right)^{(0.5)} \qquad \textit{Eq. 4}$$

where C_{total} = Total line capacitance

You can use C_{total} to determine the additional access time from the graph in *Figure 1*. For example, for a 3-in. 50Ω microstrip transmission line (C_0 = 35 pF/ft.) driving one load (C_D = 5 pF/ft.), the total capacitance is:

$$C_{total} = 35 \text{ pF/ft.}\left(1 + \frac{5}{35}\right)^{(0.5)} \times \frac{3 \text{ in.}}{12 \text{ in.}}$$

$$= 9.35 \text{ pF}$$

Using the graph, the access time decreases by 0.41 ns, for an access time of 11.6 ns. If the line is 6 in. long with two loads, the total capacitance is 19.8 pF, for an increased access time of 11.95 ns. Using Equation 3 gives 11.4 ns and 11.9 ns for the two examples.

CYPRESS SEMICONDUCTOR

Combining SRAMs Without an External Decoder

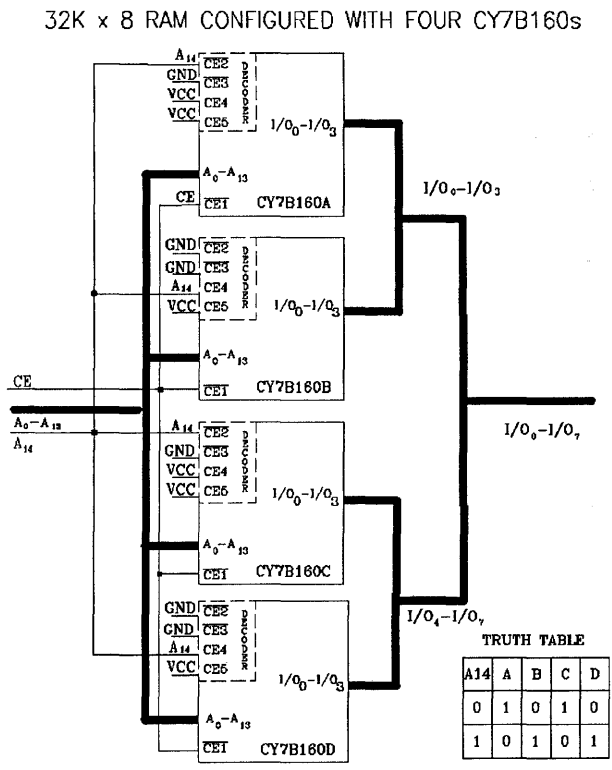


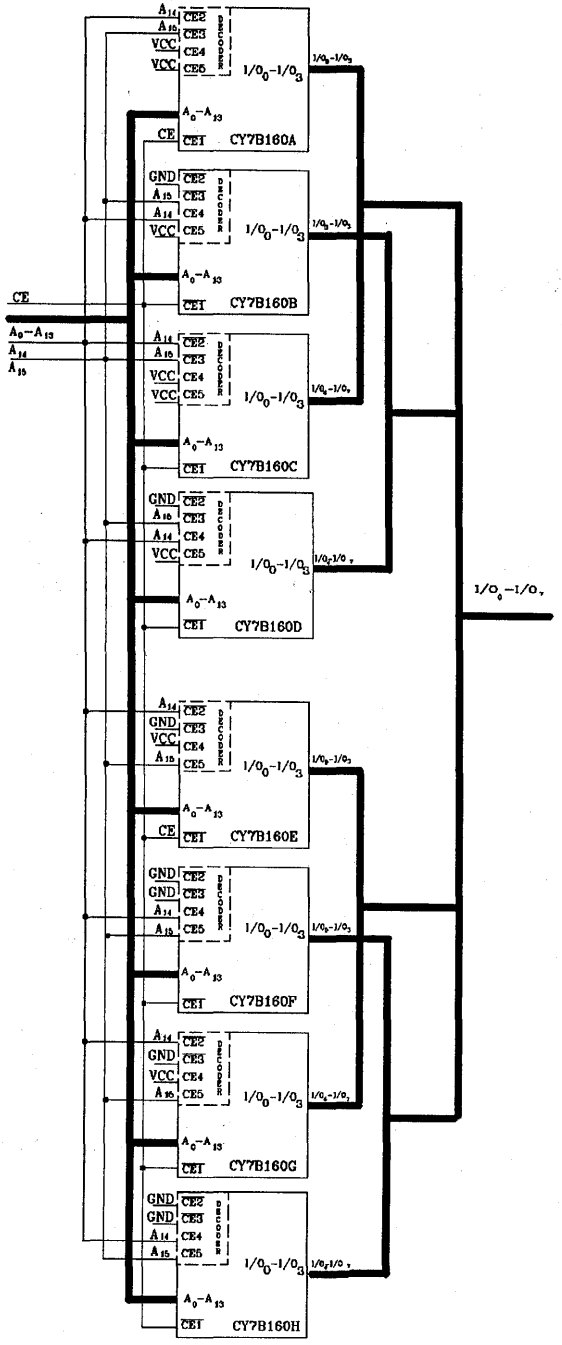
Figure 1. 32 Kbit x 8

An internal decoder with four chip-enable inputs helps designers retain the 8/10-ns access times of the CY7B160 16K x 4 BiCMOS SRAM in multiple-chip memory configurations. Without this capability, denser memory arrays require external logic, which adds 3 or more nanoseconds to the access time. This application note describes how to use the 16K x 4 SRAM to create 64K x 4, 32K x 8, or 64K x 8 memories without an external decoder.

In the x4 configuration, only one Cypress CY7B160 is active at a time. In the x8 configurations, two chips are active at once. Devices that are deselected power-down to less than 40 mA of standby current from a maximum operating current of 120 mA.

Figures 1, 2, and *3* show how two additional address lines, connected to the memories' chip-enable (CE) inputs, permit multiple-SRAM configurations without using an external decoder. You can use a fifth chip-enable input to power-down all devices.

The decoder works without external logic because two of the CE inputs (/CE2 and /CE3) are active Low, and two (CE4 and CE5) are active High. When any CE pin is pulled out of its active state, the chip is deselected. Any CE pin can deselect and power-down the device independently of the other CE pins.

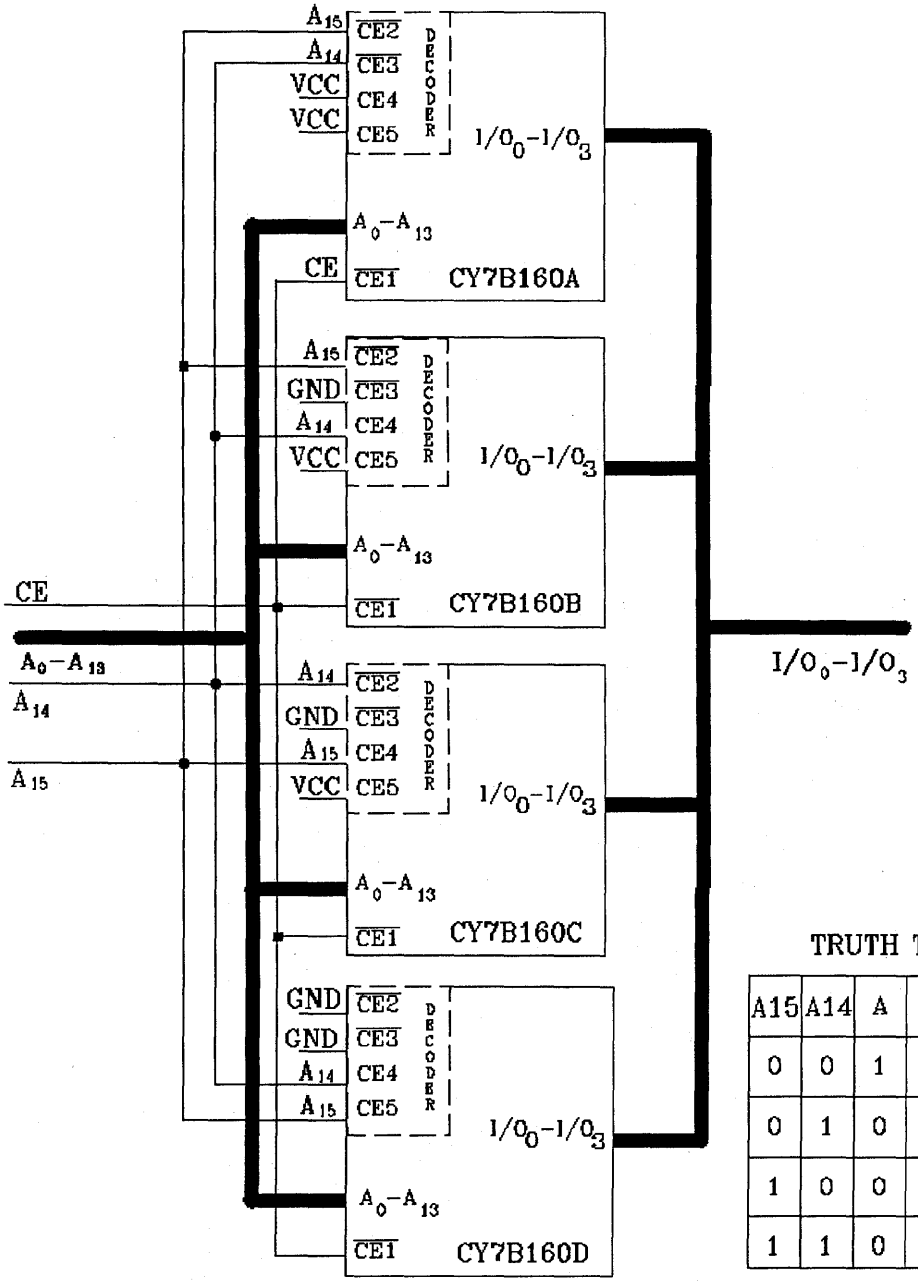


TRUTH TABLE

A16	A14	A	B	C	D	E	F	G	H
0	0	1	0	1	0	0	0	0	0
0	1	0	1	0	1	0	0	0	0
1	0	0	0	0	0	1	0	1	0
1	1	0	0	0	0	0	1	0	1

Figure 2. 64 Kbit x 8

64K x 4 RAM CONFIGURED WITH FOUR CY7B160s



TRUTH TABLE

A15	A14	A	B	C	D
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Figure 3. 64K x 4

CYPRESS SEMICONDUCTOR

BiCMOS TTL SRAMs Improve MIPS R3000 and R3000A Systems

This application note analyzes the speeds required for the cache SRAMs used in RISC systems. The focus here is on the R3000/R3000A RISC processor architecture from MIPS Computer Systems Inc.

One of the goals of RISC-type machines is to execute one instruction per CPU cycle. To achieve this goal, RISC processors employ a compact and unified instruction set, a deep instruction pipeline, and careful adaptation to optimizing compilers. However, these benefits can be rendered useless without an efficient cache memory system composed of fast SRAMs.

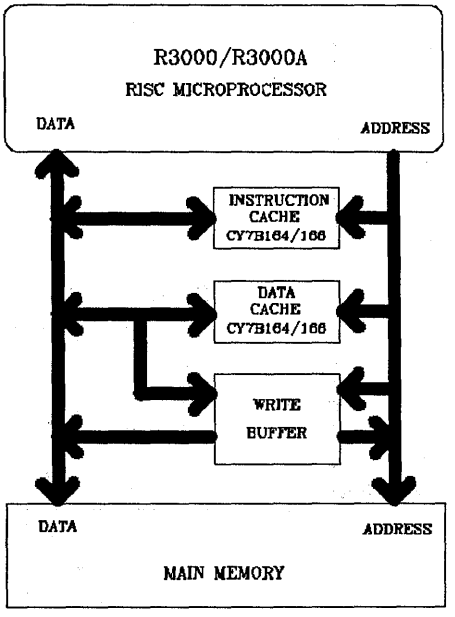


Figure 1. R3000/R3000A System with High-Performance Cache

Design Overview

A block diagram showing the memory components of an R3000A cache system appears in *Figure 1*. The memory system is designed for maximum bandwidth by utilizing separate instruction and data caches and an external write buffer for main memory. The high-speed cache is physically close to the processor and holds instructions and data that are repetitively accessed by the CPU; this reduces the number of times that slow main memory must be utilized.

The R3000 can handle up to 256 Kbytes in 64K entries. The processor provides cache control, which is direct mapped. The processor also provides tag control to verify that the correct data is read from cache. The controller can refill multiple words when a cache miss occurs.

With separate data and instruction caches on the same bus the processor can access or write data and instructions at the CPU's cycle rate. The separate cache architecture for instruction and data memory means that each are alternately accessed during each CPU cycle. This makes cache access time equal to half the cycle-time clock period.

As the processor speed increases from 25 MHz for the R3000 to 33 and 40 MHz for the R3000A, the time allowed for instruction and data fetches from cache memory decreases. The clock period is 30 ns for the 33-MHz system and 25 ns for the 40-MHz system. This leaves 15 ns to access and/or read/write data for the 33-MHz system and only 12.5 ns for the 40-MHz system

To further illustrate the cache timing, a sample read critical path in a 64-Kbyte cache system appears in *Figure 2*. Path 1 is the access time from the R3000 through the 373A latch and into the CY7B166 SRAMs. Path 2 is the time it takes data to be valid after an IRd signal is received from the R3000.

The external latch between the R3000A and the cache address inputs provides part of the pipelining used in the R3000 system and also minimizes loading between the addresses of cascaded memories and the R3000. This

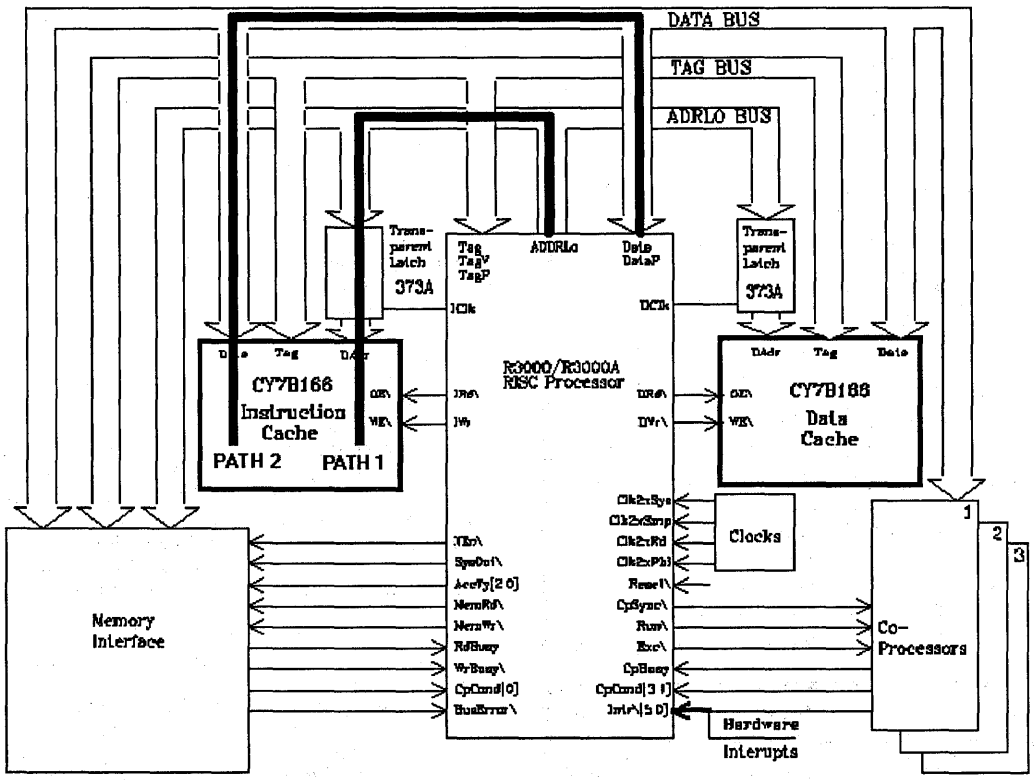


Figure 2. Data and Instruction Cache Critical Paths

extra device causes the memory's access time to become critical, however.

As shown in *Figure 3*, data is fetched from the data SRAM (path 2 for data cache) while the address for the instruction SRAM is set up (path 1 for instr. cache). During the next half cycle, the opposite operation is performed. This arrangement allows use of shared pins on the processor, which save up to 64 I/O lines; however, bus bandwidth requirements are doubled. You must therefore keep signal lines short and loading as low as possible to minimize capacitance.

For a 40-MHz system, critical path 1 in *Figure 2* includes 3.9 ns for a 74PCT373A latch, which leaves only 8.6 ns for the memory, board trace, and address set-up. 'Fortunately, memory access can overlap into the read cycle by 3 ns. Path 2 for a read cycle includes the time it takes the R3000 to send the IRd signal to the CY7B166, the CY7B166 OE-Low-to-data-valid time (t_{DOE}), and the R3000's data set-up time. The set-up time for the 40-MHz R3000A is 4 ns, and the read signal takes 3 ns to generate. This leaves only 5.5 ns for t_{DOE} and for slewing the output load capacitance.

Table 1 lists the time constraints for critical paths 1 and 2 for different system speeds. This data indicates that fast SRAMs are essential to keep up with the 33- and 40-MHz processors. Fortunately, BiCMOS processes now fill this need with 8-, 10-, and 12-ns TTL-I/O-compatible SRAMs with reduced internal propagation delays and improved driving capability.

CY7B166 16K x 4 BiCMOS SRAM

The CY7B166 SRAM is optimized using a BiCMOS process to achieve 12-, 10-, and 8-ns access times. Bipolar and CMOS technology combines to speed-up critical paths and boost output drive (see the block diagram in *Figure 1* of "A New Generation of BiCMOS TTL SRAMs"). CMOS technology reduces memory array size and keeps power to a minimum, while bipolar technology speeds-up critical paths.

BiCMOS technology allows the inputs to be CMOS for compatibility with existing products, while the on-chip bipolar bus interconnects and sense amplifiers speed the internal access timing to allow more time for the outputs to switch. On the outputs, two bipolar transistors drop two

Table 1. Delays Through Two Critical Paths

	PARAMETER	25 MHz	33 MHz	40 MHz
PATH 1	tAV R3000	1.5 ns	1 ns	1 ns
	tpd 373A	5.5 ns	4.2 ns	3.9 ns
	tAA CY7B166	12 ns	10 ns	8 ns
	BOARD DELAYS*	2 ns	2 ns	1.5 ns
	ACCESS CYCLE TIME	20 ns	15 ns	12.5 ns
PATH 2	tIRd\R3000A	5.0 ns	3.75 ns	3.125 ns
	tDOE CY7B166	6 ns	5 ns	4 ns
	tDS R3000/A	6 ns	4.5 ns	4 ns
	BOARD DELAYS*	3.0 ns	1.75ns	1.375 ns
	READ CYCLE TIME	20 ns	15 ns	12.5 ns
CLOCK PERIOD		40 ns	30ns	25ns

Board delays are critical as speed increases. The access time needed by the SRAM can overlap the path cycle time by 3 ns to make up for loss in board delays.

V_{be} levels (approximately 1.6V) to reduce the High-level output swing. One transistor is tied base-to-collector as a diode and the other transistor is the High-level drive transistor. Both transistors cause the output to conform to standard TTL-type logic levels (not CMOS rail to rail). (See *Figure 2* in "A New Generation of BiCMOS TTL SRAMs" for a diagram of this output structure.) The diode is the bipolar transistor Q3, and Q2 is the High-level drive transistor. M18 is an output Low-level pull-down MOSFET (n type). Keeping the output from swinging to the power supply rail saves time when changing states and makes the ramp rate slower (as shown in *Figure 3* of "A New Generation of BiCMOS TTL SRAMs").

The CY7B166's input side includes CMOS devices M2 and M4. Input clamping diodes are also included to provide ESD protection and meet MIL-STD-883C Method 3015 static discharge voltages of 2001V. The inputs meet standard CMOS specifications.

To reduce ground-bounce noise problems associated with full-swing, high-speed CMOS devices — as well as TTL parts to a lesser degree — the CY7B166 incorporates an internal supply-bypass capacitor between the power supply pin and the ground pin. The device also includes an inductor, whose value equals that of the package lead inductance, in parallel with the bypass capacitor to cut the overall inductance associated with output-swing ground bounce in half. Both the capacitor and inductor decrease the magnitude of the bounce on the falling edge of the output logic swing.

Substituting BiCMOS type TTL devices for CMOS parts in a new or existing TTL-I/O system creates no compatibility problems. Upgrading from a CMOS 64K TTL SRAM to Cypress' BiCMOS family of devices increases speed and noise immunity, while decreasing noise generation for overall system improvement.

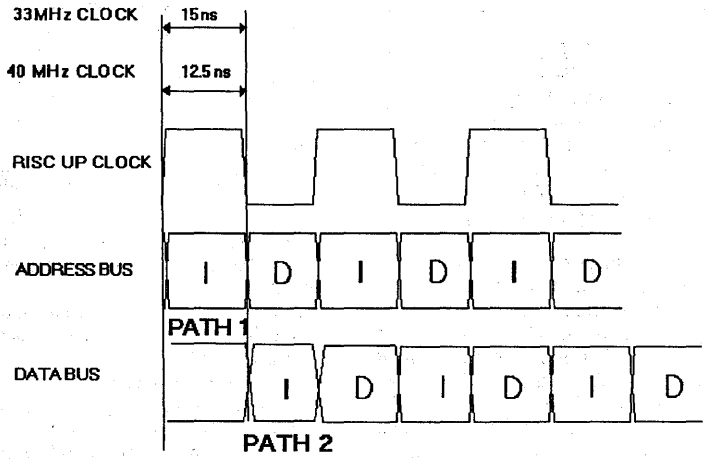


Figure 3. Cache Interleaved Instruction/Data Timing

CYPRESS SEMICONDUCTOR

Memory and Support Logic for Next-Generation ECL Systems

This application note describes the characteristics and use of ECL-I/O technology. Available for many years, this technology is now breaking into mainstream applications due to innovative process technologies. The high power requirements and low device density that once banished ECL to high-speed niche markets are fading with advanced technology and circuit designs. *Table 1* shows how performance and power utilization are evolving.

As system clocks pass 50 MHz, it becomes hard for TTL to provide the necessary low-noise drive capability for fast rise times, and ECL becomes essential. Happily, new BiCMOS SRAMs, gate arrays, and improved bipolar PLDs combine ECL I/O speed with higher density and lower power requirements.

A bipolar ECL implementation of an industry-standard PLD such as the 16P4, for example, draws a modest 220 mA (max), while exhibiting propagation delays of 3 ns (333 MHz). These specifications are for Cypress's CY10E302 and CY100E302 10KH- and 100K-compatible devices. Low-power (170 mA) versions with 4-ns propagation delays are also available.

ECL and BiCMOS

BiCMOS combines bipolar ECL I/O with both bipolar and CMOS internal functions. This helps parts such as Cypress's CY10E474/CY100E474 1K x 4 static RAMs draw only 275 mA, while exhibiting access times of 3.5 ns. Low-power (190 mA) versions exhibit 7-ns access times.

This performance is based on new approaches to combining ECL and CMOS in single devices. Historically, BiCMOS technologies were developed as either CMOS speed enhancers or bipolar power misers. The resulting BiCMOS processes were based either on CMOS or bipolar process flows, and performance for the complementary bipolar or MOSFET components was less than optimal.

In contrast, Cypress's STAR M2 process is a third-generation, 0.8µ BiCMOS technology in which the baseline process is BiCMOS. (See *Figure 1* in "BiCMOS TTL and ECL SRAMs Improve High-Performance Systems" for a simplified cross section of the STAR M2 BiCMOS process.) The STAR process utilizes a modular architecture. That is, polysilicon loads, TiW fuses, or other non-volatile elements are easily incorporated into the baseline process. This results in high-density SRAMs, high-speed PLDs, or high-density EPROMs/PLDs, respectively.

The STAR M2 process is an 18-mask, double-poly, double-metal technology that utilizes a thin epitaxial layer to achieve excellent production performance for NPNs (F_t greater than 10 GHz) and CMOS latch-up immunity. The MOSFETs use lightly doped drains for high performance and reliability.

Unlike first-generation BiCMOS processes, which were limited to SRAMs, STAR's polysilicon bipolar emitter is the same poly used for MOS gates. This enhances NPN performance and decouples the NPN from the poly

Table 1. ECL Families

Parameter	10KH	100K	ECLPS™	Cypress STAR™
Ext. Gate Delay (ps)	500	400	500	500
Flip-Flop (MHz)	250	400	800	800
Gate Power (mW)	25	30	8	3
Speed(X)Power (pJ)	25	12	2.4	0.6

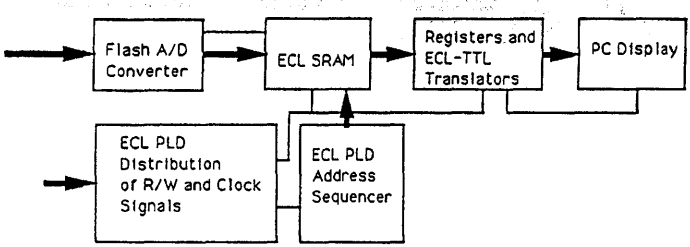


Figure 1. High-Speed A-to-D Application

load module used for 4T SRAM cells. Use of this poly load resistor allows for an 85-square-micron memory cell and small die size.

The advantages of the STAR M2 process over second-generation BiCMOS technologies include higher product performance and greater density and manufacturability.

Applications for ECL and BiCMOS ECL

Applications for ECL PLDs and SRAMs include graphics and image processing, waveform generation, and direct digital synthesis (DDS). In the case of video, ECL memory stores images. In waveform generation and DDS, ECL memory stores digital representations of analog waveforms before feeding the information to a digital-to-analog converter (DAC).

In both image and waveform applications, PLDs are used for address generation/decoding, data manipulation, and clocking schemes/timing control. These functions previously had to be either built discretely with ECL gates or added onto the DAC or memory on the same die. However, high-speed video DACs (greater than 125 MHz) use bipolar process technology, which does not lend itself to high density due to power dissipation problems. It is easier to implement the functions in ECL PLDs and BiCMOS SRAMs.

For analog-to-digital conversion, ECL PLDs work with high-speed flash A/D converters (ADCs) that have ECL outputs. These converters' clock rates range from 20 MHz up to 1 GHz. Applications include HDTV, phased-array radar, digital oscilloscopes, and single-event digitization. Here, PLDs help create high-speed specialty memories such as self-timed SRAM, pipelined SRAM, and interleaved SRAM.

Using the design shown in *Figure 1*, you can implement a fast digital oscilloscope to display analog waveforms on a PC. The flash ADC contains a string of comparators that split the signal into a digital "thermometer" code. From there the digital codes are usually decoded into 8 bits, which are latched on the outputs every clock period. The flash A/D converter feeds BiCMOS SRAMs, which can be interleaved for maximum speed. The PLDs are programmed as address decoders and counters to change the ECL SRAM's address location every clock period. Similar to the way a cache memory works, the memory stores the digital information from the ADC at top speeds. After the memory is full, you can load the data at a slower rate to a PC or digital oscilloscope for manipulation and/or measurements in software.

Instead of using the ECL PLD to implement the SRAMs' address sequencer, it might once have been necessary to incorporate the sequencer as part of the memory chip or use discrete logic. Neither approach was satisfactory, in the one case because of power dissipation and in the other because of the speed limitations imposed by multiple levels of discrete logic.

Further applications for ECL PLDs and SRAMs are found in high-performance workstations, file servers, and high-end embedded controllers. In fact, the next generation of high-end workstations will require ECL support logic. *Figure 2* shows an example based on Bipolar Integrated Technology's 10K-ECL version of Sun Microsystems Inc.'s SPARC processor. In this 80-MHz SPARC implementation, based on Bipolar Integrated Technology's ECL SPARC chip, cache and tag memories use BiCMOS SRAMs and the cache control, memory management unit (MMU), and cache data path (CDP) are implemented with ECL PLDs.

The BIT system is bipolar and consists of the main integer unit (IU), a floating-point coprocessor interface chip, a multiplier and accumulator floating point chip set, and a register file chip. The IU can handle off-chip cache of almost any size with complementary sets of 30Ω cache address drivers to split the cache into two banks. This minimizes trace length, reduces noise, and improves cycle time. The 4K or 64K BiCMOS ECL SRAMs implement the cache memory and reduce system power dissipation. The IU has a 12.5-ns cycle time and provides a Data Ready clock signal that allows a 15-ns cache access time. This access time makes up for trace propagation delays. The design can use SRAMs with access times from 3 to 12 ns, depending on the required cache size and power requirements; these SRAMs can easily keep up with the IU, as can the 3-ns PLDs.

Designing with ECL

Because ECL PLD propagation delays are as short as 3 ns, and output rise/fall times are in the sub-nanosecond region, you must adhere to strict system layout guidelines. ECL speed and noise performance are enhanced with correct transmission-line design and power-supply bypassing techniques. The underlying objectives are to minimize the

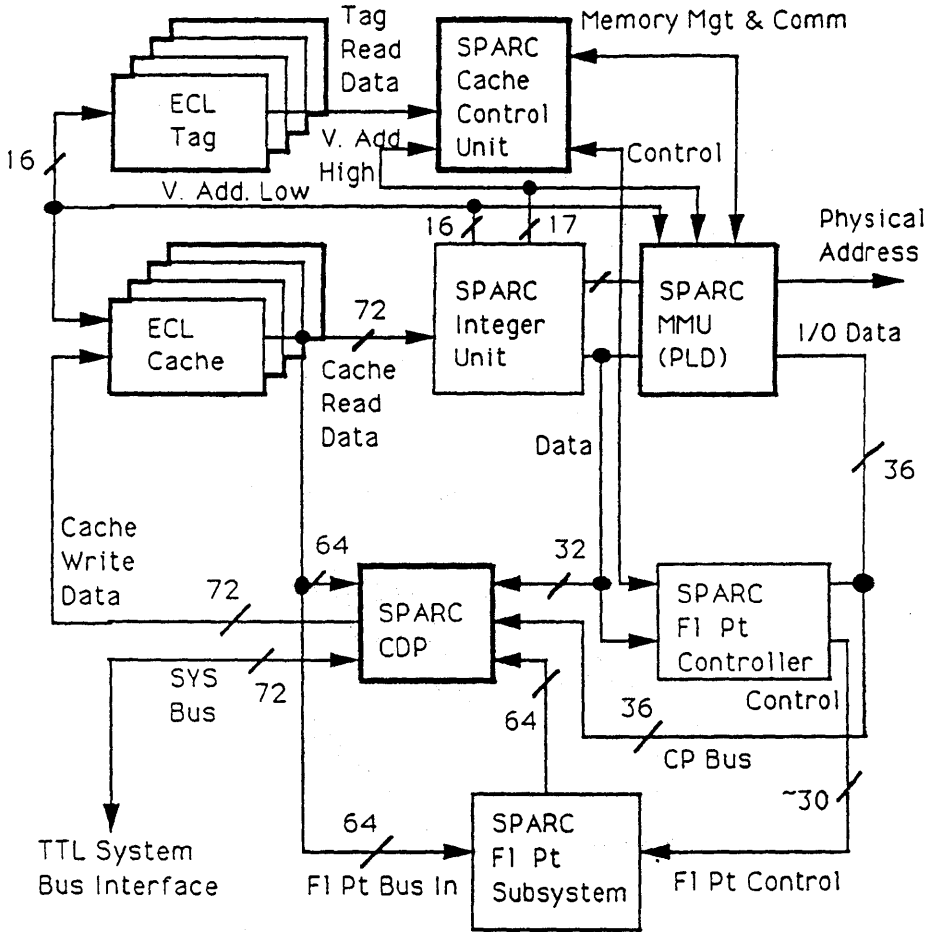


Figure 2. 80-MHz SPARC Implementation

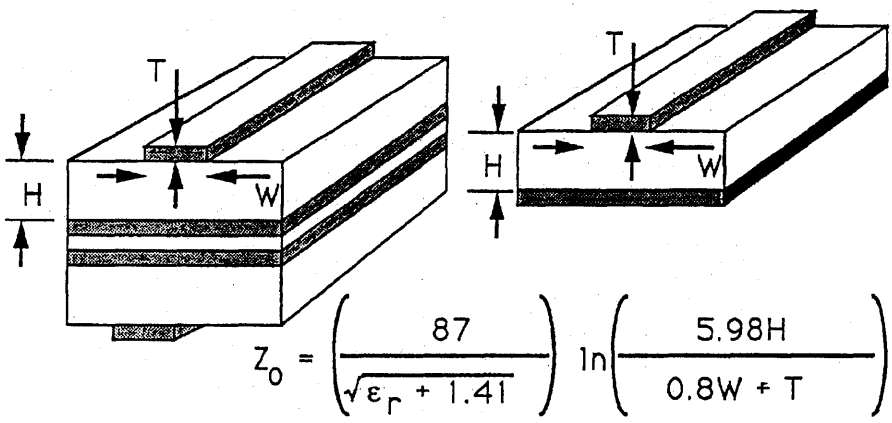
capacitive loading that slows data, prevent ringing and reflections from impedance mismatch, and minimize voltage drops that add system noise and reduce noise margin.

ECL-I/O circuits achieve the best possible match to transmission lines for maximum energy transfer. The output stage consists of a low-impedance, open-emitter transistor that can effectively drive different values of transmission-line Z_o with the addition of a pull-down termination resistor. The pull-down resistor is also necessary for operation of the output transistor and can serve a dual role as the transmission-line termination. ECL input pins are connected to a transistor's high-impedance (DC) base, which appears as a small capacitive load to a properly terminated transmission line.

It is always a good idea to use transmission lines, but they are essential when line propagation delay to the receiving end and back again is greater than or equal to the signal's rise time. Basic calculations for different etched circuit board (ECB) lines appear in *Figure 3*, along with an equation for propagation delay through the transmission line. *Table 2* lists common values for the dielectric constant.

Stripline is used in multi-layered boards and between ground planes; it consists of a trace buried between ground/power planes. The stripline calculations assume that W/(H - T) is less than 0.35 and that T/H is less than 0.25. Single and composite microstripline is used on the top and/or bottom of single- or double-ground boards; it consists of a trace on the surface, with the ground or power plane buried.

Other common high-speed practices are to use equal line lengths from device to device and rounded corners on

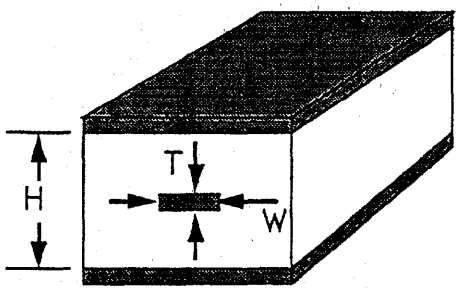


$$Z_0 = \left(\frac{87}{\sqrt{\varepsilon_r + 1.41}}\right) \ln\left(\frac{5.98H}{0.8W + T}\right)$$

$$\delta = 1.016 \bullet \sqrt{\epsilon_e}\ \text{ns/ft}$$

where δ = propagation delay

$$\epsilon_e = 0.475\ \epsilon_r + 0.67$$



$$Z_0 = \left(\frac{60}{\sqrt{\varepsilon_r}}\right) \ln\left(\frac{4H}{0.67\pi(0.8W + T)}\right)$$

Figure 3. Z_0 for Microstrip and Stripline

Table 2. Common Values for Dielectric Constant

Material	ε_r
Duroid	2.56
Quartz	3.78
G-10 (FG Epoxy)	4.7
Alumina	9.7
Silicon	11.7

traces. Component lead lengths should be short, with surface-mount passive and active components used as much as possible.

Terminations

Transmission-line terminations must match the line's characteristic impedance to minimize reflections. The termination is usually also used as the pull-down resistor for the open-emitter ECL outputs. These outputs allow Z_o values from 50 to 150Ω. This means that ECL can accommodate 75Ω video systems as well as 50Ω communications systems. Some ECL outputs even allow 25Ω trans-

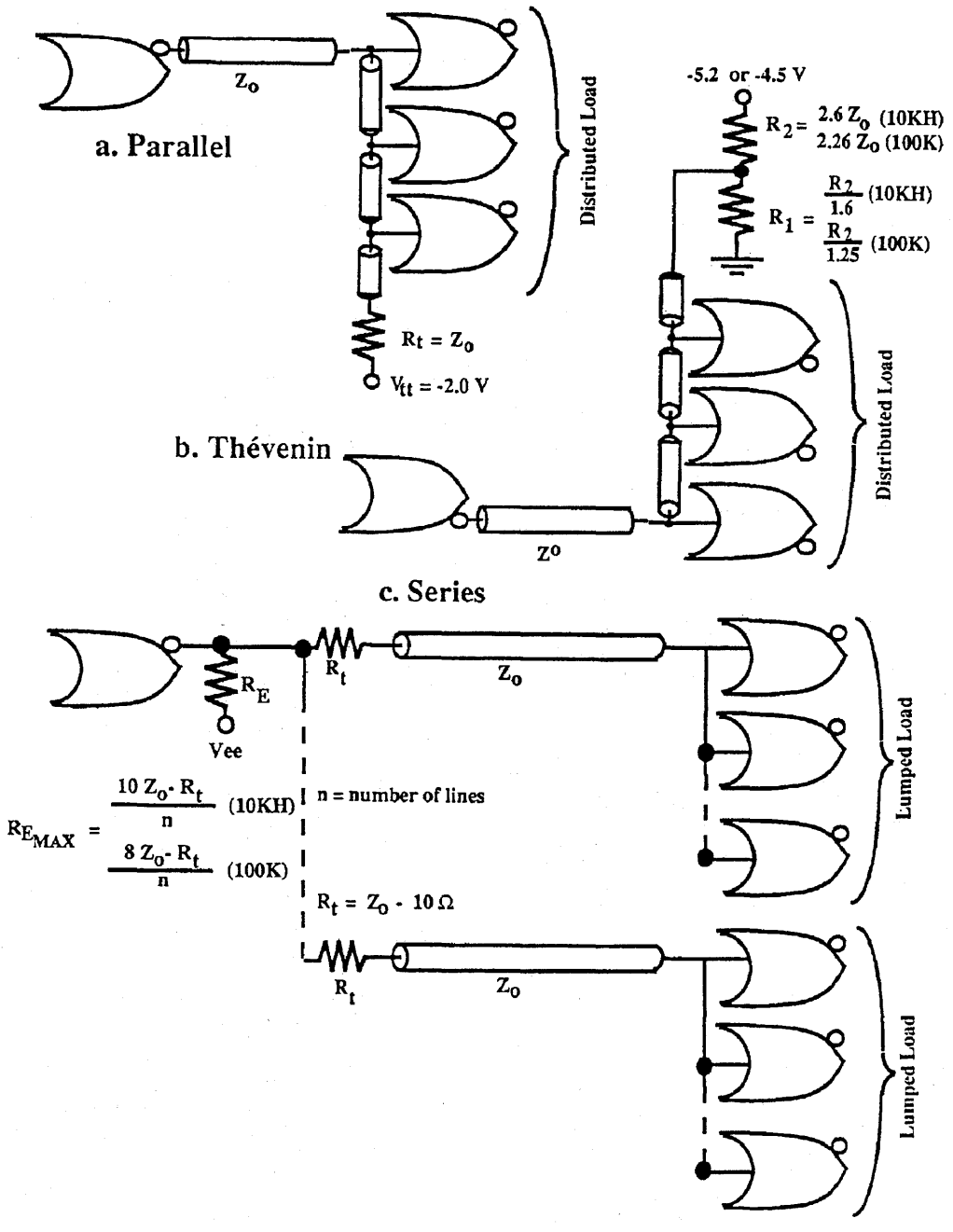


Figure 4. Three Types of Transmission Line Terminations

Table 3. ECL Output Transistor Power and Terminating/Pull-Down Resistor Power

	Dissipation in	
Terminating Resistor Value (Ω)	**ECL Output Transistor (mW)**	**Terminating Resistor (mW)**
Parallel Termination		
150	5.0	4.3
100	7.5	6.5
75	10	8.7
50	15	13
Thevenin Termination		
82/130	15	140
Series Termination		
2K	2.5	7.7
1K	4.9	15.4
680	7.2	22.6
510	9.7	30.2

mission lines to drive doubly terminated 50Ω bus lines in backplane applications.

Figure 4 shows the types of terminations with calculations. The different options have tradeoffs that include routing, power dissipation, loading, and ease of use.

The parallel termination (*Figure 4A*) is simple: The terminating resistor at the far end of the transmission line equals the line's Z_o. In reality, the line and R_t always exhibit some mismatch caused by the ECL device's input capacitance. This termination offers the fastest performance and lowest power dissipation, but requires an additional power supply for the termination resistor (R_t).

An advantage of parallel terminations over series terminations (*Figure 4C*) is that you can use the former with ECL loads distributed along the length of the transmission line. This is because the parallel termination is installed at the transmission line's receiving end and absorbs most all reflections.

The Thevenin equivalent (*Figure 4B*) of the parallel termination (called the Thevenin termination) requires two resistors but needs no separate supply because the termination relies on the system power bus. Although this feature is convenient for small systems, the Thevenin termination draws 11 times more power per termination than does the parallel termination.

The series termination is potentially the most power-efficient. It matches Z_o by means of a resistor (R_t) in series with the driving ECL gate's output impedance, which is 10Ω in STAR devices). Instead of totally preventing any reflections at the far end of the line, the series termination allows pulses to be reflected by the high impedance there, absorbing them when they are reflected back to the near end.

The series termination's efficiency depends on the value of R_E. The power dissipated by a small R_E can exceed the power dissipated in the parallel termination. A large R_E can slow negative-going transitions because the input capacitance of the following gates (typically 4 pF) are being charged through the resistor. A large R_E can also reduce noise margins.

Note that, in this case, R_E does not have to equal the transmission-line impedance. *Table 3* shows a tabulation of ECL output transistor power and R_E power dissipation.

Because the series termination is installed at the near end of the transmission line, only lumped loads can be used. Distributed loads cause problems because the full value of the pulses are seen only at the far end of the line and not along the length of the trace, as with the parallel and Thevenin terminations.

Typically, you can have up to 10 lumped loads at the end of the line. Thus, you must choose R_E to supply enough current to drive the loads. However, you must also consider the voltage drop in the series terminating resistor. One way to minimize dissipation is to make the series termination drive two or more lines with lumped loads in parallel (as in *Figure 4c*).

Measurements

After prototyping transmission lines and terminations, you can make waveform measurements on a sample board to uncover any mismatches. Simple time domain reflectometry (TDR, *Figure 5*) can show the position of discontinuities or mis-matches along the line and the type of reactance or termination needed to correct them. Discontinuities, such as gate input capacitances distributed along the line, appear as small glitches on the output waveform. The reflection's amplitude is proportional to the capacitance. You can therefore calibrate the test setup using a series of standard capacitances. Also, test equipment with TDR capability, which simplifies measurements, is available from HP.

Interfacing and Prototyping

With the increased use of ECL in new and next-generation systems, many connector and cable companies, such as W. L. Gore & Associates, are offering controlled-impedance coax ribbon cable and wrappable coax cable for prototypes and final design.

Although most ECL system prototyping is done on PC boards, alternatives exist. ECL and mixed-TTL/ECL wire-wrapping boards with extensive ground planes are available from MUPAC Corporation. You can use wrappable coax on these boards between signal pins, with additional connections to adjacent ground pins.

Programming ECL PLDs

Cypress's current ECL PLDs are bipolar devices with proven TiW fuses. This means that, unlike the company's erasable CMOS PLDs, the ECL PLDs are one-time fuse-programmable. You can program the devices using Data I/O, Stag, and Logical Devices PLD programmers; you

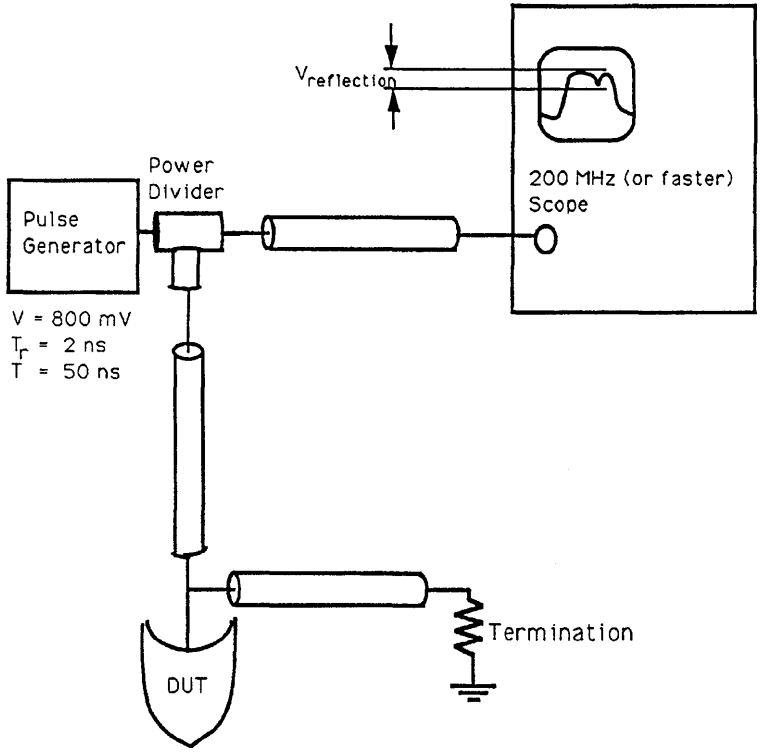


Figure 5. Time Domain Reflectometry Setup

can also use Cypress's QuickPro II. Development software, including simulation models, is available from Data I/O (ABEL) and Logical Devices (CUPL).

Section Contents

CYPRESS SEMICONDUCTOR

RAM I/O Characteristics

This application note describes the function and I/O standards of Cypress high-speed static RAMs. Manufactured using a speed-optimized CMOS technology, these RAMs meet and exceed the performance of competitive bipolar devices, while consuming significantly less power and providing superior reliability. While providing identical function, the RAMs exhibit slightly different input and output characteristics, which permit you to improve overall system performance.

Product Description

The five parts represented in *Figure 1* constitute three basic devices of 64, 1024, and 4096 bits. The CY7C189 and CY7C190 feature inverting and non-inverting outputs, respectively, in a 16 x 4-bit organization. Four address lines address the 16 words, which are accessed via separate input and output lines. Both of these 64-bit devices have separate active-Low select and write-enable signals.

The 256 x 4 CY7C122 is packaged in a 22-pin DIP, and features separate input and output lines, both active-Low and active-High select lines, eight address lines, an active-Low output enable, and an active-Low write enable.

Both the CY7C148 and CY7C149 are organized as 1024 x 4 bits and feature common pins for data input and output. Both parts have 10 address lines, a single active-Low chip select, and an active-Low write enable. The CY7C148 features automatic power-down whenever the device is not selected, while the CY7C149 has a high-speed, 15-ns chip select for applications that do not require power control.

This family of high-speed static RAMs is available with access times of 15 to 45 ns with power in the 300- to 500-mW range. These RAMs are designed from a common core approach and share the same memory cell, input structures, and many other characteristics. The outputs are similar, with the exception of output drive and the common I/O optimization for the CY7C148 and CY7C149. For more detailed information on these products, refer to the Cypress Data Book.

Generic I/O Characteristics

Input and output characteristics fall generally into two categories: when the area of operation falls within the normal limits of V_{CC} and V_{SS} plus or minus approximately 600 mV, and abnormal circumstances, when these limits are exceeded. Under normal operating conditions, inputs switch between logic Zero and logic One. This application note considers operation in a positive-True environment, and therefore a One is more positive than a Zero.

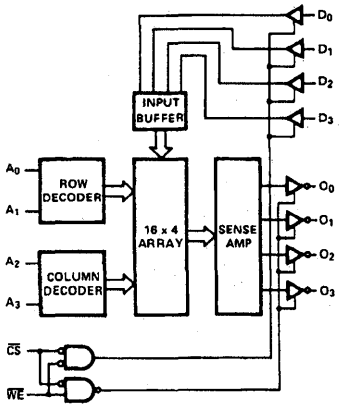
The RAMs provide TTL-compatible I/O. Therefore a One is 2.0V, while a Zero is 0.8V. To be considered a One, the input of a device must be driven greater than 2.0V, but not exceeding V_{CC} + 0.6V. To be considered a Zero, the input must be driven to less than 0.8V, but not less than V_{SS} - 0.6V.

Output characteristics represent a signal that drives the input of the next device in the system. Because the RAM levels are TTL compatible, you can assume that the V_{IL} and V_{IH} values of 0.8V and 1.0V referenced above are valid.

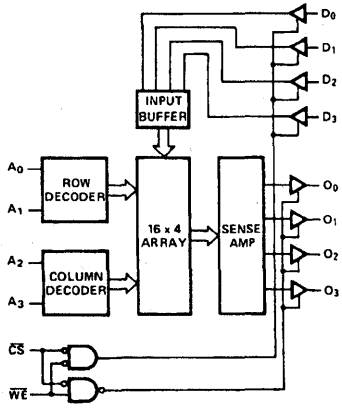
In consideration of noise margin, however, driving the input of the next stage to the required V_{IL} or V_{IH} is not sufficient. Noise margins of 200 to 400 mV are considered more than adequate. Thus, an adequate V_{OH} is 2.4V and V_{OL} is 0.4V, providing a noise margin of 400 mV.

Because the driven node consists of both a resistive and a capacitive component, output characteristics are specified such that the output driver is capable of sinking I_{OL} at the specified V_{OL}, and capable of sourcing I_{OH} at V_{OH}. Because the values of I_{OL} and I_{OH} differ depending on the device, these values are shown in *Table 1*.

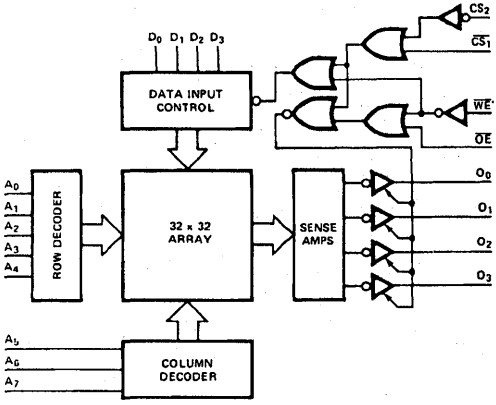
Outputs have one other characteristic to be aware of: output short-circuit current (I_{OS}). This is the maximum current that the output can source when driving a One into



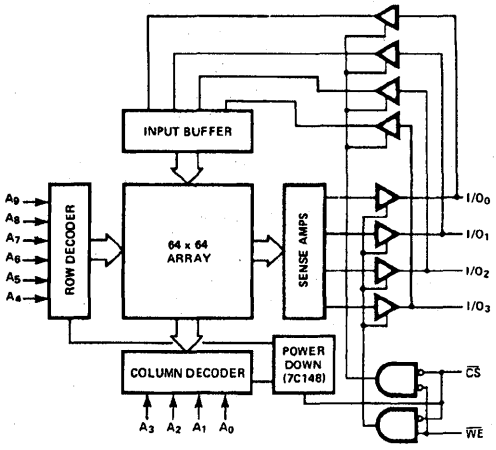
7C189



7C190



7C122



7C148/9

Figure 1. RAM Block Diagrams

V_{SS}. You need to be aware of I_{OS} for two reasons. First, the output should be capable of supplying this current for some reasonable period of time without damage. Second, this is the current that charges the capacitive load when switching the output from a Zero to a One and will control the output rise time.

Because memories such as these are often tied together, you also need to consider the output characteristics when the devices are deselected. All of the RAMs in the family feature three-state outputs; when deselected the outputs are in a high-impedance condition that does not source or sink any current. In this condition, as long as the input is driven in its normal operating mode, a three-state output appears as an open, with less than 10 µA of leakage. Thus, to any other device driving this node, the output does not exist.

Technology Dependencies and Benefits

Some of the products described in this application note were originally produced in a bipolar technology. They have since been re-engineered in NMOS technology, and Cypress has now produced them in a speed-optimized CMOS technology.

Both technology dependencies and benefits associated with each technology relate to the design of input and output structures. When you use these products, you should know about these characteristics and how they can benefit or impede a design effort.

One of the most obvious factors is that both NMOS and CMOS device inputs are high impedance, with less than 10 µA of input leakage. Bipolar devices, however, require that the driver of an input sink current when driv-

Table 1. DC Parameters

Parameters	Description	Test Conditions	CY7C122		CY7C148/9		CY7C189/90		Units
			Min.	Max.	Min.	Max.	Min.	Max.	
V_{OH}	Output High Voltage	V_{CC} = Min., I_{OH} = -5.2 mA	2.4		2.4		2.4		V
V_{OL}	Output Low Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.4		0.4		0.4	V
V_{IH}	Input High Voltage		2.1	V_{CC}	2.0	V_{CC}	2.0	V_{CC}	V
V_{IL}	Input Low Voltage		-3.0	0.8	-3.0	0.8	-3.0	0.8	V
I_{IL}	Input Low Current	V_{CC} = Max., V_{IN} = V_{SS}		10		10		10	µA
I_{IH}	Input High Current	V_{CC} = Max., V_{IN} = V_{CC}		10		10		10	µA
I_{OFF}	Output Current (High Z)	V_{OL} < V_{OUT} < V_{OH}, T_A = Max.	-10	+10	-10	+10	-10	+10	µA
I_{OS}	Output Short-Circuit Current	V_{CC} = Max., 0°C < T_A < 70°C		-70		-90		-275	mA
		V_{OUT} = V_{SS}, -55°C < T_A < 125°C		-80		-90		-350	mA

ing to V_{IL}, but appear as high impedance at V_{IH} levels. This is because the input of a bipolar device is the emitter of a bipolar NPN-type device with its base biased positive. The bias (1.5V) establishes the point at which the input changes from requiring current to be sourced to presenting a high impedance. This switching level is the reason that AC measurements are done at the 1.5V level.

Although NMOS and CMOS device inputs do not change from low to high impedance, great care is taken to balance their switching threshold at 1.5V. This allows you to consider only capacitive loading for MOS device fanout, while bipolar has both a capacitive and DC component.

The other input characteristic that differs between bipolar and MOS is the clamp diode structure, which exists in both MOS and bipolar. However, in MOS devices that use bias generator techniques (all high-speed MOS devices), the diode does not become forward biased until the input goes more negative than the substrate bias generator plus one diode drop. Because the bias generator is usually at about -3V, this factor removes the clamping effect.

CMOS/NMOS/Bipolar Input Characteristics

Although NMOS, CMOS, and bipolar technologies differ widely, the I/O characteristics are the TTL derivatives that have been covered above and are documented in *Table 1*. With the exception of the differences in input impedance between MOS and bipolar devices, all three technologies are used to produce TTL-compatible products.

Another group of devices provide a true CMOS interface, where signals swing from V_{SS} + 1.5V. In addition, loads are primarily capacitive. Only devices produced in a CMOS technology are capable of behaving in this manner. CMOS devices can, however, handle both TTL and CMOS inputs.

Devices such as the ones described in this application note have input characteristics such as those depicted in *Figure 2*. While operated in the TTL range, these devices perform normally. Operated in full CMOS mode, the devices save power because the current consumed in the input converter decreases as the input voltage rises above 3.0V or falls below 1.5V. Because the input signal is in the 1.5V-to-3.0V range only when transitioning between logic states, the power savings in a large array with true CMOS inputs can be significant. With input signals on over half of the pins of a device, significant savings in a large system can be realized by using CMOS input voltage swings even in TTL systems.

Although this application note does not directly deal with the AC characteristics of high-speed RAMs, the input and output characteristics of these devices have a great deal to do with the actual AC specifications. Conventionally, all AC measurements associated with high-speed devices are done at 1.5V and assume a maximum rise and fall time. This eliminates the variations associated with the various usage configurations (as a figure of merit when testing the device), but does not mean that you can ignore these influences when designing a system.

Maximum rise and fall time is usually included on every data sheet. For the products referred to in this application note, a 10-ns maximum rise and fall time is specified for all devices with access times equal to or

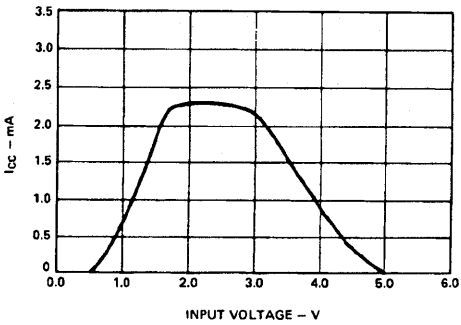


Figure 2. Input Voltage vs Current

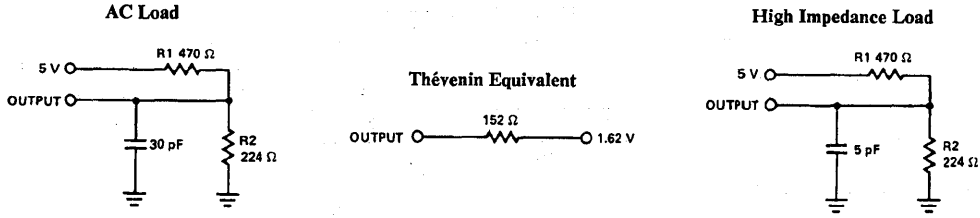


Figure 3. Test Loads

greater than 25 ns. All devices with access times less than 25 ns have a 5-ns maximum rise and fall time.

The AC load and its Thevenin equivalent in *Figure 3* represent the resistive and capacitive load components that the devices are specified to drive. With either of these loads, the device must source or sink its rated output current or its specified output voltage. The capacitance stresses the ability of the device output to source or sink sufficient current to slew the outputs at a high enough rate to meet the AC specifications.

The high-impedance load is a convenience to testing when trying to determine how rapidly the output enters a high-impedance mode. The resistive divider charges the capacitance until equilibrium is reached. Allowing for noise margin, testing for a 500 mV change is normal. By using a smaller capacitance than normal, you can make the change occur more quickly, allowing a more accurate determination of entry into the high-impedance state.

Switching-Threshold Variations

Along with input rise and fall times, switching-threshold variations can affect the performance of any device. Input rise and fall times are under your control and are primarily affected by capacitive loading or the driver and bus termination techniques. Switching threshold is affected by process variations, changes in V_{CC}, and temperature. Compensation of these variables is the responsibility of the manufacturer, both at the design stage and during the manufacture of the device. Combined threshold shifts over full military temperature ranges and process variations average less than 100 mV. This translates directly to V_{IL} and V_{IH} variations that track well within the noise margins of normal system design, particularly because the V_{OL} and V_{OH} changes track to the same 100 mV.

Electrostatic Discharge

Because of extremely high input impedance and relatively low breakdown voltage (approximately 30V), MOS devices have always suffered from destruction caused by ESD (electrostatic discharge). This problem has had two effects. First, major efforts to design input protection circuits without impeding performance have resulted in MOS devices that are now superior to bipolar devices. Second, care in handling semiconductors is now common practice.

Interestingly enough, bipolar products that once did not differ from ESD have now become sensitive to the phenomenon, primarily because new processing technology involving shallow junctions is in itself sensitive. MOS devices are in many cases now superior to bipolar products. A sampling of competitive bipolar and NMOS 64-bit, 1-Kbit, and 4-Kbit products reveals breakdown voltages as low as ±150V and greater than ±2001V.

The circuit in *Figure 4* protects Cypress products against ESD. The circuit consists of two thick-oxide field effect transistors wrapped around an input resistor and a thin-oxide device with a relatively low breakdown voltage (approximately 12V). Large input voltages cause the field transistors to turn on, discharging the ESD current harmlessly to ground. The thin oxide transistor breaks down when the voltage across it exceeds 12V; this transistor is protected from destruction by the current limiting of R_P.

The combination of these two structures provides ESD protection greater than 2250V, the limit of available testing equipment. Repeated applications of this stress do not cause a degradation that could lead to eventual device failure, as observed in functionally equivalent devices.

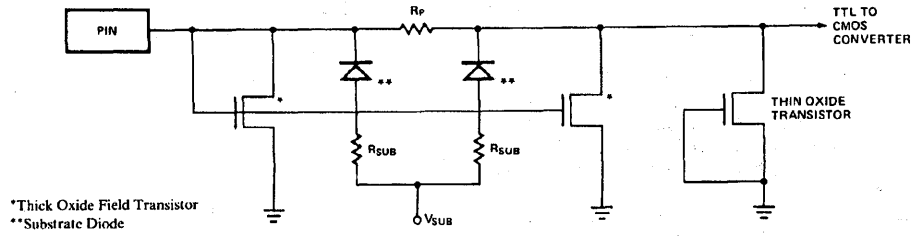


Figure 4. Input Protection Circuit

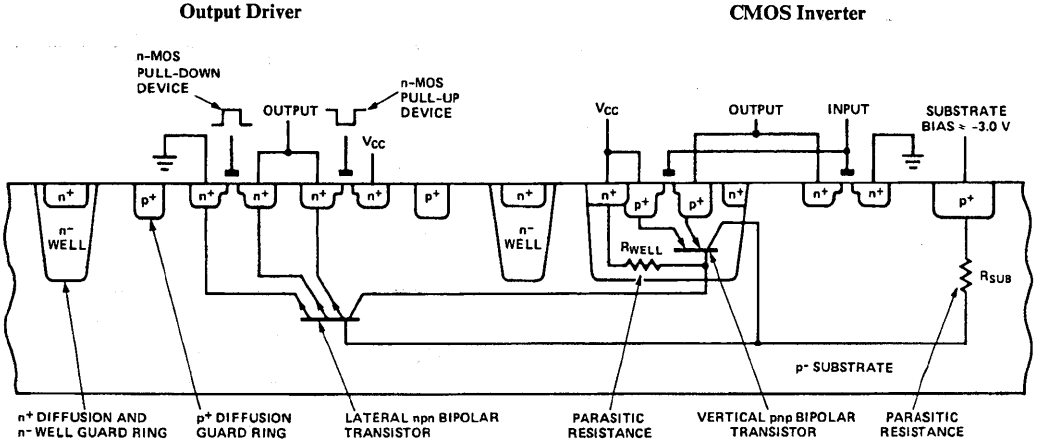


Figure 5. CMOS Cross Section and Parasitic Circuits

CMOS Latch-Up

The parasitic bipolar transistors shown in *Figure 5* result in a built-in silicon-controlled rectifier (*Figure 6*). Under normal circumstances the substrate resistor R_{SUB} is connected to ground. Therefore, whenever the signal on the pin goes below ground by one diode drop, current flows from ground through R_{SUB}, forward biasing the lower transistor in the effective SCR. If this current is sufficient to turn on the transistor, the upper PNP transistor is forward biased, which turns on the SCR and normally destroys the device.

Two possible solutions are to decrease the substrate resistance or add a substrate bias generator (*Figure 7*). The bias generator technique has several additional benefits, such as threshold voltage control, which increases device performance. The bias generator is thus employed in all Cypress products. Also used are guard rings, which effectively isolate input and output structures from the core of the device and thus decrease the substrate resistance by short-circuiting the current paths.

Latch-up can be induced at either the inputs or outputs. In true CMOS output structures such as the ones previously discussed, the output driver has a PMOS pull-up resistor that creates additional vertical bipolar PNP transistors, which compound the latch-up problem. Additional isolation using the guard ring technique can solve this problem at the expense of additional silicon area. Because all the devices of concern here require TTL outputs, the problem is totally eliminated through the use of an NMOS pull-up resistor.

Inducing Latch-Up for Testing Purposes

Exercise care in testing for latch-up because it is typically a destructive phenomenon. The normal method is to power the device under test with a current-limited supply, so that when latch-up is induced, insufficient current exists to destroy the device. Once this setup exists, driving the inputs or outputs with a current and measuring the point at which the power supply collapses allows nondestructive measurement of latch-up characteristics.

In actual testing, with the device under power, individual inputs and outputs are driven positive and nega-

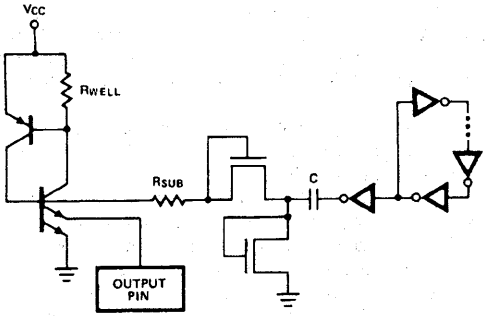


Figure 6. Parasitic SCR and Bias Generator

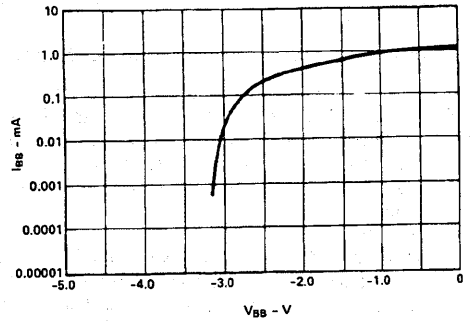


Figure 7. Bias Generator Characteristics

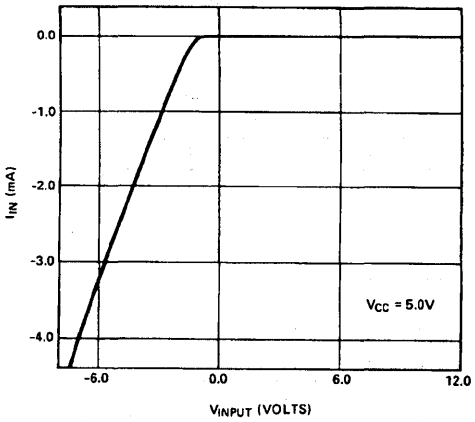
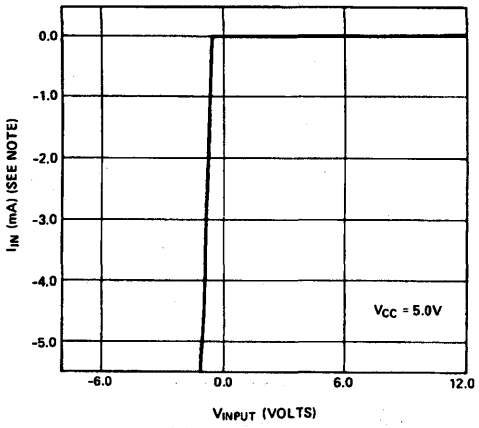


Figure 8. Input V/I Characteristics



Note: Output is in a High Impedance Condition.

Figure 9. Output V/I Characteristics

tive with a voltage. The current is measured at which the device latches up. This provides the DC latch-up data for each pin on the device as a function of trigger current.

Measuring the latch-up characteristics of devices should encompass ranges of reasonable positive and negative currents for trigger sources. Depending on the device, latch-up can occur at sink or source currents as low as a few milliamperes to as high as several hundred milliamperes. Devices that latch at trigger currents of less than 20 to 30 mA are in danger of encountering system conditions that cause latch-up failure.

Competitive Devices

Although few devices compete directly with the Cypress devices covered in this application note, the latch-up characteristics of the closest functionally similar devices were measured. The results show devices that latch-up at trigger currents as low as 10 mA all the way to devices that can sustain greater than 100 mA without latch-up. The Cypress devices covered in this application note can sustain greater than 200 mA without incurring latch-up, which is far more than it is possible to encounter in any reasonable system environment.

Eliminating Latch-Up in Cypress RAMs

The latch-up characteristic inherently exists in any CMOS device. Thus, rather than change the laws of physics, semiconductor manufacturers design to minimize latch-up effects over the operating environment that the device must endure. The environmental variables include temperature, power supply, and signal levels, as well as process variations.

Several techniques are employed to eliminate the latch-up phenomenon. One approach is to move the trigger threshold outside the operating range so that the voltage level never approaches this threshold. This can be done using low-impedance, epitaxial substrates and/or a substrate bias generator.

The use of a low-impedance substrate increases the undershoot voltage required to generate the trigger current that causes latch-up. A substrate bias generator has two effects that help to eliminate latch-up. First, by biasing the substrate at a negative (-3.0V) voltage, the parasitic devices cannot be forward biased unless the undershoot exceeds -3.0V by at least one diode drop. Second, if undershoot is this severe, the impedance of the bias generator itself is sufficient to deter enough trigger current from being generated.

The bias generator has one additional noticeable characteristic: It effectively removes the input clamp diode. This is due to the anode of the diode connecting to the substrate that is at -3.0V. Therefore, even though the diode exists, as shown in *Figure 4*, DC signals of -3.0V do not forward-bias the diode and exhibit the clamp condition. The benefits of the bias generator are apparent in higher noise tolerance, as substrate currents due to input undershoot do not occur.

Figures 8 and *9* represent the voltage and current characteristics of the devices discussed in this application note. *Figure 8* is characteristic of an input pin, and *Figure 9* an output pin in a high-impedance state. In *Figure 8*, the input covers +12V to -6V — well outside the +7V to -3V specification.

Figure 4 helps explain these characteristics. When the input voltage goes negative, the thin-oxide transistor acts as a forward-biased diode, and the slope of the the curve is set by the value of R_P. As the input voltage goes positive, only leakage current flows. The output characteristics in *Figure 9* show the same phenomenon, except that, because this is not an input, no protection circuit exists and therefore no R_P exists. An equivalent thin-film device acts as a clamp diode that limits the output voltage to approximately -1V at -5 mA.

CYPRESS SEMICONDUCTOR

Understanding Dual-Port RAMs

This application note examines the evolution of multi-port memories and explains the operation and benefits of Cypress's dual-port RAMs.

A dual-port RAM is a random-access memory that can be accessed simultaneously by two independent entities. In digital ICs, this implies a dual-port memory cell that can be accessed at the same time using two independent sets of address, data, and control lines.

A Brief History of Multi-Port Memories

The first multi-port memories were probably used in the CPU of the first computers. Many two-operand instructions are efficiently implemented using dual-port registers for the operands and the result.

For example, consider Equation 1, which describes a typical two-operand operation in the ALU (arithmetic logic unit) of a CPU:

(C) = (A) [OPERATOR] (B) *Eq. 1*

A and B could be either the operands (i.e., the data) or the addresses of the operands, in which case the data could be either in memory or in registers. In any case, Equation 1 describes two pieces of data, A and B, being operated upon by the OPERATOR and the results designated as C. C could also be the data, a register, or a memory location. OPERATOR could be arithmetic or logical.

The Combinatorial ALU

The 74181 was the first integrated circuit ALU. In this IC, the 4-bit operands, A and B, are operated upon according to a 4-bit command; the result, C, is output. The chip also provides a carry-in input, a carry-out output, and A = B outputs. A mode-control pin selects either logical or arithmetic operations. The 74181 is combinatorial; no storage is provided.

Early computers used the contents of a memory location as one operand and an accumulator in the CPU as the second operand. The results were usually stored in the accumulator.

Bringing the Registers On Chip

The 67901 was the first 4-bit slice that brought 16 4-bit registers onto the chip. The MMI 67901 was second-sourced by AMD and became the 2901. At one point, five vendors offered this industry-standard bipolar ALU. The Cypress CMOS CY7C901 is the highest-performance, TTL-compatible, 4-bit slice that is form, fit, and functionally equivalent to the original 901.

The 16-word deep, 4-bit wide register array is functionally equivalent to a 16 x 4 dual-port memory. Four A address lines and four B address lines select the contents of two of the 16 registers, whose outputs are applied to transparent latches. The latch outputs are then applied to 3:1 multiplexers, whose outputs drive the ALU inputs. The ALU outputs can be sent off chip, entered into a temporary register (Q), or written back into the register file, thus replacing one of the operands. This architecture is shown in the CY7C901 block diagram in the Cypress data book.

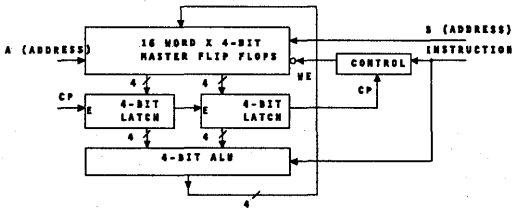


Figure 1. **901 Dual-Port Memory (Simplified)

CY7C901 Dual-Port Memory Operation

A simplified CY7C901 block diagram appears in *Figure1*. The device's A and B addresses select the contents of two registers, whose outputs are applied to two 4-bit latches. When the clock (CP) is High, the latch outputs follow their data inputs (i.e., are transparent). When the clock is Low, the ALU outputs are written ($\overline{WE}$) into the register array at the location specified by the A or B addresses, depending upon the instruction being executed. A Low on the clock causes the data in the latches not to change, so that the ALU outputs are

stable when they are written back into the register array.

Note that the CY7C901 does not perform the three-port function described by Equation 1. In the CY7C901, the C operand equals either the A or B operand, depending upon the instruction being executed. In fact, the A and B addresses can be the same. An old programming trick is to Exclusive-OR the contents of a register with itself, which clears the register.

Additionally, the CY7C901's dual-port memory does not use a dual-port memory cell. This type of cell is not required because the CY7C901 does not need the ability to simultaneously write independently to two separate memory locations.

Dual-Port Memory Using Single-Port RAM

Before the dual-port memory cell existed, designers created dual-port RAMs from single-port RAMs by adding a multiplexer between the RAM and the two entities that shared the RAM. *Figure 2* illustrates a block diagram of such an arrangement. Two processors, MP1 and MP2, share the RAM. If each processor has access to the RAM half the time, the resource is shared equally and is said to be allocated according to a fairness doctrine.

This time division multiplexing assures that there is no contention for the RAM. However, performance suffers if the RAM's access time does not equal 1/2 or less of the processors' clock period, assuming that the processors are clocked from the same source.

For example, consider two processors clocked from the same 25-MHz source, for a period of 40 ns. Because the processors are closely coupled, only one operating system is in memory. In this case, the maximum access time of the dual port has to be 20 ns or less. The highest-speed dual-port RAM available has a 25-ns access time. Therefore, each processor suffers a worst-case 20% performance degradation.

Dual-Port RAM Applications

The first applications for dual-port memories were for CPU register files. Dual-port RAMs can also serve as data or instruction cache memories. However, the largest usage of dual-port RAMs is in communications, which includes the exchange of data between processors, processes, and systems.

Virtual Dual-Port RAM

Communication between systems does not require physical dual-port RAMs. Instead, a conventional RAM memory is partitioned into virtual data-storage areas (buffers), usually to store at least two data packets. These buffers are shared between the communications controller and the intelligent element that assembles the packets and stores them (usually a microprocessor). The communications controller can also be a microprocessor. It reads the data from memory, converts the data from parallel to serial form, encodes the

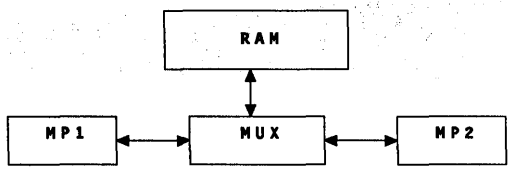


Figure 2. Dual-Port Memory Using Single-Port Ram

data, converts the data to analog form, and sends the data out over the communications channel on the transmit side. If the system contains only one processor, the data buffers are not shared, and the system needs neither a virtual nor a physical dual-port RAM.

Control information associated with each data buffer tells the communications controller the number of words in the buffer and the starting address of the data in the buffer. The control information resides in one or more memory locations whose addresses have been previously agreed upon by the two processors.

This simple software-based buffer example requires a second level of control — a mechanism or procedure that prevents the two microprocessors from getting in each other's way. In other words, the system needs a procedure control mechanism.

Another way of analyzing this requirement introduces the concept of data ownership. Say, for example, that processor A assembles and stores messages and thus owns the data while performing these tasks. Likewise, the communications processor B owns the data while performing its tasks. The procedure control mechanism amounts to a technique for transferring data ownership between processor A and B.

In large systems, where many processors perform many different operations, the processing of the information is called a job or a procedure. The procedure is divided into many tasks, which can be performed by different processors. The tasks can either be scheduled and assigned by a processor dedicated to that task or be performed by any available processor. These alternatives are referred to as autocratic and egalitarian systems, respectively. The term egalitarian implies that the processors are treated equally. In either case, the processors must have access to a shared memory location used for message passing.

Synchronizing sequential processes is the cornerstone of concurrent programming, which applies to multi-tasking, single-processor systems; distributed-processor networks; and tightly-coupled multiprocessor systems.

Message Passing

In the two-processor system under consideration, synchronization can be achieved by using a lockword or lockvariable. The lockvariable can apply either to data (as in this example) or to executable instructions.

The lockvariable is a location in shared memory that is operated upon using two synchronization primi-

tives: LOCK (v) and UNLOCK (v), where (v) is the location operated upon. These are simple binary switch operations. If a processor wishes to lock or own a critical section of code or data, the processor indivisibly sets the lockvariable if testing shows the lockvariable to be zero. If the lockvariable is not zero, then the operation is repeated until the lockvariable is zero. To unlock the critical section, a processor sets the lockvariable to zero and continues.

Most modern processors have indivisible read/modify/write instructions, also called test and set (TAS) instructions. In *Reference 1*, however, E. W. Dijkstra shows that lockvariables can be implemented without using a read/modify/write instruction. And in *Reference 2*, he develops the semaphore, a technique for managing a queue of tasks waiting for a resource. Lockvariables surround or bracket semaphores and thus provide entry and exit control on a mutual exclusion basis.

Typical TAS Instruction

The current example assumes that the processors have a TAS instruction. A typical TAS instruction operates as follows: Read, test, and set to X. The addressed memory location is read, and if its contents are zero, the value X is written into that location. If the contents are not zero, the contents are returned to the processor, and the value in the memory location is not disturbed.

The usual convention is that a value of zero in the lockvariable means that the resource associated with it is available. A non-zero value means that another processor temporarily owns the resource and that the resource is not available. After performing the task associated with the lockvariable, the processor sets the lockvariable's value to zero. The system is initialized with all lockvariables set to zero.

In the current example, processor A performs a TAS operation on the lockvariable and, finding the lockvariable zero, sets the lockvariable to a one. This tells processor B that the message is in the process of being assembled in the memory buffer area and is not ready to be transmitted. Processor A then assembles the message. After the message is assembled, processor A clears the lockvariable, sends a message to processor B saying that the message is ready to be transmitted, and gives the data's location and the number of bytes to be sent. Processor B reads the message from processor A and performs a TAS operation on the lockvariable; finding the lockvariable zero, processor B sets it to a two. This tells processor A that the message is in the process of being transmitted. Processor B then transmits the message and clears the lockvariable. Processor B sends processor A a message that the transmission task has been completed. After receiving the message from processor B, processor A performs a TAS operation on the lockvariable; finding the lockvariable zero, processor A concludes that the message has been successfully transmitted.

Note that this procedure does not require the use of a dual-port RAM. The procedure does require each processor to perform a TAS instruction, clear the lockvariable, and send a message to the other processor. Sending a message implies writing to a location in shared memory. To know that a message is waiting, the processor receiving the message must either read the memory location periodically (referred to as polling a mailbox) or the act of writing to the mailbox must generate an interrupt to the receiving processor. The interrupt-driven alternative is usually preferred because the receiving processor does not have to waste time in a polling sequence.

Dual-Port RAM Cell History

The first dual-port RAM ICs to use a dual-port RAM cell were the Synertek SY2130 and SY2131, introduced in 1983. These products are organized as 1024 words of 8 bits and use n-channel, double-polysilicon technology to achieve 100-ns access times. The SY2130 has an automatic power down feature controlled by the chip enables, and the SY2131 does not. The smaller (512 X 8) SY2132 and SY2133 were similar but unsuccessful.

The original dual-port RAMs include two mailboxes for message passing. When written to from one port, a mailbox generates an interrupt to the opposite port. Additionally, on-chip arbitration logic generates a busy signal to the loser when both left and right ports address the same memory location. If the loser was attempting to write, the write is suppressed.

Most of the dual-port RAMs on the market today are functionally equivalent to the original Synertek products. The "new features" added to several dual-port RAM products by Motorola and Integrated Device Technology (IDT) include dedicated semaphore registers. These semaphores are unnecessary, however, and the products that use them do not have second sources.

The SY2130 was second-sourced by IDT in 1984 and Advanced Micro Devices (AMD) in 1985. IDT also doubled the density to 2K X 8 and called the new part the IDT7132. Due to pin limitations (48 pins), the interrupt functions were deleted.

The AMD part (Am2130, 1024 X 8) had at least three logic errors. A busy-going-active indication failed to reset the interrupt when both ports addressed the same mailbox location. Additionally, busy going inactive failed to retrigger the address transition detection circuitry at all locations. And finally, when contention occurred and both ports were attempting to write, the losing port was not prevented from writing. The data sheet for this device does not explicitly state these conditions, but they must occur for the device to make logical sense (more on this later).

In 1985 IDT added slave companion parts to the company's dual-port family. The IDT7140 (1024 X 8) is the slave to the IDT7130, and the IDT7142 (2K X 8) is

the slave to the IDT7132. The slave device provides word-width expansion. Busy is an input to the slave from the master, and the slave contains no arbitration logic. One master can drive many slaves. This arrangement avoids the classic deadly embrace problem. This arrangement avoids the classic deadly embrace problem described in the next section.

The Deadly Embrace

The deadly embrace can occur when two masters are connected in parallel to make a wider word. If the left and right port addresses match, and the left and right port chip enables then become active to both chips at approximately the same time, it is possible to have one port of one master lose and the opposite port of the other master also lose. In other words, if an address match occurs and both ports are enabled during a small time window, or aperature of uncertainty, the dual-port RAM cannot determined which port wins or loses.

Under these conditions, if the corresponding left and right port busy pins are connected together, both ports of both masters are active (Low). This condition occurs because the busy outputs are open drain, and the loser pulls the node Low.

This condition is the simplest example of the deadly embrace. So far as the external world is concerned, both ports are busy, and the system remains locked up indefinitely, with each port waiting to be released by the other. Each master's arbiter section thinks it has lost the arbitration and is waiting to be released by the other.

In general, the deadly embrace occurs under two conditions: a processor requires one or more resources to perform a task, and one or more of the required resources is temporarily owned by another processor, which requires one or more of the same resources to perform its task.

For example, if processor A owns resource X and processor B owns resource Y, and both resources are required to accomplish the task, a stalemate occurs in which each processor waits for the other to relinquish the required resource. This is the simplest example. The concept extends to n processors and m resources.

The solution to the deadly embrace depends upon whether the system is autocratic or eglitarian, the tasks' priorities, etc., and is beyond the scope of this discussion. In the case of dual-port RAMs, however, the solution is simple: Do not cascade two masters in width; use a master and a slave.

The Cypress Dual-Port RAM Family

Table 1 lists the members of the Cypress dual-port RAM family. The package designator D26 stands for 600-mil ceramic DIP, and P25 stands for 600-mil plastic DIP. The 48-pin ceramic leadless chip carrier (LCC) is designated as L68. The 52-pin packages are designated as L69 for ceramic LCC and J69 for plastic LCC (PLCC).

Note that the interrupt function is not available at the 2048 X 8 level in a 48-pin package. This is due to pin limitations. At the 2-Kbyte level, each port requires an additional address pin for the address's most significant bit.

The M/S column in *Table 1* indicates whether the device is a master or slave. The difference between these devices is that the masters have arbitration logic and the slaves do not. The busy signals are outputs from the master and inputs to the slave. (The ramifications of this are examined later.)

Table 1. The Cypress Dual-Port RAM Family

Configuration	Part Number	M/S	Package Options				
			48-pin Dual In-Line Pkg		48-pin Square	52-pin Square	
			Ceramic	Plastic	LCC	LCC	PLCC
1KX8	CY7C130	M	D26	P25	L68	---	---
	CY7C131	M	---	---	---	L69	J69
	CY7C140	S	D26	P25	L68	---	---
	CY7C141	S	---	---	---	L69	J69
2KX8	CY7C132	M	D26	P25	L68	---	---
	CY7C136	M	---	---	---	L69	J69
	CY7C142	S	D26	P25	L68	---	---
	CY7C146	S	---	---	---	L69	J69

Note: The interrupt function is not available at the 2KX8 level in a 48-pin package

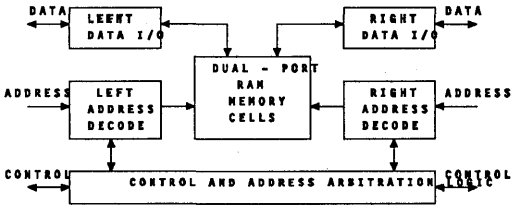


Figure 3. Dual-Port RAM Block Diagram

Cypress Dual-Port RAM Operation

A simplified block diagram of the Cypress dual port RAM appears in *Figure3*. The device interface includes three types of signals: address, data, and control. There are two sets of these signals: those of the left port and those of the right port. Each signal has either the subscript L or R to designate left or right, respectively.

The address pins are designated A0 through A9 (1024 X 8) and A0 through A10 (2048 X 8), where A0 is the least significant bit (LSB) and A9 or A10 is the most significant bit (MSB). The address pins are unidirectional inputs to the device; their states specify the memory location to be read from or written into.

The data pins are designated I/O0 through I/O7, where I/O0 is the LSB and I/O7 is the MSB. The data pins are bidirectional; their states represent either the data to be written or the data to be read.

The control pins are chip enable ($\overline{CE}$), read/$\overline{write}$ (R/$\overline{W}$), and output enable ($\overline{OE}$). Two flags are also provided, $\overline{INT}$ and $\overline{BUSY}$; both have open-drain outputs and require external pull-up resistors. A Low on the chip enable input allows that port to become functional. Data is either read from the internal dual-port RAM array or written into it, depending upon the state of the read/$\overline{write}$ signal; a Low initiates a write operation. The three-state data output drivers are enabled by a Low output enable.

When one port writes to a pre-determined mailbox, an interrupt to the other port is generated. When the interrupted port reads that memory location, the interrupt is reset.

When both ports address the same memory location and both chip enables are active (Low), contention occurs for that address. An arbitration is then performed, and ownership of the memory location is assigned to the winner. An active (Low) busy signal notifies the loser of the arbitration.

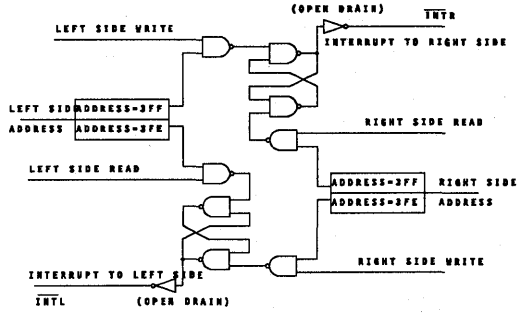


Figure 4. Interrupt Logic

Dual-Port RAM Functional Description

An important aspect of the Cypress dual-port RAMs is their interrupt logic. A simplified logic diagram of this logic appears in *Figure4*, with the chip enables deleted. A port's chip enable must be asserted for the port to either read from or write to any location, including the mailboxes. Note that you can use the mailbox locations as conventional memory by not connecting the interrupt line to the appropriate processor.

The upper two memory locations (7FF and 7FE for 2K x 8; 3FF and 3FE for 1K x 8) can be used for message passing. The highest memory location serves as the mailbox for the right processor. When the left processor writes to this mailbox, the interrupt (request) to the right processor, $\overline{INTR}$, goes Low. When the right processor reads its mailbox, the flip-flop is reset, and $\overline{INTR}$ goes High.

The second highest memory location serves as the mailbox for the left processor. When the right processor writes to this mailbox, the interrupt (request) to the left processor, $\overline{INTL}$, goes Low. When the left processor reads its mailbox, the flip-flop is reset, and $\overline{INTL}$ goes High.

Note that each port can read the other port's mailbox without resetting the associated flip-flop. If your application does not require message passing, leave the appropriate pin open. Do not connect a pull-up resistor to the pin, and do not connect the pin to the processor's interrupt request pin.

Note that the active state of the busy signal prevents a port from setting the interrupt to the winning port. Additionally, an active busy signal to a port prevents that port from reading its own mailbox and thus resetting the interrupt. These operations are ramifications of the data-ownership concept.

If both ports address the same memory location at the same time, the master performs an arbitration, so that one port wins and the other loses. Because each of the two ports can be in either the reading or writing state, there are four possible combinations of ports and states (*Table 2*).

Both Ports Reading

If both ports read the same location at the same time, you would assume that both ports should read the same data. This is true for all dual-port ICs. When arbitration occurs as a result of contention in a Cypress dual-port RAM, the port that wins the arbitration gets temporary ownership of the memory location. The

Table 2. Functional Operation of Dual-Port Masters

OPERATION			RESULT OF OPERATION AFTER ARBITRATION (MASTER)	
CASE	LEFT PORT	RIGHT PORT	AMD	CYPRESS and IDT
1	READ	READ	BOTH PORTS READ	BOTH PORTS READ
2	READ	WRITE	LOSER WRITES, WINNER IF READING, MIGHT HAVE CORRUPTED DATA AND NOT KNOW IT	LOSER PREVENTED FROM WRITING. IF LOSER IS READING AND PORTS ARE ASYNCHRONUS, DATA READ MIGHT NOT BE VALID
3	WRITE	READ		
4	WRITE	WRITE		

losing port can read the memory location but is told that it lost the arbitration by the busy signal.

To guarantee data integrity in a multiprocessor system, it is standard practice to apply the concept of data ownership. This ownership can apply to executable code, data, or control locations in memory. The control locations in memory can be associated with a resource, such as a printer, tape drive, disk drive, or communications port.

One Port Reading, the Other Writing

In the AMD dual-port RAM, the losing port is not prevented from writing. In the Cypress and IDT devices, the losing port is prevented from writing. All dual-port RAMs assert a busy signal to the losing port, so that this port can tell that the data might be corrupted.

In the Cypress dual-port RAMs, the losing port is prevented from writing so that the data cannot be corrupted. Busy is asserted to the losing port, so that the port can tell that its read or write operation might not have been successful.

Both Ports Writing

In the AMD dual-port RAMs, both are allowed to write. Busy is asserted to the losing port, indicating that the data might be corrupted. However, the winning port is not told that the data it just wrote might be corrupted by the writing of the losing port. This situation can cause system errors.

In the Cypress and IDT dual-port RAMs, the losing port is prevented from writing, so that the data cannot be corrupted. Busy is asserted to the losing port, indicating that its write operation was unsuccessful.

Arbitration Logic

Figure5 shows the arbitration logic used in Cypress dual-port RAM masters. The arbitration logic has three functions: to decide which port wins and which loses if the addresses are equal simultaneously; to prevent the losing port from writing; and to provide a busy signal to the losing port.

The arbitration logic consists of left and right address equality comparators with their associated delay buffers; the arbitration latch formed by the cross-coupled, three-input NAND gates labeled L and R; and the gates that generate the busy signals.

Operation With Unequal Addresses

When the addresses of the right and left ports are not equal, the outputs of the address comparators (nodes A and B) are both Low, and the outputs of the gates labeled L and R (nodes C and D) are both High. This condition forces both $\overline{\text{Busy}}$ signals High and both $\overline{\text{Write Inhibit}}$ signals High. The arbitration latch does not function as a latch.

Left Port Camped on an Address

Next, consider the condition where the left-port address and chip enable are quiescent, and the right port address changes to an address equal to that of the left port. Nodes A and B are initially Low.

Because the right-port address does not go through the delay buffer, the output of the right-address com-

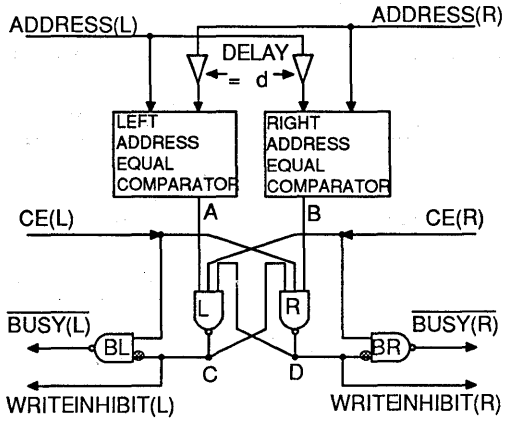


Figure 5. Arbitration Logic

parator (node B) goes High before node A goes High by a delay interval, d. The delay must be greater than the delay through the R gate, so that when node B goes High, node D goes Low, causing node C to remain High. CE(R) and CE(L) are both High; they are the inverse of the chip enable inputs. Node D going Low causes the output of the BR gate to go Low, which tells the right port that the memory location it just addressed belongs to the left port. A write-inhibit signal is also generated that prevents the right port from writing into the addressed memory location.

In summary, when the right port addresses a memory location that is already being addressed by the left port, a delay occurs that equals the sum of the propagation delays of the right-address comparator, the R gate, the BR gate, and the output driver (not shown in the diagram). Then the busy signal to the right port is asserted. Nodes A, B, and C are now High, and node D is Low. $\overline{BUSY}$ is asserted to the right port.

Due to the symmetry of the arbitration logic, the device operates the same when either the right or left ports are camped on an address.

Right and Left Addresses Equal Simultaneously

In the general case, it is possible to have both ports access the same memory location simultaneously, unless this is guaranteed not to occur by the design of the system. When nodes A and B go from Low to High at exactly the same instant, the arbitration latch settles into one of two states and determines which port wins and which port loses. The latch is designed such that its two outputs are never Low at the same time. It also has a very fast switching time.

The dual-port RAM imposes a minimum time difference between either of two events: the two chip enables going from inactive to active and the two sets of addresses going from mismatch to equal. If the events are close together in time, the probability of each port either winning or losing the arbitration is approximately equal. This parameter is called port set-up time for priority and is abbreviated as t_{PS} on the data sheets. The specified value is 5 ns. (Note, though, that Cypress product engineers have measured t_{PS} at room temperature and nominal V_{CC} (5V) and found a value of approximately 200 ps.) In other words, if one port addresses a memory location 5 ns before the other port, the first port is guaranteed to win. If not, the result of the subsequent arbitration is unpredictable.

Other Key Busy Parameters

Several other key parameters are specified with respect to the busy signal. For example, Busy Low from address match, t_{BLA}, is the maximum time it takes busy to go Low, as measured from the time the two port addresses are the same. This is the time from an address match until the losing port is notified that it has lost the arbitration. Obviously, the sooner this occurs the better. If the value of t_{BLA} is greater than the memory cycle

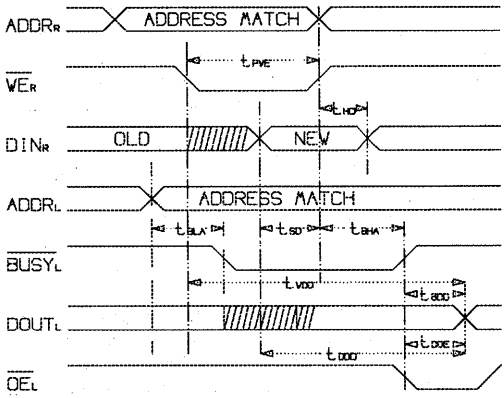


Figure 6. Busy Timing

time, another cycle must be added to detect the condition, which can severely reduce performance. This time is less than the minimum cycle time for all speed grades of all Cypress dual-port RAMs.

Another parameter, $\overline{Busy}$ High from address mismatch, t_{BHA}, is the maximum time it takes $\overline{busy}$ to go from Low to High, as measured from the time the two port addresses do not match until the $\overline{busy}$ signal goes High. The comments of the preceding paragraph also apply here.

The next two parameters are similar to the preceding two. The difference is that the chip enable controls the busy signal. The parameters are $\overline{Busy}$ Low from $\overline{CE}$ Low, t_{BLC}, and $\overline{Busy}$ High from $\overline{CE}$ High, t_{BHC}. Both of these parameters are less than the minimum cycle time for all speed grades of all Cypress dual-port RAMs.

$\overline{Busy}$ High to valid data, t_{BDD}, is the maximum time it takes the data to become valid to the losing port after $\overline{Busy}$ goes away. This parameter's value equals the address access time, t_{AA}, because a read cycle is initiated to the losing port when its $\overline{Busy}$ signal transitions from Low to High. An action by either port can cause the busy transition. The winning port can either change its address or deassert its chip enable.

To illustrate the last two parameters, *Figure 6* shows the timing for the right port performing a write operation and the left port asynchronously moving to the same address and attempting to perform a read operation. The first parameter of interest is t_{DDD}, which is the maximum time between the stabilization of the data to be written by the winning port and that same data becoming valid at the outputs of the port that received the $\overline{Busy}$. The second parameter of interest is t_{WDD}, which is the maximum time between the High-to-Low transition of the winning port's write strobe and the data becoming valid at the outputs of the port that received the $\overline{Busy}$.

It is possible for the losing port to read either the old data, the new data, or some random combination of

the two under these circumstances: the two ports are operating asynchronously (i.e., with independent clocks), and the conditions illustrated in *Figure 6* occur (winning port writing and losing port reading). If the read occurs early with respect to the write, old data is read. If the read occurs late with respect to the write, new data is read. And, if the read occurs at the same time the data is changing from old to new, the data read is not predictable. However, all is not lost. There are two general solutions. Both use the fact that the busy signal is asserted to the losing port, telling the port in this instance that the data it is reading might not be valid.

One solution is to use the High-to-Low transition of the busy signal to the losing port to generate an interrupt to the processor (or state machine) so that operation can be repeated. The drawback of this technique is that a snapshot of the states of the losing port's address lines and read/$\overline{\text{write}}$ line must be taken, so that the processor can tell what load/store operation caused the interrupt. Taking this snapshot requires latches or flip-flops for the data and control logic for doing the sampling, and the technique uses up an interrupt line. The processor must also be able to read the sampled data later.

A second solution is to use the Low level of the $\overline{\text{Busy}}$ signal to the losing port to prompt one of three types of delays: delay the reading of data until the data becomes valid, which occurs an access time after the Low-to-High transition of $\overline{\text{Busy}}$; insert wait states until $\overline{\text{Busy}}$ goes High; or stretch the clock until $\overline{\text{Busy}}$ goes High. Any of these methods probably require less hardware and control logic than the preceding approach. Use of these methods does mean that the $\overline{\text{Busy}}$ signal must eventually go from Low to High. This happens when the winning port either changes its address or deasserts its chip enable. For this reason, as well as for system noise immunity and power-saving considerations, it is recommended that blocks of addresses be decoded to generate chip enables for the dual-port RAMs.

Because the losing port has no control over the winning port in the general case, however, a question arises: What can the losing port do to successfully read the data just written, assuming the winning port does not change its address, write, or chip-enable signals? There are two possible operations:

1. Change an address line to a different address, then change back to the original address. This toggles the busy signal to the losing port.

2. Change the state of the chip enable. This also toggles the busy signal to the losing port.

Address Transition Detection

Why does changing the address or chip enable allow a losing port to read data successfully? All Cypress dual-port RAMs, both masters and slaves, use a circuit design technique called Address Transition Detection (ATD) to improve performance and reduce power dissipation.

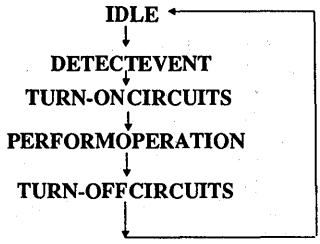


Figure 7. Simplified ATD Sequence

ATD improves performance by equilibration of differential paths, pre-charging critical nodes, and forcing the outputs to a high-impedance state. Equilibration and pre-charging bias critical nodes to voltage levels approximately in the mid-point of the small-signal operating range; when the data is sensed, it takes a shorter amount of time to transition to the Zero or One level. Forcing the outputs to their high-impedance states improves speed slightly, but more importantly, the technique reduces output switching noise by eliminating crowbar current and separating the output current into two pulses instead of one.

ATD minimizes power consumption because it turns on power-hungry circuits only when they are required. Slightly over 50 percent of a RAM's circuits are linear, and approximately 70 percent of the power is dissipated in the sense amplifiers during a read operation. When the RAM is operating at its maximum frequency, the ATD circuits are constantly triggered, so the power savings are minimal. At lower speeds or smaller duty cycles, however, the power savings are significant.

A diagram representing a typical ATD sequence is illustrated in *Figure 7*. The event that triggers the ATD sequence for either port is the transition of any address, chip-enable, or read/$\overline{\text{write}}$ signal. Equilibration and precharging are performed next, followed by either turning on the sense amplifiers and latching the data (read operation) or pulling the BIT and $\overline{\text{BIT}}$ lines to the required levels (write operation) at the addressed location. The master clock pulse lasts from 7 to 11 ns, depending upon temperature, supply voltage, and the distributions of IC processing parameters. At the end of the pulse, the data is latched and the appropriate circuits are turned off.

Master Stand-Alone Operation

Figure 8 presents a block diagram of a system using two 8-bit microprocessors, the Cypress CY7C132 dual-port RAM, static RAM, and EPROM. The address lines of each microprocessor are decoded to generate the chip enables to the dual-port RAM, the SRAM, and the EPROM. Note that pull-up resistors are required on the interrupt requests to the microprocessors and

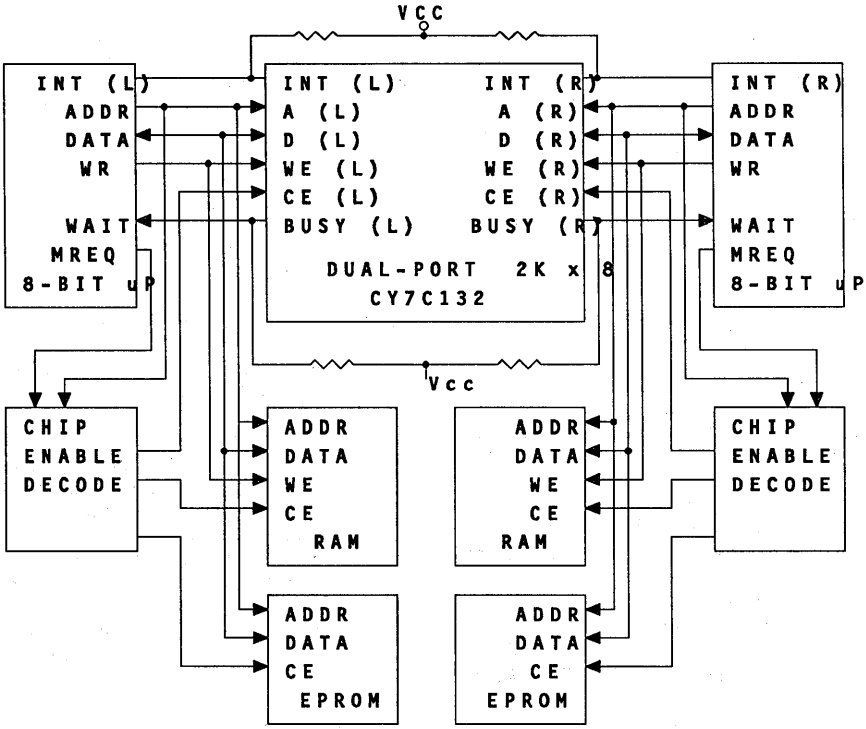


Figure 8. Typical 8-Bit Microprocessor

the busy signals, which go to the microprocessors' wait inputs.

Slave Word-Width Expansion

The block diagram in *Figure 9* shows how to interconnect a CY7C132 (2K x 8) master and a CY7C142 (2K x 8) slave to form a 16-bit-wide word. The diagram does not show the interfaces to the processors or the connections for the interrupt signals. As previously explained, the interrupt outputs are not available at the 2K X 8 level in the 48-pin DIP due to pin limitations. In the LCC and PLCC packages, the interrupt outputs are available from both the master and the slave devices. You can use either one. You do not have to tie the corresponding interrupt pins of the master and the slave together.

Delaying the Write Strobe

In width expansion, the write signals to the slave devices must be delayed by an interval at least equal to t_{BLA}, which is the time required for the master to assert the busy signal to the slave after an address match. The delay prevents the slave data at the address in contention from being overwritten. Both the write and read cycle times must be increased by this amount of time. In equation form:

$$t_{WC} = t_{PWE} + t_{BLA} \qquad \textit{Eq. 2}$$

where the delay must be at least equal to t_{BLA}.

Note that if you add more slaves to make a wider word, (e.g., 24 or 32 bits) the delay elements' outputs can connect directly to the write-strobe inputs. Additional delay elements are not required.

Slave Stand-Alone Operation

Some applications might require that you give one port permanent and absolute priority over the other. You can easily do this by implementing the memory using only slave dual-port RAMs. The $\overline{\text{Busy}}$ input to the priority port must be tied High by either connecting it directly to V_{cc} or to V_{cc} through a 10-KΩ pull-up resistor. You can connect the -ow priority port's $\overline{\text{Busy}}$ input to the high-priority port's read/$\overline{\text{write}}$ input.

In this configuration, the busy (read/$\overline{\text{write}}$) signal to the lower-priority port always prevents the port from writing when the high-priority port is writing to any location. The data of the Lower priority port is overwritten when the two ports operate asynchronously, the lower-priority port is writing, and the higher-priority

port simultaneously writes. This is not a very elegant solution because the $\overline{\text{Busy}}$ input to the low-priority port is not qualified by comparing the addresses of the two ports or their chip enables. However, this approach suggests how the slave dual-port RAMs can be used with external arbitration logic. The busy inputs can be used by control logic or under program control to dynamically change the port priorities.

If the lower-priority port is read only, you can tie its $\overline{\text{Busy}}$ input High by either connecting it directly to V_{CC} or to V_{CC} through a pull-up resistor.

Dual-Port Design Example

The following design example illustrates the methodology to follow when designing with Cypress dual-port RAMs. In this example, a dual-port memory is used for message passing and bus snooping for many bus masters on a 32-bit-wide system bus. The dual-port RAMs interface to a 32-bit system bus on the right side and a 16-bit processor on the left side. From the right port, the memory appears as 8K 32-bit words, and from the left port the memory appears as 16K 16-bit words.

The memory has the following characteristics:

1. The memory location corresponding to address zero for both ports is the same.

2. The data read from and written to the memory from both ports is in the same order. Thus, D0 of the right port corresponds to D0 of the left port. Additionally, D16 of the right port appears as D0 of the left port in address location 2048.

3. The minimum cycle time is 35 ns.

4. To conserve power, blocks of addresses are decoded to generate the required chip selects.

5. The CY7C132 and CY7C142 dual-port RAMs are used. Part of the design task is to specify the number of masters and slaves required and the way they must be interconnected.

6. The appropriate $\overline{\text{Busy}}$ signals must be generated to the correct port when contention occurs.

7. All possible mailbox locations that can be used for message passing are used.

8. The right port signals are AR0 ...AR12, DR0...DR31, $\overline{\text{CER}}$, and $\overline{\text{BusyR}}$. The left port signals are AL0...AL13, DL0...DL15, $\overline{\text{CEL}}$, and $\overline{\text{BusyL}}$.

A simplified logic diagram of the memory appears in *Figure10*. A total of 16 2K X 8 dual-port RAMs are required. The devices labeled MA (master, bank A) through MD (master, bank D) are CY7C132 masters. The devices labeled SU (slave, upper half-word) and SL (slave, lower half-word) are CY7C142 slaves. The memory consists of four masters and twelve slaves, along with the required control logic.

From the right port The memory is configured as 8K 32-bit words, with a master controlling three slaves. The one-of-four decoder labeled RB (right bank) generates chip-enable signals for each bank of 2K 32-bit words. Data is written (sampled) on the bus side, and the only reads performed are from the mailbox locations.

A general-purpose, right-port, control-logic block generates control signals that conform to the timing diagram shown in *Figure11*. The diagram does not show the generation of the output-enable control signals, but they are similar to the RB decoder signals. If your application does not require message passing to the right port, you can tie the right-port output-enable pins of all of the dual-port RAMs directly to V_{CC}.

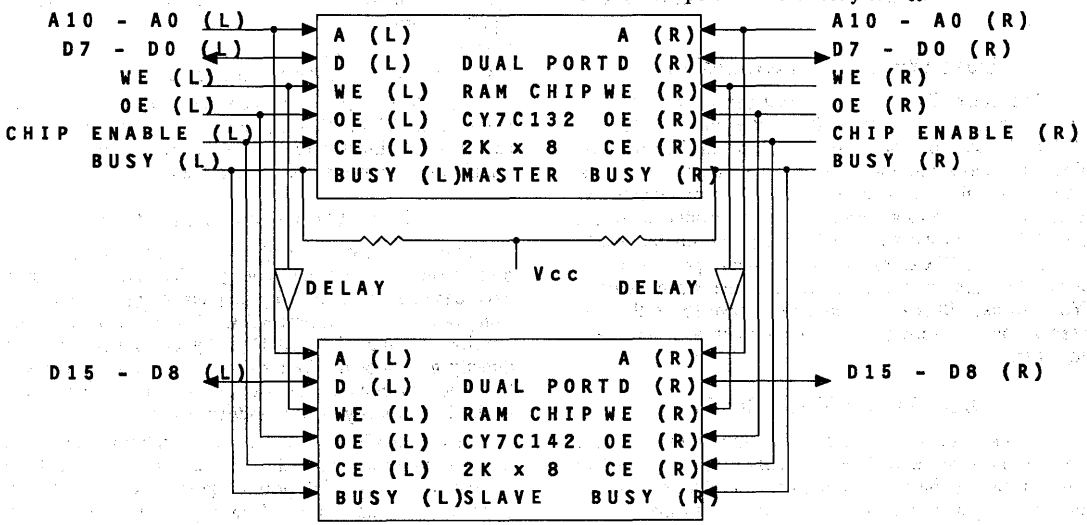


Figure 9. Expansion (2K x 16) With Slave

From the left port, the memory is configured as 16K 16-bit words. For this organization, you might think that the slave dual-port RAMs in the second column from the right in *Figure 10* should be masters. If this were the case, however, you would have to defeat the arbitration logic in them when the right port addressed the same address; this would add logic, reduce the speed, and complicate the design. Therefore, this design uses a combination of left-bank decoding (LB, 1-of-4 decoder) and upper-lower 16-bit word decoding (UL, 1 of 8 decoder) to cause the bank master to arbitrate when the right port is addressing the same bank as the left port (more on this later).

Right-Port Operation

For purposes of this discussion, "word" refers to the 32-bit word at the right-port system-bus interface. At the 16-bit processor interface, the 32-bit word is referred to as either the lower half word (right-port bits

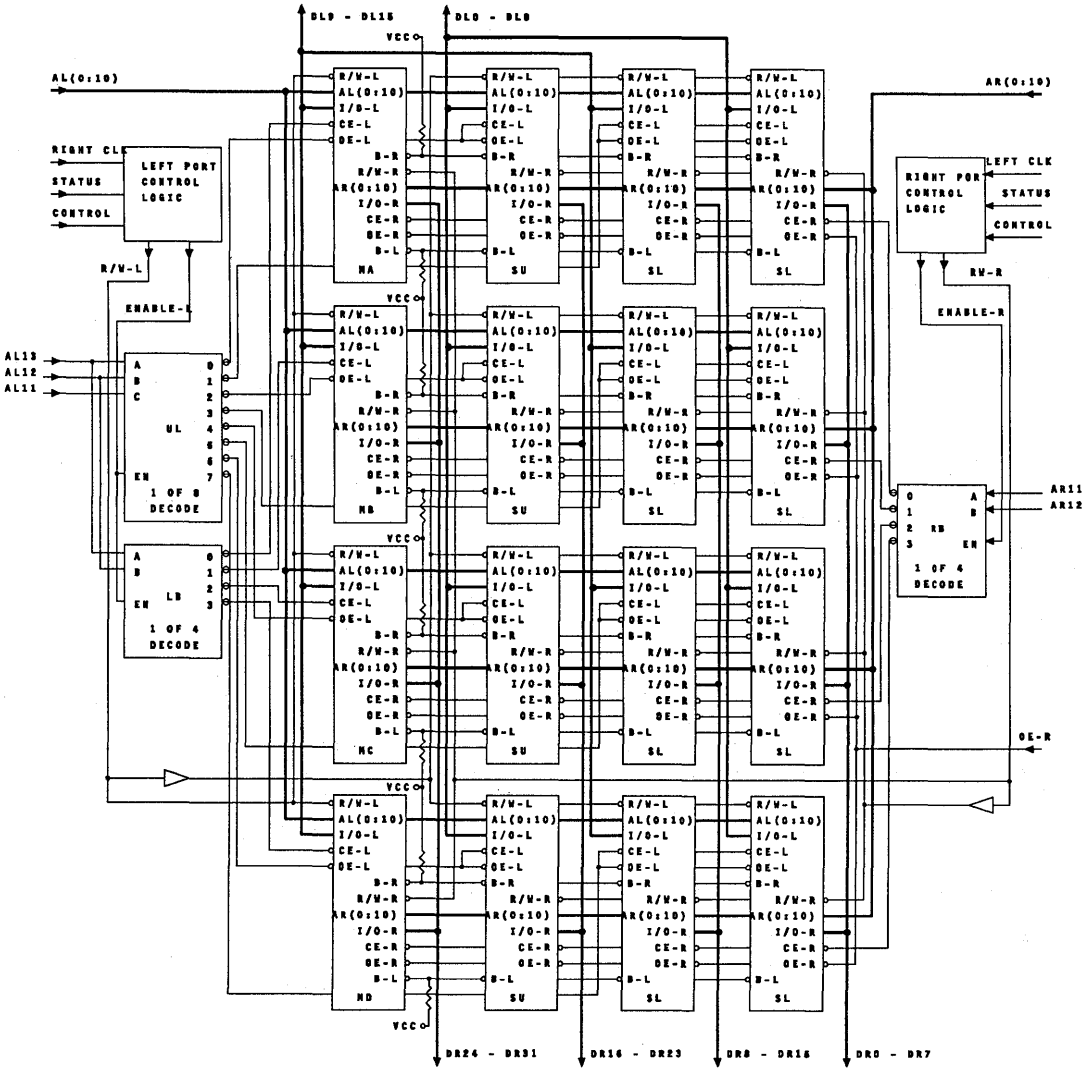


Figure 10. Logic Diagram for Dual-Port Example

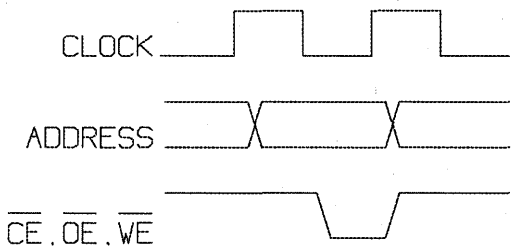


Figure 11. Dual-Port Timing for Example

0 through 15) or the upper half-word (right-port bits 16 through 31).

The bank-selection process employs the chip enables. Specifically, the 1-of-4 RB decoder decodes the four combinations of the upper two right-port address-bus signals and generates four active-Low chip enables to each bank of four dual-port RAMs. Bank A contains addresses 0 through 2047, bank B contains addresses 2048 through 4095, bank C contains addresses 4096 through 6143, and bank D contains addresses 6144 through 8191. In other words, bank A addresses 0 to 2K, bank B 2K to 4K, bank C 4K to 6K, and bank D 6K to 8K.

The lower 11 right-port address lines, AR(0:10), are connected to the A0 through A10 right-port address pins of all the dual-port RAMs.

Figure 11 does not show the generation of the write strobe, but does show the signal's timing. The write enable is applied directly to all the masters in parallel, then buffered, and then applied to all the slaves. The minimum propagation delay of the buffer must be at least as large as t_{BLA}, which is the time required for the master to assert the busy signal to the slaves after an address match occurs.

Note that all the right-port output-enable pins are connected together. These pins should be driven if reading is required; otherwise connect them to V_{CC}.

The open-drain busy outputs of the right port masters must be pulled up to V_{CC} using resistors. A value of 330Ω is recommended. The master busy outputs connect to all the right-port slave busy inputs for each bank.

For the data bus interface, the I/O pins of each RAM column connect to their respective I/O pins on each bank. This OR-tie connection is allowed because the bank-selection chip enable causes the output buffers of the un-selected banks to go to the high-impedance state.

Left-Port Operation

The 1-of-4 decoder labeled LB performs bank selection for the left port. The upper two left-port address lines, AL13 and AL12, decode bank-select chip-enable signals for the four masters only. Bank A corresponds to addresses 0 through 4095, bank B corresponds to addresses 4095 through 8191, bank C corresponds to addresses 8192 through 12,287, and bank D corresponds to addresses 12,288 through 16,383.

To perform upper and lower half-word selection, the 1-of-8 decoder labeled UL decodes the upper three right-port address signals. The decoder then generates eight chip-enable signals with a resolution of 2048. The chip enables connect to the slaves' chip-enable and output-enable pins (2048 resolution) and to the masters' output enable. Because the master chip-enable resolution is 4096, the master arbitrates for two blocks of 2048 16-bit half words.

The lower eleven left-port address lines, AL(0:10), connect to left-port address pins A0 through A10 of all the dual-port RAMs.

At the 16-bit interface, writing is only required if the left port wishes to send a message to the right port. Otherwise, you can connect the left-port write pins of all the dual-port RAMs to V_{CC}.

To implement the left-port data bus interface, the left port's data I/O pins are connected together in the same manner as those of the right port for all RAMs in the same column. In addition, to multiplex a 32-bit data word to a 16-bit half word, the least-significant bytes and the most-significant bytes of each 2048-word group are connected together. The UL decoder that controls the left-port output enable performs the selection.

If you use the masters' interrupt pins, pull them up to V_{CC} through a 330Ω resistor and connect them to the processor interrupt-request input. You can leave the slaves' interrupt pins unconnected.

If the control signal connections from their source to the dual-port memory constitute electrically long lines, they might require proper termination to avoid voltage reflections due to impedance mis-matches. Refer to the application note "Systems Design Considerations When Using Cypress CMOS Circuits" in this book for further information.

References

1. Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control." *CACM*, Vol 8, no.9, Sept. 1965, p 569.

2. Dijkstra, E.W., "Co-operating Sequential Processes." *Programming Languages*, F. Genyus (Ed.) Academic Press, New York, 1968, pp 43 - 112.

CYPRESS SEMICONDUCTOR

Using Dual-Port RAMs Without Arbitration

This application note offers several ways to implement dual-port RAMs to facilitate communication between processors. The applications covered include communication with general-purpose processors; video and radar equipment; digital signal processors; and bit-slice processors.

The most common application for dual-port RAMs is to provide a high-speed memory resource that can be shared between two processors in a system. *Figure 1* illustrates how the two processors communicate by passing data and commands via the shared memory. Both processors benefit by having access to the dual-port RAM because it is mapped just like any other memory device on the board.

Fast, local access to the shared memory eliminates the need to arbitrate for and access the system bus, when reading or writing a common resource area such as a shared memory card. In fact, many multiprocessor embedded-control systems implement dual-port RAMs for interprocessor communication and eliminate the system bus entirely. Removing the burden of a system bus, which only exists to hook the processors together, reduces the complexity of the system as well as the part count and power consumption.

Dual-Port Overview

Incorporating dual-port RAMs into a design such as the dual-processor example is straightforward. But it is important to consider the case of an address contention or busy situation that can arise when both processors simultaneously attempt to access the exact same location.

Cypress dual-port RAMs have several mechanisms that simplify simultaneous access. The simplest approach to resolving contention is to use the dual-port RAM's Busy output lines. Both right and left ports provide a Busy output signal. The arbitration logic inside the dual-port RAM activates Busy when the logic senses a match between the left and right address lines. Assertion of Busy indicates that both ports have attempted to access the same location in the RAM.

In the case of a dual-processor system, these signals can easily be gated with the processor's local Wait signal to generate a hold to the microprocessor until Busy is deasserted. Adding an occasional wait state to a microprocessor generally has no effect on the overall system performance.

Gating the Wait line and generating a hold to the processor resolves the logical problem of simultaneous address conflicts but does not address the system-level issues that can cause the conflicts. The two-processor example serves to illustrate a common underlying cause of a Busy state. Say that processor A attempts to read an array of data that was generated by processor B, but the system contains no mechanism to alert processor A when the data is ready or valid. Therefore, processor A might be updating a RAM location while processor B is reading the same address or vice versa.

This lack of overall synchronization or interprocessor communication can manifest as stale data or incomplete arrays of data in the shared memory. In a few cases, stale or incomplete date is tolerable, but in most cases it is fatal.

Locking a processor or processors out of specific memory areas until data is available guarantees that processors never receive stale data. To implement such address-space restrictions, you must provide a level of access protection above the basic gating-of-Busy technique. In most cases, you must add external hardware that signals the processors when new data is available or when

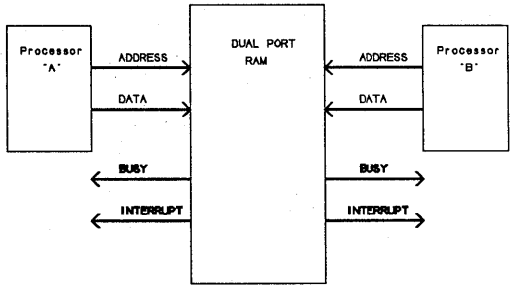


Figure 1. Dual-Processor Communication

Table 1. CY7C132 Interrupt Line Usage

Function	Result
Write to left Address 7FFh	Asserts Int_right
Read from Right Address 7FFh	Removes Int_right
Write to Right Address 7FEh	Asserts Int_left
Read from Left Address 7FEh	Removes Int_left

permission is granted to access a certain area of the dual-ported device.

Interrupts serve well as a simple means of alerting or synchronizing interdependent system elements that pass data via a shared memory. Cypress dual-port RAMs provide interrupt outputs to simplify the task of interrupting or signaling the processors; this relieves you of the need to create your own interrupt mechanism. Assertion and deassertion of these interrupt lines is accomplished by performing write and read operations to special locations within the dual-port RAM. *Table 1* lists the read and write operations.

The data word written to in these devices, 3FEh and 3FFh, can be used as a status word or semaphore. This word is presented to the data bus during the read operation of an interrupt removal cycle. The status word provides additional system-level information that augments the hardware interrupt signal by passing along some meaning with the actual interrupt event. More simply, the interrupt line alerts the processor that some action is required, and the status word provides additional information about exactly what happened or what needs to be done.

The actual meaning of the status byte is defined by the system designer. Generally, the status byte is used to indicate that data is ready, to lock a processor out of a specific range of addresses, or to prompt a processor for new data. Using the interrupt, along with status information, is an easy way of avoiding busy conditions by synchronizing processes or restricting address spaces via software.

You now have two main options for dealing with simultaneous address situations: Use Busy in a strictly hardware solution, or couple interrupts with status words for a software solution. Regardless of your preference for a hardware or software approach, Cypress dual-port RAMs provide all signals and functions necessary to ensure a simple and effective system solution that maintains data integrity and system sanity.

Using Dual-port RAMs Without Arbitration

Wait states and interrupts are a good solution for systems with microprocessor-like elements that are not affected by an occasional wait state. However, a much broader class of systems and applications cannot tolerate any type of data flow interruption or busy condition. Typically, these systems are dedicated function units that are rigidly pipelined and operate on continuous or nearly continuous streams of data.

A high-speed video processor is a good example of a system whose elements cannot be wait-stated due to the requirement that a data word or pixel be processed in every clock cycle. The block diagram in *Figure 2* shows a video data transform or look-up table.

This implementation uses a very common dual-banked or "ping-pong" RAM to realize a look-up-table translation function (*Figure 3*). A continuous stream of video data drives the address lines of RAM bank 0. The output or transformed data of bank 0 flows downstream to the post-processor units. Meanwhile, as continuous video data flows through RAM bank 0, the transform table of bank 1 is updated by a processor element, without interfering with the video data flow.

Dual banks make it impossible for a busy condition or address conflict to exist, because each system element essentially has its own discrete dedicated RAM. The processor finishes updating the look-up table, then swaps RAM banks by toggling the bank-select line. The PAL then changes the state of the buffer-enable signals, which redirects the data flow pattern of the two RAM banks.

The ping-pong arrangement is effective, but the implementation is very costly in terms of real estate. The

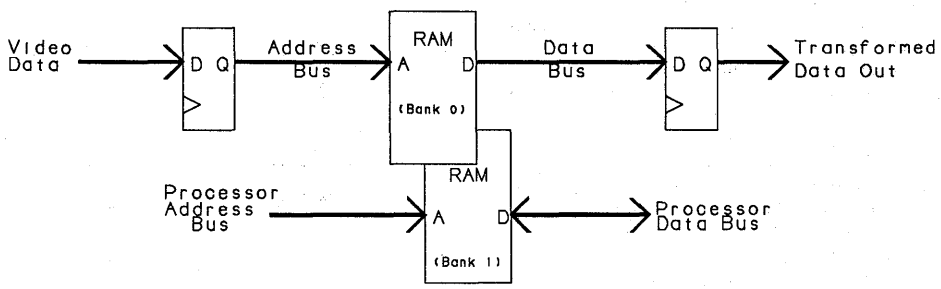


Figure 2. Video Look-Up Table

Table 2. Dual-Port vs. Ping-Pong RAM

Device	Qty	Power (mA)	Size (Sq.in.)
FCT244	6	15	0.4
FCT245	2	10	0.4
PAL16L8-D	1	180	0.4
20nsx8 RAM	2	140	0.52
Total	11	570	4.64
CY7C142-35	1	120	1.5

design requires at least 11 very high speed devices, using standard static RAMs.

Replacing the buffers, logic, and SRAMs with a single dual-port RAM (*Figure 4*) simplifies the design substantially. Video data utilizes the device's left port, while the processor communicates with the right port. Having two ports eliminates the need for any type of data and address steering buffers. During processor update cycles, however, there remains the problem of simultaneous address accesses and busy conditions.

RAM segmentation eliminates the possibility of a busy conflict and provides the key to implementing a dual-banked RAM within a single dual-port RAM. A single inverter segments the RAM. The Bank_select signal from the processor drives the left address port MSB,

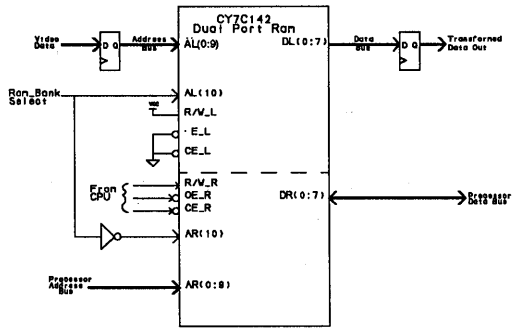


Figure 4. Video Lookup with Segmented Dual-Port RAM

and the Bank_select signal's inverse drives the right MSB. The dual-port RAM is now segmented into two 1K address spaces that do not overlap. The RAM appears as two totally separate RAMs, as it did in the ping-pong implementation. Again, because the left address can never equal the right address due to the opposite state of their MSBs, a busy condition is not possible.

Using a dual-port RAM does more than simplify the design. *Table 2* shows the tremendous savings in real estate and power consumption. Specifically, a single dual-port device reduces the board area by 68 percent and reduces the power consumption by almost 80 percent. In terms of MTBF, system reliability benefits greatly from

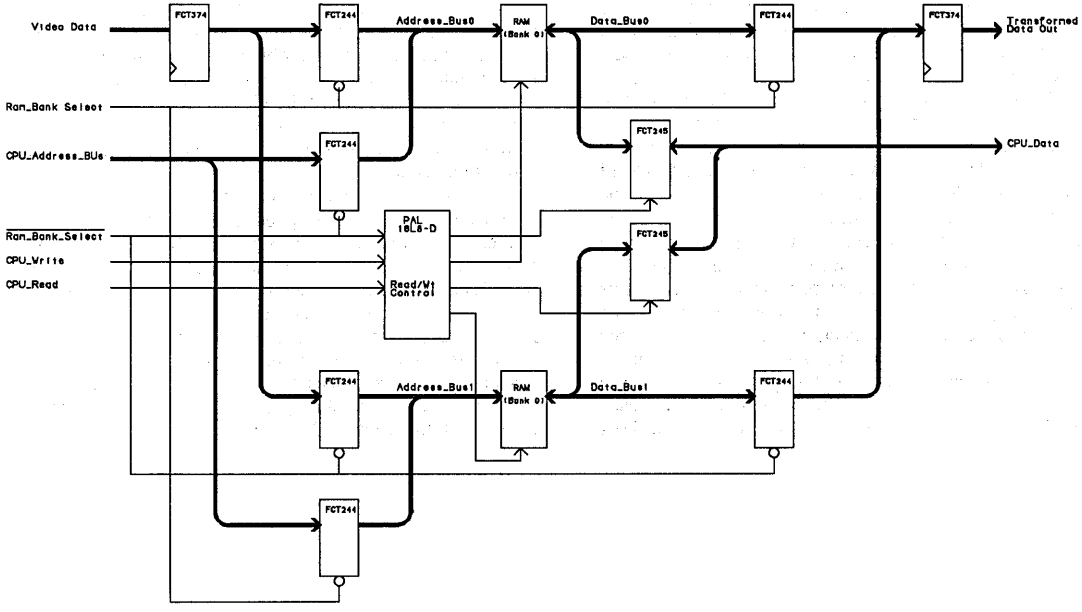


Figure 3. Ping-Pong RAM Array

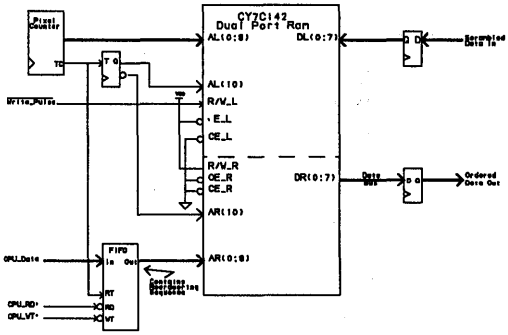


Figure 5. Data Descrambler

having fewer components and significantly lower power dissipation.

The multitude of buffers and transceivers that steer data and address signals in a ping-pong memory array take up relatively large amounts of board space as well as adding to the data propagation delay. The latter forces you to use very high speed RAMs. Dual-port RAMs do not suffer from the burden of buffer delays and can therefore operate at significantly lower speeds.

Handling Video or Radar Data

Many types of high-speed data-processing applications can benefit form the use of dual-port RAMs. For example, high-speed video or radar data is often transmitted in nonsequential or cross-interleaved order. The receiver must first descramble or reorder the data before the data can be used. Again, the incoming data stream cannot be stopped in the event of an address contention.

Figure 5 shows that a dual-port RAM is an ideal solution for this type of problem. Incoming data is written into the RAM's left port in the received order. The pixel counter provides sequential addresses to the left side of the dual-port RAM and increments after each pixel. At the end of the first line, the counter reaches terminal count and initiates a bank toggle via a T-type flip-flop. After the banks switch, the new data is accessible via the right port.

A FIFO stores the reordering sequence and thus drives the right port's address lines to read-out the stored video data. PROMs and counters can also implement the descrambling function, but this approach requires more parts and is much less flexible. Using a FIFO eliminates the need to generate addresses for the reordering sequence table. The CPU initializes the descrambling FIFO at boot

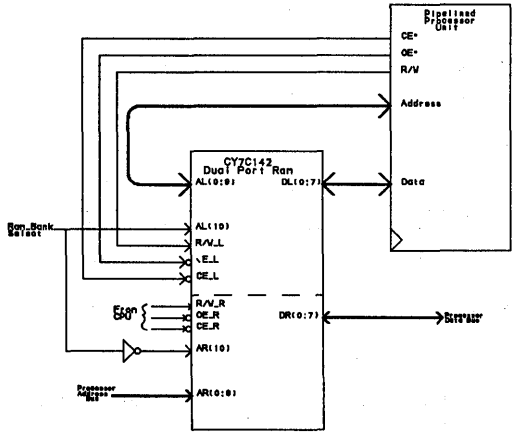


Figure 6. CPU/Pipelined Processor Interface

up. Initialization is only required once because the FIFO utilizes its retransmit function (described in the CY7C429 FIFO data sheet), unless the data ordering changes. Because this design implements the dual-port RAM as a segmented memory, you can ignore the problems caused by address contention.

For DSPs and Bit-Slice Processors

Interfacing a system's CPU to a high-speed, pipelined digital signal processor or bit-slice processor is another common system interface problem. Coefficients and commands must be passed to the pipelined processor, and final results read back by the CPU. Dual banks of RAM are often furnish a solution because they provide a shared memory space that both system elements can use without address contention.

Because the machines involved are rigidly pipelined, they cannot easily be stopped or interrupted. Thus, a single, segmented, dual-port RAM (*Figure 6*), or several dual-port RAMs in parallel with no additional glue logic, provides a simple, cost-effective solution to this problem.

If two banks of data are too restrictive, you can segment the dual-port RAM into multiple address spaces by restricting more of the upper-address-line pairs. This scheme allows the processor to easily and quickly communicate with the pipeline processor without using large amounts of real estate and power.

CYPRESS SEMICONDUCTOR

Using Cypress SRAMs to Implement 386 Cache

Because the 80386 is the most commonly used 32-bit microprocessor available today, this application note discusses some 386 cache implementations that take advantage of special features offered by Cypress's SRAM products. This application note does not offer a broad treatment of cache memories, however, and it assumes that you have a fundamental understanding of cache memories and the terminology associated with them.

Mainframe computers have used cache memories for several years. Desktop systems did not require caches until the advent of 32-bit microprocessors, such as the 80386, that run at clock frequencies of 20 MHz and above. A cache allows you to make full use of the microprocessor's available throughput. This is because the processor's bandwidth is greater than the bandwidth available from commonly available DRAMs.

In a memory hierarchy, a cache is a small, fast memory placed between the processor and main memory. A cache stores the most often used data and instructions to avoid accesses to main memory. Because of speed requirements, a cache is usually implemented with fast static RAM. The goal, then, is to implement the memory subsystem such that the processor's effective average access time approaches that of the cache, while the memory subsystem's cost per bit approaches that of the main memory.

Computer programs exhibit temporal and spatial locality, which make cache memories possible. Temporal locality refers to a program's tendency to re-reference the elements referenced in the recent past. Loops, temporary variables, and stacks are examples of constructs that conform to this property. Spatial locality refers to a program's tendency to access a portion of the address space in the neighborhood of the last reference. Sequential program execution and repeated access to array variables are examples of this property.

In addition to discrete cache implementations, several VLSI cache controllers are available today for the 80386. This application note describes two of the most popular: the Intel 82385 and the Chips and Technologies 82C307. A discrete cache implementation using Cypress products is covered first.

Discrete Implementation

You can implement a cache memory without using a VLSI cache controller. This discrete approach has the advantage of allowing you to custom tailor the cache subsystem to your specific requirements instead of being limited by a VLSI cache controller's capabilities. You can implement a low-cost cache subsystem or a cache with higher performance characteristics than can be achieved with today's VLSI cache controllers.

The discrete approach also has drawbacks. It makes high-speed caches more difficult to implement due to the delays incurred by discrete ICs input and output buffering, as well as trace delays introduced by the printed circuit board. Discrete solutions can also increase board-space and power requirements, and transmission line and noise effects become a more significant problem.

Figure 1 shows a block diagram of a simple, 64-Kbyte, direct-mapped, write-through cache. You can implement the control logic in programmable logic or a gate array (which are not detailed here). The cache tag or directory into the cache data is implemented in the CY7C150 1K X 4 resetable SRAM. The CY7B185 8K X 8 SRAM serves as the cache data RAM. CY7C408A 64 X 8 FIFOs are used as write buffers, which reduce the number of processor stalls in the write-through cache.

This example assumes that no memory references are made above 1 Gbyte. Thus, only the lower 30 address bits of the 80386 are used. Because the tag directory has 1K entries, and the data cache is organized as 8K X 32, the line size for this example is eight words or 32 bytes.

The 80386 supports two modes of local bus operation: pipelined and non-pipelined. With address pipelining enabled, the processor puts the address of the next memory access on the bus during the current access. This effectively gives the memory subsystem an extra clock cycle to decode the address. This approach has two drawbacks, however. First, entering pipeline mode incurs an additional wait state. Wait states also occur during branches, after periods when the processor's

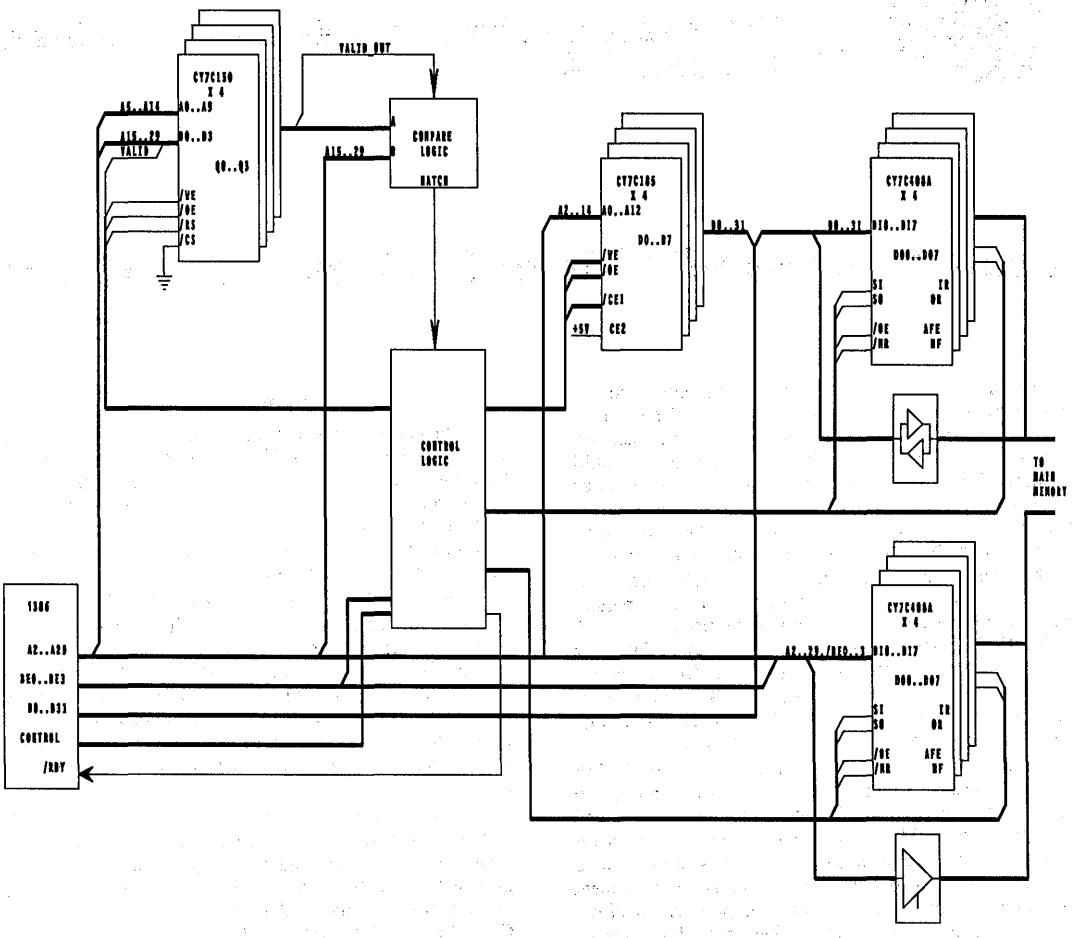


Figure 1. A Discrete Cache Implementation

pre-fetch queue is full, and after another bus master, such as a DMA controller, relinquishes the local bus to the processor. The second drawback is that the address and some of the control signals must be externally latched, requiring additional board space and complexity. Thus, for simplicity and increased performance, the 80386's address pipelining feature is disabled in this example.

During memory read accesses, address bits 5 through 14 index one of the entries in the tag RAM. Simultaneously, address bits 2 through 14 access the data RAM. After time t_{AA} of the tag RAM, the address tag appears at the comparator inputs. This tag is qualified by the valid bit and compared with 80386 address bits 15 through 29. The match output is fed to the cache control logic. If a match is found, and the cache line is valid (i.e., a read hit occurred), the cache RAMs supply data to the 80386, and the cache control logic asserts /386_RDY. If a match is not detected, or the cache line is invalid (i.e., a read miss occurred), the output enable of the cache RAMs is de-asserted, and a main memory access is initiated. The cache control logic causes the cache line to be updated from main memory. The control logic then updates the valid bit and supplies the requested data as well as /386_RDY to the processor.

This cache implements a write-through, no-write-allocate policy. Therefore, for write hits, both the cache RAM and main memory are updated before the 386

Table 1. Worst-Case Timing Calculations with the 82385

CALE : 82385 Cache Address Latch Enable
CS(3:0)# : Cache Select 3:0
COEA#,COEB# : Cache Output EnaBles A,B
WEA#, WEB# : Cache Write Enables A,B

Read Timing	
t_{AA} (max) non-pipelined mode	
4 CLK2 periods = 4 x 15 ns	60 ns
CALE valid from CLK2 (max)	- 15 ns
386 data set up time (max)	- 5 ns
t_{AA} (max)	40 ns
COE(A,B)#, CS(3:0)# to Data Valid	
4 CLK2 periods = 4 x 15 ns	60 ns
CS(3:0)# valid from CLK2	- 25 ns
386 data set up time	- 5 ns
COE(A,B)#, CS(3:0)# to data valid	30 ns
t_{OE} (max)	
2 CLK2 periods = 2 x 15 ns	30 ns
COE(A,B)# active from CLK2 (max)	- 15 ns
386 data set up time (max)	- 5 ns
t_{OE} (max)	10 ns
Write Timing	
WEA#, WEB# pulse width (min)	20 ns

can continue execution. On write misses, only main memory is updated.

Write buffers between the processor and main memory improve write performance. During write cycles, the processor writes to the write buffers, and the cache control logic updates main memory as a background task. While main memory is updated, the processor can continue executing as long as it executes read hit cycles or write cycles and as long as the write buffer has room. After a read miss, the processor halts until the write buffer has been completely flushed to main memory. Otherwise, the processor might access stale data from main memory.

The write buffers are implemented with Cypress CY7C408A 64 X 8 FIFOs. This device features speeds up to 35 MHz. It is deep enough that a full write buffer condition seldom occurs, and its output enable makes external three-state devices unnecessary.

The CY7C150 SRAM has two features that are beneficial in cache tag applications. First, access time is very fast. This product is available with a t_{AA} as fast as 10 ns. This speed is important, because the tag logic can prove to be the critical speed path in the design. Second, the CY7C150 has a memory reset function that allows the contents of the entire tag to be flushed within two memory cycles. Therefore, a cache flush operation can be performed much faster than if the processor had to invalidate the tag RAM on a line-by-line basis.

The CY7B185 SRAM is fabricated in Cypress's high-performance BiCMOS process and is organized as 8K X 8. The device is available with access times as fast as 10 ns and comes with a variety of packaging options. This part's X 8 width allows you to implement the entire data cache with only four devices.

Cypress provides a wide variety of memory width and depth configurations, all available with fast access times. You can thus implement the configuration that best suits your specific design requirements.

82385 Implementation

The 82385 is a VLSI cache controller offered by Intel that is specifically designed to work with the 80386. The device supports a 32-Kbyte cache and can be configured to operate in direct mapped or two-way set-associative modes by strapping the 2W/D# pin. *Appendix A* provides information for strapping the 82385.

The CY7C184 cache RAM connects directly to the Intel 82385 and 80386 with no external glue logic. You can configure the CY7C184 as a 2 X 4K X 16-bit device for set-associative implementations or as an 8K X 16 device for direct-mapped implementations.

During read misses, the 82385 invokes the 80386's pipeline mode to reduce the miss penalty. Therefore, the processor's address must be externally latched. The CY7C184 contains address latches, eliminating the need for discrete latches. Using discrete 4K X 4 SRAMs to implement the two-way set-associative configuration would require 18 ICs for the data cache and address latches. Only two CY7C184s can implement the same function in a space-saving 52-pin PLCC package.

The CY7C184 is configured by strapping the MODE pin High for set-associative operation or Low for direct-mapped operation. In set-associative mode, address bit A12 is a Don't Care and should be externally grounded. *Figures 2* and *3* show the connections for two-way set-associative and direct-mapped modes, respectively.

Table 1 illustrates some worst-case timing calculations for a 33-MHz system. As the CY7C184 data sheet shows, the -25 part meets or exceeds all the worst-case requirements. For the 33-MHz configuration, there is no difference in the 82385 timing specifications for set-associative and direct-mapped operation. Therefore, set-associative operation is recommended, because it yields higher hit rates. For some lower-speed grades of the 82385, the timing is less stringent for direct-mapped operation. Therefore, slower, less-expensive cache can be implemented for direct-mapped operation. Thus, you must make a price/performance decision.

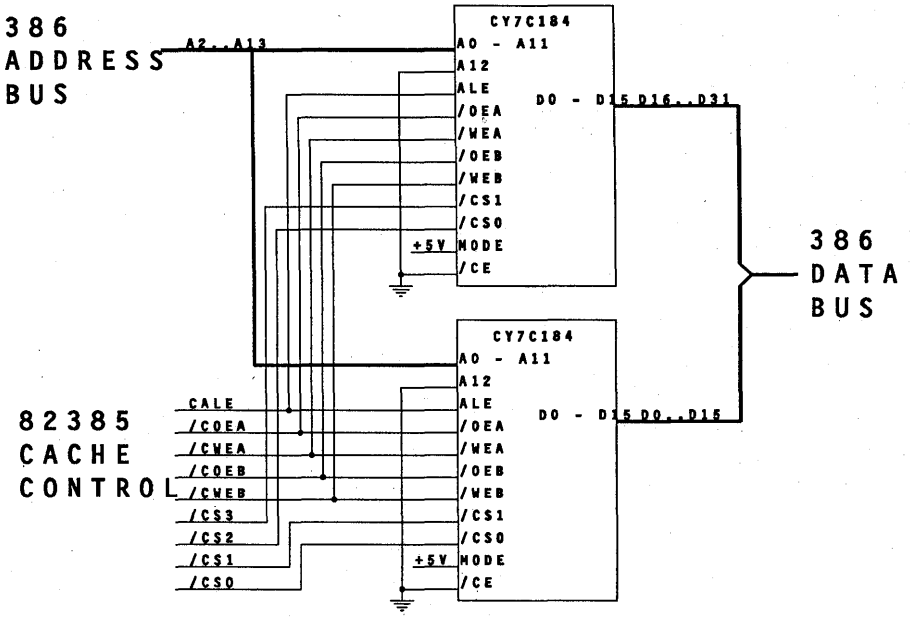


Figure 2. Set Associative Operation with the 82385

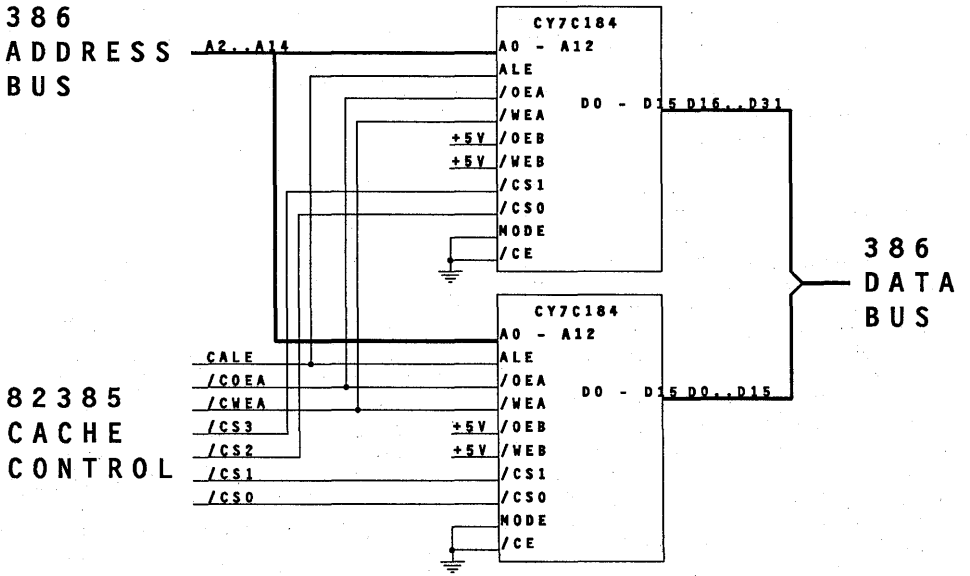


Figure 3. Direct Mapped Operation with the 82385

82C307 Implementation

The 82C307 is a combination cache/DRAM controller offered by Chips and Technologies. The device is part of a chip set designed to offer a high-performance IBM PC/AT-compatible system with a minimum number of components. Because the 82C307 has a two-way set-associative cache-mapping policy, strap the CY7C184 MODE pin High for proper operation.

The cache organization for the 82C307 is 2 X 4K X 32 bits. Two CY7C184s implement the entire data cache. The 82C307 also makes use of the CY7C184's built-in address latches when pipelined mode is required. The 82C307 has a programmable feature that allows either chip select or output enable to be supplied to the cache data RAM. This feature should always be programmed to generate a chip select when using the CY7C184. *Figure 4* illustrates how to use the CY7C184 with the 82C307. The Chips and Technologies 82C306 is used to latch the 80386 byte enables.

Table 2 illustrates some worst-case read timing calculations for a 25-MHz system in both non-pipelined and pipelined modes. As the CY7C184 data sheet shows, the -25 part meets or exceeds the worst-case requirements for non-pipelined mode, and the -45 part does the same for pipelined mode. Again, you must make a price/performance decision based on these options.

PCB Layout Considerations

As with any high-speed system, you must pay careful attention to the layout phase of a 386 cache project. The following rules of thumb help reduce noise problems and radiated EMI. A multilayer board with both power and ground planes is strongly recommended. Power and ground planes provide good, low-inductance paths for the power connections to the devices on the PCB. These paths help minimize ground bounce and other noise problems. Sandwiching power or ground planes between signal layers greatly improves the circuit board's noise characteristics. Ground-loop currents are minimized, which reduces capacitive and inductive signal coupling. A maximum center-to-center spacing of 8 mils between signal and power layers is recommended.

Good high-frequency decoupling on power and ground connections is very important for reliable high-speed operation. High-frequency bypass capacitors with NPO or X7R dielectrics are recommended. These devices store charge and supply instantaneous power required by the active devices on the PCB. For the CY7C184, one 0.1-µF and one 0.01-µF capacitor are recommended per device. Surface-mount capacitors are preferred because of the lower lead inductance these devices exhibit. Additionally, you can place surface-mount devices on the back of the PCB in the center of the device they are intended to decouple. This placement reduces the inductance between the capacitor leads and the active device's power and ground connections.

Table 2. Worst-Case Timing Calculations with the 82C307

CRD(1:0)# : 307 Cache Read 1:0

Read Timing	
t_{AA} (max) non-pipelined mode	
4 CLK2 periods = 4 x 20 ns	80 ns
CRD(1:0)# from CLK2 (max)	- 12 ns
386 data set up time (max)	- 7 ns
t_{AA} (max)	61 ns
t_{OE} (max) non-pipelined mode	
1.5 CLK2 periods = 1.5 x 20 ns	30 ns
CRD(1:0)# active from CLK2 (max)	- 12 ns
386 data set up time	- 7 ns
t_{OE} (max)	11 ns
t_{AA} (max) pipelined mode	
6 CLK2 periods = 6 x 20 ns	120 ns
CRD(1:0)# from CLK2 (max)	- 12 ns
386 data set up time (max)	- 7 ns
t_{AA} (max)	101 ns
t_{OE} (max) pipelined mode	
3 CLK2 periods = 3 x 20 ns	60 ns
CRD(1:0)# active from CLK2 (max)	- 12 ns
386 data set up time (max)	- 7 ns
t_{OE} (max)	41 ns

Avoid sockets whenever possible because of the extra inductance introduced. If sockets are necessary, high-quality sockets with gold-plated contacts are recommended.

Pay careful attention to the routing of traces. In general, traces should be kept as short as possible to reduce transmission-line effects. Point-to-point connections are recommended, as opposed to stubbed or tree-type connections. The latter causes discontinuities in the transmission line, which create reflections. Instead of 90° bends, traces should be curved; or use two 45° bends. This helps reduce EMI.

Critical signals, such as clocks and control lines, should be routed first. Whenever possible, keep these signals on the same layer, because vias cause transmission-line discontinuities. Routing these signals on the

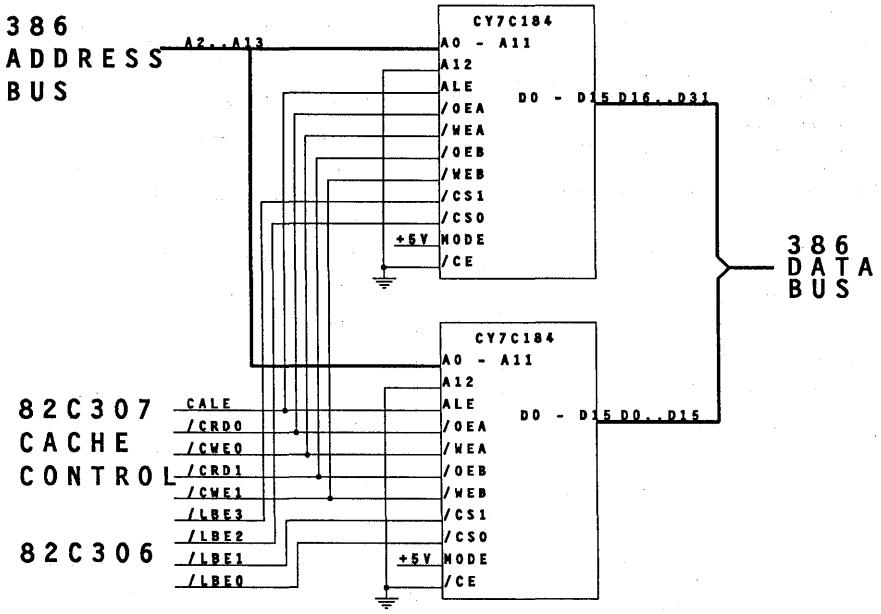


Figure 4. Operation with the 82C307

inner layers reduces radiated emissions. To minimize transmission-line effects, keep these traces to a maximum of six inches in length. To minimize crosstalk, a center-to-center minimum spacing of 16 mils is recommended for critical traces.

The signal quality of the system clock is a very important consideration. Pay careful attention to clock loading and skew. For high-speed clocks, it is usually recommended to supply each clock input from a separate driver. The clock drivers should be in a monolithic package, such as a hex inverter, so that clock skew is minimized. Keeping clock traces approximately the same length also helps minimize clock skew.

Series damping resistors in the 10 to 27Ω range might be required on clock traces to achieve good signal quality. If so, use as low a value as possible. Experimentation determines the optimal value.

Once control lines have been routed, address and data lines can be routed. These signals are somewhat less critical, because some settling time is usually provided in the worst-case timing. However, these signals should still be routed point to point, and trace length should be minimized.

Appendix A

Strapping Information for Different Steppings of the Intel 82385

Intel manufactures different versions (steps) of the 82385 cache controller. For example, the C step activates the output enables to the cache RAMs whenever the write enable signals are asserted. Step B, on the other hand, inhibits OE# while WE# is Low. Step SB, one of the new revisions, allows you to control the state of the OE# output during write cycles. Cypress recommends that pin A14 be tied Low; then OE# is de-asserted during write. There are two reasons for this:

- Although the 7C183 three-states its outputs (t_{HZWE} = 15 - 20 ns) after WE# is asserted, even if the OE# input is active, the write pulse width (t_{PWE}) in some systems might not be long enough to satisfy the t_{SD} requirement after t_{HZWE} is satisfied.
- Assuming t_{PWE} is long enough to satisfy t_{SD}, you must contend with another problem. After the 7C183 three-states its outputs, the noise caused by its buffers drives the V_{IH} level to 3V. In other words, any inputs less than 3V might not be recognized as a High level. If you want to avoid this condition, pull OE# High 10 ns before asserting WE#.

Section Contents

CYPRESS SEMICONDUCTOR

Pin-Out Compatibility Considerations of SRAMs and PROMs

This application note discusses the non-electrical parameters of pin-out and programming involved in finding socket-compatible second sources for PROMs. Included here is an example of a verified conversion from the Motorola 68764 to the Cypress 7C264, a PROM conversion that is not address-line compatible.

An SRAM Comparison

To understand how to choose second-source PROMs, consider a comparison with the process of choosing second-source SRAMs. Ignoring the AC/DC characteristics, finding a second source for an SRAM is relatively simple. So long as the power, ground, control (chip select, read, write), address, and data lines are on the same pins, the devices should be compatible. *Specifically, on SRAMs, the address and data lines need not be numbered identically* between the two devices for them to function identically in the same socket. As an example, on several Cypress SRAMs, the address pin numbering is not the same as some of our competitors.

Consider a simplified example that illustrates why address pin numbering is not a problem: Assume you have a new device, the 2-bit x 4-location SRAM shown in *Figure1*. Note that the inferior pin-out chosen by the Brand "X" 2 x 4 assigns address line 2 (A2) to pin 1, while the superior pin-out used by the Cypress device has A1 at pin 1, etc.

Assume that your engineering staff designed an infrared scanning-pattern-recognizing toaster oven using the Brand "X" SRAM, working only from the device's data sheet. Just as your company is about to ramp into volume production, Brand "X" sends out an End Of Life notice on the 2 x 4, because the company is converting all of its capacity to making DRAMs.

At this point, because you have no desire to lay out a new PCB, take a look at how the Cypress and Brand "X" SRAMs would look in your design (*Figure2*). In the figure, µP designates a microprocessor interfacing to the SRAM. The important thing to notice in *Figure2* is that the data read from an address generated by the microprocessor is the same as data written to the same location earlier. With an SRAM, any inconsistency between the address and data line numbering does not matter because the data read is the same as the data previously written.

To illustrate the point further, suppose that you write a value of 1 (µP:D2,D1 = 0,1) at location 2 (µP:A2,A1 = 1,0). If you read location 2, you obtain the value 1 that was written, because the address presented to the SRAM during the read is the same as the address for the previous write. Similarly, the data read is in the same bit order as presented during the previous write to the location. So far as the system is concerned, the two SRAM devices are compatible.

Although not significant to the system, the devices differ in where they internally store the data. In the

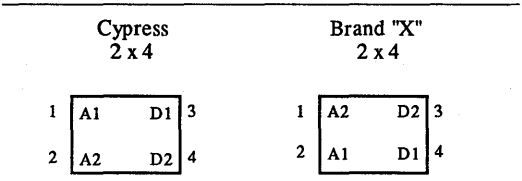


Figure 1. Example 2x4 Simplified SRAMs

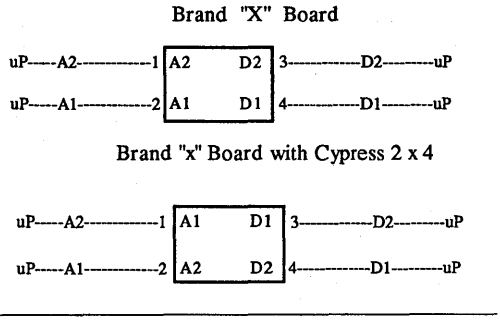


Figure 2. Example System with 2x4 SRAMs

Cypress device, the µP address of 2 (µP:A2,A1 = 1,0) actually stores the data at SRAM location 1 (Cypress:A2,A1 = 0,1). The Brand "X" RAM physically stores the data at address 2.

The address translation is transparent to the µP, however. Because the same location is accessed for the subsequent reads, the difference in address numbering between the two devices does not matter to the system. Similarly, any numbering difference on the data lines does not matter either. All writes and reads are generated in your system; thus, so long as the address and data lines are on the same pins, differences in the numbering do not matter.

Second-Sourced PROMs

For PROMS, the scenario becomes slightly more complex. Because you program PROMS using a programmer that is separate from the system in which they are used, it is more difficult to substitute PROMs that do not have the same address- and/or data-pin numbering.

Assume, for example, that the high-tech toaster oven's 2 x 4's are PROMS. If you program each location with data, you find that the Cypress device does not work properly when used in the Brand "X"-designed socket. In this case, the PROM programmer puts the data at location 2, and the board reads this data when the microprocessor requests the data at location 3. Additionally the data bits are swapped on this read. What a mess! It becomes apparent that it is easiest to replace this PROM with a device that has the same address- and data-line numbering.

There are methods that allow you to use the Cypress 2 x 4 PROM in the Brand "X" socket, however. The objective in trying to make the Cypress PROM work in the foreign pin-out socket is to have the system read the same data as when the Brand "X" device is used. In the 2 x 4 example, you encounter two problems: mismatches in the numbering of address lines and data lines.

Correcting Data-Line Mismatch

First consider the data-line mismatch. As it stands, data programmed in as bit1,bit2 is read as bit2,bit1. You could fix this problem by swapping the printed traces for D1 and D2. Unfortunately, this would also disallow the use of the Brand "X" device.

If you could internally swap the data bits programmed into the Cypress device, they would be in the correct order when read. You can, in fact, swap the data bits in the Cypress device through several means. First, you might modify your programming adapter such that D2 and D1 are swapped when programming the part. Then when the device is read, you get the bits in the same order as presented by the Brand "X" device. This is not a recommended method of solving the problem, because modifying programmers tends to make the manufacturer of the programmer unhappy.

1) Brand "X" 2 x 4	: Bit 2, Bit 1
2) Programmer (Cypress)	: Bit 1, Bit 2
3) Cypress 2 x 4	: Bit 1, Bit 2
4) System Board uP	: Bit 2, Bit 1

Figure 3. PROM Bit Swapping with Programmer

A second method of solving this problem is to alter the binary image of the PROM contents such that bits D1 and D2 are swapped in a file on your computer's disk; this altered binary image file is then used to program the Cypress PROM. This approach is less likely to cause damage than modifying a programmer, but requires some skill in altering the binary file.

Finally, the easiest solution to this problem is to trick the PROM programmer into swapping the bits for you. If you set your programmer for the Cypress device type, read a programmed Brand "X" device into memory, then program the Cypress part with the image in programmer memory, the bits are swapped for you.

You can see how this bit swapping works by examining *Figure 3*. The bits in the Brand "X" device are stored in the order Bit2,Bit1 — the same order in which the toaster's µP reads them. When you set the programmer to read the Cypress part, the data lines are logically swapped from the Brand "X" ordering. Thus, when you read the Brand "X" part, the data bits are swapped as shown.

When the Brand "X" part is removed from the socket, and the Cypress device is plugged in and programmed, the bits are programmed into the Cypress part in this same "reversed" order. When you place the Cypress part into your board, the bits are swapped again due to the difference in numbering between the Cypress part and the board layout, and the µP gets the data in the correct order.

Correcting Address-Line Mismatch

The second problem in substituting PROMs is the difference in address-line numbering. You can resolve this problem in exactly the same manner as the data swap problem. By simply setting the programmer to the Cypress device type, reading the Brand "X" part, then programming the Cypress part, any addressing differences are solved. The location of data words are swapped to allow for the difference in pin-outs, just as the bits were swapped in the data-line mismatch.

Working with PROM Programmers

Many programmers allow you to read a device different than the part selected, complaining only during programming if the device types do not match. With

PIN	Cypress 7C264	Motorola 68764
21	A10	A12
19	A11	A10
18	A12	A11

Figure 4. Cypress 7C264 vs. Motorola 68764 Pin-out

such a programmer, carrying out the procedures to convert a PROM should not present a problem.

Some programmers, however, do not allow you to read a device if it is different from the part selected. These programmers prevent the conversion method from working. Fortunately, the Cypress CY3000 QuickPro programmer does permit use of the conversion method. Cypress Field Applications Engineers, sales offices, and distributors can use their QuickPro programmers to generate a Cypress master PROM that you can use as a source for copying with uncooperative programmers.

Conversion Example

As an example of a PROM conversion, consider the Motorola 68764 8K x 8 PROM. It has a similar pinout to the Cypress CY7C264, with the exception of address lines 10, 11, and 12.

To program a Cypress CY7C264 to work properly in a socket designed to accept the Motorola device, use this procedure:

Invoke the Cypress QuickPro or other appropriate programmer and select the Cypress CY7C264 as the device to be programmed.

Place the Motorola part in the programmer adapter socket and read the device. Optionally, write the device contents to a disk file.

Place a Cypress CY7C264 in the programmer adapter socket, and program the part. Optionally, you can read the contents of the disk file as the source for programming.

The programmed device now works in the socket designed for the Motorola part.

CYPRESS SEMICONDUCTOR

Introduction to Diagnostic PROMs

This application note provides a basic understanding of the concept of a diagnostic PROM, as well as a brief introduction to possible applications.

Beginning with a short tutorial on system diagnostics, this application note presents the reason for incorporating diagnostics into a design and the special testability problems associated with sequential designs. The concept of shadow-register-based diagnostics is presented, and the benefits of this approach are outlined.

Next, a description of diagnostic PROMs is given. This covers the similarity of diagnostic PROMs to standard registered PROMs, as well as the fundamental operation of a diagnostic PROM. Next is a description of the Cypress CY7C268 and CY7C269 8K x 8 diagnostic PROMs. An application example is also included.

Introduction to System Diagnostics

As electronic systems continue to grow in size, function, and complexity, it is becoming increasingly difficult to test them and determine their reliability, as well as to service the end product in the field. One way to simplify the task of testing electronic systems is to design some form of testability into the system.

Controllability and observability are the key points of testability. These two qualities are easily obtained for a combinatorial system in which the outputs are strictly a function of the current inputs. Test vector methods are easily devised and implemented for combinatorial systems. But, for a sequential system, in which the outputs are a function of both the current inputs and the previous state(s), controllability and observability can be lost due to lack of access to the internal states of the machine. Consequently, building testability into a system means being able to control and observe all possible states of the system.

Consider the simple sequential machine in *Figure 1*. Access to internal states is either denied or difficult to obtain. The obvious way to add testability to this system is to permit access to these internal states.

One way to gain this access is through addition of a diagnostic shadow register, as shown in *Figure 2*. Observability is effected by adding a serial data output path (SDO) to allow shifting internal state information out of the system. Controllability is gained by permitting a serial data input path (SDI) to set the state of the internal registers. As a result, relatively simple test vector methods can be used to test the system.

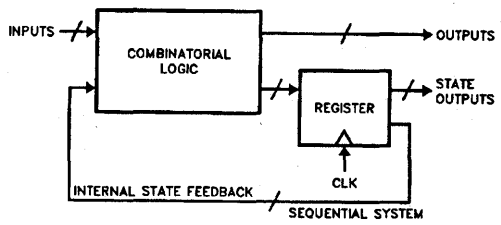


Figure 1. Simple Sequential Machine

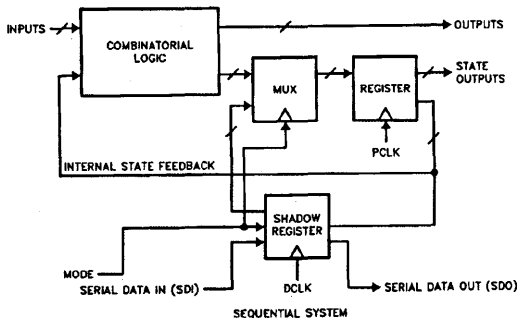


Figure 2. Simple Sequential Machine with Diagnostic Capability

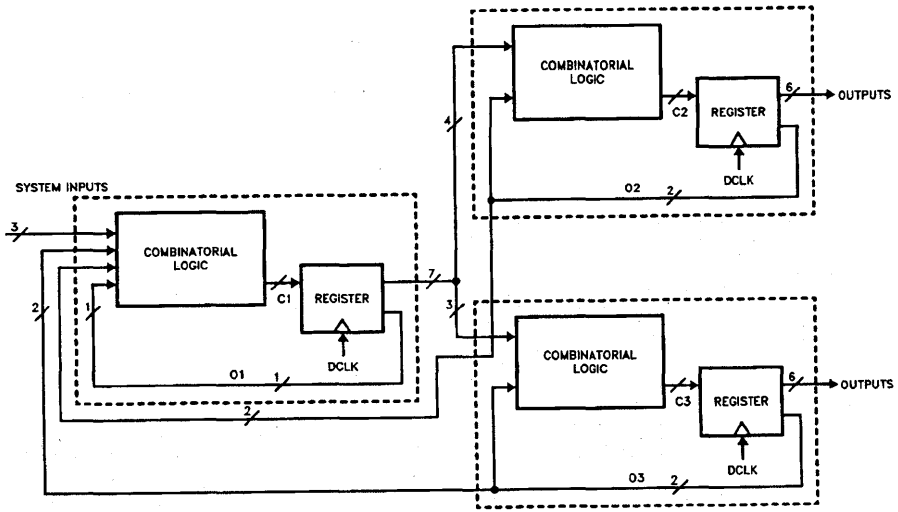


Figure 3. Complex Sequential Machine

Consider, for example, the complex sequential machine shown in *Figure 3*. This system would be virtually impossible to test in the current configuration because you cannot control or observe the machine's internal states. To increase this machine's testability, observability must be added at points 01, 02, and 03. If this were accomplished, you would be able to observe the internal states of the machine. Additionally, controllability must be added at points C1, C2, and C3. This would allow you to set the internal states of the machine.

This controllability and observability can be attained by adding shadow registers, as depicted in *Figure 4*. The result is a complex sequential machine with a high degree of testability. As a result of these actions, simple test vector methods can now be used to fully test the machine. For instance, the state of the register at point C1 can be set, the machine can be clocked through some known number of cycles, and the state of the machine can be observed at points 01, 02, and 03.

Knowing what state the machine should be in at a specific time at each observation point (the machine's "known-correct" state) can be compared with the observed machine state. This comparison determines if the machine is functioning correctly, and if it is not, which "machine primitive" is not functioning correctly (fault detection).

Note that this approach to sequential design also permits testing to see what the machine would do if a glitch caused a jump into an unused state. This capability makes the design task of forcing the machine back into a known state much less complex.

The real advantage of this approach is that it requires no changes in architecture, minimal hardware changes, and results in a minimal (5 - 10 percent) area penalty when integrated into existing integrated circuits.

Diagnostic PROMs

Diagnostic PROMs are a relatively minor migration from standard registered PROMs. A block diagram of a diagnostic PROM appears in *Figure 5*. The addition of diagnostic capability to a registered PROM includes the addition of:

- Shadow register
- Multiplexer
- MODE pin
- SDI (Serial Data In) pin
- SDO (Serial Data Out) pin
- Diagnostic clock

The shadow register is dynamically configured, based on the value of the mode signal. If the mode is set to input data to the PROM, the shadow register is configured as serial-in, parallel-out; if you want to extract information from the PROM, the shadow register is configured as a parallel-in, serial-out.

The shadow register thus serves two purposes. First, it can be configured to serially receive state information that will appear at the outputs during the next cycle. This feature allows you to preset a condition to be sent through the part of the system fed by the PROM; i.e., you can insert state information into the system. This feature adds controllability to the system.

The second purpose that the shadow register serves is to allow you to transfer state information from the register and to serially shift that data out of the PROM. This feature adds observability by allowing you to observe the state of the PROM's pipeline register at any given time.

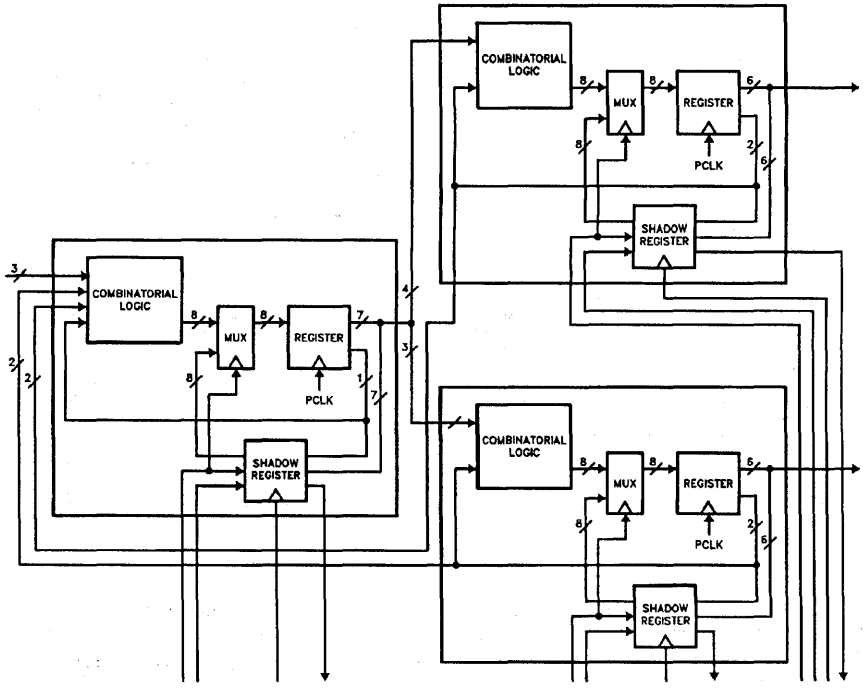


Figure 4. Complex Sequential Machine with Diagnostic Capability

Including the features listed above in a registered PROM can therefore add testability to any system. Note that this increase in function is effected without loss of other desirable registered-PROM features, such as programmable initialization, programmable output enable, etc.

Cypress Diagnostic PROMs

Cypress Semiconductor manufactures two diagnostic PROMs: the CY7C268 and CY7C269. These 64K-byte-wide diagnostic PROMs are manufactured in CMOS for an optimum speed/power tradeoff.

Both PROMs contain an edge-triggered pipeline register and on-chip diagnostic shift register. Both PROMs can withstand 2001V ESD. Both PROMs are produced in Cypress's EPROM-based process, which allows testing for 100-percent programmability. Both PROMs are available in PLCC/LCC and dual-inline packages, and both PROMs are available in a windowed package for reprogrammability.

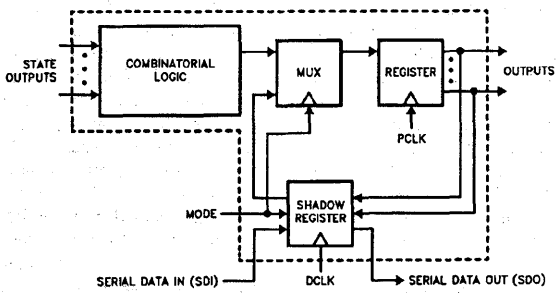


Figure 5. Diagnostic PROM Block Diagram

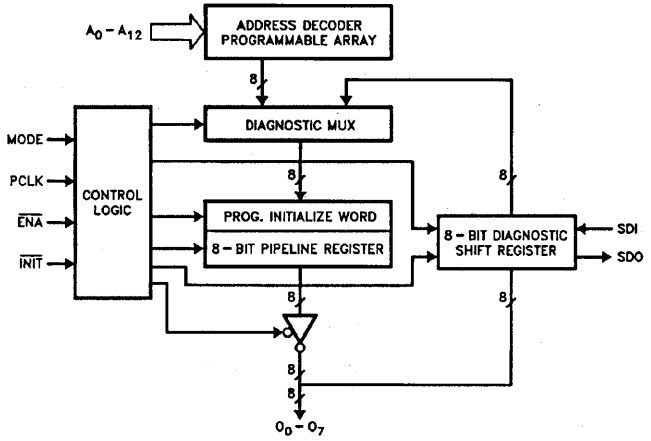


Figure 6. Condensed Block Diagram of the CY7C268

Table 1. CY7C268 Pin Functions

Name	I/O	Function
A_0-A_{12}	I	Address Input
O_0-O_7	O	Data Lines
$\overline{ENA}$	I	Synchronous or Asynchronous Output Enable
$\overline{INIT}$	I	Asynchronous Initialize
MODE	I	Sets PROM to Operate in Pipelined or Diagnostic Mode
DCLK	I	Diagnostic Clock (Used to Clock the Shadow Register)
PCLK	I	Pipeline Clock (Used to Clock the Output Registers)
SDI	I	Serial Data In (Used to Serially Shift Data into the Diagnostic Register)
SDO	O	Serial Data Out (Used to Serially Shift Data Out of the Diagnostic Register)

Table 2. CY7C268 Operational Modes

Data Flow Description	Mode	$\overline{ENA}$[1]	SDI	SDO	DCLK	PCLK
Normal Operation[1]	L	H,L	Data In	SDO	--	Rising Edge
Shadow to Pipeline[1]	H	H,L	X	SDI	--	Rising Edge
Pipeline to Shadow	H	L	L	SDI	Rising Edge	--
Data In to Shadow	H	H	L	SDI	Rising Edge	--
Shift Shadow Reg.[1]	L	H,L	Data In	SDI	Rising Edge	--
No Operation[1]	H	H,L	H	SDI	Rising Edge	--

Note: 1. For the asynchronous-enable operation, data out is enabled on the first Low-to-High clock transition after $\overline{E}$ is brought Low. When $\overline{E}$ goes from Low to High (enable to disable), the outputs go to the high-impedance state after a propagation delay if the asynchronous enable was programmed. If the synchronous enable was selected, a Low-to-High transition is required.

The CY7C268 features full diagnostic capacity and is available in 32-lead PLCC/LCC or 32-pin 0.5-inch DIPs. The CY7C269 features limited diagnostic capability and is available in 28-lead PLCC/LCC or 28-pin 0.3-inch DIPs.

For an in-depth description of the PROMs' functions, refer to the data sheets. The following discussion briefly describes the diagnostic functions available in each device.

CY7C268

A condensed block diagram of the CY7C268 appears in *Figure 6*. *Table 1* lists the pin names and functions of the CY7C268.

Note that full diagnostic capability is realized through the use of four control signals: SDI (Serial Data In), SDO (Serial Data Out), MODE, and DCLK (diagnostic clock). Including both DCLK and PCLK ensures that serial data can be shifted into or out of the diagnostic register while the PROM is operating in normal pipeline fashion. As a result, the CY7C268 has three possible modes of operation:

- Normal (pipelined)
- Diagnostic
- Pipelined and diagnostic simultaneously

Table 2 summarizes the operational modes of the CY7C268.

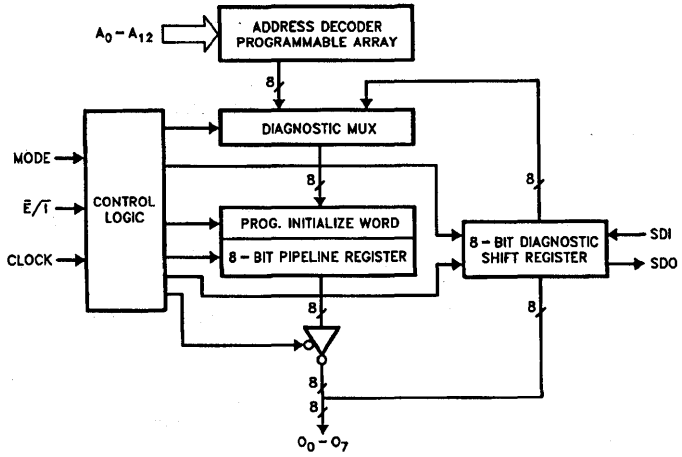


Figure 7. Condensed Block Diagram of the CY7C269

Table 3. CY7C269 Pin Functions

Name	I/O	Function
A_0-A_{12}	I	Address Input
O_0-O_7	O	Data Lines
$\overline{E}$,$\overline{I}$	I	Enable or Initialize
Clock	I	Pipeline and Diagnostic Clock
MODE	I	Sets PROM to Operate in Either Diagnostic or Regular Pipelined Mode
SDI	I	Serial Data In
SDO	O	Serial Data Out

CY7C269

A condensed block diagram of the CY7C269 appears in *Figure 7*. The CY7C269 has reduced diagnostic function relative to the CY7C268. The CY7C269 is ideal for applications requiring limited diagnostics with a premium on board-space conservation. This PROM is available in 28-pin, 300-mil DIPs (windowed or opaque) and in 28-lead PLCC/LCC packages. The pin names and functions of the CY7C269 are listed in *Table 3*.

Note that limited diagnostic capability is realized through inclusion of three diagnostic signals: MODE, SDI, and SDO. Because there is only one clock, the regular and diagnostic modes are mutually exclusive. *Table 4* summarizes the operating modes of the CY7C269.

Design Example

As an example of using diagnostic PROMs, consider the complex sequential machine presented earlier. This machine could be easily implemented using CY7C268s or CY7C269s, as shown in *Figure 8*. Note that the block labeled "diagnostic control" could consist of PLDs, PROMs, a sequencer, or a small microcontroller. Choosing between the CY7C268 and the CY7C269 is based on the complexity of the diagnostic function required. For full diagnostics that can function simultaneously with regular pipelined operation, use the CY7C268. For an application where limited diagnostic capability is required — perhaps only a function at power-up or some other well-defined time — use the CY7C269.

Table 4. CY7C269 Operating Modes

Data Flow Description	Mode	$\overline{E}$,$\overline{I}$	Clock	SDI	SDO
Normal Operation	L	[1],[2]	Rising Edge	X	High Z
Shadow to Pipeline	H	L	Rising Edge	L	SDI
Pipe or Bus to Shadow	H	L	Rising Edge	H	SDI
Shift Shadow	H	H	Rising Edge	Data In	SDO

Notes:
1. The $\overline{E}$ or $\overline{I}$ function is selected during programming.
2. If I is selected, the outputs are always enabled. If $\overline{E}$ is selected, the outputs are enabled synchronously or asynchronously, as programmed.
3. If $\overline{I}$ is selected, the outputs are always enabled. If $\overline{E}$ is selected, during diagnostic operation the data outputs remain in the state they were in when the mode was entered. When enabled, the data outputs reflect the outputs of the pipeline register. Any changes in the data in the pipeline register appear on the output pins.

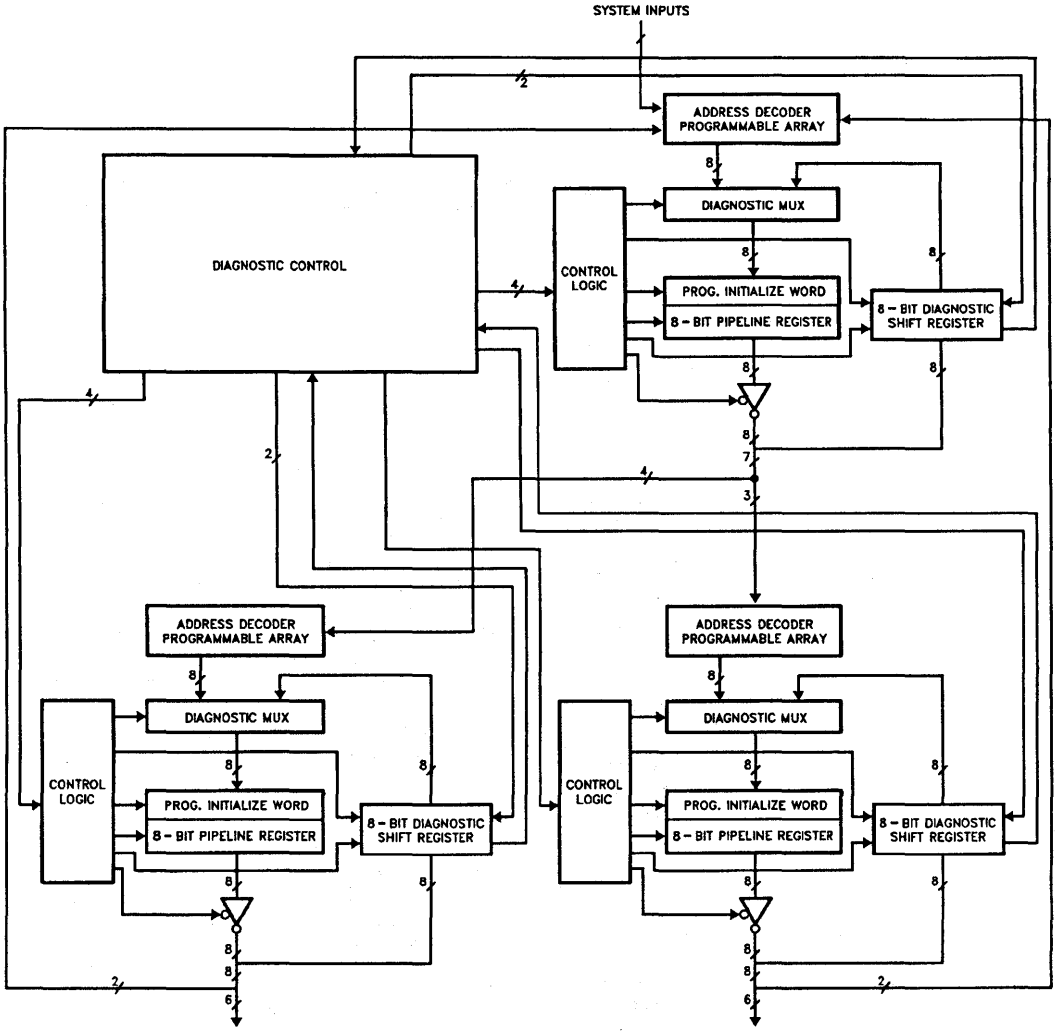


Figure 8. Complex Sequential Machine Implemented with Cypress Diagnostic PROMs

CYPRESS SEMICONDUCTOR

Interfacing the CY7C289 to the AM29000

This application note describes how to use high-speed Cypress CY7C289 PROMs to design an instruction memory system with virtually zero wait states for a 33-MHz AMD AM29000. The design includes 1 Mbyte of CY7C289 PROMs in addition to the interface circuitry used to support processor bursts. A logic schematic and the equations for the PLDs used in the memory interface are included.

Traditionally, PROMs have been much slower than RAMs. System designers used PROMs only for the boot process, immediately transferring the information into RAMs once power-up was complete. This inefficient solution wasted a considerable amount of board space, but system performance was generally considered more important.

The need for this tradeoff is now evaporating. Cypress PROMs have narrowed the speed gap between RAMs and PROMs to almost nothing. The CY7C289 PROMs use a fast-column-access architecture to produce on-page access times of just 20 ns (for registered mode) at a 512-Kbit (64K x 8) density. This architecture takes advantage of the burst mode feature common in many current microprocessors. Because most 32-bit processors burst just 16 bytes in a wrap-around fashion, the burst mode accesses fall within a single page of the CY7C289 PROMs. Thus, each access in a burst to the PROM is always completed in 20 ns.

Even with a processor that generates bursts considerably longer than 16 bytes, the CY7C289 can supply all the data in a burst from a single page. An excellent example of this capability is the 29000 instruction memory design described in this application note. Even though 29000 bursts can be up to 1 Kbyte long, the memory design described here never requires a wait state during a processor burst. Wait states are only required during an initial access, and the maximum wait in a 33-MHz system is just two clock cycles.

Figure 1 displays a block diagram of the instruction memory system design for the 29000. The design has three basic blocks: the 29000 microprocessor, the control logic, and 1 Mbyte of CY7C289 PROM.

CY7C289 PROMs

The CY7C289 is one of four new 64K x 8 reprogrammable PROMs offered by Cypress Semiconductor. Two of these PROMs, including the CY7C289, feature the unique fast-column-access architecture. On these devices, the PROM array is divided into 1024 pages that are each 64 bytes long. Any consecutive access to the same CY7C289 page requires just 20 ns to complete. If an access crosses an internal PROM page, the device delivers data within 65 ns. To indicate an internal page crossing to the external circuitry, the CY7C289 generates a WAIT\ signal.

Along with the unique array architecture, the CY7C289 provides a variety of programmable features to simplify the memory interface. Among these programmable features is the ability to capture the input address with on-chip registers or latches.

If you select the address latch option, the address flows into the PROM during the active portion of the ALE signal and is captured when ALE is deasserted (the ALE signal's polarity is programmable). This option is appropriate for most CISC processors, which supply a valid address after the system clock's rising edge. The ALE option can improve system performance by allowing the PROM to capture the address as soon as it becomes available, as opposed to waiting for the system clock's next rising edge. The drawback to the address latch option is that external logic might be required to generate the ALE signal.

If you select the CY7C289's registered option, the address at the input is captured at the CLK input's rising edge. The advantage of the registered mode is that the memory interface is often simpler. This configuration is particularly useful when interfacing to RISC processors. Most of these processors generate addresses around the rising edge of a system clock, making it easy to capture the address with the CY7C289 input registers. (See the application note, "Interfacing the CY7C289 to the CY7C601.")

Another important CY7C289 feature is the ability to program the polarity of two chip selects (CS1 and

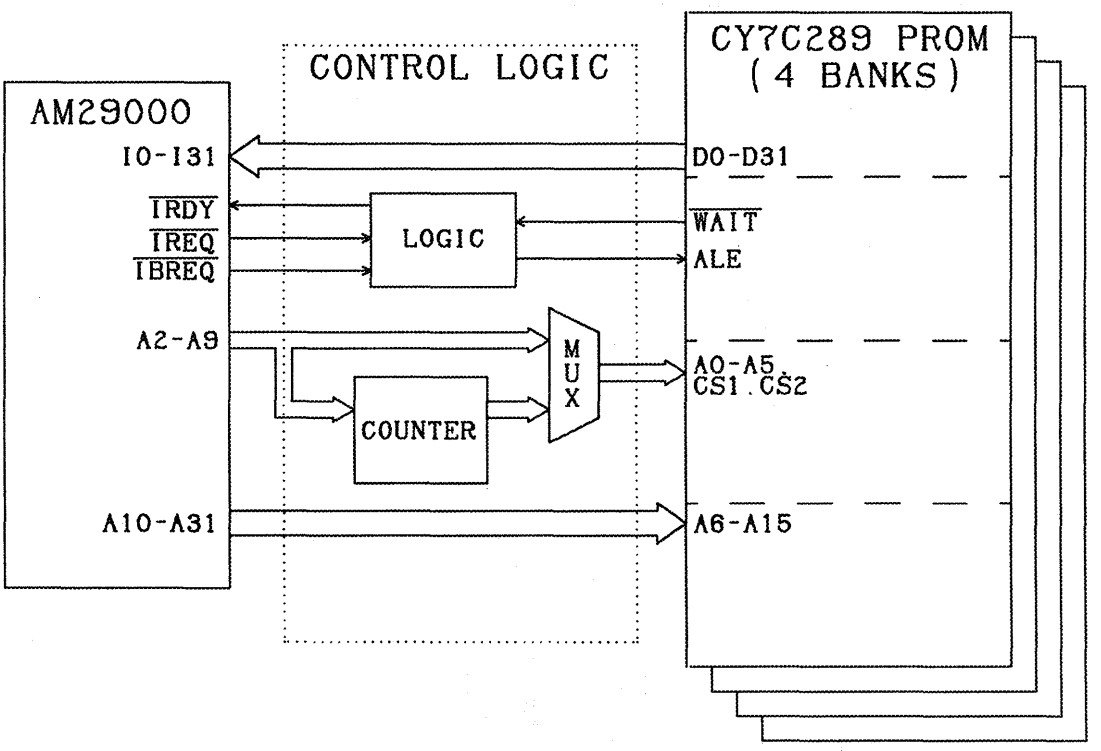


Figure 1. AM29000 Instruction Memory Block Diagram

CS2), which facilitates automatic bank selection for up to four banks of PROM. Proper use of the chip selects also allows you to extend the PROM page's length beyond 64 words when using multiple banks of PROM. This capability improves the system's performance by effectively increasing the size of a page in the CY7C289s (more on this later).

Here is a complete list of the programmable features available on the CY7C289:

- The input address can be either registered at CLK's rising edge or latched by the ALE input.
- You can program the address set-up and hold window.
- You can program the WAIT output's polarity.
- You can program the ALE input's polarity.
- The WAIT output can be generated off the falling or rising edge of CLK for the registered-mode CY7C289.
- You can program the polarity of both chip selects (CS1 and CS2).

You can set each of these options by appropriately programming a reserved PROM location. Therefore, the devices are configured at the same time the array is programmed.

AM29000 Microprocessor

The 29000 is a 32-bit general-purpose microprocessor used mainly in embedded controller applications. The version used in this design operates at 33 MHz—the highest-speed 29000 currently available. The processor's pipelined RISC architecture attempts to execute an instruction in every clock cycle. To do this, the 29000 relies heavily on burst-mode accesses.

The 29000 contains three buses, one each for address, data, and instruction. During a normal access, the three-bus architecture behaves essentially like a two-bus system (address and data), because the dedicated instruction and data buses must wait for the shared address bus. In burst mode, however, only the initial data

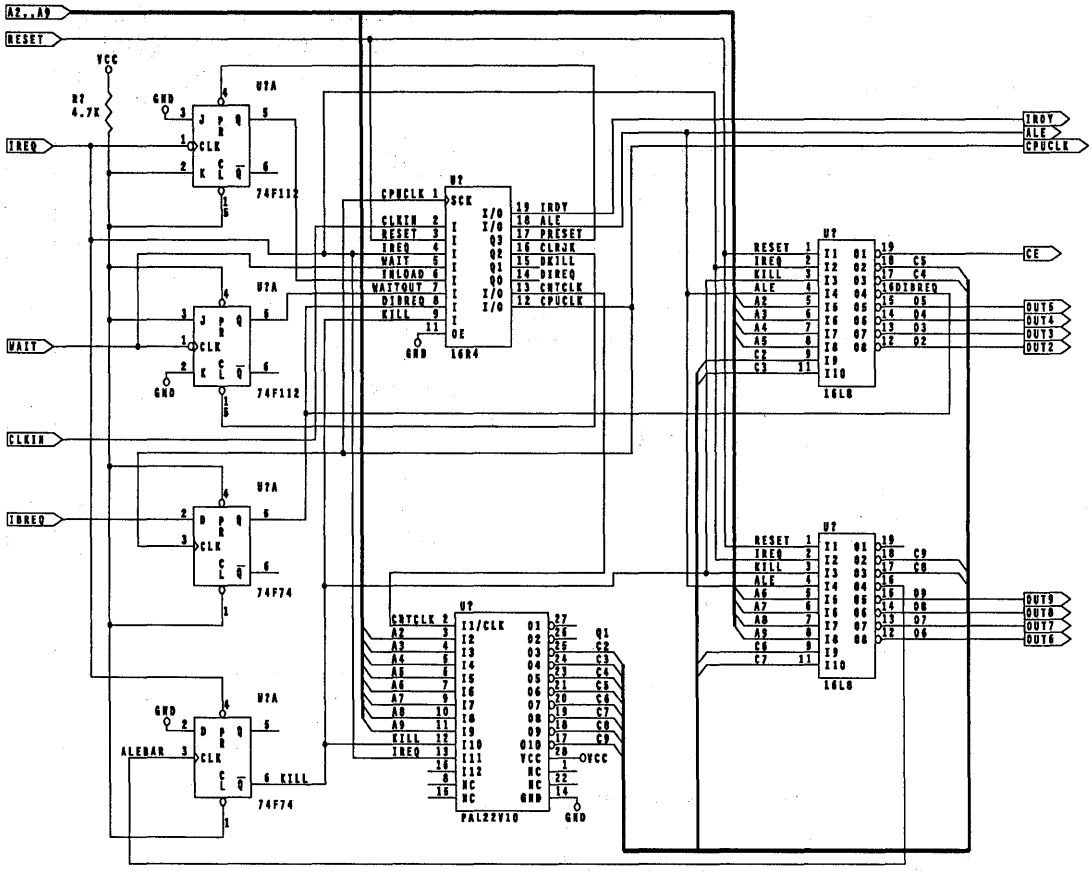


Figure 2. Control Logic for the 29000 Instruction Memory

or instruction address is sent to the corresponding memory space, and the task of incrementing the address is left to the interface circuitry. Thus, during burst, both the data and instruction buses can operate simultaneously without having to wait for the shared address bus. In other words, a 29000 in burst mode can fully utilize the separate data and instruction bus architecture.

Although the 29000 achieves maximum throughput during bursts, AMD did impose a limit on a burst's length: The 29000 only performs bursts within a 1-Kbyte boundary. Therefore, an 8-bit counter suffices to increment the burst addresses. Note that a 29000's maximum burst length depends on where it begins. For example, if a burst begins four words before the end of a 1-Kbyte boundary, the burst can at most be four words long.

Control Logic

The memory interface logic required for this design is detailed in *Figure2* and appears symbolically in *Figure 3*. In addition, PLD ToolKit source code for the PLDs used in the control logic appears in *Appendices A* through *D*.

Referring to the block diagram in *Figure 1*, note that the interface circuitry performs two primary functions. One is to generate all necessary interface signals, and the other is to increment the instruction address to

support processor bursts. The hardware required to implement the interface consists of two SSI devices (a 74F74 and a 74F112) and four small PLDs. The 22V10 PLD is a 15-ns version, while the remaining three PLDs (one 16R4 and two 16L8s) have a maximum propagation delay of 5 ns.

Memory Interface

The 29000 has a few peculiarities that affect the memory system design. For example, the instruction bus is unidirectional. *The 29000 can only READ from instruction memory.* This limitation makes it difficult to use RAMs for instruction memory, because there is no mechanism to load the instructions into the RAMs to begin with, but the nonvolatile nature of PROMs makes them ideal for this application.

One way to use RAM for the instruction memory is to trick the 29000 into thinking it is writing to data memory (the data bus is bidirectional), but route the information back to the RAMs in instruction memory. Implementing this memory subsystem requires two 32-bit 2:1 multiplexers on the data and instruction buses, in addition to the associated glue logic necessary to control the transfer. To use the memory subsystem, the system copies the instruction information (from boot PROMs located elsewhere on the board) onto the data bus and subsequently into the RAMs on the instruction side of memory. This solution is costly, wastes board space, and slows system operation by adding multiplexer delays into both the instruction- and data-bus paths.

A much better solution is to use PROMs in the instruction memory. Because they are nonvolatile, the instruction information is programmed into the device prior to assembling the system, eliminating the extensive logic needed to write to the instruction bus. Further, with the CY7C289's high speed, the system has no need for shadow RAMs. The resulting circuit occupies much less board space than the RAM-based version and provides better system performance. Moreover, lessening the number of components improves the circuit's reliability.

Another unusual 29000 feature is the processor's ability to *suspend* bursts to instruction memory. At any time during an instruction burst, the 29000 can suspend the sequence by deasserting the burst request signal (IBREQ\). The instruction memory must respond by discontinuing its operation while the IBREQ\ signal is inactive. When the processor reasserts IBREQ\, the memory system must resume from the point at which the burst was suspended. Note that because the 29000 does not send a new address at this point, the interface logic has to remember the address at which the processor suspended the burst. An instruction burst is not complete until the 29000 asserts the instruction request signal (IREQ\) and sends a new address. The interface logic described in this design fully supports suspended bursts.

PROM Configuration

In this application, 16 CY7C289 PROMs constitute a 1-Mbyte instruction memory, distributed in four banks. The CY7C289s' 22-ns access time (in latch mode) allows on-page accesses to complete in a single clock cycle at 33 MHz. Proper use of the programmable chip selects ensures that all burst accesses fall within the same PROM page and never require a processor wait cycle.

The CY7C289s are configured with address latches to take full advantage of the 29000's mid-clock address release. Latch mode minimizes the number of wait cycles during a single access or during a burst's first access. The set-up and hold window for the address latches should be programmed to minimize the hold time required after latch close. This setting is critical to proper operation of the address increment circuitry.

The CY7C289s' chip selects are programmed on a bank-to-bank basis, such that each bank has a unique polarity combination of CS1 and CS2. This arrangement permits PROM bank selection without external address decoding. The other applicable programmable features on the CY7C289 are the polarity of the WAIT\ and ALE signals. In the design implemented here, WAIT\ is active Low and ALE is active High.

Address Connection Scheme

For the most part, the placement of the CY7C289 PROMs in this design is straightforward. However, there are two important memory design features that bear clarification. The first is the address connection scheme used for the CY7C289 PROMs. In *Figure 3*'s display of the address input to the CY7C289s, notice that the addresses fed to the PROMs are not entirely sequential. This non-sequential addressing scheme is used with the chip selects to extend the effective PROM page length to 1 Kbyte, and thus achieve no-wait-state burst performance.

To understand how this is done, consider some internal details of the CY7C289. In this PROM, the lowest six address inputs (A0 - A5) designate a specific byte within a 64-byte internal PROM page. Inputs A6 - A15 select one of 1024 PROM pages. When any of the inputs at pins A6 - A15 changes, a new page is selected, and the CY7C289 asserts the WAIT\ output.

You can think of the CY7C289's chip selects (CS1 and CS2) as additional address inputs in a multi-bank memory system. Like A0 - A5, changes at these inputs do not result in an internal CY7C289 change. With four banks of PROM, you have a total of 8 address bits (A0 - A5, CS1, and CS2) that do not affect the internal PROM page, as opposed to just 6 (A0 - A5) when using one bank of PROM. The 8 bits of on-page addresses translate into a PROM page length of 256 words, or 1 Kbyte, which equals the 29000's maximum possible burst.

The schematic in *Figure 3* reveals how this page-lengthening scheme is implemented. Note that all the

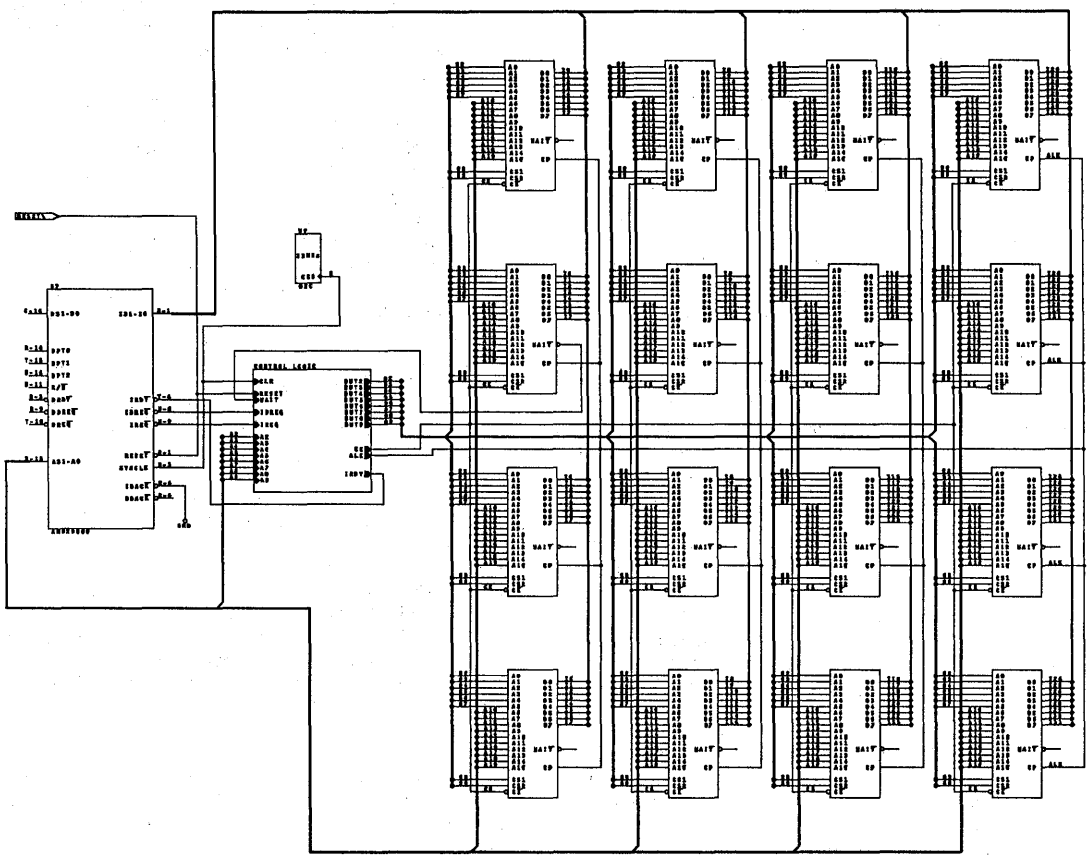


Figure 3. AM29000 Instruction Memory Design

outputs from the control logic (O2 - O9) connect to CY7C289 inputs that do not cause a page change (A0 - A5, CS1, and CS2). The lowest address connected directly from the CPU to the PROMs is A10. The 29000 is guaranteed never to change A10 - A31 during a burst, because this would constitute crossing a 1-Kbyte boundary. All the addresses that can change during burst connect to A0 - A5, CS1, and CS2; thus, the CY7C289 never crosses an internal page—and never causes a wait state—during a 29000 burst. The chip selects in this design effectively quadruple the PROM page length, allowing a greater percentage of single accesses and *all burst accesses* to finish within a single clock cycle.

To make the extended page useful, note that you need to locate sequential code on the same PROM page. Because this design extends each PROM page across all four banks, you must segment code into page-length blocks; this is analogous to using interleaved DRAMs. Because each CY7C289 PROM has a 64-byte internal page, your code must be separated into 64-

word blocks. In other words, place the first 64 words of code in bank 1, the next 64 words in bank 2, and so on. You can accomplish this segmentation with a simple program.

Using the WAIT\ Signal

The second memory design issue that bears clarification is the connection of the CY7C289's WAIT\ signal. The CY7C289 asserts this signal when the input address crosses an internal page boundary (at least one of the inputs A6 - A15 changes). WAIT\ tells the 29000 that the PROMs need an additional clock cycle to deliver the requested instruction.

Note in the schematic in *Figure 3* that only one WAIT\ output connects to the control logic. This is because all the PROMs examine the same upper-order address inputs to determine if an internal page has been crossed. Therefore, only one PROM is required to identify a page crossing and assert the WAIT\ signal, even if the chip selects (CS1 and CS2) are deasserted at the time. The only time the PROM does not generate WAIT\ is when the chip enable signal (CE\) is inactive during an address change.

Burst Counter

The 22V10 and the two 16L8s support the 29000's burst mode capability. The 22V10 implements an 8-bit loadable counter, which loads a new address from the 29000 when the IREQ\ signal is asserted. On each subsequent clock rise in which IBREQ\ is active, the 22V10 increments the current address and delivers the result to a multiplexer. Note that the clock to the 22V10 is not the system clock. The 16R4 generates a special counter clock that properly times the loading of the counter and halts the count during a suspended burst.

The pair of 16L8s are utilized primarily as two high-speed multiplexers. Each 16L8 implements a 4-bit 2:1 multiplexer that selects the instruction address from the 29000 or the counter. During an initial access (IREQ\ Low), the 16L8s feed the processor address to the instruction memory. When the 29000 is bursting, the counter address is routed to the PROMs.

Signal Generation

The primary function of the remaining interface logic (the 16R4, 74F74, and 74F112) is to generate the necessary system control signals. These signals include the instruction ready signal (IRDY\) to the 29000 and the address latch enable signal (ALE) for the PROMs. The IRDY\ input to the 29000 halts the processor when accessing slower memory. Based on the WAIT\ output from the CY7C289s, the interface circuitry deasserts IRDY\ when the PROM requires more time to complete an access. Because this design never requires a wait during a burst access, the control logic simply holds IRDY\ Low while a burst is in progress. For the PROM interface, IRDY\ is only used during a single access or during a burst's initial access.

The ALE input controls the input of addresses to the CY7C289s. The CY7C289's latch mode takes full advantage of the 29000's mid-clock address release and minimizes wait states during the initial access. The drawback to latch mode is that an ALE signal must be generated externally. In this design, the 16R4 creates ALE based on the input clock, IRDY\, WAIT\, and IBREQ\. The remaining signals generated by the 16R4 control the burst counter logic implemented in the 22V10 and the 16L8s.

Note that most of the logic displayed in *Figure 2* is required regardless of the memory device you choose. Implementing the burst-counter interface to the 29000 requires the 22V10, both 16L8s, and a portion of the 16R4. Thus, only the two SSI components and part of one 16R4 are needed to create the appropriate communication signals between the 29000 and the CY7C289 PROMs.

System Timing

Figures 4 and *5* illustrate the communication between processor and memory that supports burst mode and inserts wait states. The 29000 generates the instruction request (IREQ\) and instruction burst request (IBREQ\) signals to initiate instruction accesses. To begin an access, the 29000 asserts IREQ\ and places a valid address on the address bus a maximum of 12 ns after CLK's rising edge. If this is the beginning of an instruction burst, the processor asserts IBREQ\ no more than 10 ns after the system clock's falling edge. At each subsequent rising edge of CLK, the 29000 samples the instruction ready (IRDY\) input before reading data. Therefore, by deasserting IRDY\, the external memory system can hold the CPU until an access is completed.

When the access is finished, the memory system must assert IRDY\ at least 10 ns before CLK's next rising edge. The data must appear on the bus at least 4 ns before CLK's rising edge.

No-Wait Timing

The control logic in this design generates IRDY\ based on the WAIT\ output from the CY7C289 PROMs and the IREQ\ and IBREQ\ signals from the processor. During a single access or a burst's first access, the interface automatically inserts one wait cycle due to the 29000's late delivery of valid address; completing a single access without a wait state would require a 12-ns PROM access time. The interface inserts the wait state by deasserting IRDY\ in the cycle in which IREQ\ was asserted. In the scenario illustrated in *Figure 4*, this access falls on the same page as the previous PROM access and therefore does not generate a WAIT\. The interface logic asserts IRDY\ in the following cycle, and data is delivered prior to CLK's next rising edge. The CY7C289 PROMs' 22-ns on-page access time is well within the 44-ns window that results from the single inserted wait state.

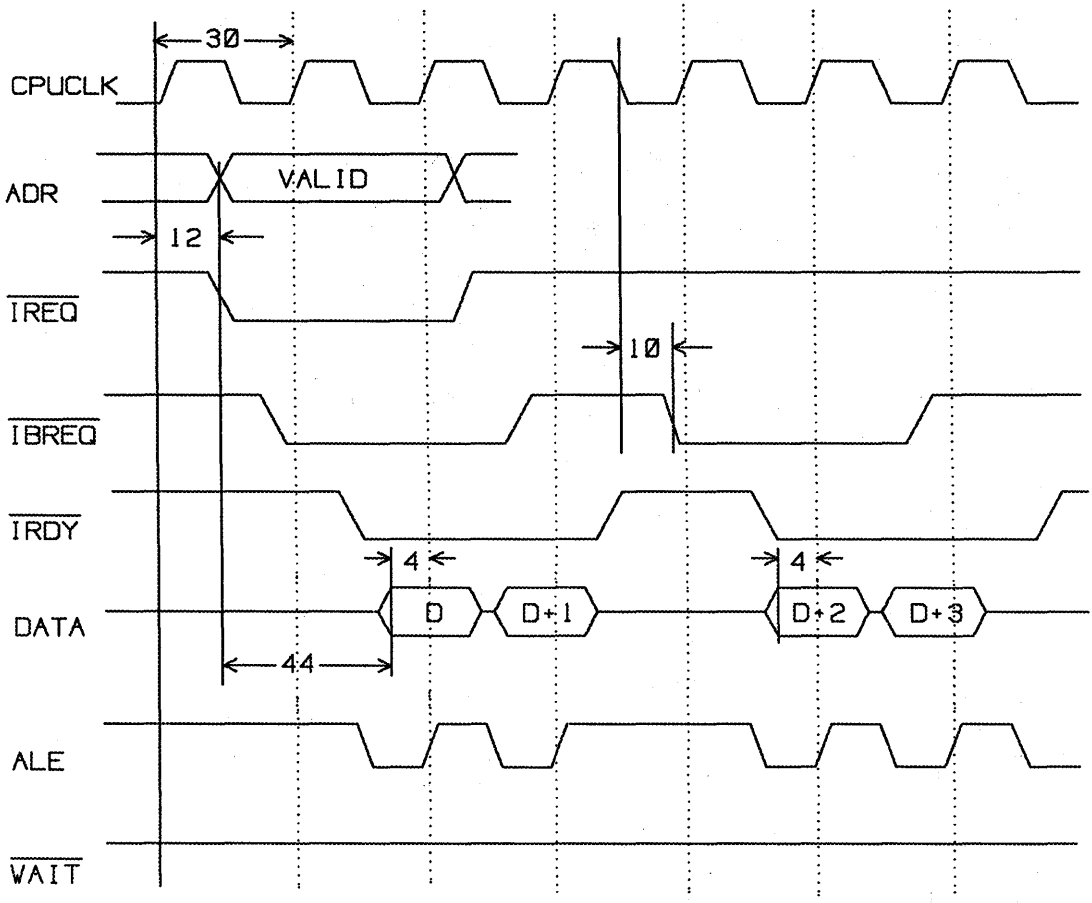


Figure 4. Instruction Memory Timing (WAIT deasserted)

Once the initial access is delivered, the memory can complete each burst access within a single cycle. The control logic therefore keeps IRDY\ asserted as long as the IBREQ\ signal from the CPU is active. In *Figure4*, note that the 29000 temporarily deasserts IBREQ\—the method the processor uses to suspend an instruction burst. In response, the instruction memory suspends data delivery until the IBREQ\ is reasserted. When IBREQ\ reasserts, the data is delivered from the point at which the burst was suspended, as illustrated in the timing diagram in *Figure4*.

To govern the operation of the instruction PROMs, the control logic generates the address latch enable signal (ALE), also shown in *Figure 4*. In this design, the ALE input is programmed as active High. Thus, when the ALE input is active, the latch is transparent, and the address at the input flows into the PROM. On the transition of ALE from High to Low, the PROMs latch the address and ignore further changes to the address while ALE is Low.

In this design, the ALE input remains active (open) until a burst sequence begins. During a burst, the ALE signal advances the counter and controls the loading of the counter address into the PROM. Because ALE's falling edge increments the count, the PROM's address inputs change only after the address latch closes.

Note in the schematic in *Figure 3* that the 16R4 generates the clock input to the AM29000. This clock arrangement ensures that the ALE and CPUCLK sig-

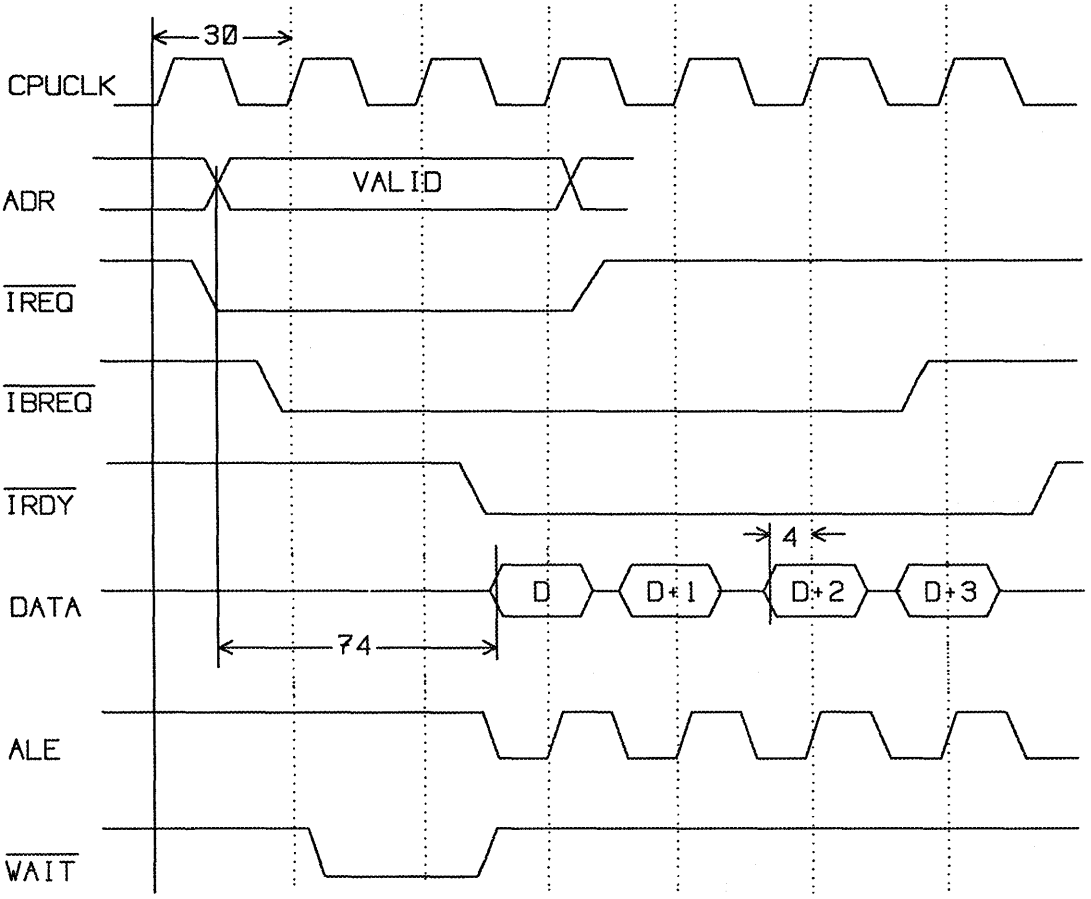


Figure 5. Instruction Memory Timing (WAIT asserted)

nals track each other and are as closely synchronized as possible.

PROM Wait Timing

If WAIT\ is asserted during a single access or during the initial access of a burst, the control logic inserts one additional wait cycle (*Figure 5*). This wait cycle occurs if a PROM address crosses a page boundary; the WAIT\ signal is then asserted a maximum of 21 ns after the address is loaded. The control logic displayed in *Figure2* uses this WAIT\ output's falling edge to send an additional wait signal to the 29000. This wait signal is created by keeping the IRDY\ signal High for one additional cycle.

As shown in *Figure 5*, this added wait provides a total of 74 ns for the PROM to complete the access. An access that involves crossing an internal PROM page actually requires only 65 ns. Note once again that after the initial data has been delivered, all subsequent burst accesses are delivered within a single clock cycle.

Appendix A. PLD Toolkit Source Code for the 16R4

```
C16R4;

{ Norman Taffe
 Cypress Semiconductor
 April 23, 1990
 Control Logic for CY7C289 PROM interface to the AMD 29000. }


CONFIGURE

{inputs}

 CLK, CLKIN, RESET, IREQ, WAIT, INLOAD, WAITOUT, DIBREQ, KILL, OE(node= 11),

{outputs}

 IRDY(node= 12), ALE, PRESET, CLRJK, DKILL, DIREQ, COUNTCLK, CPUCLK,


EQUATIONS;

 !PRESET =    < sum>  !INLOAD & !RESET;

 !DIREQ =     < sum>   !IREQ & !RESET;

 !ALE =       < oe>
              < sum>  !DIREQ & !WAITOUT & !CLKIN & !RESET
                     # !DIBREQ & !WAITOUT & !CLKIN & !RESET;

 !CPUCLK =    < oe>
              < sum>  !CLKIN & !CLKIN & !CLKIN & !CLKIN
                     # !CLKIN & !CLKIN & !CLKIN & !CLKIN;

 !IRDY =      < oe>
              < sum>  WAIT & !WAITOUT & !DIBREQ & !RESET
                     # WAIT & !WAITOUT & !PRESET & !RESET
                     # !WAITOUT & !DIBREQ & !DKILL & !RESET
                     # !WAITOUT & !PRESET & !DKILL & !RESET
                     # !CLRJK & !RESET;

 !CLRJK =     < sum>  !PRESET & WAITOUT & !RESET;

 !COUNTCLK =          < oe>
                      < sum>  !WAITOUT & DIBREQ & PRESET & CLRJK & DIREQ
                             # CLKIN & !WAITOUT & PRESET & CLRJK;

 !DKILL =     < sum>  !CLRJK & !RESET;
```

Appendix B. PLD Toolkit Source Code for the Upper 16L8

```
C16L8;

{ Norman Taffe
 Cypress Semiconductor
 April 23, 1990
 Control Logic for 285/9 PROM interface to AMD 29000. }

CONFIGURE;

{inputs}

 RESET, IREQ, KILL, ALE, A2, A3, A4, A5, C2, C3(node= 11),
 DIBREQ(node= 16), C4, C5,

{outputs}

 O2(node= 12), O3, O4, O5, CE(node= 19),


EQUATIONS;

  !O2 =  < oe>
         < sum>  !RESET & !A2
                 # !IREQ & !KILL & ALE & !A2
                 # RESET & !ALE & !C2
                 # RESET & KILL & !C2
                 # RESET & IREQ & !C2
                 # !A2 & !C2;

  !O3 =  < oe>
         < sum>  !RESET & !A3
                 # !IREQ & !KILL & ALE & !A3
                 # RESET & !ALE & !C3
                 # RESET & KILL & !C3
                 # RESET & IREQ & !C3
                 # !A3 & !C3;

  !O4 =  < oe>
         < sum>  !RESET & !A4
                 # !IREQ & !KILL & ALE & !A4
                 # RESET & !ALE & !C4
                 # RESET & KILL & !C4
                 # RESET & IREQ & !C4
                 # !A4 & !C4;

  !O5 =  < oe>
         < sum>  !RESET & !A5
                 # !IREQ & !KILL & ALE & !A5
                 # RESET & !ALE & !C5
                 # RESET & KILL & !C5
                 # RESET & IREQ & !C5
                 # !A5 & !C5;

  !CE =  < oe>
          < sum>  !IREQ #  !DIBREQ;
```

Appendix C. PLD Toolkit Source Code for the Lower 16L8

```
C16L8;

{ Norman Taffe
 Cypress Semiconductor      April 23, 1990
 Control Logic for 285/9 PROM interface to AMD 29000. }

CONFIGURE;

{inputs}

 RESET, IREQ, KILL, ALE, A6, A7, A8, A9, C6, C7(node= 11), C8(node= 17), C9,

{outputs}

 O6(node= 12), O7, O8, O9, ALEBAR,


 EQUATIONS;

 !O6 =  < oe>
        < sum>  !RESET & !A6
                # !IREQ & !KILL & ALE & !A6
                # RESET & !ALE & !C6
                # RESET & KILL & !C6
                # RESET & IREQ & !C6
                # !A6 & !C6;


 !O7 =  < oe>
        < sum>  !RESET & !A7
                # !IREQ & !KILL & ALE & !A7
                # RESET & !ALE & !C7
                # RESET & KILL & !C7
                # RESET & IREQ & !C7
                # !A7 & !C7;


 !O8 =  < oe>
        < sum>  !RESET & !A8
                # !IREQ & !KILL & ALE & !A8
                # RESET & !ALE & !C8
                # RESET & KILL & !C8
                # RESET & IREQ & !C8
                # !A8 & !C8;


 !O9 =  < oe>
        < sum>  !RESET & !A9
                # !IREQ & !KILL & ALE & !A9
                # RESET & !ALE & !C9
                # RESET & KILL & !C9
                # RESET & IREQ & !C9
                # !A9 & !C9;

 !ALEBAR =  < oe>
            < sum> ALE;
```

Appendix D. PLD Toolkit Source Code for the 22V10

```
C22V10;

{ Norman Taffe
  Cypress Semiconductor
  April 25, 1990
  8-bit counter for AMD29000 PROM interface. }


CONFIGURE;

{inputs}

CLK,A2,A3,A4,A5,A6,A7,A8,A9,KILL,IREQ,

{outputs}

O9(node= 14),O8,O7,O6,O5,O4,O3,O2,Q1(noreg),


EQUATIONS;

   !Q1 :=  < sum>  !KILL & !IREQ;

   !O9 :=  < oe>
          < sum>  O2 & O3 & O4 & O5 & O9 & O8 & O7 & O6 & Q1
                # !O2 & !O9 & Q1
                # !O3 & !O9 & Q1
                # !O4 & !O9 & Q1
                # !O5 & !O9 & Q1
                # !O9 & !O6 & Q1
                # !O9 & !O7 & Q1
                # !O9 & !O8 & Q1
                # A2 & A3 & A4 & A5 & A6 & A7 & A8 & A9 & !Q1
                # !A2 & !A9 & !Q1
                # !A3 & !A9 & !Q1
                # !A4 & !A9 & !Q1
                # !A5 & !A9 & !Q1
                # !A6 & !A9 & !Q1
                # !A7 & !A9 & !Q1
                # !A8 & !A9 & !Q1;

  !O8 :=  < oe>
          < sum>  O2 & O3 & O4 & O5 & O8 & O7 & O6 & Q1
                # !O2 & !O8 & Q1
                # !O3 & !O8 & Q1
                # !O4 & !O8 & Q1
                # !O5 & !O8 & Q1
                # !O8 & !O6 & Q1
                # !O8 & !O7 & Q1
                # A2 & A3 & A4 & A5 & A6 & A7 & A8 & !Q1
                # !A2 & !A8 & !Q1
                # !A3 & !A8 & !Q1
                # !A4 & !A8 & !Q1
                # !A5 & !A8 & !Q1
                # !A6 & !A8 & !Q1
                # !A7 & !A8 & !Q1;
```

Appendix D. PLD Toolkit Source Code for the 22V10 (cont.)

```
!O7 :=  < oe>
        < sum>  O2 & O3 & O4 & O5 & O7 & O6 & Q1
             #  !O2 & !O7 & Q1
             #  !O3 & !O7 & Q1
             #  !O4 & !O7 & Q1
             #  !O5 & !O7 & Q1
             #  !O7 & !O6 & Q1
             #  A2 & A3 & A4 & A5 & A6 & A7 & !Q1
             #  !A2 & !A7 & !Q1
             #  !A3 & !A7 & !Q1
             #  !A4 & !A7 & !Q1
             #  !A5 & !A7 & !Q1
             #  !A6 & !A7 & !Q1;


!O6 :=  < oe>
        < sum>  O2 & O3 & O4 & O5 & O6 & Q1
             #  !O2 & !O6 & Q1
             #  !O3 & !O6 & Q1
             #  !O4 & !O6 & Q1
             #  !O5 & !O6 & Q1
             #  A2 & A3 & A4 & A5 & A6 & !Q1
             #  !A2 & !A6 & !Q1
             #  !A3 & !A6 & !Q1
             #  !A4 & !A6 & !Q1
             #  !A5 & !A6 & !Q1;


!O5 :=  < oe>
        < sum>  O2 & O3 & O4 & O5 & Q1
             #  !O2 & !O5 & Q1
             #  !O3 & !O5 & Q1
             #  !O4 & !O5 & Q1
             #  A2 & A3 & A4 & A5 & !Q1
             #  !A2 & !A5 & !Q1
             #  !A3 & !A5 & !Q1
             #  !A4 & !A5 & !Q1;


!O4 :=  < oe>
        < sum>  O2 & O3 & O4 & Q1
             #  !O2 & !O4 & Q1
             #  !O3 & !O4 & Q1
             #  A2 & A3 & A4 & !Q1
             #  !A2 & !A4 & !Q1
             #  !A3 & !A4 & !Q1;


!O3 :=  < oe>
        < sum>  O2 & O3 & Q1 #  !O2 & !O3 & Q1 #  A2 & A3 & !Q1
             #  !A2 & !A3 & !Q1;

!O2 :=  < oe>
        < sum> O2 & Q1 #  A2 & !Q1;
```

CYPRESS SEMICONDUCTOR

Interfacing the CY7C289 to the CY7C601

This application note describes how to use high-speed CY7C289 PROMs to design an instruction memory for a 40-MHz CY7C601 RISC processor. The design features 1 Mbyte of PROM and requires no interface circuitry. Utilizing a unique fast-column-access architecture, the CY7C289 supplies data in a 40-MHz system with only occasional wait states. A schematic of the design is included at the end of this application note.

Because microprocessor performance improvements have outpaced access-time advances in high-density memory devices, system designers have resorted to memory interleaving and high-speed SRAM caches to more fully utilize a processor's performance capability. In embedded control applications, the alternative has been to compromise system performance by slowing every processor access to PROM memory with wait states or by using PROMs only for the boot process and running instruction code from SRAMs. The necessity for faster, nonvolatile memory in high-performance embedded applications has prompted Cypress to design high-speed PROMs that you can easily interface to a variety of microprocessors.

Using the CY7C289, high-speed embedded applications can run code directly from PROM and eliminate the extra board space, cost, and logic required to transfer code into "shadow" RAMs. To achieve this level of performance, the CY7C289 PROMs employ an innovative architecture that accentuates local speed. The memory array is split into 64-byte pages that allow on-page access times of just 20 ns in a 512-kbit (64K x 8) PROM. This performance equals that of the fastest static RAMs at similar densities. SRAM-like performance, combined with the non-volatility of EPROM technology, makes these devices ideal for high-performance embedded control applications.

Another important CY7C289 feature is the availability of on-chip address registers. The CY7C601 memory design presented in this application note is an example of the address registers' usefulness. Like many RISC architectures, the CY7C601 delivers its address and memory signals unlatched prior to the system clock's rising edge. Ordinarily, you must latch these signals externally with several 74F74s or the like. However, the CY7C289's on-chip registers capture the address bits at the system clock's rising edge. This feature, as well as the CY7C289's automatic WAIT-signal generation, allow for a straightforward connection between the memory and the processor.

Figure1 displays a block diagram of the instruction memory system design for the CY7C601. As the diagram shows, the design has only two major components: the CY7C601 32-Bit RISC Processor and one Mbyte of CY7C289 PROM.

CY7C289 PROMs

The CY7C289 is part of a high-density (512K), high-speed CMOS PROM family offered by Cypress Semiconductor. The CY7C289, along with another of the family members, features a unique fast-column-access architecture. The PROM array is arranged into 1024 pages, each 64 bytes long. Consecutive accesses to the same page require only 20 ns to complete. When an access crosses a page within the PROM, the data is delivered in 65 ns. The 7C289 generates a WAIT signal to alert external circuitry of an off-page access.

The CY7C289 emphasizes fast local accesses—within a 64-byte page. The principle behind the CY7C289 derives from a statistical approach to performance improvement. Many microprocessors linearize memory access requests because of on-chip cache burst-fill modes or instruction pre-fetch queues, in effect localizing the instruction fetch sequences. In the CY7C289, Cypress uses the fast-column-access architecture to improve local performance and take advantage of instruction stream linearity and locality. Fast access is possible when consecutive PROM retrievals are within the current page.

When a memory cycle requests data that is not on the current page, the chip must power up the correct page. Because processor code tends to be linear in nature, though, PROM accesses usually fall on the same PROM page and therefore require only 20 ns to complete.

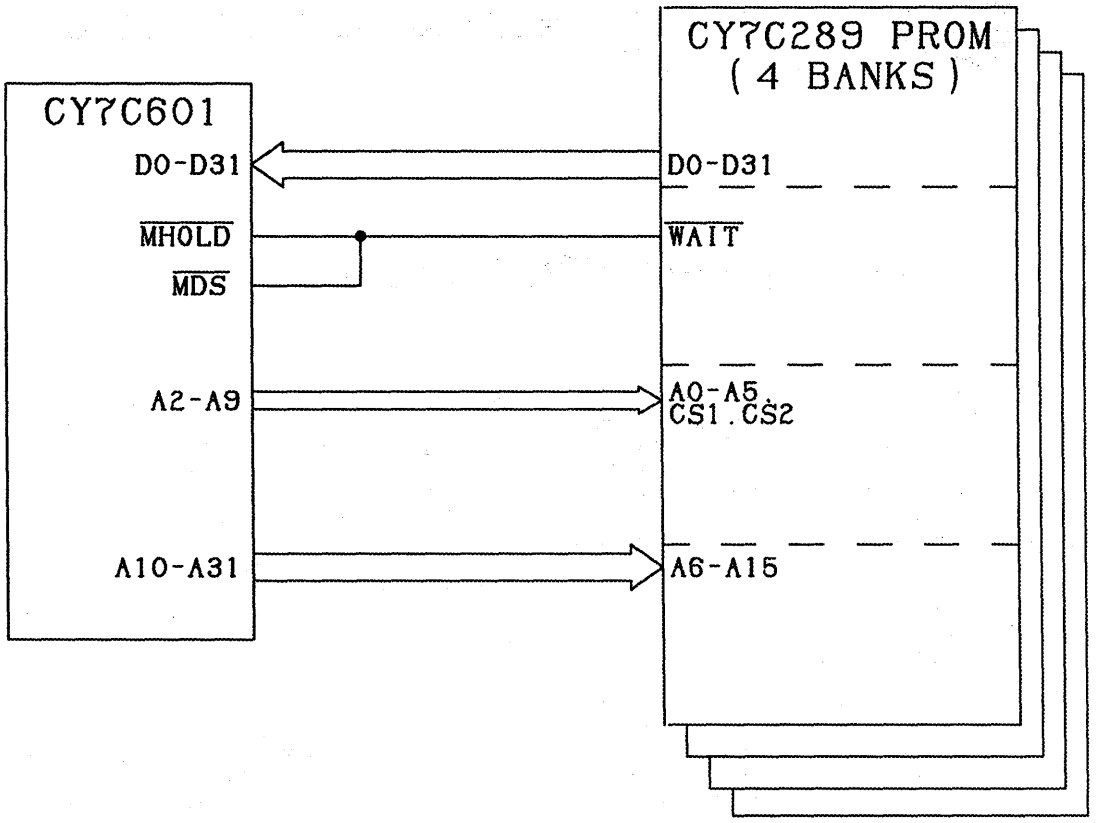


Figure 1. Block Diagram of CY7C601 Memory Design

Along with the unique array architecture, the CY7C289 simplifies system design by providing the on-chip logic necessary to generate a WAIT signal. This signal is used to automatically insert microprocessor wait states during an off-page access.

To simplify the memory interface with a variety of microprocessors, the CY7C289 contains a rich set of programmable features. For example, you can latch the input address with the ALE input or register the address at CLK's rising edge. The CY7C289 provides a programmable bit to select between latched and registered address inputs. The default is registered inputs, which samples the address on CLK's rising edge and captures the address in the address register. This configuration suits most RISC processors, which generate addresses around the system clock's rising edge.

When in LATCH mode while the ALE pin is active, the PROM recognizes any address changes and latches the address into the address registers on the user-defined edge of ALE. This option is particularly useful when interfacing with CISC processors (see *Reference*). Most CISC processors generate a valid address some time following the system clock's rising edge. Instead of waiting for the next rising clock edge (and sacrificing performance), you can capture the address immediately using the ALE input. The drawback to LATCH mode is that it might require external interface circuitry. If you do select the ALE function, you can define the ALE signal's polarity, with the default being positive.

To eliminate external bank decoders, the CY7C289 includes two programmable chip selects (CS1 and CS2). The polarity of these inputs is user programmable, facilitating automatic bank selection of up to four banks of PROM. The programmable chip selects provide an additional advantage for multibank PROM designs. If you arrange them correctly, you can effectively extend

the length of the CY7C289 pages from 64 to as many as 256 words. This extension improves system performance by increasing the likelihood of on-page PROM accesses (more on this feature later).

The CY7C289 includes these programmable features:

1. You can either register the input address at CLK's rising edge or latch the address using the ALE input.
2. You can program the address set-up and hold window.
3. You can program the WAIT output's polarity.
4. You can program the ALE input's polarity.
5. You can generate the WAIT output from CLK's falling or rising edge for the registered-mode CY7C289.
6. You can program the polarity of both chip selects (CS1 and CS2).

Each of these options is set by appropriately programming a reserved PROM location. Therefore, the devices are configured at the same time the array is programmed.

CY7C601 Microprocessor

The CY7C601 is a 32-bit general-purpose microprocessor that offers extremely high performance for embedded controller applications. The system described in this application note, for example, operates at 40 MHz. The CY7C601 is Cypress's CMOS implementation of Sun Microsystems' SPARC (Scalable Processor Architecture). This architecture achieves 29 MIPS by executing most instructions in a single clock cycle.

A CY7C601 architectural feature that affects the memory interface is an internal pipeline. To achieve an instruction execution rate approaching one instruction per clock cycle, the CY7C601 uses a four-stage instruction pipeline. All four stages operate in parallel, working on up to four different instructions at a time. The stages are:

1. Fetch—The processor sends out the instruction address to fetch an instruction.
2. Decode—The instruction is placed in the instruction register and decoded. The processor reads the operands from the register file and computes the next instruction address.
3. Execute—The processor executes the instruction and saves the results in temporary registers.
4. Write—The processor writes the result to the destination register.

A basic single-cycle instruction enters the pipeline and completes four cycles later. Normally, once the pipeline is full, an instruction is executed during every clock cycle. The existence of the instruction pipeline affects the memory interface (as described in the System Timing section of this application note). Otherwise, the memory interface design is straightforward.

PROM Configuration

In this application, four banks (16 CY7C289s) of PROM are used to provide 1 Mbyte of memory. Like most RISC architectures, the CY7C601 sends out valid address information immediately preceding a rising clock edge (and removes it soon afterward). Thus, the CY7C289s are configured in registered mode. The on-chip address registers capture the input at CLK's rising edge and ignore all unclocked address changes.

The chip selects on the CY7C289s are programmed on a bank to bank basis. Each bank is programmed with a unique polarity combination of CS1 and CS2 to permit PROM bank selection without external address decoding.

The other programmable features relevant to this design involve the CY7C289's WAIT signal. For compatibility with the CY7C601, the WAIT signal should be active Low and generated with respect to CLK's falling edge.

PROM Interface

Because this design involves no glue logic, the CY7C289 PROM's circuit connections are relatively straightforward. The CY7C601 communicates with external memory via a 32-bit address bus and a 32-bit data/instruction bus. Note, in *Figure2*, however, that the addresses fed to the PROMs are not entirely sequential.

The reason for the nonsequential addresses lies in the way the CY7C289 is organized. To improve the system's performance, the CY7C289 chip selects (CS1 and CS2) are used to extend the effective PROM page length to 256 32-bit words (1 Kbyte). To understand how this is done, consider that the CY7C289's lowest six address inputs (A0 - A5) designate a specific byte within a 64-byte internal PROM page. The CY7C289 uses inputs A6 - A15 to select one of 1024 PROM pages. When any of the inputs at pins A6 - A15 changes, a new page is selected and the CY7C289 asserts the WAIT\ output.

You can think of the CY7C289's chip selects as additional address inputs in a multibank memory system. As with A0 - A5, changes at the chip select inputs do not result in an internal page change.

With four banks of PROM, you have a total of 8 address bits (A0 - A5, CS1, CS2) that do not affect the internal PROM page, as opposed to just 6 (A0 - A5) when using one bank of PROM. The 8 bits of on-page addresses translate into a PROM page length of 256 words or 1 Kbyte.

The schematic in *Figure 2* reveals how this page-lengthening scheme is implemented. Note that the lowest 8 address bits from the CPU (A2 - A9) connect to the CY7C289 inputs that do not cause a page change (A0 - A5, CS1, CS2). The lowest address that connects directly from the CPU to the PROMs is A10. The chip selects in this design have effectively quadrupled the

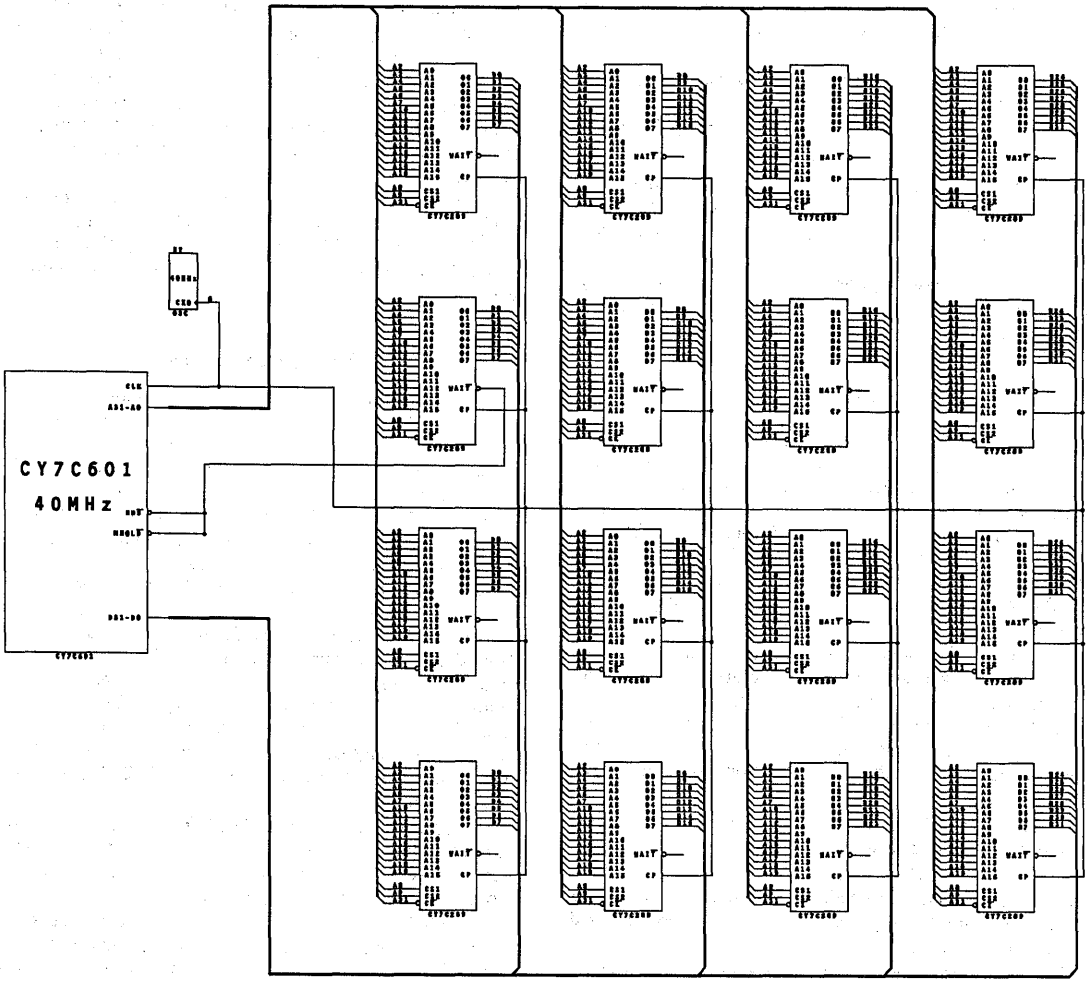


Figure 2. CY7C601 Memory Design

PROM page length, allowing a greater percentage of PROM accesses to complete within a single clock cycle.

Note that the extended-page-length feature of this design affects the software that runs on the system. To make the extended page useful, sequential code needs to be located on the same PROM page. In this design, where each PROM page extends across all four banks, code must be segmented into page-length blocks. This situation is analogous to interleaving DRAMs. Because each CY7C289 PROM has a 64-byte internal page, the users' code must be separated into 64-word blocks. In other words, place the first 64 words of code in bank 1, the next 64 words in bank 2, and so on. A simple program can accomplish this segmentation.

Another design issue that bears clarification is the connection of the WAIT\ signal generated by the CY7C289. This signal is asserted when the input address crosses an internal page boundary on the PROMs. WAIT\ connects directly to the CPU's Memory Hold A (MHOLDA\) and Memory Data Strobe (MDS\) inputs to tell the CY7C601 that an additional clock cycle is required to deliver the requested instruction from PROM. In the schematic in *Figure2*, only one WAIT\ output is connected to the CY7C601. This is because all 16 PROMs examine the same upper-order address inputs to determine if an internal page has been crossed. Therefore, only one PROM is needed to assert the WAIT\ signal when an off-page access is detected. It is important to note that the PROM will not generate WAIT\ if the chip enable signal (CE\) is inactive when the address changes. This ensures that when the CPU addresses some other portion of memory, such as RAM, the internal PROM page does not change, and a WAIT\ signal is not generated.

CY7C601 Interface

As shown in *Figure2*, the instruction memory interface requires only two control inputs (MHOLDA\, MDS\). MHOLDA\ freezes the clock to the instruction pipeline during a cache miss (for systems with cache) or when accessing a slow memory, such as the 65-ns page-miss operation in the CY7C289. Whenever the CY7C289 generates a WAIT\ signal, MHOLDA\ is asserted and the instruction pipeline is frozen. The processor freezes with the next instruction's address on the address bus. MHOLDA\ must be presented to the CY7C601 at the beginning of each processor clock cycle and be stable during the processor clock's falling edge.

The other control signal, MDS\, signals the processor when slow or missed (cache-miss) data is ready on the bus. The signal must be asserted only while the processor is frozen by either MHOLDA\ or Memory Hold B (MHOLDB\). Assertion of MDS\ enables the clock to the on-chip instruction register during an instruction fetch and effectively strobes the valid data into the CPU.

System Timing

This section provides a brief description of the CY7C601 timing interface to the CY7C289 PROMs. The timing diagram in *Figure3* illustrates a typical communication sequence between the CPU and the PROMs.

The memory interface's timing depends on whether or not the access is on the same page as the previous access. The case where an internal PROM page is crossed is illustrated in the left side of *Figure 3*. Address 1 (displayed as A1) is an access to PROM that causes an internal page change. WAIT\ is asserted by the CY7C289 to freeze the processor until the PROMs can deliver valid data. Note in *Figure3* that WAIT\ is not asserted until the next processor clock cycle. This delay is possible, using either MHOLDA\ or MHOLDB\, because of the CY7C601's pipelined architecture. The delay allows memories or interface logic more time to examine the address and determine if a wait state is required.

The processor samples MHOLDA\ on the processor clock's falling edge. An active MHOLDA\ indicates that the address in the *previous* clock cycle requires at least one wait state to complete. However, as shown in *Figure3*, by the time MHOLDA\ is detected active, the processor has already read the data corresponding to A1. Reading this false data is perfectly acceptable due to the CY7C601's internal instruction pipeline. The CPU has the time to invalidate the erroneous data before it reaches the execution stage. The MDS\ signal strobes in the correct data when the data becomes available.

The CY7C289s are configured to generate the WAIT\ signal with respect to CLK's falling edge to ensure proper operation of the wait-state mechanism. If the rising-edge option were selected, it is possible that the WAIT\ signal would be generated too *early* by the PROMs. Consequently the CY7C601 would recognize an active level on MHOLDA\ during the first cycle and invalidate the data from the bus cycle prior to the PROM access. Generating the WAIT\ signal from the falling edge ensures that the CPU does not detect the hold until the access's second cycle.

Another important aspect of the memory interface's operation during a PROM page change is that WAIT\ connects directly to MDS\ as well as to MHOLDA\. This arrangement causes MDS\ to be asserted for two clock cycles instead of just one, but this does not affect the system's operation. Although the CY7C601 copies data erroneously during the first cycle of MDS\, the erroneous data is overwritten with valid data in the next cycle. This approach works because MHOLDA\ remains asserted and does not allow the internal pipeline to advance until the correct data arrives. The advantage to feeding WAIT\ directly into MDS\ is that it avoids the use of any external logic for the memory interface.

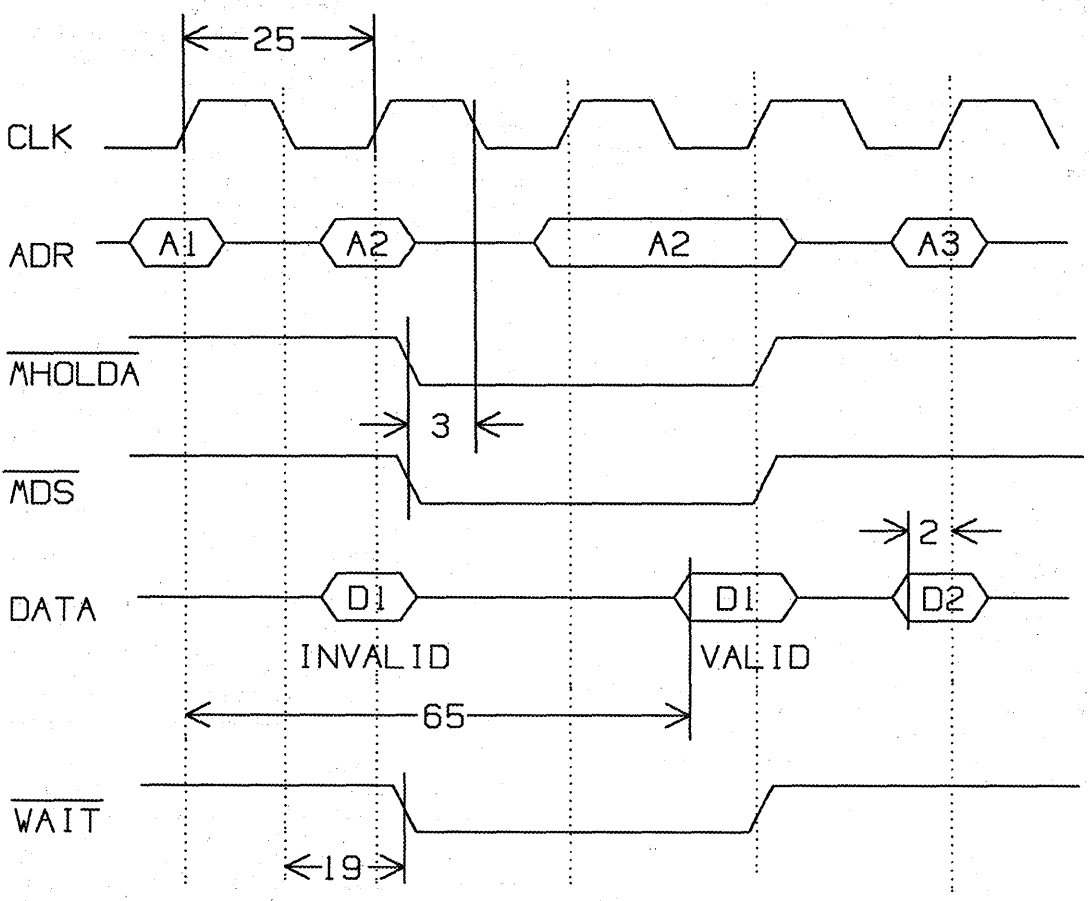


Figure 3. Memory Interface Timing

Figure 3 also displays some of the speed requirements that must be met in the instruction memory interface. In the case of an internal page change, the CY7C289 PROMs require two wait cycles to complete an access. The 40-MHz CY7C601 requires 2 ns of data setup time before the system clock's rising edge. This sequence results in a total of 73 ns available for the memory to return valid data. The CY7C289 meets this requirement with the 65-ns off-page access time.

The relatively trivial timing of sequential accesses falling on the same PROM page is illustrated in the right portion of *Figure 3*. The PROM latches A2 into the on-chip registers at CLK's rising edge and delivers data a maximum of 20 ns later.

Reference

For information on using the CY7C289 in latched mode, see the application note entitled "Interfacing the CY7C289 to the AM29000."

Section Contents

CYPRESS SEMICONDUCTOR

Introduction to Programmable Logic

Why Use a PLD?

ASICs (Application Specific Integrated Circuits) are one of the fastest growing segments of the semiconductor market for good reason. In addition to increasing packaging density and reducing board real estate by integrating SSI/MSI logic functions, ASICs reduce power requirements, improve reliability, and provide product secrecy.

ASICs include several different types of devices: full-custom devices, standard cells, gate arrays, and PLDs. Full-custom devices offer the greatest degree of integration, but they are expensive, and the development cycles can be on the order of nine months to a year. Full-custom designs are justified only for very large volume applications.

Standard cell devices can be turned around much more quickly (in about four months) and cost less than full-custom devices. However, the level of integration, and thus the speed, are lower than with the full-custom product.

Gate arrays offer even less dense integration, but because only two metal masks must be fabricated, the design turnaround can be as short as six weeks. One drawback of all these ASICs is that the design logic must be set at the start of the fabrication cycle. If the design changes, the whole product cycle must start over. In addition, because each device is application specific, you must watch inventory very carefully to make sure that just enough of each device is ordered to meet demand.

An alternative to custom or semicustom devices is the PLD (Programmable Logic Device). Although PLDs do not offer the same level of integration as the other ASICs, the board-space reduction is still significant. The reduction factor is application dependent and ranges from 4:1 and 10:1 for smaller PLDs (20 to 24 pins) to 75:1 for high-density/pin-count devices such as the LCA or MAX families. Additional benefits include reduced parts inventory, faster design and turnaround times, and simplified timing considerations.

Because a PLD is sold as a "generic" array of logic, customized by the user, you can use the same PLD in many different applications, spanning any number of projects. Cypress's PLDs are based on EPROM technology, thus making them EPLDs, which are erasable using an ultraviolet light source. You can make design changes at any point in the product cycle more easily than you can with other ASICs. The design cycle of a moderately complex PLD can be a week or less, and after the one-time purchase of a good development software package and programmer, the parts are relatively inexpensive. PLDs simplify logic timing because all logical functions take approximately the same path through the device. Thus, the same propagation delays apply to all device outputs (more on this later).

PLD Technology

All Cypress EPLD families except the CY7C360 family utilize the familiar sum-of-products architecture. You can implement Boolean transfer functions of this form by programming the AND array whose output terms feed a fixed OR array. This scheme can implement most combinatorial logic functions and is limited only by the number of product terms available in the AND-OR array. PLDs come in a variety of different sizes and with additional architectural features such as flip-flops.

TTL PLDs use a fuse as their programmable element. During the manufacturing process, fuses are built into all the connections between input pins and product terms. All unwanted connections are then blown during the programming process. Bipolar products are programmed using 20V pulses from 50 µs to 100 ms long. These 100- to 300-mA pulses blow unwanted fuses. Fuses are blown one at a time so that the heat generated does not damage or weaken the IC. Because of the high currents required, bipolar PLDs have to be programmed one at a time. Because physical fuses are blown, you can program these devices only once.

In contrast, the Cypress CMOS EPLD family uses an EPROM cell instead of fuses. This structure allows Cypress to functionally test and then erase all devices prior to packaging, thus facilitating 100-percent programming yields. The EPROM cell used by Cypress serves the same purpose as the fuse used in most bipolar PLD devices. Before programming, the AND gates (product terms) are connected via the EPROM cells to both true and complement inputs.

You program the EPROM cells using high-voltage pulses that produce 50 mA of programming current. Eight cells are programmed at a time. Because the current is lower than with bipolar PLDs, you can program 10 to 20 devices in parallel. When you program an EPROM cell, you disconnect that input to an AND gate (or product term). Programming alters the transistor cell's threshold so that no conduction can occur, which has the effect of disconnecting the input from product terms. This process equates to blowing a bipolar device's fuses, except that exposing the EPLD to ultraviolet light returns the cell's threshold to normal, effectively erasing the device. Selectively programming EPROM cells enables you to implement the desired logic function.

Cypress also offers the highest performance silicon PLDs that are available in ECL and BiCMOS technology. Aspen Semiconductor Corporation, a Cypress subsidiary, has developed a series of bipolar ECL PLDs using an advanced process that incorporates proven Ti-W fuses, and a BiCMOS process that incorporates CMOS and ECL. These devices achieve maximum input-to-output propagation delays of 3 to 6 ns.

PLD Notation and Fuse Maps

The Cypress *Data Book* and *PLD ToolKit Manual* provide logic diagrams for Cypress parts. These logic diagrams employ a common logic convention that is easy to use (*Figure 1*). In this convention, an X represents an unprogrammed EPROM cell used to connect an input term (a vertical line) to the input of the AND operation (a horizontal line). A missing X means that the EPROM cell on that connection has been programmed or disconnected. This convention does not imply that the input terms are connected on the common line that is indicated; rather, the input terms are being wire-ANDed. *Figure 2* shows a further extension of this convention that illustrates the implementation of a simple transfer function. *Figure3* shows the traditional representation of the same function.

Figure 4 shows the logic diagram for the PALC16L8. As mentioned earlier, all vertical lines in the array are connected to an array input. These inputs come from the input pins and the bidirectional I/O pins. Each horizontal line is a wired-AND function, or product term, which is either connected to an output driver's output enable or acts as one of seven inputs to an OR gate that connects to the output driver.

An EPROM lies at each intersection of an input and product term. These cells are numbered, starting with 0 as the top left fuse, increasing to the right, and then down. Thus, in *Figure 4*, cell 0 is the intersection between pin 2, noninverted, and the output-enable product term for pin 19. Cell 32 is the intersection between pin 2, noninverting, and the first product term for pin 19. The numbering proceeds until cell 2047, which is the intersection of pin 11, inverted, and the seventh product term for pin 12.

A fuse map represents a PLD's fuse array in software. A fuse map is an array of binary digits, arranged so that each digit corresponds to a cell in the device. For the PALC16L8 pictured in *Figure 4*, this array is 32 x 64. If a fuse is to be programmed, or disconnected, the corresponding digit is a 1. If the fuse is to be left intact, the corresponding digit is a 0. Because a virgin device has all cells conducting, or unprogrammed, its fuse map is all 0s. A product term, or horizontal line, of all zeros, is logically false because it is the AND of each input's true and complement. A product term of all 1s has no conducting path because all fuses are programmed, and thus nonconductive. This programming allows the product term to stay continuously at an asserted state.

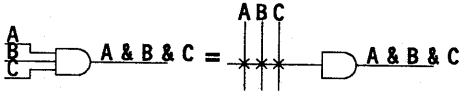


Figure 1. PLD Logic Notation

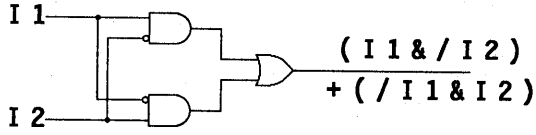


Figure 2. Transfer Function in PLD Logic Notation

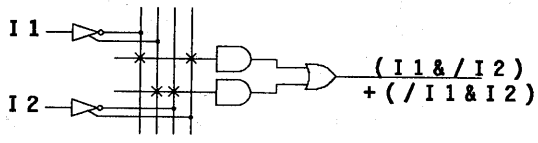


Figure 3. Conventional Schematic of Transfer Function in Figure 2

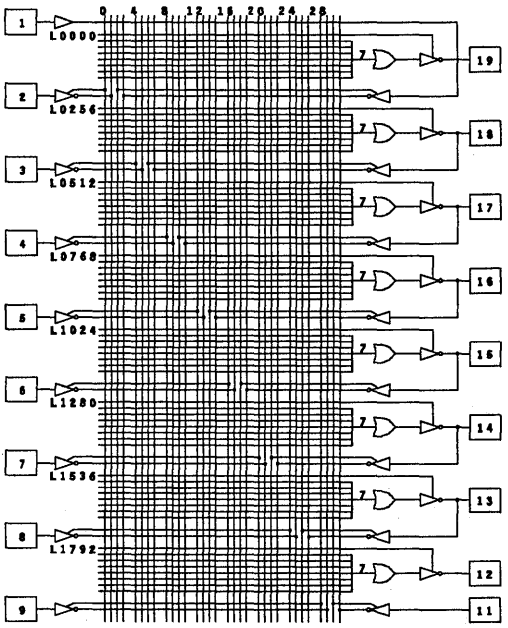


Figure 4. The 16L8 Block Diagram.

The official, standardized version of a fuse map is called a JEDEC map. This map can contain various informational fields and/or comments in addition to the 1s and 0s. *Figure5* shows the JEDEC map that implements the function shown in *Figures 2* and *3*. Each number starting with L in the leftmost column represents the first fuse number in that row. An N denotes a note or comment. QF precedes the total number of fuses in this device—QF2048 in this example. F0 means that the fuse default is 0, or unprogrammed. G0 specifies an unprogrammed security fuse, whereas G1 denotes a programmed security fuse (more on this later). C precedes a checksum value for the file. An * specifies the end of a field. A JEDEC file can also contain test vectors, which are not shown here.

For more information on the JEDEC Standard, refer to "JEDEC Standard No.3-A, Standard Data Transfer Format Between Data Preparation System and Programmable Logic Device Programmer" available from:

Solid State Products Engineering Council
2001 Eye Street N.W.
Washington, DC 20006

Most PLD design packages compile the design and translate it into a JEDEC map. The map is then downloaded to the programming hardware, which programs the device(s) accordingly.

```
Cypress C16L8 Jedec file
File: PLDAPP3.JED produced: 5/2/1989
From Cypress PLD Toolkit CPT100*
*QF2048* G0*N Security bit Unprogrammed*
L00000 11111111111111111111111111111110*N OE PT, pin= 19*
L00032 10011111111111111111111111111111*N Sum PT, pin= 19*
L00064 01101111111111111111111111111111*N Sum PT, pin= 19*
L00096 00000000000000000000000000000000*N Sum PT, pin= 19*
L00128 00000000000000000000000000000000*N Sum PT, pin= 19*
L00160 00000000000000000000000000000000*N Sum PT, pin= 19*
L00192 00000000000000000000000000000000*N Sum PT, pin= 19*
L00224 00000000000000000000000000000000*N Sum PT, pin= 19*
L00256 00000000000000000000000000000000*N OE PT, pin= 18*
L00288 00000000000000000000000000000000*N Sum PT, pin= 18*
L00320 00000000000000000000000000000000*N Sum PT, pin= 18*
L00352 00000000000000000000000000000000*N Sum PT, pin= 18*
L00384 00000000000000000000000000000000*N Sum PT, pin= 18*
L00416 00000000000000000000000000000000*N Sum PT, pin= 18*
L00448 00000000000000000000000000000000*N Sum PT, pin= 18*
L00480 00000000000000000000000000000000*N Sum PT, pin= 18*
L00512 00000000000000000000000000000000*N OE PT, pin= 17*
L00544 00000000000000000000000000000000*N Sum PT, pin= 17*
L00576 00000000000000000000000000000000*N Sum PT, pin= 17*
L00608 00000000000000000000000000000000*N Sum PT, pin= 17*
L00640 00000000000000000000000000000000*N Sum PT, pin= 17*
L00672 00000000000000000000000000000000*N Sum PT, pin= 17*
L00704 00000000000000000000000000000000*N Sum PT, pin= 17*
L00736 00000000000000000000000000000000*N Sum PT, pin= 17*
L00768 00000000000000000000000000000000*N OE PT, pin= 16*
L00800 00000000000000000000000000000000*N Sum PT, pin= 16*
L00832 00000000000000000000000000000000*N Sum PT, pin= 16*
L00864 00000000000000000000000000000000*N Sum PT, pin= 16*
L00896 00000000000000000000000000000000*N Sum PT, pin= 16*
L00928 00000000000000000000000000000000*N Sum PT, pin= 16*
L00960 00000000000000000000000000000000*N Sum PT, pin= 16*
L00992 00000000000000000000000000000000*N Sum PT, pin= 16*
L01024 00000000000000000000000000000000*N OE PT, pin= 15*
L01056 00000000000000000000000000000000*N Sum PT, pin= 15*
L01088 00000000000000000000000000000000*N Sum PT, pin= 15*
L01120 00000000000000000000000000000000*N Sum PT, pin= 15*
L01152 00000000000000000000000000000000*N Sum PT, pin= 15*
L01184 00000000000000000000000000000000*N Sum PT, pin= 15*
L01216 00000000000000000000000000000000*N Sum PT, pin= 15*
L01248 00000000000000000000000000000000*N Sum PT, pin= 15*
L01280 00000000000000000000000000000000*N OE PT, pin= 14*
L01312 00000000000000000000000000000000*N Sum PT, pin= 14*
L01344 00000000000000000000000000000000*N Sum PT, pin= 14*
L01376 00000000000000000000000000000000*N Sum PT, pin= 14*
L01408 00000000000000000000000000000000*N Sum PT, pin= 14*
L01440 00000000000000000000000000000000*N Sum PT, pin= 14*
L01472 00000000000000000000000000000000*N Sum PT, pin= 14*
L01504 00000000000000000000000000000000*N Sum PT, pin= 14*
L01536 00000000000000000000000000000000*N OE PT, pin= 13*
L01568 00000000000000000000000000000000*N Sum PT, pin= 13*
L01600 00000000000000000000000000000000*N Sum PT, pin= 13*
L01632 00000000000000000000000000000000*N Sum PT, pin= 13*
L01664 00000000000000000000000000000000*N Sum PT, pin= 13*
L01696 00000000000000000000000000000000*N Sum PT, pin= 13*
L01728 00000000000000000000000000000000*N Sum PT, pin= 13*
L01760 00000000000000000000000000000000*N Sum PT, pin= 13*
L01792 00000000000000000000000000000000*N OE PT, pin= 12*
L01824 00000000000000000000000000000000*N Sum PT, pin= 12*
L01856 00000000000000000000000000000000*N Sum PT, pin= 12*
L01888 00000000000000000000000000000000*N Sum PT, pin= 12*
L01920 00000000000000000000000000000000*N Sum PT, pin= 12*
L01952 00000000000000000000000000000000*N Sum PT, pin= 12*
L01984 00000000000000000000000000000000*N Sum PT, pin= 12*
L02016 00000000000000000000000000000000*N Sum PT, pin= 12*
C0B65*
0000
```

Figure 5. A 16L8 JEDEC Map.

First-Generation PLDs

The first PLDs were strictly combinatorial logic with three-state outputs, like the PALC16L8. Then D flip-flops, a clock input, and internal feedback were added, allowing a single PLD to implement sequential logic or state machines. The 16L8, 16R4 (four registered outputs), 16R6 (six registered outputs), and 16R8 (eight registered outputs) became industry-standard parts.

Testability was a problem in some of the earlier devices. Because a blank device had all fuses intact, out-

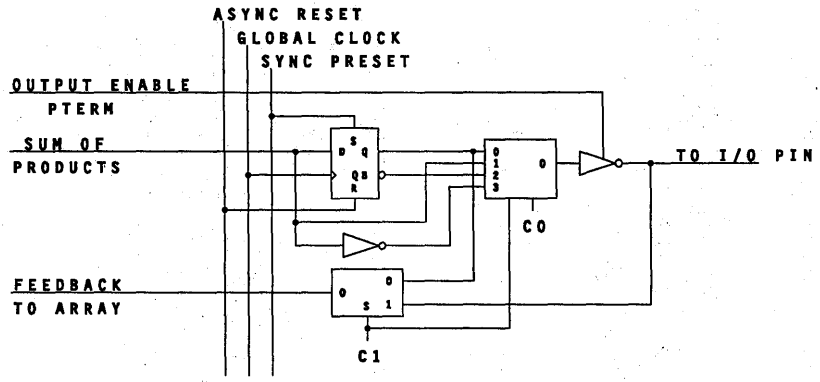


Figure 6. The 22V10 Macrocell.

put enables were all turned off, configuring all device pins as inputs. This scheme made it difficult to test blank devices and to check whether the fuses could be blown without actually blowing any of them.

To get around these problems, a phantom array was added to the device. The 16L8, for example, has 256 additional bits in its phantom array. These bits are used to test the PLD functionally and verify dynamic (AC) operation after the chip is packaged, without using the normal array. The phantom array is so named because it does not function in regular operating mode. The device must be in a special mode to access the phantom array.

The phantom array is usually programmed and verified as part of the final electrical test procedure during the manufacturing process. This procedure verifies both the PLD programmability and function. Cypress's EPLDs are programmed, tested, and then erased before they are packaged. You can also use the phantom array as part of incoming inspection.

Another feature of today's PLDs is register preload, which loads data into the registers of registered devices for testing purposes. This arrangement greatly simplifies and shortens the testing procedure. You can use this feature to check illegal state resolution—a state machine's ability to pass from an accidental illegal state to a legal one. Preloading is accomplished by applying a super-voltage (usually in the range of 12 to 14V) pulse of at least 100-µs duration to a specific pin, while holding a second pin at V_{IH}. The super voltage acts as a write strobe, which clocks data applied to the I/O pins into the corresponding registers.

A security fuse has also become a standard PLD feature. In addition to writing a fuse map into a device, any good device programmer can read a device's fuse map. This capability tends to negate the PLD's advantage of hiding proprietary logic from observers. But if you do not want your PLDs to be read by a programmer, you can program the security bit, which disconnects the lines used to verify the array. In a Cypress EPLD, the security EPROM cell is designed to retain its charge longer than any of the other cells in the array.

The Programmable Macrocell

The basic 20-pin PLDs of the past still had some limitations. For instance, they provided no way to control output-pin polarity without doing DeMorgan operations on the logic equations. Quite often the DeMorgan version has too many product terms to fit in such as device, even after several hours of reduction using a logic-optimization program.

Another drawback is that you have to stock a variety of the basic 20-pin PLDs and/or their 24-pin equivalents to get the best fit for a given design. Often extra registers are left unused when the design is finished. Even though these PLDs tend to be pin limited, the pins associated with the extra registers end up being wasted because you cannot use them for anything else.

The 24-pin 22V10 overcame earlier limitations and revolutionized PLDs by introducing the programmable macrocell (*Figure 6*). The programmable macrocell allows you to select one of four output configurations: combinatorial inverting, combinatorial noninverting, registered inverting, and registered noninverting. You can use the "output" pin as an input or for bidirectional I/O if you specify the macrocell as combinatorial.

Each of the 22V10's ten I/O pins have all four configuration options. You select the option using two fuses, or cells, identical to those in the array. These 20 bits (two for each of ten macrocells) appear at the bottom of the fuse map that represents the array.

Another innovation of the 22V10 is that some pins have a larger sum of products than others—an approach called variable product term distribution. In the 22V10, I/O pins have sums from eight to 16 product terms wide. This variable distribution accommodates

applications such as D flip-flop counters, where several outputs require a large number of product terms.

The 22V10 offers yet another improvement over PLDs such as the 16R8, which powers up with all registers in the reset state. The only way you can change this is by clocking in new data. The 22V10 avoids this problem by adding two extra product terms. One sets all registers, the other resets all registers. Because the set and reset are each a product term, they can be programmed to be the AND of any array input(s). For additional flexibility, the set is designated as a synchronous operation, and the reset is asynchronous.

Because of the 22V10's versatility, it has become something of an industry standard. It is available in TTL, CMOS, and GaAs. Many companies have introduced similar architectures with slightly different features. For example, the Cypress PLDC20G10 uses a similar macrocell that adds the capability to choose between a product-term output enable and a pin-controlled output enable. To make the PLDC20G10 faster and less expensive than the 22V10, Cypress has reduced the array to nine product terms per I/O macrocell and removed the set and reset product terms.

Another device introduced around the same time as the 22V10 is the 20RA10, which targets asynchronous registered applications. Like the 22V10, the 20RA10 has I/O pins with programmable polarity bits. You can configure the 20RA10's I/O pins as registered or combinatorial, but not with dedicated fuses. Instead, each I/O pin has a sum of four product terms that connects, through a polarity switch, to the D input of a flip-flop. Each of these flip-flops has dedicated product terms connected to its clock, set, and reset functions. When both the set and reset of a flip-flop are asserted (High), the flip-flop becomes transparent, thus making the output combinatorial.

In addition, the 20RA10 has an unusual output-enable scheme. Pin 13 is inverted and ANDed with an output-enable product term. If pin 13 is High, all I/O pins are at high impedance. The 20RA10 also offers a synchronous register preload in operating mode. When pin 1 goes Low, any data driven onto an I/O pin is latched into the corresponding flip-flop. An 20RA10's I/O pin is illustrated in *Figure 7*. This device's flexibility and asynchronous nature make it ideal for bus-arbiter and interrupt-controller applications.

Second-Generation PLDs

The architectural features introduced by the 22V10 greatly enhance PLD flexibility, but this device still has some limitations. It offers only D-type flip-flops, for example, which are cumbersome for applications such as counters. Further, each flip-flop and its feedback still use a pin, even if the flip-flop's output is not needed outside the PLD. Bidirectional, registered pins cannot be implemented. High-speed applications often require flip-flops outside the PLD's inputs to latch data because propagation delays impose relatively long set-up times for output flip-flops.

Cypress solves all these problems with the CY7C330. In addition to the output registers on the I/O pins, each pin except power and ground has an input register with a choice of two clocks. This input macrocell makes the 28-pin CY7C330 ideal for pipelined control and high-speed state machine applications.

Another CY7C330 feature is its ability to emulate T and JK flip-flops—a useful alternative in counter designs. In each I/O macrocell, the sum of products from the array drives one input of an exclusive-OR (XOR) gate. The second input to the XOR gate is another product term. This gate's output connects to the D input of the output flip-flop in the macrocell (*Figure 8*). If the flip-flop's Q output is fed back and connected to the single product term driving the XOR gate, the sum-of-products acts as the T input of a T flip-flop. The macrocell can also emulate a JK flip-flop in this way, using the relation T = J!Q + KQ. If you require a D flip-flop, you can use the XOR gate to control polarity.

Close examination of *Figure 8* reveals two paths into the array. The first is a multiplexer that selects feedback from either the output register or the input

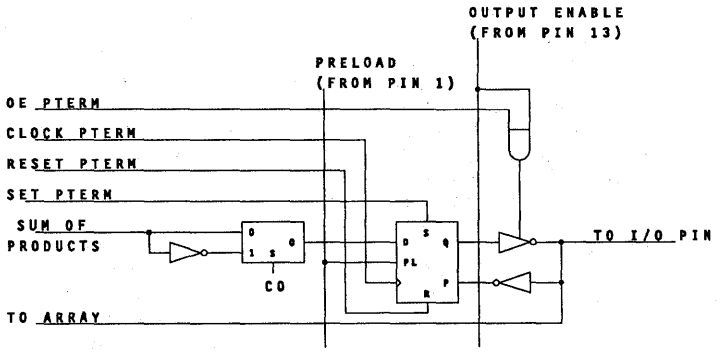


Figure 7. The 20RA10 Macrocell.

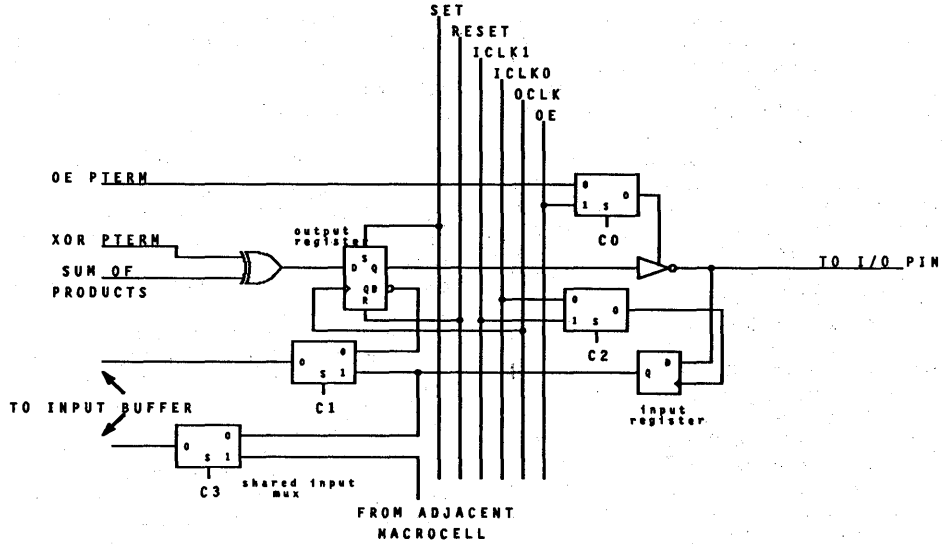


Figure 8. The CY7C330 Macrocell.

register's Q output. This multiplexer is called the feedback mux. The inputs to the second path, called the shared input mux, are the Q outputs of input registers belonging to adjacent I/O macrocells. This path allows you to feed back the Q output of a macrocell's output register, and still utilize the pin associated with that macrocell as an input. You can do this for six of the 12 I/O macrocells. If you need more registers for an application, the CY7C330 contains four additional buried registers. These registers are identical to the output register portion of the I/O macrocell, except they are not connected to any pin.

Just as the CY7C330 can be considered as an extended, enhanced version of the 22V10, the CY7C331 represents an extension of the 20RA10. The 20RA10 has many of the same limitations as the 22V10, with the additional limitation that the sum of products is only four product terms wide. The CY7C331 has 12 I/O macrocells. In addition to the 20RA10-like output flip-flops, the CY7C331 has identical flip-flops in the input path. As in the 20RA10, each flip-flop has a product-term-controlled clock, set, and reset. If the set and reset product terms are both asserted, the flip-flop becomes transparent. The 20RA10 polarity fuse has been replaced in the CY7C331 by an XOR gate, which has as inputs the sum of products and a dedicated product term. Thus, you can control the output's polarity or have the flip-flops emulate T or JK flip-flops, as in the CY7C330. The CY7C331 macrocell appears in *Figure 9*

Like the 22V10 and CY7C330, the CY7C331 has variable-product-term distribution with sums from four to 12 product terms wide. The CY7C331 borrows the shared input mux and output enable schemes from the CY7C330. The CY7C331 does not support the 20RA10's operating mode preload, but you can preload the CY7C331's registers using a super voltage.

The CY7C331 is designed especially for self-timed applications such as high-speed I/O interfaces. The device supports self-timed designs with programmable clock inputs, well-controlled internal timing relationships, and ultra-fast metastable resolution. No other PLD has this self-timed capability.

Another PLD architectural trend is to put registered inputs in combinatorial devices. These PLDs generally serve in sophisticated decoding applications, where the address or data is only stable for a short time.

In the past, an MSI chip with latches or flip-flops was used to capture transient data, and the latched data fed into a PLD. Now PLDs such as the CY7C332 feature an input macrocell that you can program as combinatorial, registered, or latched. You have a choice of two clocks, and you can program the clock polarity as well.

The CY7C332 I/O macrocell (*Figure 11*) incorporates the input macrocell and a combinatorial output path. The latter includes a variable sum of products that drives one input of an XOR gate; a dedicated product term drives the XOR's other input. An output-enable mux allows a product term (pin 14) to control the output enable. This combinatorial output path can act as an input to the programmable-input register/latch, thus allowing you to create state machines.

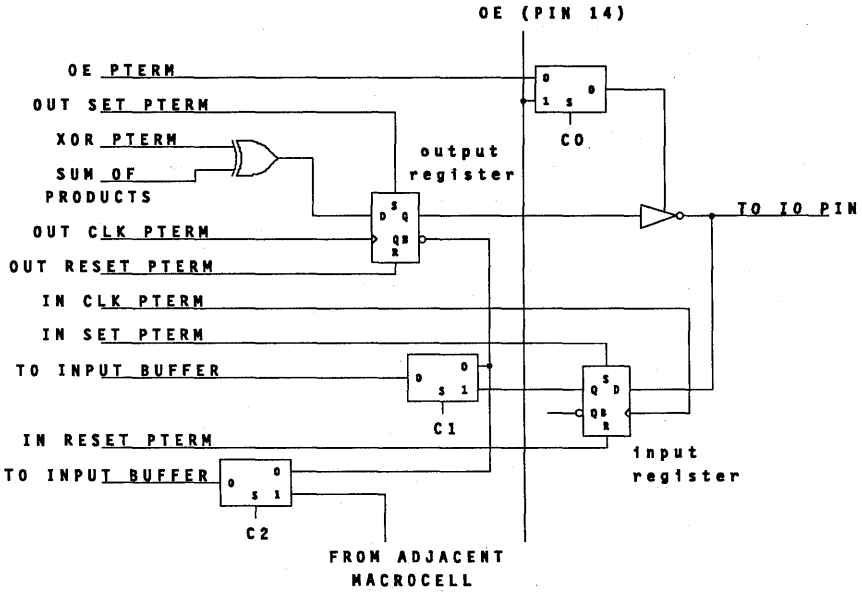


Figure 9. The CY7C331 Macrocell.

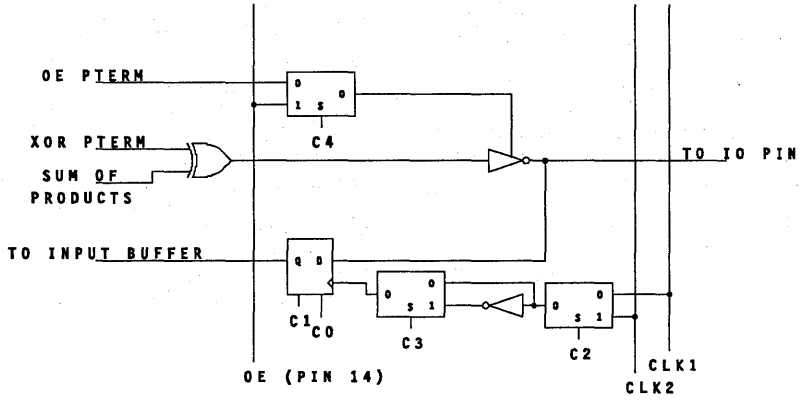


Figure 10. The CY7C332 Macrocell.

High-Density PLDs

Because of its low power consumption, CMOS can achieve higher integration than can bipolar technologies. Several manufacturers are taking advantage of this fact to produce very high density PLDs. The CY7C342, for example, is a 68-pin member of the new MAX family and contains 128 flip-flops and over 1000 product terms. Up to 256 additional latches can be configured using expander product terms. Each of these product terms is called a logic array block (LAB). The CY7C342 contains eight LABs, which connect together via a programmable interconnect array (PIA).

The CY7C342 macrocell (*Figure12*) contains a sum of three product terms driving one input of an XOR. The other XOR input is a dedicated product term. The XOR drives a programmable flip-flop, which you can

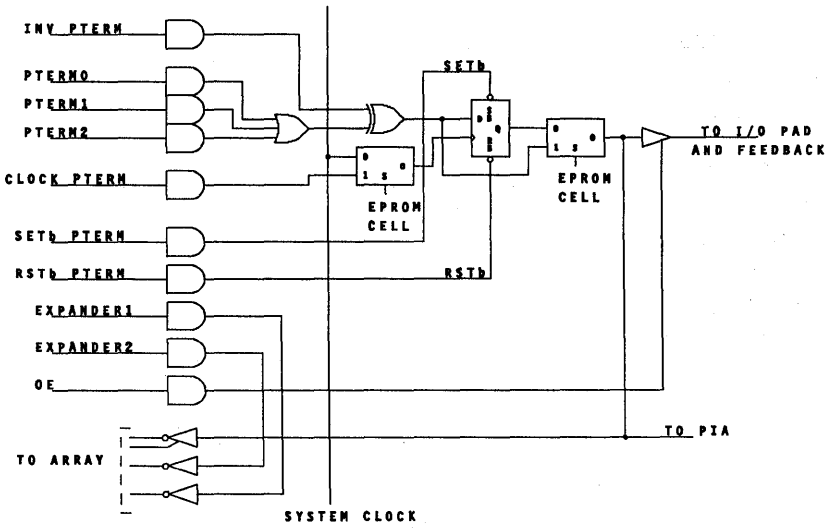


Figure 11. The CY7C342 Macrocell.

configure as a D, T, JK, or SR flip-flop or as a latch. The flip-flop has asynchronous set and reset product terms. It also offers a choice of asynchronous clock product term or a synchronous clock. Alternatively, the macrocell provides a combinatorial path.

The CY7C342's block diagram appears in *Figure 13*. In addition to a high level of integration, the device is fast. Its typical clock frequency equals 50 MHz. This combination of density, speed, and flexibility allows the CY7C342 to replace over 50 standard TTL devices.

PLD Software Packages

Parts as sophisticated as the MAX chips require equally sophisticated design software. The MAX+PLUS software offers schematic capture, state machine syntax, Boolean algebra entry, logic reduction, synthesis and fitting, and timing simulation. Similar packages that support a variety of devices are available from Data I/O, MINC, and several CAD software vendors.

More conventional support is available from ISDATA's LOGiC, Data I/O's ABEL, and Logical Devices' CUPL. These packages offer Boolean equation entry and logic reduction, as well as various higher-level language constructs, state machine syntax, and simulation. All these packages cover a variety of devices from several vendors.

Cypress offers a support package called the PLD ToolKit. This software supports all Cypress PLDs, with the exception of the MAX devices.

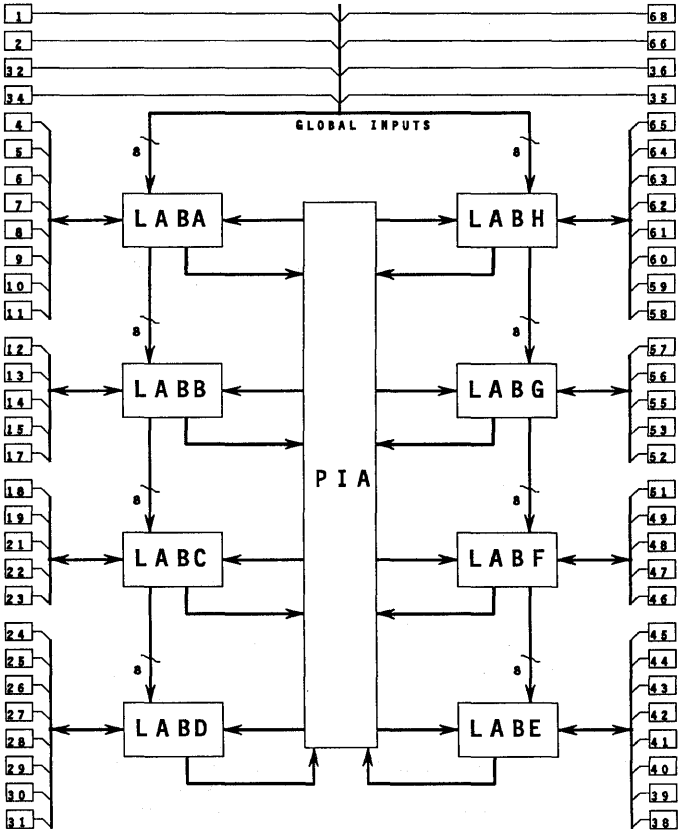


Figure 13. The CY7C342 Block Diagram.

CYPRESS SEMICONDUCTOR

CMOS PAL Basics

This application note provides a basic description of the Cypress CMOS PAL C devices, including their architecture and design, the technology used in their fabrication and programming, and how their reliability is guaranteed. The PAL C devices are functionally equivalent, pin compatible, and superior in performance to their bipolar counterparts.

This application note also furnishes information on the design techniques that Cypress uses on all products to eliminate latch-up and improve ESD (electrostatic discharge) protection.

PAL Definition

The functional structure of a PAL (programmable array logic) consists of a programmable AND array, whose outputs feed into a fixed OR array. The pertinent parameters are the number of inputs, the number of outputs, the number of factors (width) in the AND array, and the width of the OR array. The Boolean equation implemented by a PAL is a sum of products, or minterm form.

The first PALs included only combinatorial logic. Then someone realized that adding latches (D flip-flops), a clock input, and internal feedback made it possible to implement a programmable, sequential state machine in a single package. Three-state outputs, the "security fuse," flip-flop initialization, and testability were added later. Today, you have many PAL options to chose from.

Applications and User Benefits

PALs are used to replace SSI/MSI chips and glue logic, primarily to increase packaging density. A single PAL is the functional equivalent of many SSI ICs (up to 200 - 500 equivalent gates). Therefore, when you use PALs to replace standard logic gates, the resulting reduction in PC card area is significant. The reduction factor is application dependent and varies between 4 to 1 and 10 to 1. That is, one 20- or 24-pin PAL (in a 0.3-inch DIP) replaces between four and 10 14-pin ICs. Secondary benefits include reduced parts inventory, reduced power, higher reliability, faster design and turnaround time, product secrecy, and equal (matched) propagation delays through the AND-OR array.

PALs give you more functions and, more importantly, more interconnections within a single IC. This affects the reliability of your designs. Studies have proven that system reliability is inversely proportional to the number of interconnections among system elements. However, the reliability of ICs is *not* a function of their complexity.

The failure rate of mature ICs in volume production, during their useful life — the nearly horizontal part of the bathtub-shaped curve — is 0.1 percent per 1000 hours. This figure has remained essentially constant over the last 20 years, in spite of the fact that circuit complexity has increased by more than two orders of magnitude.

IC manufacturers have put more functions and interconnections in a single package, which results in a system with fewer components and, therefore, higher reliability. A definite benefit to you.

Programming

A ramification of using PALs is that they must be programmed. You can either design and build a programmer or buy a commercially available one for $4,000 to $10,000 from one of a dozen or so companies.

The programming process puts stress on the PAL, especially if fuses are blown. Bipolar PROMs have historically used fuses; the same technology has been applied to bipolar PALs.

All of the connections in a PAL are made during the wafer fabrication process. Then the unwanted connections are "unmade" by blowing fuses during the programming process. The first fuse materials were nichrome compounds and suffered from reliability problems. If the right amount of energy was not used to blow the fuse, the residue ash could become conductive over a period of time (100 - 500 operating hours) and the fuse could "regrow." These problems have been corrected, and materials such as platinum, silicide, and polysilicon are

currently used for the fuse material. However, the programming technique is the same: blowing fuses.

Bipolar PALs are programmed using 20V pulses lasting from 50 µs to 10 ms and carrying from 100 to 300 mA. One fuse is blown at a time, primarily because so much heat is generated that blowing more than one could either permanently damage the IC or stress it so much that it could fail later. In fact, some programming algorithms take into account the physical locations of the fuses and avoid sequentially blowing fuses that are physically close to each other; this prevents excessive localized heating of the chip. Because of the high currents required, bipolar PALs are not "gang" programmed, as are EPROMs.

Programming Cypress CMOS PALs

Cypress PALs are programmed by storing charge on the floating gate of a FAMOS (Floating Gate Avalanche Metal On Oxide) transistor in an EPROM cell. Thus, during programming Cypress PALs are stressed significantly less than fuse programmable PALs. In addition, every cell is programmed, tested and erased as part of the manufacturing process. This 100-percent testing guarantees a 100-percent programming yield to you, which is impossible to guarantee with bipolar PALs.

The storage mechanism is well understood. Products using it have been in volume production for the past 15 years. Numerous reliability studies have been performed by many independent organizations and all have concluded that the technology is reliable.

Cypress PAL C devices are programmed using 13.5V pulses lasting from 100 µs to 10 ms, during which 50 mA of current exist. Eight bits are programmed at the same time and, because of the lower currents required, 10 to 20 devices can be gang programmed in parallel.

Before programming, AND gates or product terms are connected via EPROM cells to both true and complement inputs. Programming an EPROM cell disconnects an input from a gate or product term. Selective programming of these cells enables a specific logic function to be implemented. PAL C devices are supplied in four functional configurations: 16L8, 16R8, 16R6, and 16R4. These functional variations offer you the choice of combinatorial as well as registered paths to implement logic functions.

Cypress PAL C devices are fabricated using an advanced 0.8µ N-well CMOS technology. The use of proven EPROM technology to achieve memory non-volatility, combined with novel circuit design and a unique architecture, provides you with a superior product in terms of performance, reliability, testability, and programmability.

Cypress PAL Functions

The variations of PAL C functions available appear in *Table 1*. The 16L8, for example, is purely combinatorial and consists of eight groups of seven-input AND gates; each group can have up to 32 possible inputs. One of the AND gates in each group enables the output driver, so that the other seven AND gates each feed one OR gate, whose output is inverted.

The 16R8 is similar to the 16L8, except that the outputs are latched using D flip-flops (with a common clock), the inputs to the eight OR gates are the outputs of eight AND gates, and the three-state output drivers are enabled by a common enable input.

All PAL C devices are manufactured using the same masks, except for the metal mask.

Refer to the PAL C data sheet for a more detailed description of the other members of the family. The 16R4, 16R6, and 16R8 have four, six, or eight registered outputs with feedback.

Register Preload

In registered devices, the preload function loads data into the internal register for testing purposes. This significantly simplifies and shortens the testing procedure.

Loading is accomplished by applying a supervoltage (13.5V) pulse of at least 100 µs duration to pin 5 as a write pulse while pin 11 is held at V_{IH} and data is applied to pins 12 through 19.

Security Function

The security function prevents the contents of the regular array from being electrically verified. This enables you to safeguard proprietary logic. The EPROM technology prevents the state of the cell from being visually ascertained.

The security function is implemented by programming an EPROM cell that disconnects the lines that are used to verify the array. This cell has been designed to retain its charge longer than any of the other cells in the array.

Arrays

There are 2048 EPROM cells in the regular PAL array that are used to select up to 32 inputs for eight groups of seven-input AND OR gates and up to 32 inputs for eight AND output enable gates. In normal usage, no more than 16 inputs are connected to any AND gate, because connecting both a true and a complement input of the same signal to the input of an AND gate results in a constant Low output.

The PALs have an additional 256 bits in a phantom array that are used to test the PAL functionally. These bits also serve to verify dynamic (AC) operation without using the normal array and after the PAL chip is packaged.

The phantom array is programmed and verified as part of the final electrical test procedure during the manufacturing process. You can use it as part of an incoming inspection to verify programmability as well as operation. Three input pins are used to verify operation of the phantom array. One (pin 2) has a worst-case (longest physical length) propagation delay path through the regular array.

You program the phantom array in the same manner as the regular array. Both are addressed as byte arrays for programming. The normal array has 256 bytes to program and the phantom array has 32 bytes.

Programming the PAL C EPROM Cell

A schematic of the two-transistor EPROM cell used in all PAL C devices appears in *Figure 1*. Conventional EPROMs use one transistor per cell. The transistor's design is a compromise between being able to program (write) rapidly and read. Cypress uses a two-transistor cell that enables the PAL C devices to achieve superior performance because the read and write transistors are optimized for their respective functions. The cell measures 20.4 by 6.7µ. Note that the selection gates, the floating gates, and the sources of both transistors are connected together.

In the unprogrammed state, the read transistor has a threshold voltage of approximately 1V and the program transistor, approximately 3.5V.

Table 1. PAL C Selection Guide

Commercial Parts

Generic Part Number	Logic	Output Enable	Outputs	I_{CC} (mA)		t_{PD} (ns)		t_S (ns)		t_{CO} (ns)	++
				L	STD	-25	-35	-25	-35	-25	-35
16L8	(8) 7-wide AND-OR-Invert	Programmable	(6) Bidirectional (2) Dedicated	45	70	25	35	--	--	--	--
16R8	(8) 8-wide AND-OR	Dedicated	Registered Inverting	45	70	--	--	20	30	15	25
16R6	(6) 8-wide AND-OR	Dedicated	Registered Inverting	45	70	25	35	20	30	15	25
	(2) 7-wide AND-OR-Invert	Programmable	Bidirectional								
16R4	(4) 8-wide AND-OR	Dedicated	Registered Inverting	45	70	25	35	20	30	15	25
	(4) 7-wide AND-OR-Invert	Programmable	Bidirectional								
20G10	(10) 8-wide AND-OR-Invert with MACRO	Programmable or Dedicated	Programmable Bidirectional or Registered	--	55	25	35	15	30	15	25
22V10	(10) variable AND-OR-Invert with MACRO	Programmable	Programmable Bidirectional or Reigistered	55	90	25	35	15	30	15	25

Military Parts

Generic Part Number	Logic	Output Enable	Outputs	I_{CC} (mA)	t_{PD} (ns)				t_S (ns)				t_{CO} (ns)			+
					-20	-25	-30	-40	-20	-25	-30	-40	-20	-25	-30	-40
16L8	(8) 7-wide AND-OR-Invert	Programmable	(6) Bidirectional (2) Dedicated	70	20	NA	30	40	--	NA	--	--	--	NA	--	--
16R8	(8) 8-wide AND-OR	Dedicated	Registered Inverting	70	--	NA	--	--	20	NA	25	35	15	NA	20	25
16R6	(6) 8-wide AND-OR	Dedicated	Registered Inverting	70	20	NA	30	40	20	NA	25	35	15	NA	20	25
	(2) 7-wide AND-OR-Invert	Programmable	Bidirectional													
16R4	(4) 8-wide AND-OR	Dedicated	Registered Inverting	70	20	NA	30	40	20	NA	25	35	15	NA	20	25
	(4) 7-wide AND-OR-Invert	Programmable	Bidirectional													
20G10	(10) 8-wide AND-OR-Invert with MACRO	Programmable	Programmable Bidirectional or Registered	80	NA	--	30	40	NA	--	25	35	NA	--	20	25
22V10	(10) variable AND-OR-Invert with MACRO	Programmable	Programmable Bidirectional or Registered	100	NA	25	30	40	NA	20	25	35	NA	20	20	25

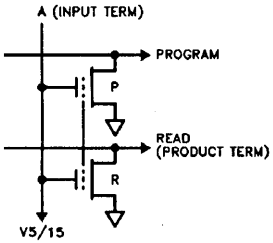


Figure 1. PAL EPROM Cell Schematic

To program the cell, you raise the input line (A) to 15V, which causes charge to be stored on the floating gate of the program transistor. This in turn causes the program transistor's threshold to increase to approximately 7V. Because the floating gates of both transistors are connected together, the threshold of the read transistor increases by the same amount (7 - 1 = 6V).

To read from the cell, you raise the input line (A) to 5V. If the cell has been programmed, this voltage is not sufficient to turn on the read transistor. However, if the cell has not been programmed, the read transistor turns on. Under this condition, the current through the read transistor is 120 to 150 μA — approximately an order of magnitude greater than that used in a conventional EPROM cell. The larger current is required to achieve the specified performance.

Operational Overview

The PAL operates in two basic modes: PAL and Program. In the PAL mode, either the regular array or the phantom array can be used along with the data inputs to determine the state of the outputs. In the Program mode, either the regular array or the phantom array can be programmed using the eight outputs (pins 12 - 19) as data inputs and pins 2 - 9 as address inputs.

Table 2 illustrates the various modes of operation for the PAL C 20 Series devices. The modes are decoded by high-voltage-sensitive on-chip circuits. You can go from any of the modes to any other mode. Note that the normal data output pins (12 - 19) serve as data input pins for programming.

Programming

Tables 3 and *4* indicate how the regular and the phantom arrays are addressed. The regular array is addressed as a 256-word (8 X 32-bit) memory. The phantom array is

Table 2. PAL C 20 Series Operating Modes

Pin Name	V_{PP}	PGM/$\overline{OE}$	A1	A2	A3	A4	A5	D7-D0	Notes
Pin Number	(1)	(11)	(3)	(4)	(5)	(6)	(7)	(12-19)	
Operating Modes									
PAL	X	X	X	X	X	X	X	Programmed Function	3, 4
Program PAL	V_{PP}	V_{PP}	X	X	X	X	X	Data In	3, 5
Program Inhibit	V_{PP}	V_{IHP}	X	X	X	X	X	High Z	3, 5
Program Verify	V_{PP}	V_{ILP}	X	X	X	X	X	Data Out	3, 5
Phantom PAL	X	X	X	X	X	V_{PP}	X	Programmed Function	3, 6
Program Phantom PAL	V_{PP}	V_{PP}	X	X	X	X	V_{PP}	Data In	3, 7
Phantom Program Inhibit	V_{PP}	V_{IHP}	X	X	X	X	V_{PP}	High Z	3, 7
Phantom Program Verify	V_{PP}	V_{ILP}	X	X	X	X	V_{PP}	Data Out	3, 7
Program Security Bit	V_{PP}	V_{PP}	V_{PP}	X	X	X	X	High Z	3
Verify Security Bit	X	X	Note 8	V_{PP}	X	X	X	High Z	3
Register Preload	X	X	X	X	V_{PP}	X	X	Data In	3, 9

Notes:

1. V_{PP} = 13.5 ±0.5V, I_{PP} = 50 mA, V_{CCP} = 5 ±0.25V, V_{IHP} = 3V, V_{ILP} = 0.4V.
2. Measured at 10 and 90% points.
3. $V_{SS} < X < V_{CCP}$.
4. All "X" inputs operational per normal PAL function.
5. Address inputs occupy pins 2 through 9 inclusive; for both programming and verification, see programming address *Tables 3* and *4*.
6. All "X" inputs operational per normal PAL function except that they operate on the function that occupies the phantom array.
7. Address inputs occupy pins 2 through 9 inclusive; for both programming and verification, see programming address *Tables 3* and *4*. Pin 7 is used to select the phantom mode of operation and must be taken to V_{PP} before selecting phantom program operation with V_{PP} on pin 1.
8. The state of pin 3 indicates whether the security function has been invoked. If pin 3 = V_{OL}, security is in effect. If pin 3 = V_{OH}, the data is unsecured and can be directly accessed.
9. For testing purposes, the output latch on the 16R8, 16R6, and 16R4 can be preloaded with data from the appropriate associated output line.

selected using the same addresses as columns 0, 1, 2, and 3, but with pin 7 at V_{pp} (as shown in *Tables 2* and *4*).

For either the normal or phantom array, the product terms are addressed in groups of eight, as shown in *Table 3*. There is a one to one correspondence between the data to be programmed and the D0 - D7 inputs and the product terms, as modified modulo 8, by the address on pins 2, 3, and 4 (*Figure 2*). In other words, a One on D0 corresponds to deselecting the product term input at input line 0 and product term 0. A One on D1 corresponds to de-selecting the product term input at input line 0 and product term 8, etc.

One method of programming the array is to program and verify the bits corresponding to the first product term address, then increment a counter that generates the OR gate addresses (pins 2, 3, and 4), then program and verify the second row of *Table 3*, and continue this process eight times until all 64 product terms associated with input line 0 have been programmed and verified. To select the second (1) input term, address pins 6, 7, 8, and 9 are held Low (as before) and pin 5 = High. The preceding sequence is then repeated 31 more times, incrementing pins 5 through 9 in a binary sequence to program and verify the entire array. The other members of the PAL C family are programmed in an identical manner.

Figure 3 shows a simplified block diagram of a 16L8 PAL C. *Figure 4* shows the method of programming and sensing.

Table 3. PAL C 20 Series Product Term Addresses

Product Term Addresses									
Binary Address Pin Numbers			Line Number						
(4)	(3)	(2)							
V_{ILP}	V_{ILP}	V_{ILP}	0	8	16	24	32	40	48	56
V_{ILP}	V_{ILP}	V_{IHP}	1	9	17	25	33	41	49	57
V_{ILP}	V_{IHP}	V_{ILP}	2	10	18	26	34	42	50	58
V_{ILP}	V_{IHP}	V_{IHP}	3	11	19	27	35	43	51	59
V_{IHP}	V_{ILP}	V_{ILP}	4	12	20	28	36	44	52	60
V_{IHP}	V_{ILP}	V_{IHP}	5	13	21	29	37	45	53	61
V_{IHP}	V_{IHP}	V_{ILP}	6	14	22	30	38	46	54	62
V_{IHP}	V_{IHP}	V_{IHP}	7	15	23	31	39	47	55	63
			D0	D1	D2	D3	D4	D5	D6	D7
			Programmed Data Input							

Table 4. PAL C Series Input Term Addresses

Input Term Addresses					
Input Term Numbers	Binary Addresses Pin Numbers				
	(9)	(8)	(7)	(6)	(5)
0	V_{ILP}	V_{ILP}	V_{ILP}	V_{ILP}	V_{ILP}
1	V_{ILP}	V_{ILP}	V_{ILP}	V_{ILP}	V_{IHP}
2	V_{ILP}	V_{ILP}	V_{ILP}	V_{IHP}	V_{ILP}
3	V_{ILP}	V_{ILP}	V_{ILP}	V_{IHP}	V_{IHP}
4	V_{ILP}	V_{ILP}	V_{IHP}	V_{ILP}	V_{ILP}
5	V_{ILP}	V_{ILP}	V_{IHP}	V_{ILP}	V_{IHP}
6	V_{ILP}	V_{ILP}	V_{IHP}	V_{IHP}	V_{ILP}
7	V_{ILP}	V_{ILP}	V_{IHP}	V_{IHP}	V_{IHP}
8	V_{ILP}	V_{IHP}	V_{ILP}	V_{ILP}	V_{ILP}
9	V_{ILP}	V_{IHP}	V_{ILP}	V_{ILP}	V_{IHP}
10	V_{ILP}	V_{IHP}	V_{ILP}	V_{IHP}	V_{ILP}
11	V_{ILP}	V_{IHP}	V_{ILP}	V_{IHP}	V_{IHP}
12	V_{ILP}	V_{IHP}	V_{IHP}	V_{ILP}	V_{ILP}
13	V_{ILP}	V_{IHP}	V_{IHP}	V_{ILP}	V_{IHP}
14	V_{ILP}	V_{IHP}	V_{IHP}	V_{IHP}	V_{ILP}
15	V_{ILP}	V_{IHP}	V_{IHP}	V_{IHP}	V_{IHP}
16	V_{IHP}	V_{ILP}	V_{ILP}	V_{ILP}	V_{ILP}
17	V_{IHP}	V_{ILP}	V_{ILP}	V_{ILP}	V_{IHP}
18	V_{IHP}	V_{ILP}	V_{ILP}	V_{IHP}	V_{ILP}
19	V_{IHP}	V_{ILP}	V_{ILP}	V_{IHP}	V_{IHP}
20	V_{IHP}	V_{ILP}	V_{IHP}	V_{ILP}	V_{ILP}
21	V_{IHP}	V_{ILP}	V_{IHP}	V_{ILP}	V_{IHP}
22	V_{IHP}	V_{ILP}	V_{IHP}	V_{IHP}	V_{ILP}
23	V_{IHP}	V_{ILP}	V_{IHP}	V_{IHP}	V_{IHP}
24	V_{IHP}	V_{IHP}	V_{ILP}	V_{ILP}	V_{ILP}
25	V_{IHP}	V_{IHP}	V_{ILP}	V_{ILP}	V_{IHP}
26	V_{IHP}	V_{IHP}	V_{ILP}	V_{IHP}	V_{ILP}
27	V_{IHP}	V_{IHP}	V_{ILP}	V_{IHP}	V_{IHP}
28	V_{IHP}	V_{IHP}	V_{IHP}	V_{ILP}	V_{ILP}
29	V_{IHP}	V_{IHP}	V_{IHP}	V_{ILP}	V_{IHP}
30	V_{IHP}	V_{IHP}	V_{IHP}	V_{IHP}	V_{ILP}
31	V_{IHP}	V_{IHP}	V_{IHP}	V_{IHP}	V_{IHP}
P0	V_{ILP}	V_{ILP}	V_{PP}	X	X
P1	V_{ILP}	V_{IHP}	V_{PP}	X	X
P2	V_{IHP}	V_{ILP}	V_{PP}	X	X
P3	V_{IHP}	V_{IHP}	V_{PP}	X	X

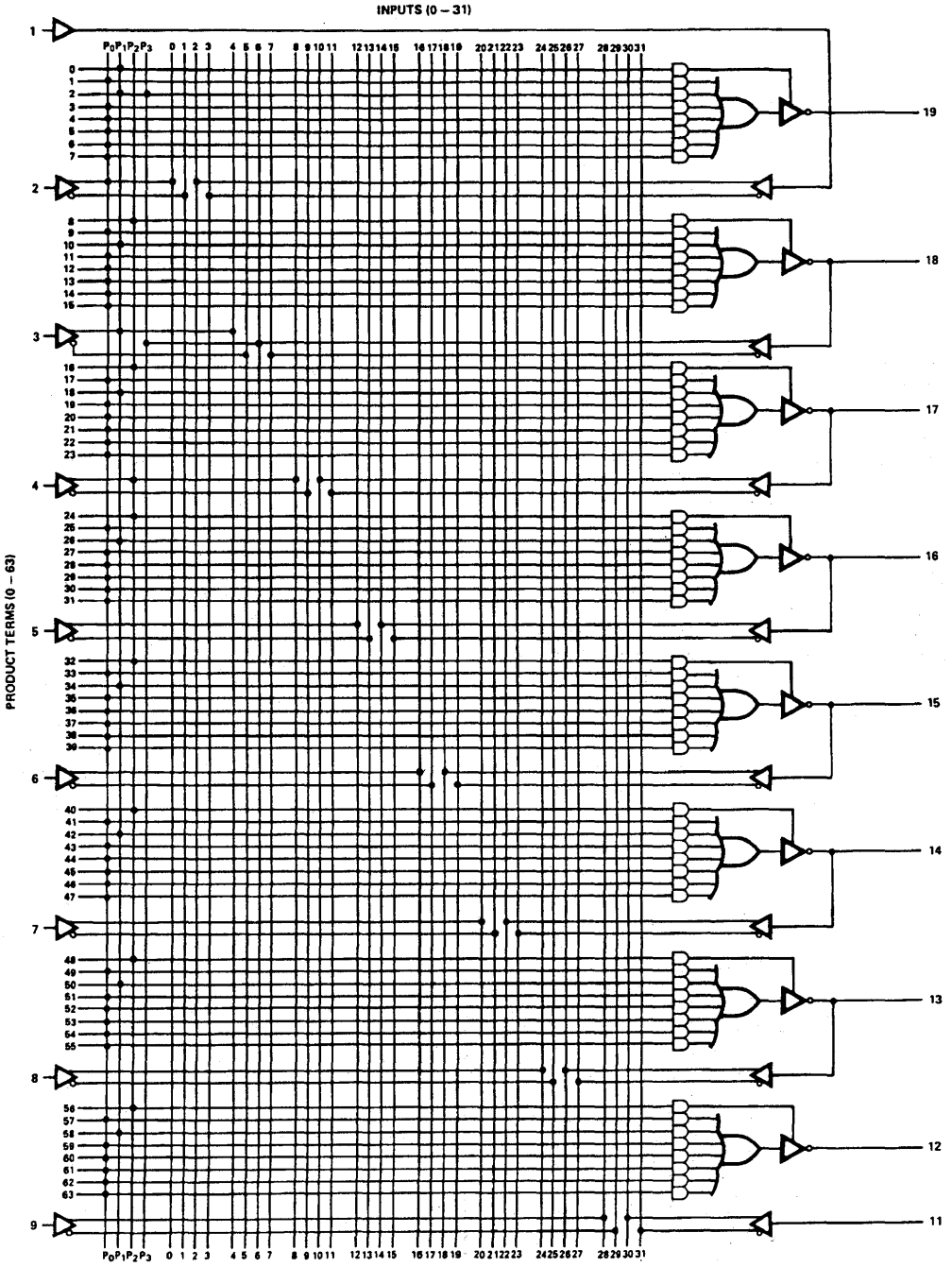


Figure 2. Functional Logic Diagram of PAL C 16L8A

Programming Operation

In a PAL C device, pins 5 - 9 are decoded (*Table 4*) in a one of 32 decoder, whose outputs correspond to the inputs labeled 0 -31 in *Figure 2*. For programming, 15V is applied to the bottom of the word line through a weak-depletion-mode device. The EN (enable) signal to all of the three-state drivers is Low, which prevents the normal PAL input signals from driving the word lines during programming. The D0 - D7 inputs (pins 19 - 12) drive the program transistors (0, 8, 16, 24, etc.) as selected by pins 2, 3, and 4 (*Table 3*). To disconnect a word line from a bit line, the program transistor is forward biased, which increases the threshold of the read transistor.

Verify Operation

To verify the programmed cells, the device must go from the Program PAL mode to the Program Inhibit mode to the Program Verify mode. This is accomplished by reducing the voltage on pin 11 to V_{IHP} (3V) and then to V_{ILP} (0.4V). Inside the device (*Figure 4*), the voltage changes disable the 1-of-32 decoder, bring the EN signal Low, and put 31 of the 32 input term lines at 0V. The line being verified is at 5V. The input address lines (pins 2 through 9) do not need to change when going from Program to Verify mode.

Because the Ones that were programmed cause the thresholds of the R transistors to increase, these transistors do not turn on during Verify mode. The unprogrammed transistors do turn on, however; the complement (inverse) of the data programmed is thus read during verify.

Regular (Normal) PAL Operation

The PAL implements the programmed function when no supervoltages are applied to any of the pins. During regular PAL operation, the 1-of-32 decoder and the D0 - D7 decoder are disabled, the EN signal is High, and all 32 input term lines are at 5V. Under these conditions, the data at the PAL C input pins is applied to all 64 of the product term lines. If any of the P transistors (16 per product term line) have not been programmed, they turn on and pull the lower input of the corresponding sense amplifier (SA) to 2V or less. Because this voltage is lower than the reference (V_{ref}), the sense amplifier's output is Low.

The reference is an unprogrammed EPROM cell that tracks the same process, voltage, and temperature variations that affect all the cells in the array. The reference is approximately 3V at room temperature and nominal V_{CC} (5V).

Phantom PAL Operation

The PAL is in the Phantom PAL operation mode when a supervoltage (V_{PP} = 13.5V) is applied to pin 6. The phantom array is programmed as shown in *Figure 2*. When the device is in Phantom PAL mode, you can measure the worst-case propagation delay from the pin 2 input to the outputs (pins 12 through 17). The truth table for the phantom array appears in *Table 5*.

Reliability

Reliability is designed into all Cypress products from the beginning by using design techniques to eliminate latchup and improve ESD and by paying careful attention to layout. All products are tested for all known types of CMOS failure mechanisms.

Failure mechanisms can be either classified as those generic to CMOS technology or those specific to EPROM devices. *Table 6* lists both categories of failures, their relevant activation energies, E_a in electron volts, and the detection method used by Cypress. In both cases, the mechanisms are aggravated by HTOL (high temperature operating life) tests and HTS (high temperature storage) tests.

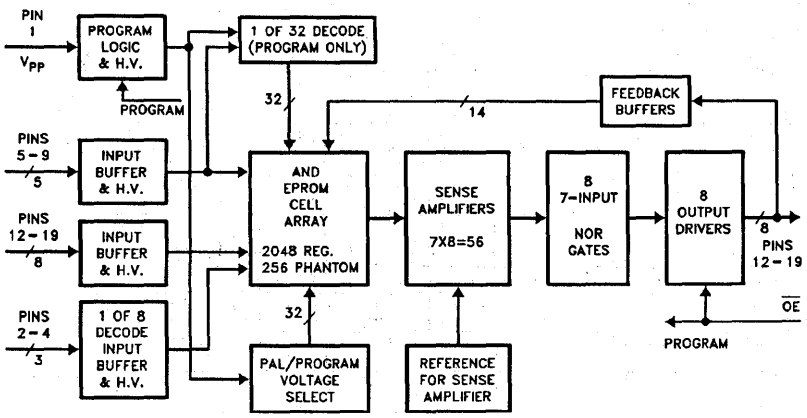


Figure 3. 16L8 Device Simplified Block Diagram

Table 5. Phantom Array Truth Table

	Inputs			Outputs							
Pin	2	3	4	19	18	17	16	15	14	13	12
	0	0	1	X	X	1	1	1	1	1	1
	1	0	1	X	X	0	0	0	0	0	0
	0	1	X	1	0	X	X	X	X	X	X
	0	1	X	0	1	X	X	X	X	X	X

Specific EPROM failure mechanisms include charge loss, charge gain, and electron trapping. Thermal energy and field emission effects accelerate charge loss.

Thermal charge loss failures usually occur on random bits and are often related to latent manufacturing defects. In many instances a dramatic difference between typical and worst-case bits are observed. Field emission effects are generally detected as weakly programmed cells. The high voltages used to program a selected bit might disturb an unselected bit as a result of a defect.

Charge gain is due to electrons accumulating on a floating gate as a result of bias or voltage on the gate. This results in a reduced read margin. The effects of this mechanism are generally negligible.

Electrons might become trapped in the gate oxide during programming and cause diminished reprogrammability. For one-time-programmable devices, this failure mode has little significance. This is because Cypress PAL C devices are programmed only three times: twice during manufacture and once by the customer.

HTOL Testing

High temperature operating life test (or burn-in) detects most generic CMOS failure mechanisms. Units are placed in sockets under bias conditions with power applied and at elevated temperatures for a specific number of hours. This test weeds out the "weak sisters" that would fail during the first 100 to 500 hours of operation under normal operating temperatures. HTOL tests are also used to measure parameter shifts to predict and screen for failures that would occur much later.

HTS Testing

High temperature storage tests are used to thermally accelerate charge loss. These tests are performed at the

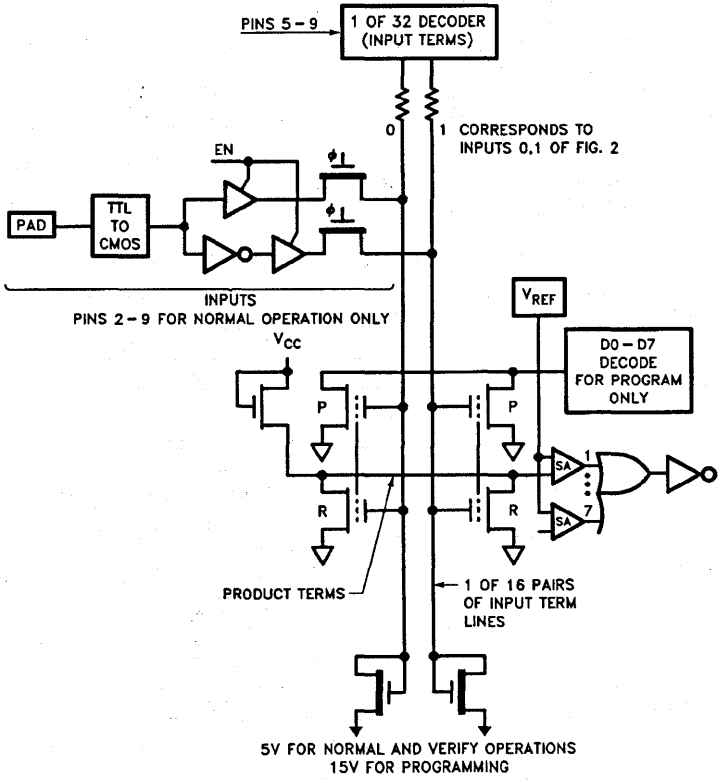


Figure 4. Programming Method

wafer level and under unbiased conditions. Both pass/fail data as well as shifts in thresholds are measured. For a more detailed discussion of charge loss screening, see the *References*.

The generally accepted screening method for identifying charge loss is a 168-hour bake at 250°C. This correlates with more than 220,000 years of normal operation at 70°C using a failure activation energy of 1.4 ev. The sample size chosen guarantees that at least 99 percent of the units will not fail during their useful operating life.

Initial Qualification

The process in general and the PAL C design specifically was qualified using HTS (bake) at 250°C for 256 hours, in conjunction with an HTOL test at 125°C for 1000 hours.

In the qualification process, four wafers were erased using ultraviolet light, and the linear thresholds of the cell's read transistors measured at 25 sites on each wafer. The wafers were then programmed, and the linear thresholds measured and recorded.

The wafers were alternately baked at 250°C and the linear thresholds measured and recorded at 0.25, 0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 hours. The number of device hours was therefore 100 x 256 = 25,600.

The results of this process revealed that the average threshold reduction due to charge loss was 0.66V. The range was 8 to 10 percent of the average initial threshold of 7.7V. This reduced threshold is more than 4V above the sense amplifier voltage reference. There were no failures.

If the charge loss failure activation energy is assumed to be 1.4 ev, the HTS time of 256 hours at 250°C translates to 438,356 years of operation at 70°C. This time translation was computed using the industry-standard Arrhenius equation, which converts the time to failure (operating lifetime) at one temperature and time to another temperature and time.

To summarize the results:

- Sample size: 100
- Device hours: 25,600
- HTS conditions: 256 hours at 250°C
- Average initial threshold: 7.7V
- Average threshold decrease: 0.66V
- Standard deviation: 0.12
- Lifetime (1.4 ev): 438,356 years at 70°C

These results confirm that the data retention characteristics of the EPROM cell used in all Cypress PALs and PROMs guarantees a minimum operating lifetime of 438,356 years for activation energies of 1.4 ev.

Production Screen

Units from the same population were assembled without being subjected to HTS and were subjected to an HTOL of 150°C for 1000 hours. The units were tested at 12, 24, 48, 96, 168, 336, and 1008 hours and the measurements recorded. Variations in the thresholds of the EPROM cells were measured and correlated to the units tested in the HTS test to determine a maximum acceptable rate of charge loss. This data allows Cypress to guarantee data retention over the devices' normal operating lifetime.

PAL C Advantages Over Bipolar PALs

The most pertinent data sheet parameters of Cypress PAL C devices are compared with those of representative bipolar PALs in *Table 1*. The supply current and propaga-

Table 6. Generic CMOS Failure Mechanisms

Mechanism	Activation Energy (eV)	Detection Method
Surface charge	0.5 to 1.0	HTOL, Fabrication monitors
Contamination	1.0 to 1.4	HTOL, Fabrication monitors
Electromigration	1.0	HTOL
Micro-cracks	--	Temperature cycling
Silicon defects	0.3	HTOL
Oxide breakdown	0.3	High-voltage stress, HTOL
Hot electron injection	--	LTOL (low-temperature operating life)
Fabrication defects	--	Burn in
Latchup	--	High-voltage stress, burn in, characterization
ESD	--	Characterization
Charge loss	0.8 to 1.4	HTS (high-temperature storage)
Charge Gain (oxide hopping)	0.3 to 0.6	HTOL
Electron trapping in gate oxide	--	Program/erase cycle

Table adapted from "An Evaluation of 2708, 2716, 2532, and 2732 Types of U-V EPROMs, Including Reliability and Long Term Stability." Danish Research Center for Applied Electronics, Nov. 1980.

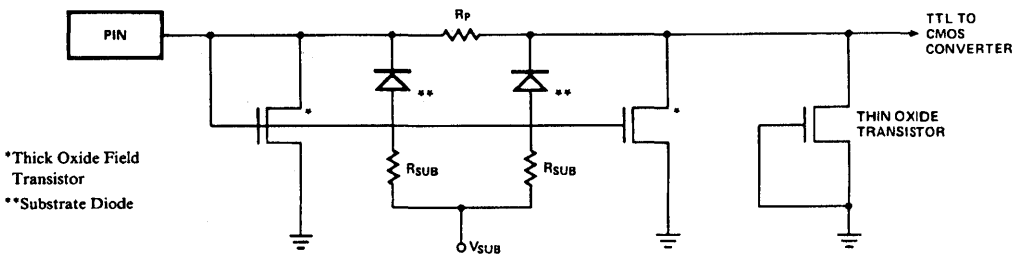


Figure 5. Input Protection Circuit

tion delay specifications are compared under identical test conditions. The output current sinking specifications are also identical. Cypress PAL C devices are clearly superior to bipolar PALs.

The lower power advantage of the PAL C results in several benefits:

- Lower capacity power supplies, which therefore cost less
- Reduced cooling requirements
- Increased long term reliability due to lower die junction temperatures

You can further reduce the power dissipation by driving the PAL C inputs between 0.5V or less and 4V or more. This reduces the power dissipation in the input TTL-to-CMOS buffers, which dissipate power when their inputs are between 0.8 and 3V.

PAL C Technology

The PAL C devices' 0.8µ, double-layer-polysilicon, single-layer-metal, N-well, CMOS technology has been optimized for performance. Careful attention to design details and layout techniques has resulted in superior-performance products with improved ESD input protection and improved latch-up protection.

The circuit shown in *Figure 5* is used at every input pin in all Cypress products to provide protection against ESD. This circuitry withstands repeated applications of high voltage without failure or performance degradation. This is accomplished by preventing the high ESD voltage from reaching the internal transistors' thin gate oxides.

The circuit consists of two thick-oxide field transistors wrapped around an input resistor (R_P) and a thin-oxide gate transistor with a relatively low breakdown voltage (12V). Large input voltages cause the thick-oxide transistors to turn on, discharging the ESD current to ground. The thin-oxide transistor breaks down when the drain-to-source voltage exceeds 12V. This transistor is protected from destruction by the current-limiting action of R_P. Experiments confirm that this input protection circuitry results in ESD protection in excess of 2000V.

Latch-up

Latch-up is a regenerative phenomenon that occurs when the voltage at an input or output pin is either raised above the power supply voltage potential or lowered below the substrate voltage potential, which is usually ground. Current rapidly increases until, in effect, a short circuit from V_{CC} to ground exists. If the current is not limited, it will destroy the device, usually by melting a metal trace.

The CMOS processing used to fabricate both N- and P-channel MOS transistors also inherently creates parasitic bipolar transistors — both NPNs and PNPs. Latch-up is caused when these parasitic transistors are inadvertently turned on.

So long as the voltages applied to the package pins of the CMOS IC remain within the limits of the power supply voltages (usually 0 to 5V), the parasitic bipolar transistors remain dormant. However, when either negative voltages or positive voltages greater than the V_{CC} supply voltage are applied to input or output pins, the parasitic bipolar transistors might turn on and cause latch-up.

Figure 6 shows a cross section of a typical CMOS inverter using a P-channel pull-up transistor and an N-channel pull-down transistor. Also shown is an N-channel output driver that is isolated from the CMOS inverter by a guard ring (channel stopper). The latter is necessary to prevent parasitic MOS transistors between devices. P+ guard rings surround N-channel devices, and N+ guard rings surround P-channel devices. The parasitic SCR (PNPN) and bias generator appear in *Figure 7*, which does not show the output driver schematic.

For latch-up to occur, two conditions must be satisfied: The product of the betas of the NPN and PNP transistors must be greater than one, and a trigger current must exist that turns on the SCR.

Because the SCR structure in bulk CMOS cannot be eliminated, the task of preventing latch-up is reduced to keeping the SCR from turning on. If either R_{well} or R_{sub} equal 0, the SCR cannot turn on. This is because the base and emitter of the PNP transistor are tied together and thus the base/emitter junction cannot be forward biased; and the base/emitter junction of the NPN cannot be forward biased because the base is connected to ground. Note, however, that the NPN can be turned on by a negative voltage on the output pin if the right end of R_{sub} is grounded.

Preventing Latch-Up

The traditional cures for latch-up include increased horizontal spacing, diffused guard rings, and metal straps

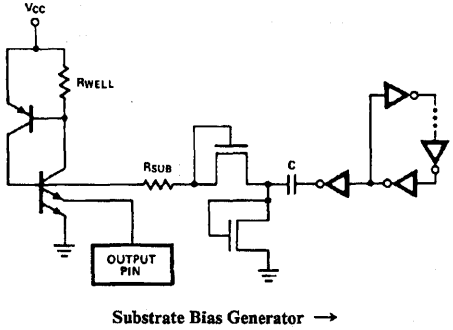


Figure 6. Parasitic SCR and Bias Generator

to critical areas. These solutions are obviously opposite to the goal of greater density.

A brute-force approach that has been successful in reducing latch-up has been to increase the conductivity of the N well and the substrate. Changing the well conductivity is unacceptable because it affects the characteristics of the P-channel MOS transistors. Using an epitaxial layer to reduce the substrate resistivity (R_{sub}) is also unacceptable because the price per wafer with a P+ epi-layer is approximately three times the cost of the industry-standard 5-inch, 50Ω per square, P- wafer.

Cypress uses several design techniques in addition to careful circuit layout and conservative design rules to avoid latch-up.

Conventional CMOS technology uses a P-channel MOS transistor as a pull-up device on the output drivers. This has the advantage of being able to pull the output voltage High to within 100 mV of the positive voltage supply. However, this is of marginal value when TTL compatibility is required. In addition, the P-channel pull-up transistor is sensitive to overshoot and introduces another vertical PNP transistor that further compounds the latch-up problem. Cypress uses N-channel pull-up transistors that eliminate all of these problems and still maintain TTL compatibility.

Cypress is the first company to use a substrate bias generator with CMOS technology. The bias generator keeps the substrate at approximately -3V DC, which serves several purposes.

The parasitic diodes shown in *Figure 5* cannot be forward biased unless the voltage at an input pin is at least one diode drop more negative than -3V. This translates into increased device tolerance to undershoot at the input pins caused by inductance in the leads. If the undershoot is larger than 3V, the output impedance of the bias generator itself is sufficient to prevent trigger current from being generated.

The same reasoning applies to negative voltages at the output pins (*Figure 7*). To turn on the NPN transistor, the voltage at the output pin must be at least one V_{BE} more negative than -3V.

To protect the core of the die from free-floating holes and stray currents, Cypress uses a diffused collection guard ring that is strapped with metal and connected to the bias generator. This provides an effective wall against transient currents that could cause mis-reading of the EPROM cells.

References

Woods, Murray H. "An E-PROM's integrity starts with its cell structure," *Electronics* magazine, August 14, 1980, pg. 132.

Rosenberg, Stuart. "Tests and screens weed out failures, project rates of reliability," *Electronics* magazine, August 14, 1980, pg. 136.

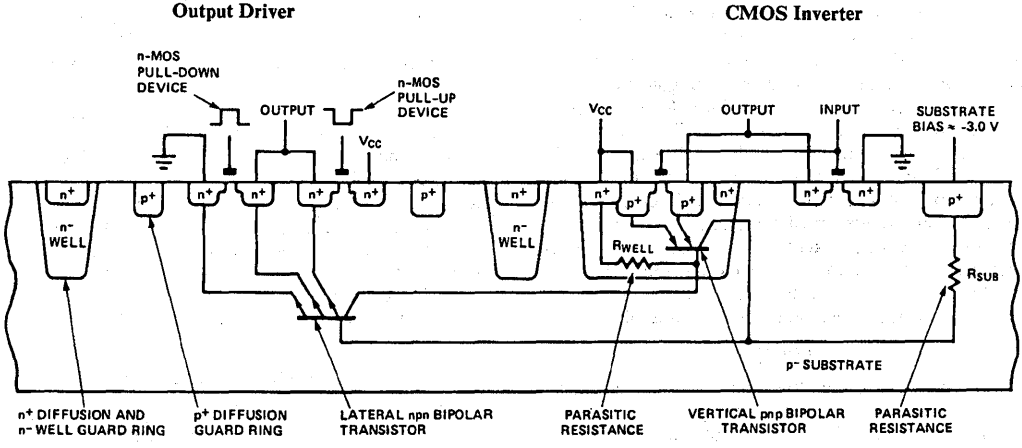


Figure 7. CMOS Cross Section and Parasitic Circuits

CYPRESS SEMICONDUCTOR

Are Your PLDs Metastable?

This application note provides a detailed description of the metastable behavior in PLDs from both circuit and statistical viewpoints. Additionally, the information on the metastable characteristics of Cypress PLDs presented here can help you achieve any desired degree of reliability.

Metastable is a Greek word meaning "in between." Metastability is an undesirable output condition of digital logic storage elements caused by marginal triggering. This marginal triggering is usually caused by violating the storage elements' minimum set-up and hold times.

In most logic families, metastability is seen as a voltage level in the area between a logic High and a logic Low. Although systems have been designed that did not account for metastability, its effects have taken their toll on many of those systems.

In most digital systems, marginal triggering of storage elements does not occur. These systems are designed as synchronous systems that meet or exceed their components' worst-case specifications. Totally synchronous design is not possible for systems that impose no fixed relationship between input signals and the local system clock. This includes systems with asynchronous bus arbitration, telecommunications equipment, and most I/O interfaces. For these systems to function properly, it is necessary to synchronize the incoming asynchronous signals with the local system clock before using them.

Figure 1 shows a simple synchronizer, whose synchronous input comes from outside the local system. The synchronizer operates with a system clock that is synchronous to the local system's operation. On each leading edge of this system clock, the synchronizer attempts to capture the state of the asynchronous input. *Figure 2* shows the expected result. Most of the time, this synchronizer performs as desired.

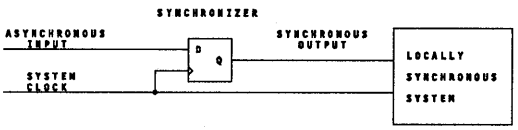


Figure 1. Simple Synchronizer

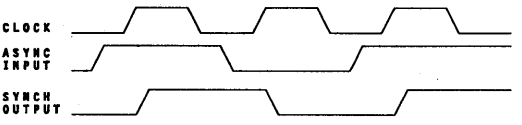


Figure 2. Expected Synchronizer Output

Digital systems are supposed to function properly all the time, however. But because there is no direct relationship between the asynchronous input and the system clock, at some point the two signals will both be in transition at very nearly the same instant. *Figure 3* shows some of the synchronizer's possible metastable outputs when this input condition occurs. These types of outputs would not occur if the synchronizer made a decision one way or the other in its specified clock-to-output time. A flip-flop, when not properly triggered, might not make a decision in this time. When improperly triggered into a metastable state, the output might later transition to a High or a Low or might oscillate.

When other components in the local system sample the synchronizer's metastable output, they might also become metastable. A potentially worse problem can occur if two or more components sample the metastable signal and yield different results. This situation can easily corrupt data or cause a system failure.

Such system failures are not a new problem. In 1952, Lubkin (*Reference 1*) stated that system designers, includ-

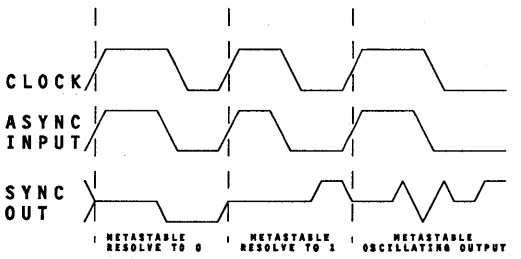


Figure 3. Possible Metastable States of Synchronizer

ing the designers of the ENIAC, knew about metastability. The accepted solution at that time was to concatenate an additional flip-flop after the original synchronizer stage (*Figure 4*). This added flip-flop does not totally remove the problem but does improve reliability. This same solution is still in wide use today.

Recovery from metastability is probabilistic. In the improved synchronizer, the first flip-flop's output might still be in a metastable state at the end of the sample clock period. Because the flip-flops are sequential, the probability of propagating a metastable condition from the second flip-flop stage is the square of the probability of the first flip-flop remaining metastable for its sample clock period. This type of synchronizer does have the drawback of adding one clock cycle of latency, which might be unacceptable in some systems.

As system speeds increase and as more systems utilize inputs from asynchronous external sources, metastability-induced failures become an increasingly significant portion of the total possible system failures. So far, no known method totally eliminates the possibility of metastability. However, while you cannot eliminate metastability, you can employ design techniques that make its probability relatively small compared with other failure modes.

Explanation of Metastability

In a flip-flop, a metastable output is undefined or oscillates between High and Low for an indefinite time due to marginal triggering of the circuit. This anomalous flip-flop behavior results when data inputs violate the specified set-up and hold times with respect to the clock.

In the case of a D-type flip-flop, the data must be stable at the device's D input before the clock edge by a time known as the set-up time, t_s. This data must remain stable after the clock edge by a time known as the hold time, t_h (*Figure 5*). The data must satisfy both the set-up and hold times to ensure that the storage device (register, flip-flop, latch) stores valid data and to ensure that the outputs present valid data after a maximum specified clock-to-output delay t_{co_max}. As used in this application note, t_{co_max} refers to the interval from the clock's rising edge to the time the data is valid on the outputs. In most cases, t_{co_max} equals the maximum t_{co} found in data sheets, as opposed to the average or typical t_{co} value.

If the data violates either the set-up or hold specifications, the flip-flop output might go to an anomalous state for a time greater than t_{co_max} (*Figure 5*). The outputs can take anywhere from an additional few hundred picoseconds to tens of microseconds to reach a valid output level. The amount of additional time beyond t_{co_max} required for the outputs to reach a valid logic level is known as the metastable walk-out time. This walk-out time, while statistically predictable, is not deterministic.

Figure 6, from *Reference 2*, shows the variation in output delay with data input time. The left portion of the graph shows that when the data meets the required set-up time, the device has valid output after a predictable delay, which equals t_{co}. The middle portion of the graph indicates the metastable region. If the data transitions in this region, valid output is delayed beyond t_{co_max}. The closer the input transitions to the center of the metastable region, violating the device's triggering requirements, the longer the propagation delay. If the data transitions after the metastable region, the device does not recognize the input at that clock edge, and no transition occurs at the output. As given in *Reference 3*, you can predict the region t_w, where data transitions cause a propagation delay longer than t, from the formula:

$$t_w = t_{co}\, e^{\frac{-(t - t_{co})}{\tau}} \qquad \textit{Eq. 1}$$

where τ depends on device-specific characteristics such as transistor dimensions and the flip-flop's gain-bandwidth product.

Figure 7 shows another way of looking at metastability. A flip-flop, like any other bistable device, has two minimum-potential energy levels, separated by a maximum-energy potential. A bistable system has stability at either of the two minimum-energy points. The system can also have temporary stability — metastability — at the energy maximum. If nothing pushes the system from the maximum-energy point, the system remains at this point indefinitely.

A hill with valleys on either side is another bistable system. A ball placed on top of the hill tends to roll toward one of the minimum-energy levels. If left undisturbed at the top, the ball can remain there for an indeterminate amount of time. As this figure indicates, the characteristics of the top of the hill as well as natural factors affect how long the ball stays there. The steepness of the hill is analogous to the gain-bandwidth product of the flip-flop's input stage.

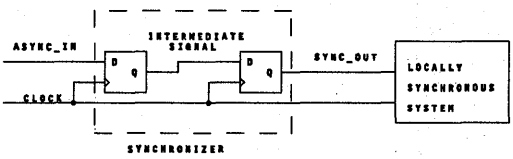


Figure 4. Two-Stage Synchronizer

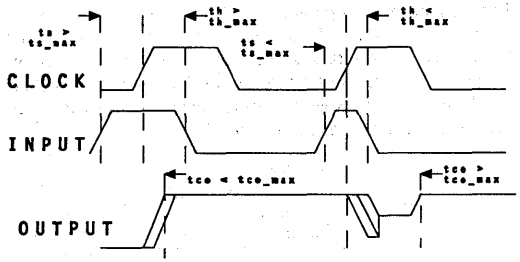


Figure 5. Triggering Modes of a Simple Flip-Flop

Causes of Metastability

Systems with separate entities, each running at different clock rates, are called globally asynchronous systems (*Reference 4*). The entities might include keyboards, communication devices, disk drives, and processors. A system containing such entities is asynchronous because signals between two or more entities do not share a fixed relationship.

Metastability can occur between two concurrently operating digital systems that lack a common time reference. For example, in a multiprocessing system, it is possible that a request for data from one system can occur at nearly the exact moment that this signal is sampled by another part of the system. In this case, the request might be undefined if it does not obey the set-up and hold time of the requested system.

When globally asynchronous systems communicate with each other, their signals must be synchronized. Arbitration must occur when two or more requests for a shared resource are received from asynchronous systems. An arbiter decides which of two events should be serviced first. A synchronizer, which is a type of arbiter with a clock as one of the arbited signals, must make its decision within a fixed amount of time. A device can synchronize an input signal from an external, asynchronous device in cases such as a keyboard input, an external interrupt, or a communication request.

Care must be taken when two locally-synchronous systems communicate in a globally-asynchronous environment. A synchronization failure occurs when one system samples a flip-flop in the other system that has an undefined or oscillating output. This event can distribute non-binary signals through a binary system (*Reference 5*).

In synchronizers, the circuit must decide the state of the data input at the clock input's rising edge. If these two signals arrive at the same time, the circuit can produce an output based on either decision, but must decide one way or the other within a fixed amount of time.

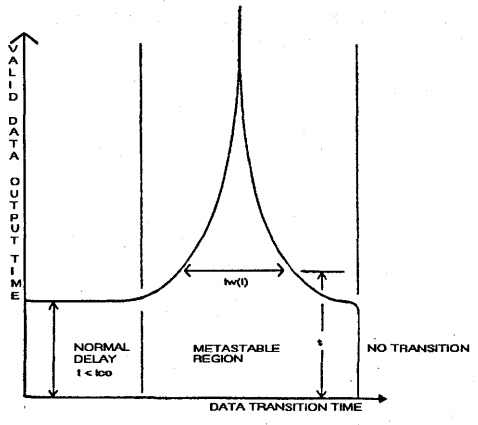


Figure 6. Output Propagation vs. Data Transition

Figure 7. Graphical View of a Bistable System

Attacking Metastability

The design of synchronous systems is much different than the design of globally-asynchronous systems. The design of a synchronous digital system is based on known maximum propagation delays of flip-flops and logical gates. Asynchronous systems by definition have no fixed relationship with each other, and therefore, any propagation delay from one locally-synchronous system to the next has no physical meaning.

Two different methods are available to produce locally-synchronous systems from globally-asynchronous systems. The first method involves creating self-timed systems. In a self-timed system, the entity that performs a task also emits a signal that indicates the task's completion. This handshaking signal allows the use of the results when they are ready instead of waiting for the worst-case delay. Such handshaking signals allow communications between locally-synchronous systems.

The advantage of the self-timed method is that it permits machines to run at the average speed instead of the worst-case speed. The disadvantages are that a self-timed system must have extra circuitry to compute its own completion signals and extra circuitry to check for the completion of any tasks assigned to external entities.

Petri Nets, data flow machines, and self-timed modules all use the self-timed method of communication among locally-synchronous systems. Self-timed structures do not completely eliminate metastability, however, because they can include arbiters that can be metastable. Most systems do not include self-timed interfaces due to the additional circuitry and complexity.

The second method of producing locally-synchronous systems from globally-asynchronous systems is the simple synchronizer. This is the most common way of communicating between asynchronous objects. The metastability errors that might arise from these systems must be made to play an insignificant role when compared with other causes of system failure.

Many metastability solutions involve special circuits (*References 6* and *7*). Some of these solutions do not reduce metastability at all (*Reference 13* and *8*). Others, however, do reduce metastability errors by pushing the occurrence of metastability to a place where sufficient time is available for resolving the error. Most of these circuits are system dependent and do not offer a universal solution to metastability errors.

The easiest and the most widely used solution is to give the synchronizing circuit enough time to both

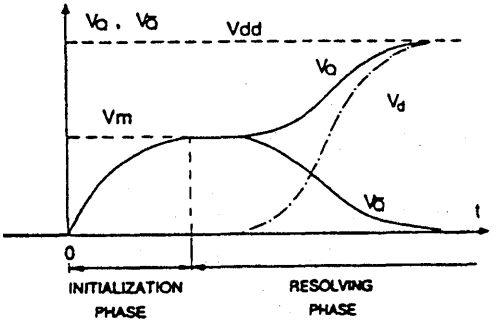


Figure 8. Two Phases of Metastability

synchronize the signal and resolve any possible metastable event before other parts of the system sample the synchronized output. This solution requires knowledge of the metastable characteristics of the device performing the synchronization.

Many semiconductor companies have developed circuits such as arbiters, flip-flops, and latches that are specifically designed to reduce the occurrence of metastability. Although these parts might have good metastability characteristics, they have very limited application. The circuits can only function as flip-flops or arbiters and do not have the flexibility of PLDs. Cypress Semiconductor has designed the flip-flops in the company's PLDs to be metastable hard. This allows you to use Cypress PLDs in a wide range of systems requiring synchronization.

Circuit Analysis of Metastability

Many authors have written papers detailing the analysis of metastability from a circuit standpoint (*Reference 5, 7, 8, 9, 10, 11,* and *12*). In *Reference 11*, for example, Kacprzak presents a detailed analysis of an RS flip-flop's metastable operation. He states that a flip-flop has two stages of metastable operation (*Figure 8*).

During the initialization phase, the Q and $\overline{Q}$ outputs move simultaneously from their existing levels to the metastable voltage V_m, which is the voltage at which $V_q = \overline{V}_q$.

The second or resolving phase occurs when the outputs once again drift toward stable voltages. Once a flip-flop has entered a metastable state, the device can stay there for an indeterminate length of time. The probability that the flip-flop will stay metastable for an unusually long period of time is zero, however, due to factors such as noise, temperature imbalance within the chip, transistor differences, and variance in input timing. During the second phase of metastability, for very small deviations around the metastable voltage, V_m, the flip-flop behaves like two cross-coupled linear amplifier stages that gain $V_d = V_q - \overline{V}_q$. When the gain of the cross-coupled loop exceeds unity, the differential voltage increases exponentially with time.

The length of time the flip-flop takes to resolve cannot be exactly determined. The probability that the flip-flop will resolve within a specific length of time, however, can be predicted. This probability depends on the electrical parameters of the flip-flop acting as a linear amplifier around the metastability voltage. The solution (*Reference 11*) to the differential voltage $V_d(t)$ driving the resolving phase is given by

$$V_d(t) = V_d(t_0)\, e^{\frac{(t-t_0)}{\tau}} \qquad \text{Eq. 2}$$

where τ depends directly on the amplifier gain and capacitance, and where $V_d(t_0)$ represents the differential voltage at some time t_0. You can use this equation to determine the length of time that the output voltage will take to drift from the metastable voltage V_m to a specified voltage difference V_d.

Horstmann (*Reference 5*) states that a flip-flop, like any other system with two stable states, can be described by an energy function with two local energy minima where P(x) = 0 (*Figure 9*). Any bistable system has at least one metastable state, which is an unstable energy level within the system and represents the local maximum of the energy function. The system's gradient can be represented by a force, F(x), that is zero at stable and metastable states (inflection points of the energy function).

Figure 10 shows a simplified first-order model of an RS flip-flop used to predict and visualize metastability. A flip-flop energy transfer curve (*Figure 11*) shows the relationship between the two outputs. The two stable states are local energy minima of the system. The metastable state, M, is a local energy maximum and represents an unstable state with loop gain near M that is greater than one.

Figure 12 show the trigger line for the first-order approximation of the flip-flop. The dashed line RS represents the device's normal trigger line, which does not follow the transfer curve because, during triggering, the feedback loop has not been established. If at varying points along the trigger line the feedback loop is re-established, the nodes of the device follow the curves that lead to the

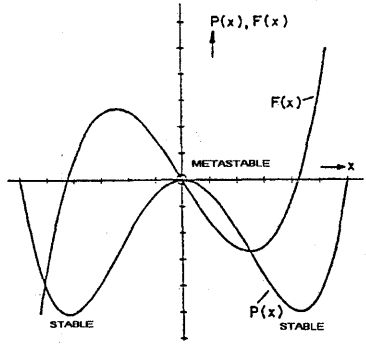


Figure 9. Energy/Force Function of a Bistable System

line S_0 - S_1. Once on this line, the circuit exponentially drifts toward stability at either S_0 or S_1, depending on which side of the line $Q = \overline{Q}$ the feedback loop was re-established. The curves are solutions to the first-order model circuit equations for the device shown in *Figure 10.*

When the feedback loop is restored near the line $Q = \overline{Q}$, the system moves toward the unstable state M and can take an indefinite amount of time to exit from this metastable state. You can see this from the graph by noticing that S_0 and S_1 are equally likely solutions for system stability from M. Once the feedback loop is re-established, the system exponentially decays toward M and then exponentially grows toward S_0 or S_1.

Figure 13 shows the system's possible trigger events using the implied time scale of the state-space curves. The solution of these simplified first-order equations indicates that the fastest metastable resolution time occurs when the circuit's gain-bandwidth product is maximized.

Flannagan (*Reference 12*), in an attempt to maximize the gain-bandwidth product, solves simplified flip-flop equations to determine the phase trajectory near the metastable point. His results, which are supported by other authors, indicate that p and n devices with equal geometries produce the optimal gain-bandwidth product for metastable event resolution.

Statistical Analysis of Metastability

To begin the analysis of metastability, assume that the flip-flop's probability of resolving its metastable state does not depend on its previous metastable state. In other words, the metastable device has no memory of how long it has been in a metastable region. The analysis of metastability also assumes that the flip-flop's probability of resolving its metastable state in a given time interval does not depend on the metastable resolution in another disjoint time interval. The probability that a metastable event will resolve in a given interval (0,t) is only proportional to the length of the interval.

These assumptions yield an exponential distribution that describes the probability that the flip-flop resolves its metastability at a time t. The exponential distribution has the form

$$f(x) = \mu e^{-\mu t} \qquad Eq.\ 3$$

where μ is the expected value of metastability resolution per unit time (settling rate).

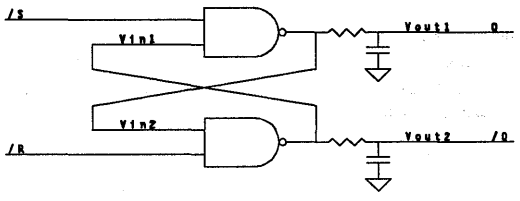


Figure 10. First-Order Flip-Flop Approximation

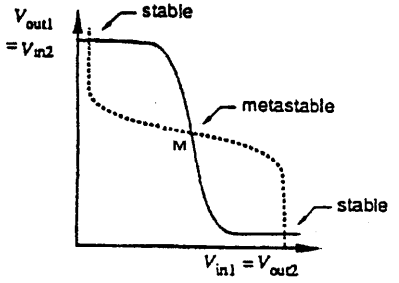


Figure 11. Energy Transfer Diagram of Simple RS Flip-Flop

Using this equation and given that the flip-flop was metastable at time t = 0, the probability of a metastable event lasting a time t or longer is

$$P(met_t \mid met_{t=0}) = \int_t^\infty \mu e^{-\mu t} dt = e^{-\mu t} \qquad Eq.\ 4$$

The next part of the analysis involves the probability that the flip-flop is metastable at time t = 0. This part of the analysis assumes that the probability that the data transitions in a given time interval depends only on the length of the interval. A Poisson process with rate f_d describes the probability of the data transitioning at a time t:

$$p(x) = \frac{e^{-f_d t}(f_d t)^x}{x!} \qquad Eq.\ 5$$

where x is the number of transitions.

If a data transition within a bounded time interval, W, of the clock edge causes a metastable condition, the expected number of transitions of this Poisson process with rate f_d in time interval W is

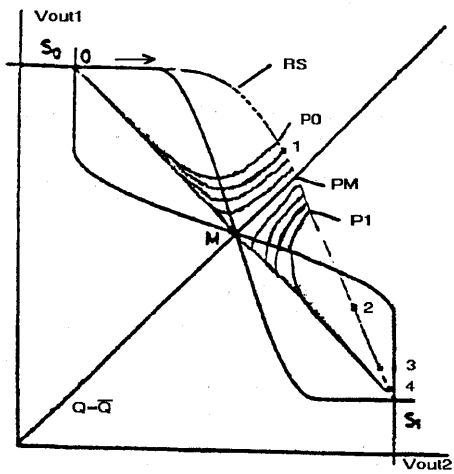


Figure 12. Energy Transfer Curves showing Trigger Paths

$$E(X)=\sum_{x=0}^{\infty}\frac{x e^{-f_d W}(f_d W)^x}{x!}=f_d W \qquad \textit{Eq. 6}$$

Because this expected number of transitions is the same as the probability that the flip-flop is metastable at t = 0, the equation for the probability at t = 0 is

$$P(met_{t=0})=f_d W \qquad \textit{Eq. 7}$$

Using *Equations 5* and *7*, the probability that a given clock cycle results in metastability that lasts at most a time t is

$$P(met_t)=P(met_t \mid met_{t=0})P(met_{t=0}) \qquad \textit{Eq. 8}$$
$$=f_d W e^{-\mu t}$$

Substituting $\frac{1}{t_{sw}}$ for μ allows this variable to be expressed as a settling time constant of the flip-flop. Further, a synchronization failure for a given clock cycle exists whenever a metastable event lasts a specified time (t_r) or longer. Using these two substitutions, the probability that the flip-flop is metastable in a given clock cycle is:

$$P(fail_{1\ clock})=f_d W e^{\frac{-t_r}{t_{sw}}} \qquad \textit{Eq. 9}$$

Because the data transitions are independent, the number of failures in n clock cycles has a binomial distribution with an expected number of failures:

$$E(fail_{n\ cycles})=n\,P(fail_{1\ cycle}) \qquad \textit{Eq. 10}$$

Assuming a sample clock frequency, f_c, that represents the number of clock cycles, n, per unit time, the expected number of failures per unit time is

$$E(fail_{unit\ time})=f_c f_d W e^{\frac{-t_r}{t_{sw}}} \qquad \textit{Eq. 11}$$

Assuming that all data transitions are independent and that the clock has a fixed period, the mean time between failures (MTBF) is

$$MTBF=\frac{1}{E(fail_{unit\ time})}=\frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W} \qquad \textit{Eq. 12}$$

where MTBF is a measure of how often, on the average, a metastable event lasts a time t_r or longer.

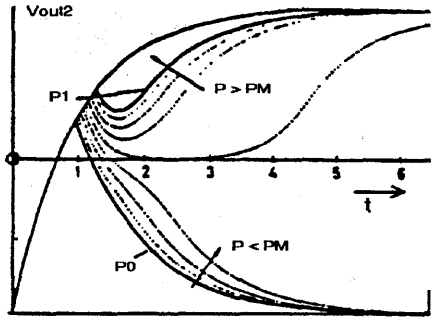


Figure 13. Time Scale Showing Trigger Paths

Metastability Data

Equation 12 shows a strong resemblance to *Equation 2* that is based on the predictions of the first-order circuit analysis of an RS flip-flop. In fact, the metastability resolving time constant, t_{sw}, is directly related to the variable τ, which is based on the flip-flop's gain-bandwidth product.

The device-dependent variable W depends mostly on the window of time within which the combination of the input and clock generate a metastable condition. This parameter also depends on process, temperature, and voltage levels. The MTBF equation is usually plotted with t_r (the resolving time allowed for metastable events) on the X axis and the natural log of the MTBF plotted on the Y axis (*Appendix*). Because the metastability equation is plotted on semi-log paper, the graph of t_r vs ln(MTBF) is a line described by the equation

$$\ln(MTBF)=\frac{t_r}{t_{sw}}-\ln(f_c f_d W) \qquad \textit{Eq. 13}$$

Graphically, the parameter t_{sw} is 1/slope of the line on this graph. The equation for t_{sw} from the graph is

$$t_{sw}=\frac{t_{r1}-t_{r2}}{\ln(MTBF_1)-\ln(MTBF_2)} \qquad \textit{Eq. 14}$$

To determine how often, on the average, a given synchronizer in a system will go metastable (MTBF), you must know the two device-specific parameters W and T_{sw}, which should be available from the manufacturer. *Table 1* lists these values for Cypress PLDs. Additional values you need are the average frequency of both the system data and the synchronizer clock and the amount of time after the synchronizer's maximum clock-to-Q time that is allowed to resolve metastable events.

For example, consider the method for determining the MTBF for a Cypress PALC22V10 registered PLD used as a synchronizer in a system with the following characteristics:

W = 0.125 ps

t_{sw} = 190 ps

f_c = system clock frequency = 25 MHz

f_d = average asynchronous data frequency = 10 MHz

In addition to these values, the PLD's maximum operating frequency, f_{max}, is taken directly from the data sheet. The frequency is specified as the internal feedback maximum operating frequency. It is calculated as

$$f_{max}=\frac{1}{t_{cf}+t_s}=41.6\,MHz$$

where t_{cf} is the clock-to-feedback time. If the data sheet does not specify t_{cf}, you can use t_{co} as t_{cf}'s upper bound.

Using f_{max}, you calculate the amount of time that a metastable event is allowed to resolve, t_r, with

$$t_r=\frac{1}{f_c}-\frac{1}{f_{max}}=\frac{1}{25\,MHz}-\frac{1}{41.6\,MHz}=16\,ns$$

Now you enter these values into the MTBF equation, making sure to keep all units in seconds:

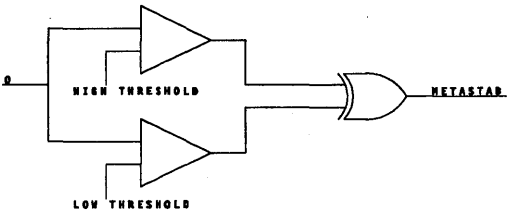


Figure 14. Intermediate Voltage Sensor

$$MTBF = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W}$$

$$= \frac{e^{\frac{16\times10^{-9}s}{190\times10^{-12}s}}}{25\times10^{6}s^{-1}\times20\times10^{6}s^{-1}\times0.125\times10^{-12}s}$$

$$= 59.7\times10^{33}s = 1.89\times10^{27}years = Almost\ forever$$

If the operating frequency of the system, f_c, is simply changed to 33.3 MHz,

$$MTBF = \frac{e^{\frac{6\times10^{-9}s}{190\times10^{-12}s}}}{33.3\times10^{6}s^{-1}\times20\times10^{6}s^{-1}\times0.125\times10^{-12}s}$$

$$= 623\times10^{9}\,s$$

the system fails, on the average, about every 19,700 years — still beyond the system's normal lifetime.

And if f_c is changed to f_{max} (41.6 MHz),

$$MTBF = \frac{e^{\frac{0\times10^{-9}s}{190\times10^{-12}s}}}{41.6\times10^{6}s^{-1}\times20\times10^{6}s^{-1}\times0.125\times10^{-12}s}$$

the system fails, on the average every 9.62 ms.

A 16-ns difference in resolve time, t_r, results in almost 36 orders of magnitude difference in MTBF. Obviously, accurate data is needed to design a system with a high degree of reliability without being overly cautious.

Characterization of Metastability

Many authors (*References 6, 8, 9, 10, 11,* and *12*) have performed numerous experiments on circuits to predict the likelihood of device metastability. These researchers have used several testing theories and apparatus

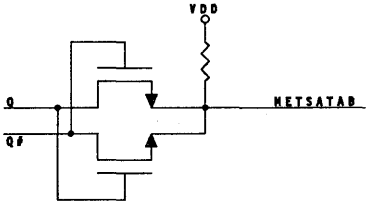


Figure 15. Output Proximity Sensor

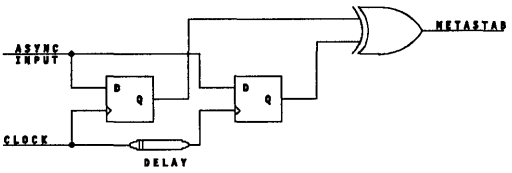


Figure 16. Late Transition Sensor

that can be classified into three basic types (*Reference 14*).

Intermediate voltage sensors constitute the first type. Two voltage comparitors determine whether the output voltage, Q, lies between two given voltages. The fixture produces an error output if Q has a level that is neither High nor Low, hence metastable. *Figure 14* shows an intermediate voltage sensor.

The second type of apparatus uses an output proximity sensor to determine if the Q and $\overline{Q}$ outputs have approximately the same voltages, which would indicate that the device is metastable. *Figure 15* shows an output proximity sensor.

The last type of apparatus uses a late-transition sensor to test for metastability. Note that if one or more gates separate the sensor from the metastable signal, the metastability might not be detected. The test circuitry must infer the occurrence of metastability by some other means. *Figure 16* shows an example of a late-transition sensor. The sample input is detected at time t_1, then at a later time t_2. If these two signals disagree, the device under test was metastable at t_1.

Information from Manufacturers

Many semiconductor companies provide metastability data on their parts. However, most companies do not present the data in a format the engineer can use. They either present inconclusive and incomplete data or they assume the engineer can use the data without further explanation. Few companies compare their devices with similar devices to provide correlation between comparable devices.

PLD manufacturers provide little data largely because of a fear that telling the design community that devices can fail in synchronizing applications will cause designers to use a competitor's parts. The truth: No company can provide a device that is guaranteed not to become metastable if used as a synchronizer. At a given operating frequency, with a given asynchronous input, and given enough time, the device becomes metastable.

Cypress provides you with data you can use to build a system to any given level of reliability when using Cypress PLDs. Cypress has performed numerous tests and collected extensive data on Cypress PLDs, as well as PLDs from other companies. This data gives you a perspective of the parts that are best suited for a specific application. Specific data on the metastability characteristics of Cypress PLDs is found in this application note

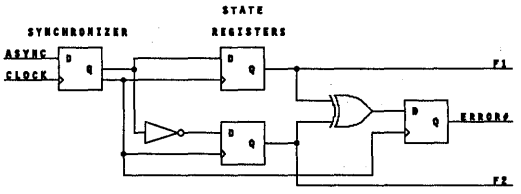


Figure 17. Metastability Test Circuit

in the "Test Results" section. Metastability data collected by Cypress for other companies' PLDs is available upon request.

The Test Circuit

Cypress uses a test that falls into the category of the late-transition detection. Directly measuring the outputs of the flip-flop in a PLD are impossible due to the additional circuitry that lies between the flip-flop and the outside world. The metastability detection circuitry must, instead, infer the flip-flop's state.

Figure 17 shows the metastability test circuit implemented in each test PLD. This circuit allows the PLD under test to effectively test itself. The device under test will both produce and record metastable conditions.

Figure 18 is a state diagram showing the operation of the device. During normal operation, the two flip-flops' outputs (F_1, F_2) transition between states S_1 and S_2, depending on the synchronizer's state. During normal operation, the Exclusive-OR on these outputs produces a High. This indicates either that metastability has not occurred within the device or that metastability that has occurred has resolved before the next clock cycle.

If a metastable event cannot resolve before the next clock cycle, the state machine move to states S_3 or S_4. In this case, the state flip-flops have interpreted the signal from the synchronization register differently; Exclusive-ORing this signal produces a Low at the device's output, indicating that unresolved metastability has occurred.

This test circuit does not catch all metastable events. Specifically, it does not record metastable events that resolve before the next clock cycle. But metastability causes an error only when it has not resolved by the time the signal is needed. The Cypress tests thus reveal the information designers need to know: how often metastability creates an error in the system.

The test circuit also includes the ability to check the maximum operating frequency of the device under test (*Figure 19*). At each clock edge, the first register's output toggles. When the device reaches its maximum operating frequency, the PLD array cannot resolve the changing signal fast enough to produce a valid output. At this speed, one register might resolve the signal correctly and one might not, or both might produce invalid signal resolutions. In any case, when Exclusive-ORing the state T_1/T_2 of the two maximum-frequency testing registers results in anything other than a High, the part's maximum operating frequency is exceeded.

The Test Board

A four-layer printed circuit board with two signal planes, a ground plane, and a power plane is used to perform the metastability measurements. Using this four-

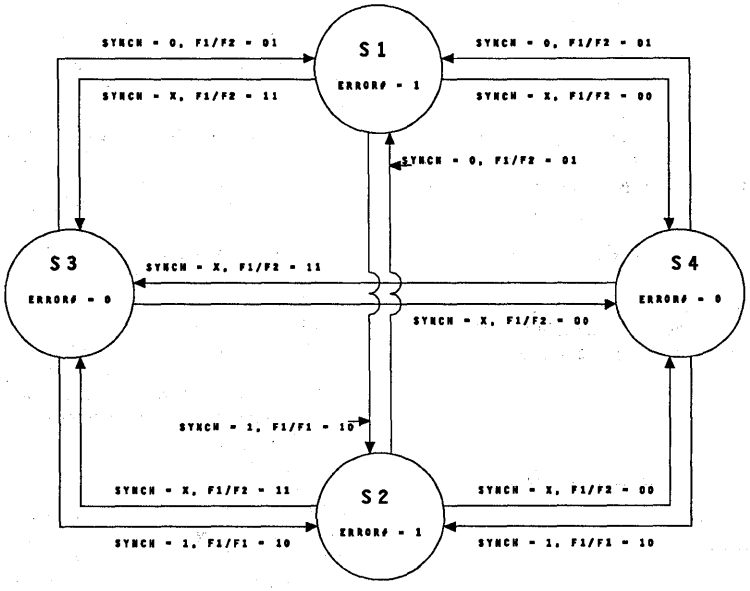


Figure 18. Metastability Testing State Diagram

layer board gives a quiet testing environment with reliable, repeatable results. *Figure 20* shows a block diagram of the test board, with the complete schematic shown in *Figure 21*. The device under test (DUT) is decoupled with 0.01-µF and 100-pF capacitors. The test circuit is designed to fit all industry-standard and Cypress-proprietary PLDs. The socket allows DUT pins 1, 2, and 4 to serve as clock pins. Pin 3 is the device's asynchronous input. The $\overline{\text{ERROR}}$ condition is located on pin 27 of a 28-pin device, and the $\overline{\text{FAIL}}$ condition is on pin 20. Two additional outputs, F_1 and F_2, monitor the state of the metastability test circuit flip-flops.

All inputs and outputs connect with BNC connectors located around the board. The clock line, which is terminated with a 50Ω resistor to match the coax input impedance, is buffered with a 74AS04 and isolated from other signals by a ground trace. The input line is also terminated with a 50Ω resistor and buffered with a 74AS04. Four PLDs drive a four-digit LED display that counts metastability occurrences.

After going Low in response to a metastable event, the $\overline{\text{ERROR}}$ signal automatically transitions High again at the next system clock. This Low-to-High pulse produces a clock to the input of the first PLD, which in turn increments the display of metastable events. When a digit reaches 9, the next occurrence of metastability generates a cascade signal to the next higher digit.

In this way, the test board can record a maximum of 9,999 metastable events. If a metastable event is received at 9,999, all LEDs switch to E, indicating that an overflow condition occurred. A reset button resets all counters and initializes the DUT.

Test Setup

Figure 22 shows a block diagram of the test setup used for metastability testing. Two independent pulse generators (Hewlett-Packard 8082As) produce the CLOCK and the ASYNC_IN signal to the test board. A Tektronix DAS9200 logic analyzer records metastable events. A 2465 CTS digital oscilloscope with frequency counter accurately determines the DUT's maximum operating frequency and the ASYNC_IN and CLOCK frequencies.

Test Procedure

Cypress has tested all its PLDs of 28 pins or less. The fastest speed grades of each device type were tested because these devices have the best metastable resolution time and thus make the best synchronizers. Several parts from each device type were tested to ensure an average metastability characteristic for that product. Where possible, parts from different date codes were selected to eliminate variations among different wafer lots.

Testing for a specific device starts by creating the equations used to program the device. *Figure 23* lists the equations for programming the 22V10. All devices were tested using bit maps produced by the PLD ToolKit, except for the CY7C344. The MAX+PLUS development environment was used to produce a design file for this device.

Each part is programmed, then tested for its maximum operating frequency, f_{max}. By attaching the $\overline{\text{FAIL}}$ output to the oscilloscope and observing the clock frequency at which the device started to malfunction ($\overline{\text{FAIL}}$ going Low periodically), the maximum operating frequency for that part is determined. f_{max} indicates the maximum rate at which metastability measurements can be taken with accurate results. Above this frequency, metastable events are indistinguishable from errors caused by exceeding f_{max}.

To determine each device's metastability characteristics, measurements are taken of the number of metastable events that occurred in a given time interval for several different clock and data frequencies.

Equation 13 can be used to describe the graph of the metastability characteristics of the device:

$$\ln(MTBF) = \frac{t_r}{t_{sw}} - \ln(f_c f_d W)$$

The slope of the line, t_{sw}, can be determined only by forcing the Y intercept of the graph ($\ln(f_c f_d W)$) to a constant value when using *Equation 14*:

$$t_{sw} = \frac{t_{r1} - t_{r2}}{\ln(MTBF_1) - \ln(MTBF_2)}$$

Note that t_{sw} is a constant, device-specific parameter.

Because W is also a constant, device-specific parameter, it is only necessary to hold the product $f_c f_d$ constant to make $\ln(f_c f_d W)$ constant. The independent variable t_r is varied by changing f_c to produce changes in the dependent variable ln(MTBF). Decreasing the frequency f_c from its f_{max} value increases the metastable resolu-

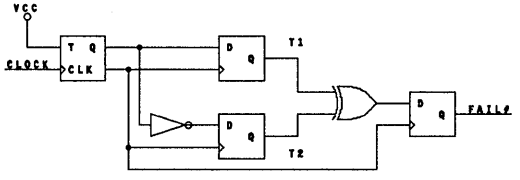


Figure 19. Maximum Operating Frequency Test

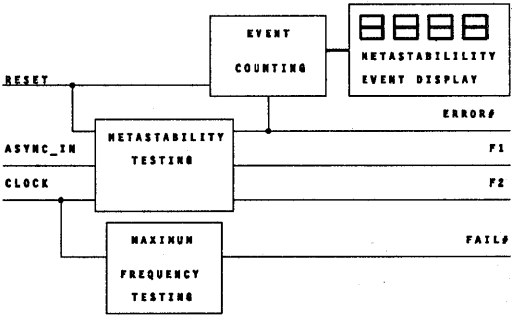


Figure 20. Metastability Test Board Block Diagram

tion time, t_r, and decreases the probability that a metastable event will last longer than t_r.

As f_c is decreased below a certain limit, the MTBF becomes too large to measure accurately. A metastable event occurring every minute is chosen as the upper limit for MTBF measurements. The range of clock rates for metastability testing is then between f_{max} and the metastable-event-per-minute clock rate. Between these two rates, a selected frequency constant ($f_c f_d$) ensures that no point in this range has a clock frequency less than twice the data frequency. This is because a data signal that transitions more than once per clock period cannot be effectively sampled.

After determining this constant, data is taken from several test points within the test range by varying f_c and f_d. The data at each test point is averaged among all test devices, and the equation for the line through these points is determined using a linear regression analysis. The correlation between the line and the data points verifies that the metastability equation accurately describes the test data. From the calculated results, the constants W and t_{sw} are extracted.

Test Results

Table 1 and the *Appendix* list the results of the metastability analysis of Cypress PLDs. *Table 1* also lists the maximum data book operating frequency, f_{max}; the metastability equation constants, W and t_{sw}; the metastability resolve time, t_r, required for a 10-year MTBF; and the process for that part.

You can use this data to determine the maximum metastability resolve time (t_r) that you must use in a system to yield a given degree of reliability. The graphs and constants (W and t_{sw}) can be used with any speed grade of the device, but it is suggested that the fastest speed grade of the specific PLD be used for optimum synchronizer performance. These graphs indicate the time (t_r) and the device's minimum clock period that must be used to produce a desired degree of reliability.

For example, to determine the operating parameters of the Cypress PALC22V10-20 from *Table 1* when using the device as a synchronizer, determine the desired MTBF. With a 10-yr (315×10^6 *s*) MTBF, for instance, a synchronization failure will occur once every 10 years on the average. The maximum operating frequency (f_{max})

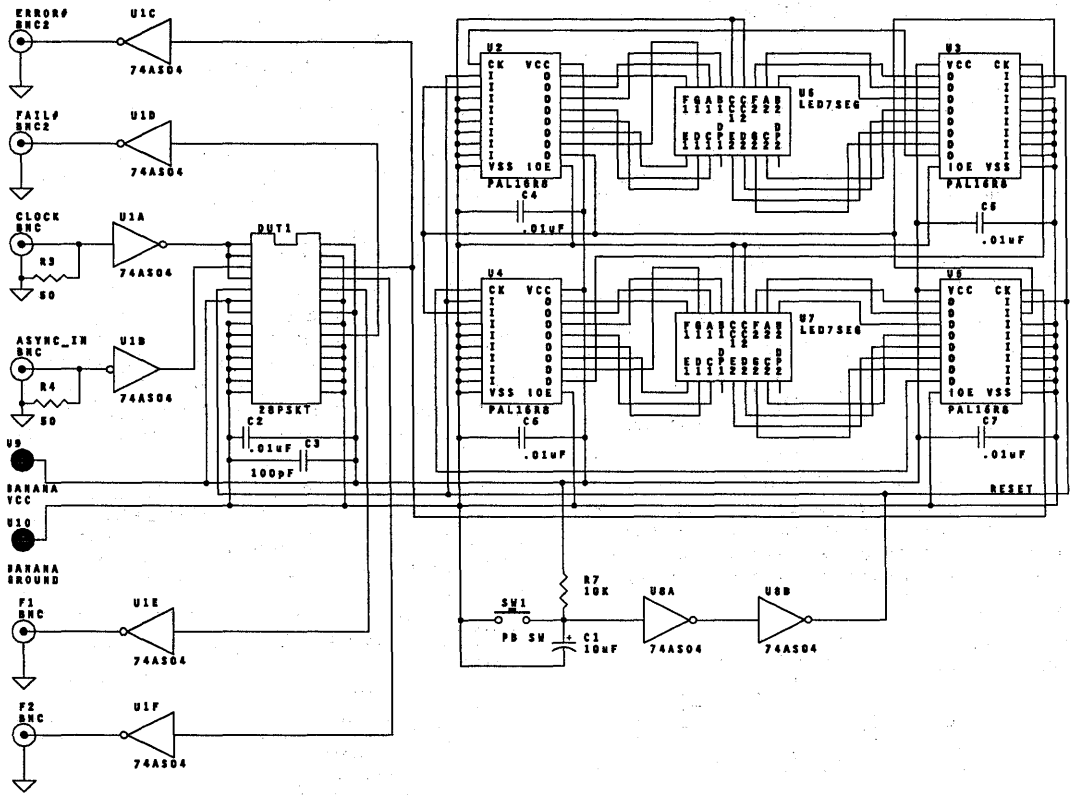


Figure 21. Metastability Test Board Schematic

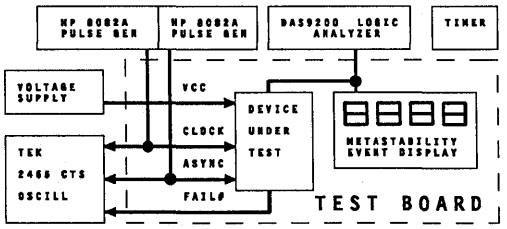


Figure 22. Metastability Test Setup

from the PALC22V10's data sheet is 41.6 MHz. From this information, you can determine the minimum time (t_r) beyond the device's minimum operating period that must be added for metastability resolution:

$$MTBF = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W}$$

$$t_r = t_{sw}(\ln(MTBF) + \ln(f_c f_d W))$$

$$t_r = (0.190 \times 10^{-9} s)[\ln(315 \times 10^{6} s) + \ln(41.6 \times 10^{6} \times 41.6 \times 10^{6} \times 0.125 \times 10^{-12})]$$

$$= 4.73\ ns$$

This analysis assumes that the clock, f_c, operates at f_{max} (41.6 MHz) and that the average asynchronous data frequency is no more than half the clock frequency. The latter condition ensures effective data sampling by the synchronizer. f_d, as explained in the section "Statistical Analysis of Metastability," represents the rate at which the data changes state. f_d is twice the average frequency of the asynchronous data input because, during any given asynchronous data period, the asynchronous data changes state twice: once from Low to High and again from High to Low. Because either of these state changes can cause a metastable event, f_d must be set to twice the average asynchronous data frequency when determining the worst-case MTBF.

Due to the real-world uncertainty in factors such as trace delays and the skew in clock generators, 5 ns is used instead of 4.74 ns for t_r. The synchronizer's maximum operating frequency, f_c, in this system is then

$$f_c = \frac{1}{t_s + t_{cf} + t_r} = \frac{1}{10\ ns + 12\ ns + 5\ ns} = 37.0\ MHz$$

The effective MTBF using these new values for t_r and f_c is

$$MTBF = \frac{e^{\frac{5 \times 10^{-9} s}{0.190 \times 10^{-9} s}}}{37.0 \times 10^{6} s^{-1} \times 37.0 \times 10^{6} s^{-1} \times 0.125 \times 10^{-12} s}$$

$$= 1.57 \times 10^{9} = 49.7\ yrs$$

Another example focuses on the CY7C330-50 used as a synchronizer in a system whose output registers are clocked at an f_c of 35.7 MHz, and the data has an average frequency of 10 MHz. The MTBF for this device used as a synchronizer is calculated by first determining the metastable resolution time, t_r, allowed for synchronization. The maximum operating frequency of the part is specified in the Cypress Data Book as

$$f_{max} = \frac{1}{t_{co} + t_{is}}$$

where t_{co} in this case specifies the clock-to-feedback delay, and t_s specifies the set-up time of the output registers. t_r is calculated with the equation:

$$t_r = \frac{1}{f_c} - \frac{1}{f_{max}} = \frac{1}{35.7\ MHz} - \frac{1}{50.0\ MHz} = 8\ ns$$

With this result, the MTBF is

$$MTBF = \frac{e^{\frac{8 \times 10^{-9} s}{0..290 \times 10^{-9} s}}}{35.7 \times 10^{6} s^{-1} \times 20.0 \times 10^{6} s^{-1} \times 1.02 \times 10^{-12} s}$$

$$= 1.31 \times 10^{9}\ s = 41.6\ yrs$$

This equation uses the same values for W and t_{sw} with this 50-MHz device as with the 66-MHz device listed in *Table 1*. As stated previously, the constants listed in *Table 1* are valid for all speed grades of a specific device. Also note that the 10-MHz average data frequency is doubled to produce the frequency of data transitions, f_d.

Table 1. Metastability Characteristics of Cypress PLDs

DEVICE	Fmax (MHz)	W (s)	t_{sw} (s)	t_r for 10yr MTBF	Process
PALC16R8-25	28.5	9.503E-12	.515E-9	14.68nS	PROM1
PALC20G10-20	41.6	3.73E-12	.173E-9	4.91nS	PROM1
PALC20RA10-15	33.3	2.86E-12	.216E-9	5.87nS	PROM2
PALC22V10C-10	90.0	8.08E-15	.547E-9	13.0nS	BICMOS
PALC22V10B-15	50.0	55.76E-12	.261E-9	8.19nS	PROM2
PALC22V10-20	41.6	.125E-12	.190E-9	4.73nS	PROM1
CY7C330-66	66.6	1.02E-12	.290E-9	8.12nS	PROM2
CY7C331-20	31.2	.298E-9	.184E-9	5.91nS	PROM2

The last example illustrates how to use a Cypress PALC22V10C-10 as a synchronizer. For a 10-year MTBF, assuming the maximum f_c from the Cypress Data Book and f_d, the required t_r is

$$t_r = (0.547 \times 10^{-9} s)\,[\ln(315 \times 10^{6} s) + \ln(90.9 \times 10^{6} \times 90.9 \times 10^{6} \times 8.08 \times 10^{-15})]$$
$$= 13.0\ ns$$

Using this result, the synchronizer's maximum operating frequency is reduced from 90.9 MHz to

$$f_c = \frac{1}{\frac{1}{f_{max}} + t_r} = \frac{1}{\frac{1}{90.9\ MHz} + 13.0\ ns} = 41.6\ MHz$$

Two-Stage Synchronization

As explained earlier, you can use a second register in series to perform two-stage synchronization (*Figure 4*). This is accomplished by feeding the output of the first synchronization register to the input of the second synchronization register. In PLDs, this method is common

```
C22V10;
{
 Cypress Semiconductor
 Revision: 06/28/90
 These are the equations to perform metastability testing on the PALC22V10
}

CONFIGURE;

CLOCK,                  {CLOCK input on pin 1}
ASYNC_IN(node=3),       {Asynchronous input signal}
RESET(node=5),          {RESET signal}
TSYNC(node=15),         {Synchronization for Fmax }
T1(node=17),            {State node for Fmax}
T2(node=18),            {State node for Fmax}
FAIL(node=16),          {Fmax indication}
SYNC(node=19),          {Synchronization for Meta test}
F1,                     {State node for Meta test }
F2,                     {State node for Meta test }
ERROR,                  {Metastable Event indication}

EQUATIONS;

/SYNC  =        <sum> ASYNC_IN;                 {Synchronize Asynchronous input}

/F1     =       <oe>
                <sum> SYNC;             {Have two registers hold the}
                                                {true and inverted sense of }
/F2     =       <oe>                            {the synchronization register }
                <sum> /SYNC;

/ERROR  =       <oe>
                <sum> /RESET * F1 * /F2         {ERROR# goes low when the XOR }
                        + /RESET * /F1 * F2     {of F1 and F2 is false, ERROR#}
                        + RESET * ERROR;        {also toggles on RESET}

/TSYNC  =       <sum> TSYNC;            {Fmax reg toggles on every clock }

/T1     =       <sum> TSYNC;            {Have two registers hold the}
                                                {true and inverted sense of }
/T2     =       <sum> /TSYNC;           {Fmax reg }

/FAIL   =       <oe>                            {FAIL# goes low when the XOR}
                <sum> T1 * /T2          {of T1 and T2 is false, indicating }
                        + /T1 * T2;             {Fmax has been exceeded }
```

Figure 23. PLD Equations for Metastability Testing

because the first synchronization stage can synchronize the asynchronous input signal, and the second synchronization stage can perform a Boolean function on a combination of the input and output signals. Boolean functions can be performed at either stage; the metastability characteristics listed in *Table 1* apply to PLD registers' asynchronous inputs that are used directly as well as asynchronous inputs used as a Boolean combination of existing inputs and outputs.

When implementing a two-stage synchronizer in a PLD, the probability that a synchronizer is metastable after the second stage of synchronization is the square of the probability that a synchronizer is metastable after the first stage of synchronization. The MTBF equation is

$$MTBF = \left(\frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W}\right)^2$$

From this result, the equation for t_r becomes

$$t_r = \frac{t_{sw}\left(\ln\left(MTBF\right) + 2 \times \ln\left(f_c f_d W\right)\right)}{2}$$

Using this result for a two-stage synchronizer in a Cypress PALC22V10C, the t_r for a 10-year MTBF is reduced from 13.0 ns to

$$t_r = (0.5)(0.547 \times 10^{-9} s)[\ln(315 \times 10^{6} s) + \ln(90.9 \times 10^{6} \times 90.9 \times 10^{6} \times 8.08 \times 10^{-15})]$$
$$= 7.65\ ns$$

The maximum f_c increases from 41.6 MHz to

$$f_c = \frac{1}{\frac{1}{f_{max}} + t_r} = \frac{1}{\frac{1}{90.9\ MHz} + 7.65\ ns} = 53.6\ MHz$$

This example shows that if the cycle of latency caused by the additional synchronization stage is acceptable, you can dramatically increase the synchronizer's maximum operating frequency.

References

1. Lubkin, S., (Electronic Computer Corp.), "Asynchronous Signals in Digital Computers," *Mathematical Tables and Other Aids to Computation*, Vol. 6, No. 40, Oct 1952, pp. 238 - 241.

2. Nootbaar, Keith, (Applied Microcircuits Corp.), "Design, Testing, and Application of a Metastable-Hardened Flip-Flop," WESCON 87 (San Francisco, CA, Nov. 17 - 19, 1987), Electronic Conventions Management, Los Angeles, CA 90045.

3. Stoll, Peter A., "How to Avoid Synchronization Problems," *VLSI Design*, November/December 1982, pp. 56 - 59.

4. Chapiro, Daniel M., *Globally-Asynchronous Locally-Synchronous Systems*, Department of Computer Science Report No. STAN-CS-84-1026, October 1984.

5. Horstmann, Jens U., Eichel, Hans W., Coates, Robert L., "Metastability Behavior of CMOS ASCI Flip-Flops in Theory and Test," *IEEE Journal of Solid-State Circuits*, Vol 24, No 1, Feb 1989, pp. 146 - 157

6. Wormald, E.G., "A Note on Synchronizer or Interlock Maloperation," *Professional Program Session Record 16*, WESCON 87, November 17 - 19, 1987, Electronic Conventions Management, Los Angeles, CA 90045.

7. Pechouchek, Miroslav, "Anomalous Response Times of Input Synchronizers," *IEEE Trans. Computers*, Vol. C-25, No. 2, Feb 1976, pp. 133 - 139.

8. Chaney, T. J., "Comments on 'A Note on Synchronizer or Interlock Maloperation,'" *IEEE Trans. Computing*, Vol C-28, No 10, Oct. 1979, pp. 802 - 804.

9. Couranz, George R., Wann, Donald F., "Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region," *IEEE Trans. Computers*, Vol C-24, No. 6, June 1975, pp. 604 - 616

10. Veendrick, Harry J.M., "The Behavior of Flip-Flop Used as Synchronizers and Prediction of Their Failure Rate," *IEEE Journal of Solid-State Circuits*, Vol SC-15, No. 2., April 1980, pp. 169 - 176.

11. Kacprzak, Tomasz, Albicki, Alexander, "Analysis of Metastable Operation in RS CMOS Flip-Flops," *IEEE Journal of Solid-State Circuits*, Vol SC-22, No 1, Feb 1987, pp. 57 - 64.

12. Flannagan, Stephen T., "Synchronization Reliability in CMOS Technology," *IEEE Journal of Solid-State Circuits*, Vol. SC-20, No. 4, Aug 1985, pp. 880 - 882.

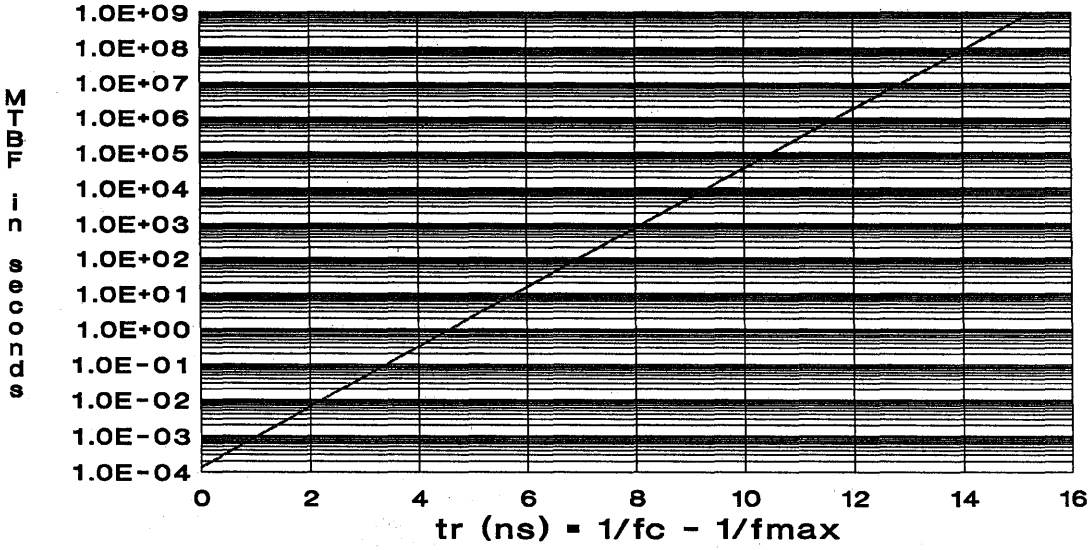
13. Wakerly, John F., *A Designers Guide to Synchronizers and Metastability*, Center for Reliable Computing Technical Report, CSL TN #88-341, February, 1988Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA.

14. Freeman, Gregory G., Liu, Dick L., Wooley, Bruce, and McClusky, Edward J., *Two CMOS Metastability Sensors*, CSL TN# 86-293, June 1986, Computer Systems Laboratory, Electrical Engineering and Computer Science Departments, Stanford University, Stanford, CA.

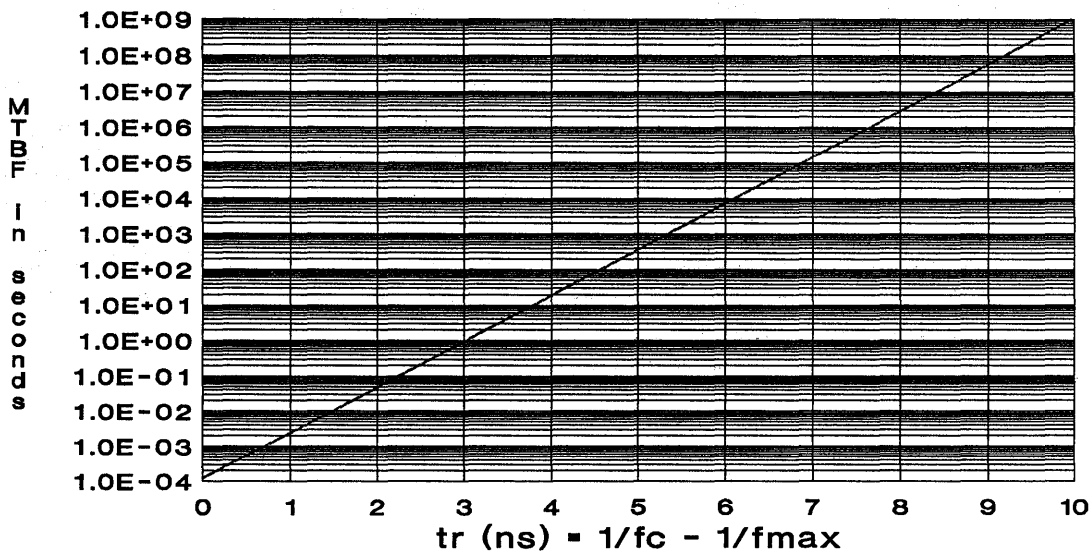
15. Rubin, Kim, "Metastability Testing in PALs," WESCON 87 (San Francisco, CA, Nov. 17 - 19, 1987), Electronic Conventions Management, Los Angeles, CA 90045. 16/1.

Appendix. Metastability Graphs of Cypress Devices

CYPRESS PALC16R8-25

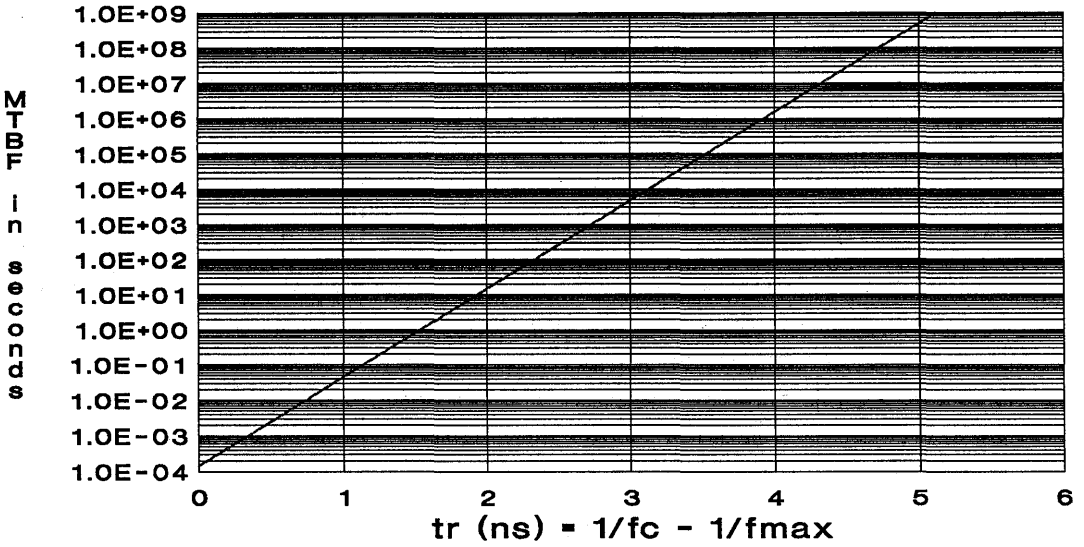


CYPRESS PLDC18G8-12

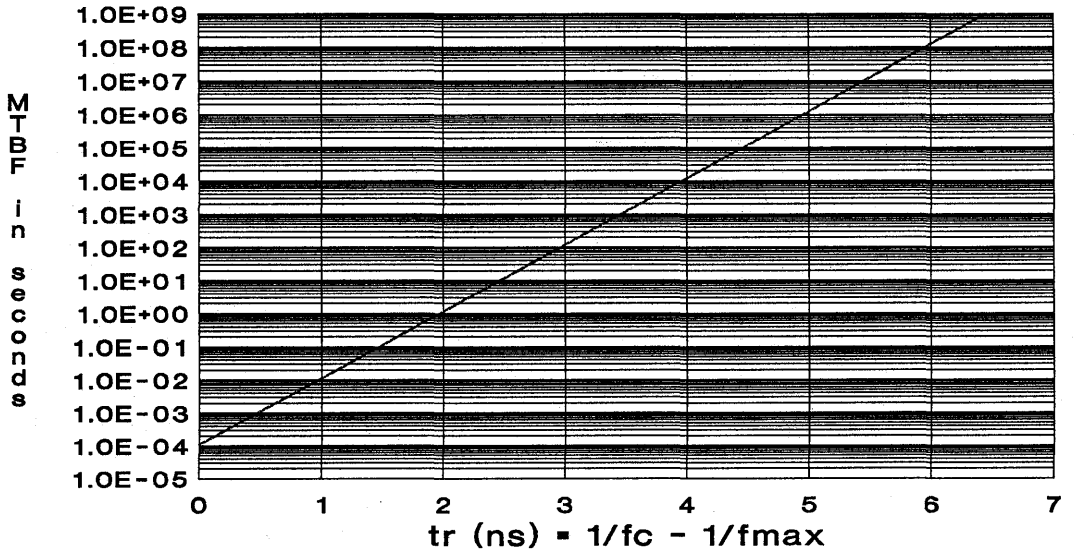


Appendix. Metastability Graphs of Cypress Devices

CYPRESS PALC20G10-20

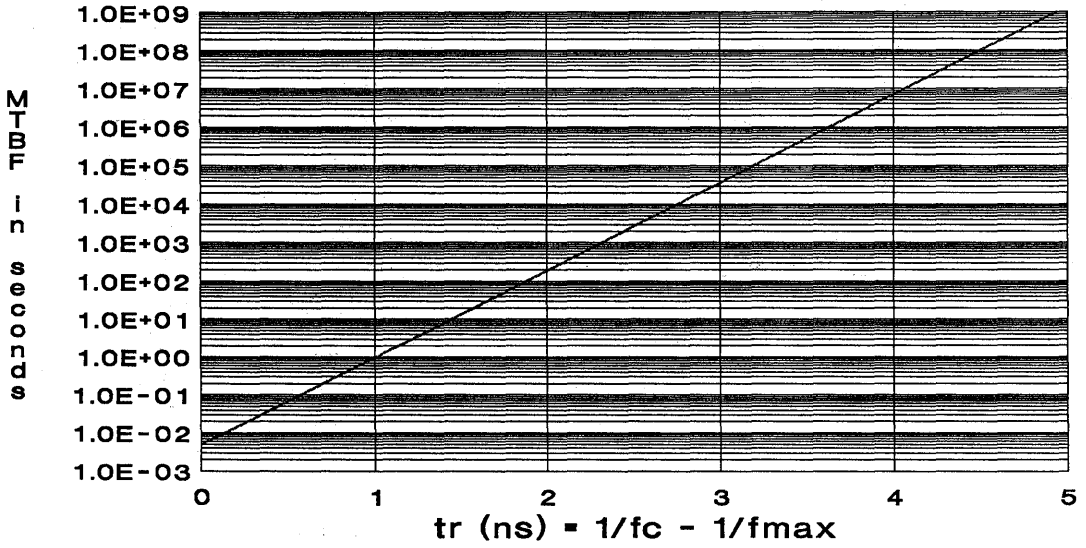


CYPRESS PALC20RA10-15

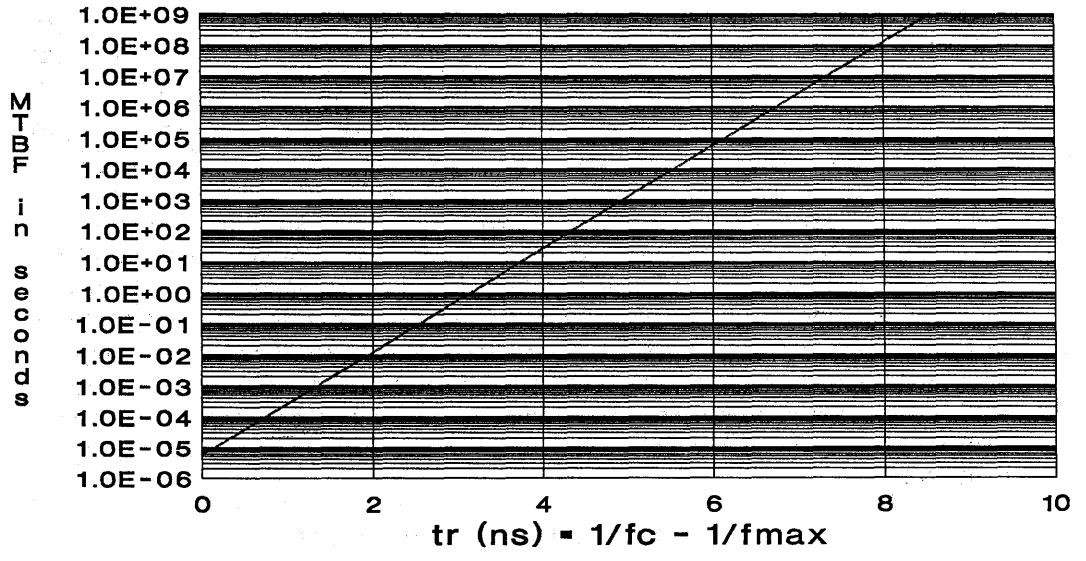


Appendix. Metastability Graphs of Cypress Devices

CYPRESS PALC22V10-20

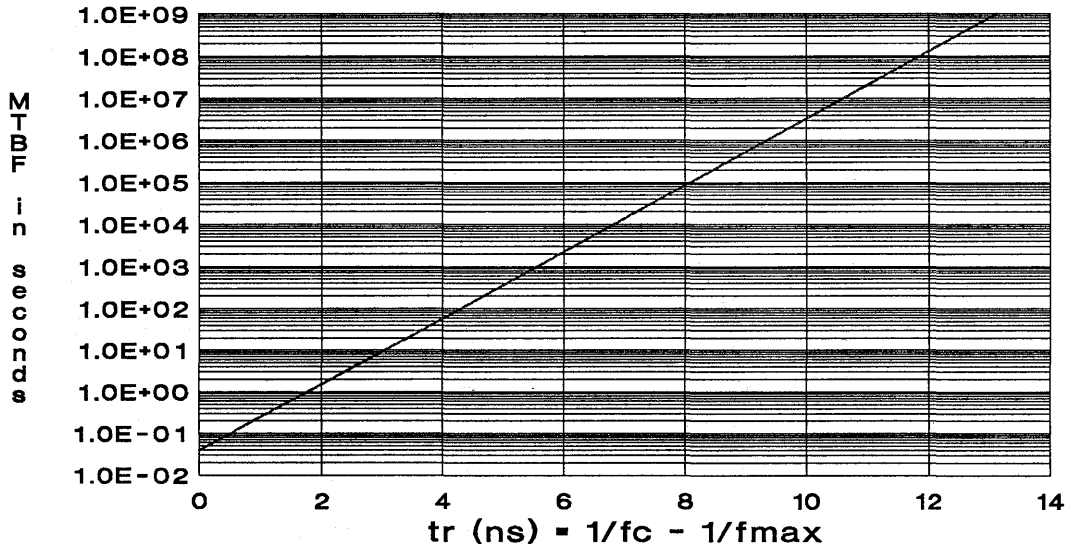


CYPRESS PALC22V10B-15

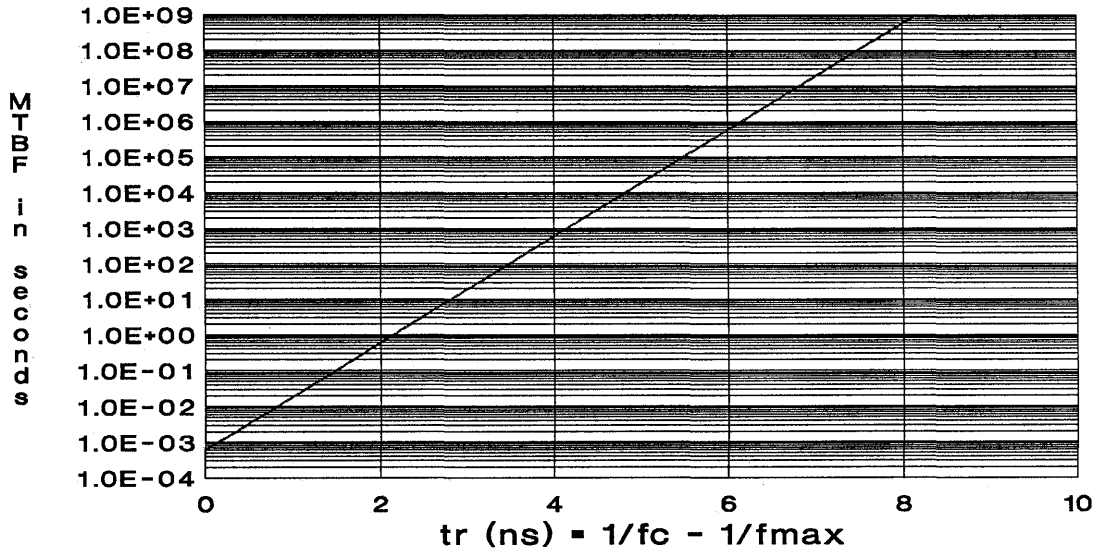


Appendix. Metastability Graphs of Cypress Devices

CYPRESS PALC22V10C-10

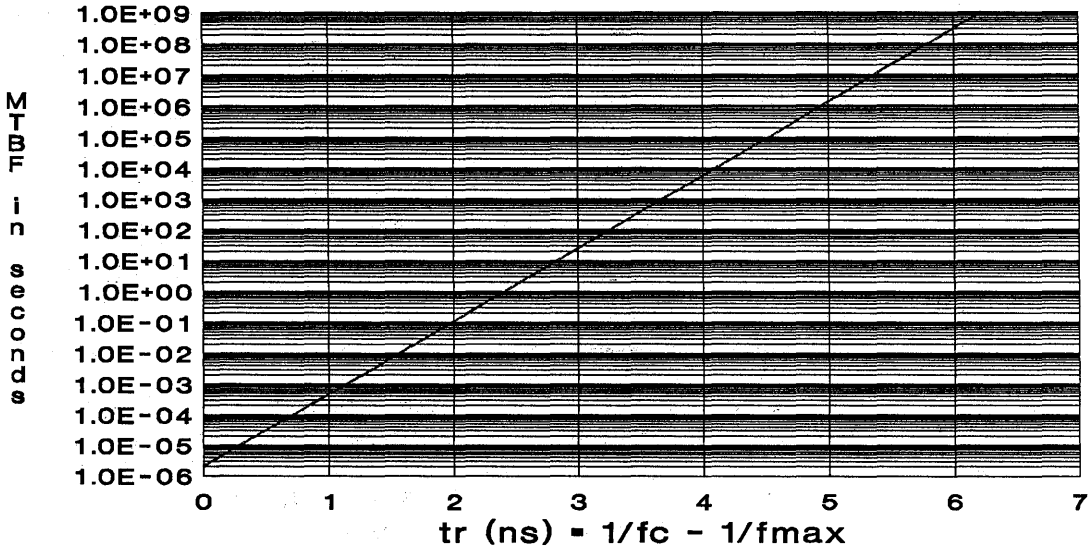


CYPRESS CY7C330-66

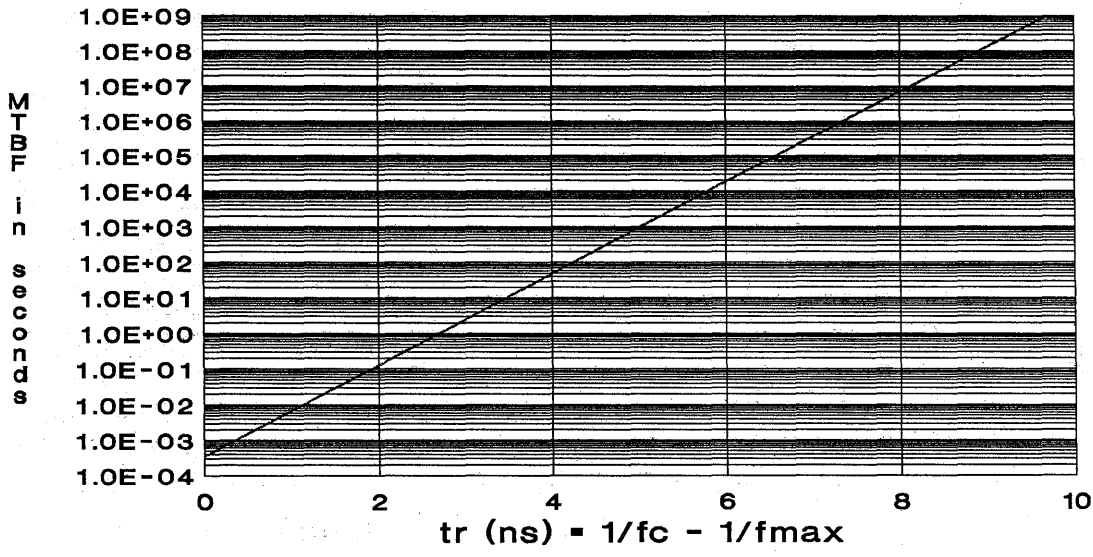


Appendix. Metastability Graphs of Cypress Devices

CYPRESS CY7C331-20

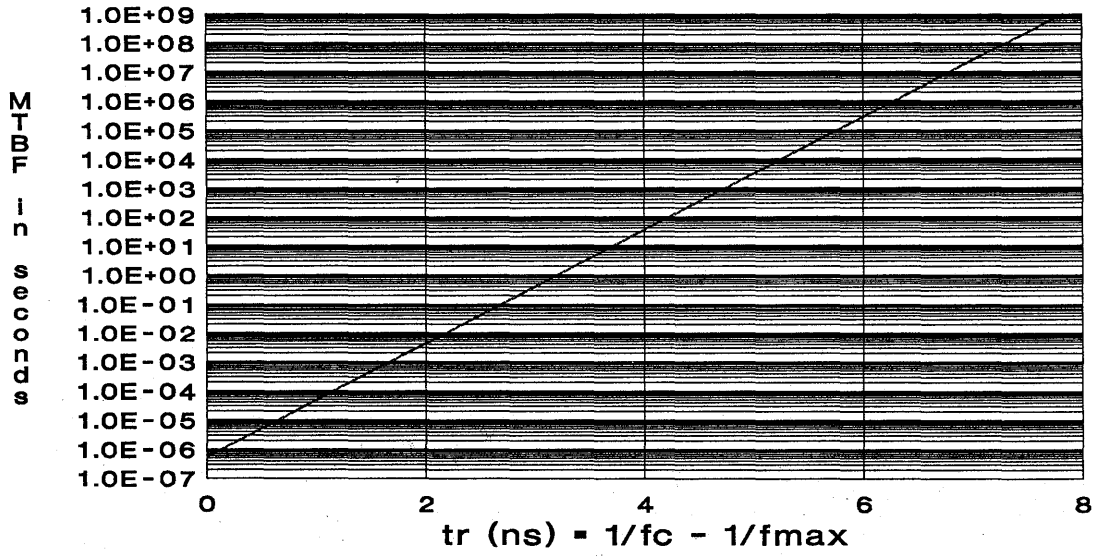


CYPRESS CY7C332-15



Appendix. Metastability Graphs of Cypress Devices

CYPRESS CY7C344-20



CYPRESS SEMICONDUCTOR

PLD-Based Data Path for SCSI-2

This application note begins by describing the major differences between the original SCSI standard and the new SCSI-2 document, with special emphasis on SCSI-2's high-speed signal timing. This information is then put to use in a PLD-based, high-speed data-path design for a SCSI-2 host bus adapter.

Small Computer System Interface

The SCSI-2 standards document is based on the original SCSI-1 standard (ANSI X3.131-1986) developed by the X3T9.2 Accredited Standards Technical Subcommittee. The SCSI-2 specification, generated by this same subcommittee, offers substantial improvements over the existing SCSI-1 standard in documentation, function, performance, interoperability, and command-set standardization.

With the new SCSI-2 ANSI standard, companies that use SCSI for their peripheral I/O now face difficult decisions: Which of the new capabilities offered by SCSI-2 should they support?

The changes in the SCSI-2 document affect both hardware and software. Although it is possible to implement the changes affecting software drivers over time, as these new features appear in peripherals delivered to the marketplace, companies must decide now which hardware features a host bus adapter (HBA) should support. After deliveries to customers, hardware changes made as field upgrades or retrofits always bear high costs and often present a negative picture to the customer.

The physical differences between the original SCSI and the new standard fall into four main categories: SCSI-1 options that are now requirements, new connector/cable options, faster transfer rates, and wider data buses.

SCSI-1 Options

To be considered SCSI-2 compliant, an HBA must support both the parity and arbitration options of SCSI-1. SCSI-2-compliant HBAs should be software configurable by SCSI device address to allow use of older SCSI-1 peripherals that do not have both capabilities.

Connectors/Cables

SCSI-2 documents a 50-mil-pitch connector system. This connector family allows fully shielded assemblies for the 50-wire A cable and optional 68-wire B cable. Many SCSI manufacturers use this micro-D-type connector in volume. You can use the cable/connector scheme in a mix-and-match system with SCSI-1 connector/cable types through the use of adapter cables that have different connector types on each end.

One of the de facto (non-ANSI-standard) SCSI cable schemes, the 25-pin D-sub connector made popular by the Apple Macintosh, does not support SCSI's differential signal implementation. This cable system achieves its low pin count by removing a large number of the ground signals specified for single-ended operation. Because the single-ended transmission scheme is not recommended for SCSI-2's fast synchronous information transfer mode, users of this connector/cable system limit the data rates, cable lengths, and noise margins at which they can operate.

Transfer Rates

SCSI supports two types of information transfer; asynchronous (interlocked) and synchronous (data streaming/offset interlock).

In asynchronous transfers, a four-way handshake occurs between the SCSI peripheral (target) and the HBA (initiator) for each piece of information transferred on the SCSI bus. The SCSI bus's REQ (request) and ACK (acknowledge) control signals are used in this handshake operation, with the SCSI I/O signal determining the direction of information flow. This asynchronous transfer mode is the default mode for all SCSI devices and is required for all MESSAGE, COMMAND, and STATUS transfers. On SCSI systems implemented with very short cables and fast turn-around times in both the target and the initiator, theoretical burst-transfer rates can exceed 10 Mtransfers/s. None of the commercial LSI SCSI controller chips available at this time support this high rate for asynchronous trans-

fers. Most of these controllers handle asynchronous transfers at 50 Ktransfers/s to 3 Mtransfers/s.

SCSI-2 implements the synchronous transfer mode to remove device turn-around time and cable and transceiver delays as factors affecting transfer rates. Unlike asynchronous transfers, which are limited by the interface's four-way path delay, synchronous transfers are limited by interface skew—the difference in transmission delays among signals on the interface.

SCSI-2 allows use of the synchronous method only for data transfers and only after enabling it with a SCSI MESSAGE negotiation between the initiator and target. Synchronous transfers exist in SCSI-1, but few commercial LSI SCSI controllers or peripherals implement this capability. The SCSI-1 implementation defines synchronous transfers for data transfer periods of 200 ns and slower. This specification limits the synchronous data rate to 5 Mtransfers/s.

With tighter-tolerance parts and low-pair-to-pair-skew cables now available, SCSI-2 defines an additional form of synchronous data transfer with a 100-ns minimum period. This change pushes the SCSI-2 maximum data rate to 10 Mtransfers/s. Because of the tighter timing defined for the fast synchronous transfer mode, the SCSI-2 document does not recommend this mode's use with single-ended transceivers, even for short cable lengths.

Wide Data Bus

The last hardware addition allows use of wider SCSI data buses. In SCSI-1 the interface's data-bus portion was only eight bits wide. SCSI-2 allows two additional bus widths of 16 and 32 bits. Because of these different bus widths, SCSI-2 information transfer rates are usually specified in transfers/second rather than bytes/second. You determine the bytes/second rate by multiplying the SCSI data-bus width in bytes by the number of transfers per second on the interface.

The wide SCSI bus is currently defined as a secondary 68-signal B cable that can contain an additional three bytes of bus width. Because this B cable contains only the SCSI control signals necessary for information transfer, you must use it in conjunction with a 50-signal A cable for proper communications.

Use of the wide SCSI option at the maximum 32-bit data-bus width, along with the fast synchronous transfer mode, provides data transfer operations as high as 40 Mbytes/s.

New Problems

SCSI users who require no more performance than they currently have need not make any changes to accommodate SCSI-2. The SCSI-1 standard's capabilities exist as a subset of SCSI-2. However, users experiencing an I/O bottleneck imposed by their current SCSI implementation must implement one or more of the new SCSI-2 features to get additional performance.

The vast majority of the SCSI-2 changes are not really changes at all, just better definitions of items documented in the existing SCSI-1 standard. The arbitration and parity capabilities carry over unchanged from the SCSI-1 standard. The connectors and cables are now well defined, with multiple component sources. The wide bus options require only a replication of existing data-path hardware, but the data-path hardware itself has undergone a significant change.

The new fast synchronous data-transfer mode requires much tighter timing control than was necessary with SCSI-1. If you plan on using the fast synchronous transfer capability, you must contend with differential transceivers, low-skew cables, three data-transfer modes (asynchronous, synchronous, and fast synchronous), and short set-up and hold times.

With all these challenges, it might seem doubtful whether anyone will use the fast synchronous transfer mode. However, a system analysis shows that implementing fast synchronous mode will cost less than any of the wide-bus implementations and still yield a burst data rate as high as 10 Mbytes/s with the standard 50-pin cables. This data rate is twice the maximum offered in SCSI-1 and equal to that offered by the competing Intelligent Peripheral Interface (IPI) in its 2-byte-wide standard implementation. The wide-bus requirement of a second cable also causes problems in weight, cost, and space. Many of the newer 3.5-in. peripherals just do not have room for an additional 68-pin connector.

SCSI Transfer Timing

Of the 23 different interface timing values specified in the SCSI-2 document, 11 apply directly to the different forms of information transfer. These values are:

Cable skew delay	10 ns
Deskew delay	45 ns
Synchronous REQ/ACK assertion period	90 ns
Synchronous data hold time	45 ns
Synchronous REQ/ACK negation period	90 ns
Synchronous/fast synchronous transfer period	Selectable
Fast synchronous REQ/ACK assertion period	30 ns
Fast synchronous cable skew delay	5 ns
Fast synchronous deskew delay	20 ns
Fast synchronous data hold time	10 ns
Fast synchronous REQ/ACK negation period	30 ns

Of these 11 timing values, only the cable skew delay and the deskew delay apply to the asynchronous mode of information transfer. The remaining values apply to the two modes of synchronous data transfers.

These timing values are all specified for the transmitting end of the SCSI interface. Sufficient margins are included in these values to allow proper interface operation under worst-case configurations of transmitters, receivers, and cables. The fast synchronous mode cuts many of the timing parameters by half or more from those of the synchronous mode. Because the interface must still operate over the same distance (up to

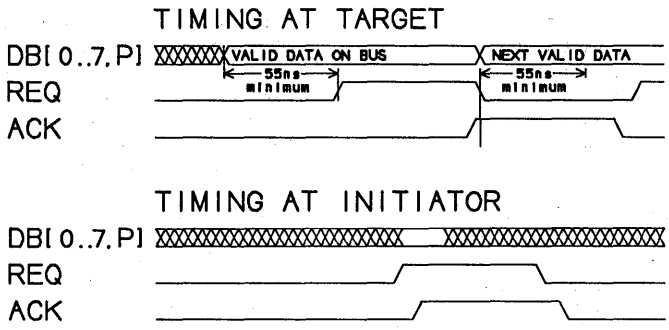


Figure 1. Asynchronous Transfer Timing, Target Transmit

25m), usage of fast synchronous mode demands tighter tolerances for many of the electrical components.

SCSI Transfers

All information transfers on the SCSI bus are controlled by the target device. The initiator cannot send or receive information until it first has received a valid REQ signal from the target device.

Asynchronous Mode Transfers

The interface timing for asynchronous transfers is common to all SCSI devices. Because MESSAGE, COMMAND, and STATUS transfers require support for this mode, all SCSI devices must support it. The interface timing for asynchronous operation varies slightly, depending on whether the SCSI initiator or SCSI target is sending information.

When the target sends information, it must first place the correct data on the SCSI bus, delay a minimum of 55 ns, then assert REQ. The 55-ns delay accounts for all possible data-transmission-time variations caused by transceivers, bias and termination networks, cables, and the information present on them. Because the data has been on the SCSI bus for at least this long prior to REQ's assertion, the initiator knows that the data present at its inputs is supposed to be valid when it receives the asserted REQ signal. Because no set-up time is guaranteed at the initiator, it should not assert its ACK signal to respond to the REQ signal until after delaying long enough to ensure that it (the initiator) can properly capture the data (*Figure1*).

When the initiator sends information, it must first wait until it receives the REQ signal from the target. This is necessary because the bus phase, which determines the information to be sent and the direction of the SCSI bus, does not begin until the REQ signal is asserted for that phase's first transfer. After receiving this first REQ, the initiator can place its data on the SCSI bus, delay a minimum of 55 ns, and respond by asserting ACK. The SCSI target must delay its negation of REQ until it has captured the data.

Because the initiator is not supposed to drive the SCSI bus until a transfer's first REQ occurs, the total delay for this first transfer is longer than the delay for a first transfer from the target to the initiator. To get around this longer delay, many initiators prestage the data for subsequent transfers. The initiator does this by driving the data bus with the next byte of information as soon as the REQ signal from the previous transfer goes Low (*Figure2*).

Synchronous Mode Transfers

The synchronous mode of information transfer is an option for SCSI-1 and SCSI-2 devices. This mode is only usable for data transfers and is not valid for MESSAGE, COMMAND, and STATUS transfers.

SCSI target devices with the ability to use synchronous mode default to asynchronous transfer mode following either a SCSI reset or power-up sequence. To allow synchronous transfers to occur, the target device must first be placed into synchronous mode through a MESSAGE negotiation sequence with an initiator. This sequence sets both the minimum synchronous transfer period and a maximum REQ/ACK offset count.

The synchronous transfer period specifies the minimum period between successive leading edges of any two consecutive REQ pulses or ACK pulses while operating with synchronous transfers. If the negotiated period is less than 200 ns but not less than 100 ns, the data transfer is specified as operating in the fast synchronous mode and must meet the interface timing requirements specified for fast synchronous transfers. If the negotiated period is 200 ns or longer, the data transfer is specified as operating in the synchronous mode and must meet the interface timing requirements specified for synchronous transfers. If the negotiated period is ever set to zero, the data transfer mode reverts to asynchronous.

Unlike asynchronous transfers, where REQ and ACK are directly interlocked to each other to control the transfer's speed, synchronous mode data transfers impose no direct timing relationship between the

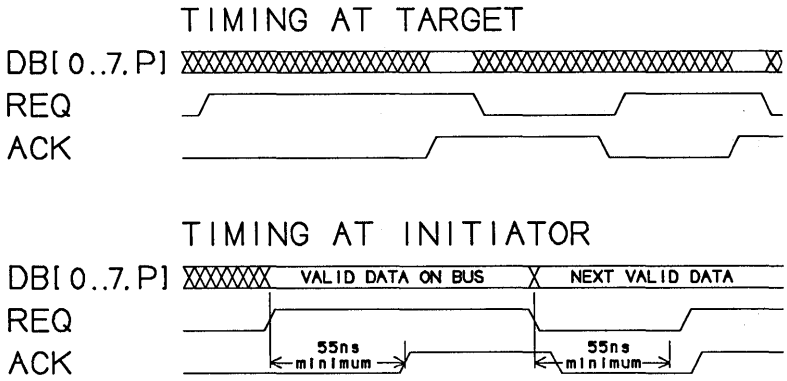


Figure 2. Asynchronous Transfer Timing, Initiator Transmit

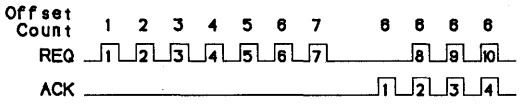


Figure 3. REQ/ACK Offset Count

target's REQ pulses and the initiator's ACK pulses. Instead, the initiator uses a count relationship, known as the REQ/ACK offset count, to slow the transfer. Maintained by both the initiator and the target, this count keeps track of the difference between the number of REQ and ACK pulses. When the count in the target device reaches the negotiated maximum value (*Figure 3*), the target device stops sending REQ pulses until the initiator brings the count below the maximum by returning an ACK pulse. A proper synchronous transfer requires that an equal number of REQ and ACK pulses be sent.

The timing relationships of the REQ and ACK pulses and the data passed with them is specified by the two values used for asynchronous transfers and values specifically identified for synchronous transfers. SCSI synchronous-mode transfers do not require a 50-percent duty cycle for REQ or ACK timing. When operated at or near the maximum transfer rate the required interface timings approach this ratio, but at slower rates the duty cycle is allowed wide variability.

When the target sends information in synchronous mode, the target must place it's data on the SCSI bus a minimum of 55 ns before asserting REQ. The target can then remove or change the data a minimum of 100 ns following REQ's assertion. REQ must remain active for a minimum of 90 ns and, once negated, cannot be reasserted for a minimum of 90 ns. In addition to these requirements, the minimum negotiated period must be maintained. A data transfer is completed when the target has no more data to send and the REQ/ACK offset count has returned to zero. As with the asynchronous transfer mode, the specified delays guarantee valid data at the initiator on REQ's leading edge and not before (*Figure4*).

When the initiator sends information to the target, the initiator must wait until it receives the REQ signal from the target. Once the initiator receives REQ's lead-

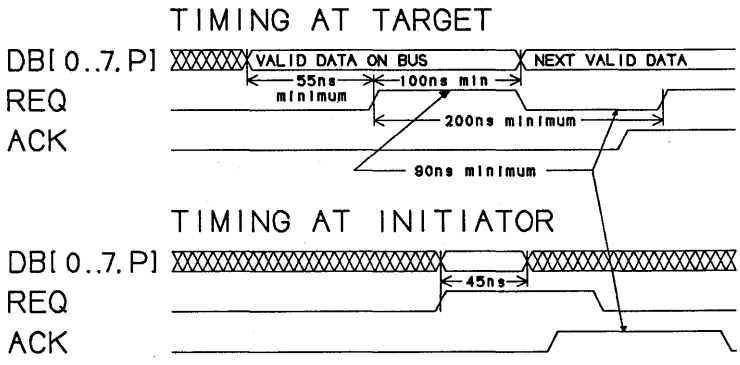


Figure 4. Synchronous Transfer Timing, Target Transmit

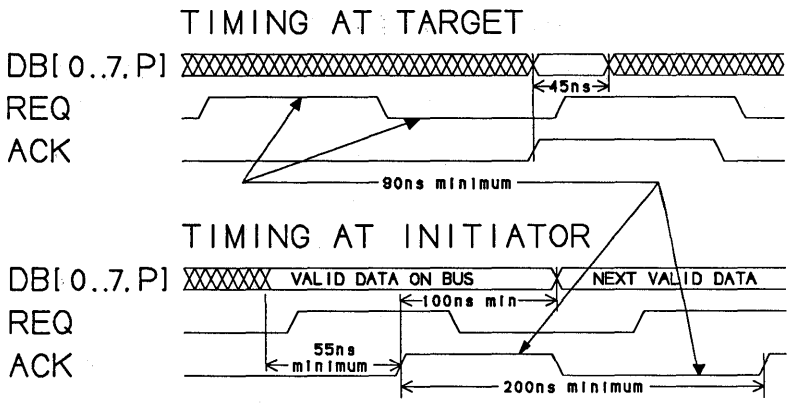


Figure 5. Synchronous Transfer Timing, Initiator Transmit

ing edge, the REQ/ACK Offset count in the initiator is no longer zero. So long as the initiator has data available to send and the REQ/ACK Offset count is non zero, the initiator can continue to send data to the target.

The timing for this transfer (*Figure5*) is like that of the transfer from the target described above. Synchronous mode provides valid data at the SCSI bus's receiving end during a 45-ns interval immediately following REQ/ACK reception.

Fast Synchronous Mode Transfers

Fast synchronous transfers function the same as synchronous transfers but with different timing parameters. These transfers only exist for REQ/ACK pulse periods shorter than 200 ns and longer than or equal to 100 ns.

With fast synchronous transfers, the REQ/ACK minimum assert and negate times decrease to one third their previous size. Thus, SCSI-2 permits REQ and ACK pulses as short as 30 ns when operating in fast synchronous mode. Additionally, the minimum data set up prior to transmitting a REQ/ACK pulse decreases to 25 ns, and the data hold time after REQ/ACK's leading edge is only 35 ns. This timing provides data specified as valid, at the receive end of the SCSI bus, for only 10 ns immediately following REQ/ACK reception. See *Figures 6* and *7* for fast synchronous mode timing diagrams.

SCSI-2 Data Path Design

Synchronous and asynchronous data transfers, 10-ns timing windows, fixed and variable delays, and programmable pulse widths are all necessary functions of a SCSI-2 data path. The simpler techniques used with SCSI-1's 45-ns data-availability windows are quite different from those needed to operate with SCSI-2's 10-ns windows. Fortunately, designing a data path that handles all possible SCSI-2 information transfer modes is not as difficult as it might appear. By carefully selecting some of the newer PLD and interface parts, you can implement the design quite efficiently.

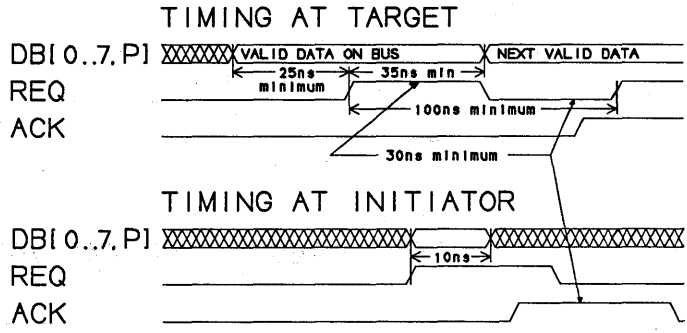


Figure 6. Fast Synchronous Transfer Timing, Target Transmit

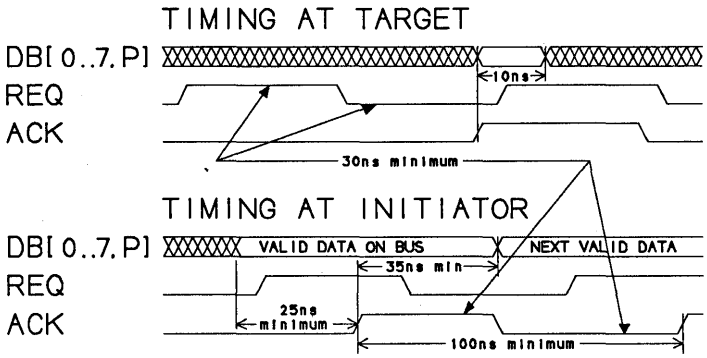


Figure 7. Fast Synchronous Transfer Timing, Initiator Transmit

To successfully meet the needs of fast transfer rates and operability for a wide variety of peripherals, the SCSI-2 design must be capable of:

- Asynchronous data transfers at up to 5 Mtransfers/s
- Synchronous data transfers at a maximum transfer rate of 5 Mtransfers/s, with selectable lower transfer rates for peripherals that cannot operate at the maximum synchronous rate
- Fast synchronous data transfers at a maximum transfer rate of 10 Mtransfers/s, with selectable lower transfer rates between 10 and 5 Mtransfers/s for peripherals that cannot operate at the maximum fast synchronous rate, yet can operate faster than the maximum synchronous rate
- Operation with differential transceivers

Design Partitioning

Correct partitioning is probably the most critical part of achieving an efficient implementation of any SCSI design. When partitioning the design, list the necessary functions and, where possible, combine multiple functions into a single, more global function. A SCSI-2 data path must include these functions:

- SCSI interface transceivers
- Receive data register
- Transmit data register
- Receive data buffer
- Transmit data buffer
- REQ/ACK offset counter
- Asynchronous receive control
- Asynchronous transmit control
- Synchronous receive control
- Synchronous transmit control
- Fast synchronous receive control
- Fast synchronous transmit control

Although the transmit and receive control functions must operate with different timing values, the asynchronous, synchronous, and fast synchronous control operations for receive or transmit must perform the same function: receiving or transmitting information. Grouping the receive and transmit control functions into two separate and more generalized functional units reduces the design's complexity.

The necessary operations of the receive control function are:

- Clocking information into the receive data register
- Returning and removing the ACK signal at the proper time
- Writing the received data into the data buffer

The necessary operations of the transmit control function are:

- Reading the data from the data buffer and clocking the data into the transmit data register
- Returning and removing the ACK signal at the proper time
- Timing the necessary data set-up time
- Timing the necessary data hold time
- Timing the necessary ACK assertion time
- Timing the necessary ACK negation time

The data buffer function is another area where some consolidation can occur. Because the SCSI interface cannot send and receive data at the same time, a single common buffer is used for both transmit and receive functions.

With these functions combined, the design now comprises seven functions:

- SCSI interface transceivers
- Receive data register
- Transmit data register
- Data buffer
- REQ/ACK offset counter
- Receive control
- Transmit control

SCSI Interface Transceivers

The SCSI interface supports both single-ended and differential transceiver types. The single-ended variety is

most common today because it is relatively inexpensive and most commercial LSI SCSI controller chips incorporate this type. Single-ended transceivers suit cable lengths less than 6m long and synchronous data rates of 5 Mtransfers/s or less.

SCSI devices using fast synchronous mode require differential transceivers. This transceiver type meets the electrical specifications of the EIA RS-485 standard. Operating from a single +5V supply, these transceivers can handle large swings in common mode noise, are guaranteed glitch free during power-up and -down operations, and have short-circuit and thermal-shutdown protection. SCSI applications that use cables longer than 6m also require differential transceivers. Although currently limited in the SCSI standard to operation at no more than 25m, this transceiver type can drive signals much farther, as shown by the Intelligent Peripheral Interface usage of the same parts at 65m.

Differential transceivers have one other advantage that is often overlooked. Because two differential signals determine the output state of each receiver, it is possible to achieve either active High or active Low TTL inputs and outputs by reversing the connection of the + and - differential signal lines on the SCSI bus. This programmable inversion can often eliminate the need for an inverter, and its associated delay, from many of the differential signals paths.

All existing SCSI applications that use differential transceivers place these parts external to the LSI SCSI controller chips. This practice is due primarily to the transceivers' power dissipation and partially analog operation. Until recently you could only get differential transceivers in singles—one transmitter and receiver in an 8-pin part. This packaging required 18 parts to implement the transceivers for a SCSI-1 bus.

Due to the growing usage of these parts and improvements in power control technology, manufacturers now offer triple and quad transceiver parts. Some of these parts are designed specifically for the SCSI environment. To allow for the selection and arbitration sequences, for example, the transceivers have separate transmitter enables that allow individual transmitters to be turned on within the part. These transceivers meet all signal and skew requirements of the SCSI-2 fast synchronous mode.

Receive Data Register

The information from the transceivers is used for arbitration, selection, and reselection sequences, as well as information transfers. Of the transfer sequences, the fast synchronous transfer mode has the most stringent timing concerns.

Because of the fast synchronous mode's 10-ns data-availability window, the receive data register must have a very short set-up and hold time. The 74F823, a 9-bit D-type register, fits this application nicely. With a maximum set-up-and-hold-time total of 5.5 ns, the register leaves room for a 4.5-ns skew in clock timing for proper operation. Because of this timing, the clock path to the receive data register can afford only a single gate delay. To meet the defined 10-ns data window and work with the 74F823, the single gate must have a minimum propagation delay of 3 ns and a maximum delay of 7.5 ns for the Low to High output transition. Depending on the gating function needed, any parts such as the 74F08, 74F11, or 74F32 meet the timing window.

Transmit Data Register

The same part type, 74F823, also works on the transmit side of the interface. Because both the transmit and receive data registers are as wide as the full SCSI data bus, they implement a nearly seamless design.

Data Buffer

You can implement the data buffer for a SCSI interface in many ways. Host bus adapters that support data-caching functions might require a large piece of memory. Because the data cache usually exists several logic levels away from the physical SCSI interface, the HBA needs a smaller piece of memory to act as a "rubber band" between the SCSI target and the host or HBA memory. Using such a front-end buffer allows data to move quickly on the SCSI physical interface.

Because the SCSI interface is asynchronous to most of the logic activity in any HBA, the cleanest form of this front-end data buffer has an asynchronous interface, which permits the buffer to accept data as the data becomes available. Memories of this type fall into two categories: dual-port RAMs and FIFOs. The latter is an excellent fit because the information transferred over the SCSI interface is order dependent and does not contain memory-address information. The FIFO eliminates any need for address-sequencing logic for moving information in and out of the data buffer.

The data buffer must also be bidirectional to allow the HBA to send and receive information. You can create a bidirectional FIFO using unidirectional FIFO memories with external bus-steering and control logic. Unfortunately, a bidirectional FIFO built in this manner requires many extra parts, power, and board space. A much better choice is to use a monolithic bidirectional FIFO.

Although most available bidirectional FIFOs are register programmable and require a processor connection to control their operation, the Cypress CY7C439 bidirectional FIFO does not. This 2K x 9-bit FIFO supports the full 9-bit SCSI data bus, in addition to the pin programmability necessary for simple state machine control.

REQ/ACK Offset Counter

The HBA uses the REQ/ACK offset counter (*Figure 8*) for synchronous and fast synchronous transfers. The counter keeps track of how many unanswered REQ pulses the HBA has received and must respond to. Both transmit and receive operations employ this logic.

Just how big a counter is needed? Although it would be easy to pick an arbitrary number, you can calculate the size of the counter needed to keep the SCSI interface operating at its peak rate. This task requires a counter of N bits, where R outstanding REQ and ACK pulses can be active, such that R=2N-1. This same R value applies to the target device as the maximum REQ/ACK offset count.

The value of R depends on the SCSI cable's length, the velocity of the cable signals' propagation (Vp), the fastest synchronous period to be used, the turnaround time of a REQ pulse to an ACK pulse in the initiator, and the recognition time for an ACK pulse in the target. Many of these values are specified or can be calculated from the information in the SCSI-2 document. You can approximate the remaining values to arrive at a number accurate to within a power of 2 (1 counter bit).

The cables specified for the SCSI interface use a solid dielectric whose Vp ranges from 60 to 66 percent. Additionally, the use of twisted-pair cables is strongly recommended to reduce crosstalk. When wires are twisted together to form a cable, longer wires are needed to reach a specific physical cable length. Depending on the amount of twist in the pairs, the longer wires can lengthen the physical signal from 2 to 30 percent. The cables specified for fast synchronous transmission have a very tight pair-to-pair signal skew specification that is partially achieved by having a very

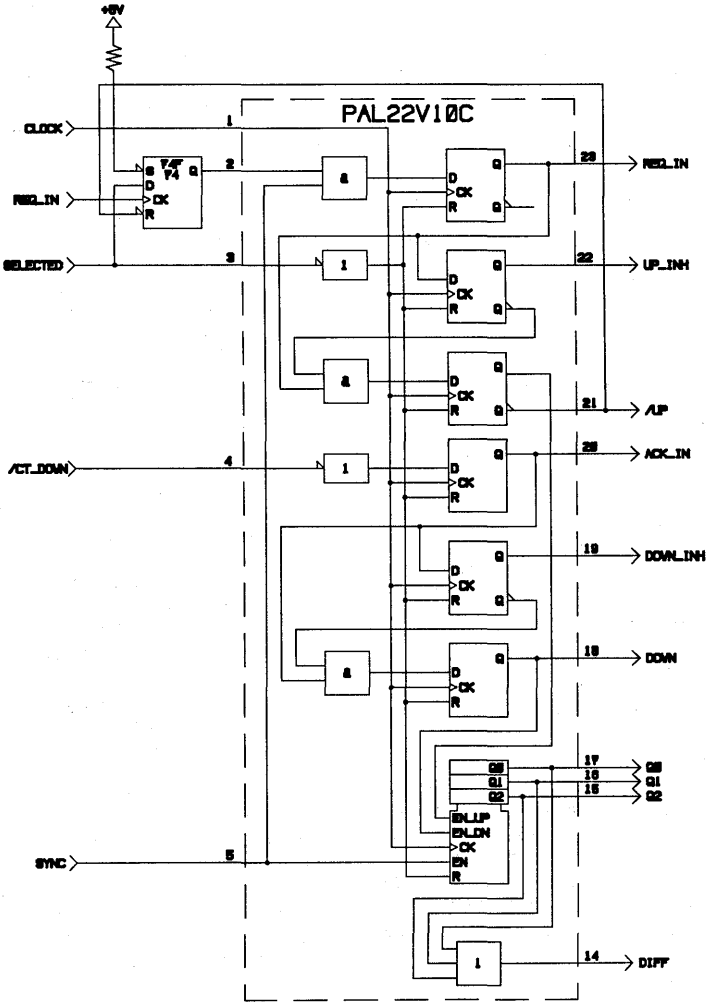


Figure 8. REQ/ACK Offset Counter

loose twist in the signal pairs. In these cables, each line's internal physical signal length is approximately 2 to 10 percent longer than the external physical length.

With a maximum external cable length of 25m, the calculated one-way maximum signal delay through the cable is

$$t = (25m + 2.5m) * 5.56 \text{ ns/m}$$
$$t = 153 \text{ ns}$$

Because the SCSI target does not know that an ACK has occurred until the ACK propagates to the target's end of the cable, this one-way delay must be doubled to allow for the return path time.

In addition to cable delay, the transceivers themselves contribute a major portion of the total loop delay. The data sheet for a DS36954 quad differential transceiver lists a maximum delay value of about 20 ns for each transmitter and receiver that the REQ and ACK signals pass through. This delay adds 80 ns to the loop delay.

The next delays to consider are the turnaround and recognition times in the initiator and target. These delays must be approximated by examining the operations that must occur. Because both the REQ and ACK signals are asynchronous when they are received, they must go through a metastable-prevent circuit before they can be used. The faster forms of TTL-compatible logic can execute a metastable prevention procedure in less than 20 ns and still provide a reasonable MTBF. Following this procedure, a counter must operate on the signal and generate a status value, which determines whether the transfer can proceed or must suspend. For worst-case operations, a miss must be assumed for the first stage of the metastable-prevent circuit. This assumption yields a maximum REQ/ACK offset counter delay of 80 ns.

The REQ/ACK send delay is the last piece of the delay loop. The REQ/ACK send delay assumes the necessary data set-up time before generation of the REQ or ACK pulse to send the data. For the fastest transmission mode, this delay could be as long as 70 ns.

Adding these values yields a loop delay of

306 ns	Cable delay
80 ns	Transceiver delay
80 ns	Initiator REQ/ACK offset counter delay
80 ns	Target REQ/ACK offset counter delay
70 ns	Data set-up delay
616 ns	Total loop delay

Considering this figure and the 100-ns minimum period for fast synchronous transmission, achieving continuous data flow demands that there be at most six outstanding REQ pulses at the target. This task requires a minimum of a 3-bit REQ/ACK offset counter to maintain data streaming for fast synchronous transfers.

This counter must operate under the following rules:

1. Each received REQ pulse generates a single count up.

2. Each generated ACK pulse generates a single count down.

3. The counter does not change if REQ and ACK are recognized simultaneously.

Although the simplest approach would be to run the REQ signal from the receiver straight into the metastable-prevent circuit, this could cause problems in some systems. Because the REQ signal is allowed to be as narrow as 30 ns at the cable's transmitting end, this pulse might shrink under some conditions such that the received pulse is less than the 20-ns sample period (plus set-up and hold time). This situation could occur under worst-case conditions of intersymbol interference, cable imbalance, and bias distortion, causing the the REQ/ACK offset counter to miss the REQ pulse and create a transmission error.

To make sure the counter does not miss the REQ pulse, you need to add a D flip-flop, configured as an edge detector, just before the metastable-prevent circuit. This flip-flop forces the received REQ signal to remain at the counter input until it is recognized.

Although you can build the REQ/ACK counter with a small handful of MSI/SSI parts, a superior approach is to use a single Cypress PAL22V10C PLD. This one part can include the entire 3-bit up/down counter, two single-count-per-pulse filters, and both REQ and ACK metastable-prevent structures. Because of the PAL22V10C's synchronous operation, the asynchronous edge-detector function still requires a single 74F74 flip-flop external to the PAL22V10C REQ/ACK offset counter. The equation list for this PLD appears in *Appendix A*.

Receive Control

Data reception from the SCSI bus is handled the same for all modes of information transfer. This is possible because the information on the SCSI bus is always valid at REQ's leading edge for asynchronous, synchronous, and fast synchronous transfer modes. Every received REQ pulse can thus clock the receive data register. Even when the initiator sends data to the target, and therefore clocks invalid data into the receive data register, the next REQ pulse overwrites the invalid data.

It is necessary to delay the received REQ signal's leading edge by a gate delay that matches the 74F823 Received Data Register's set-up and hold times. The 74F08 fits nicely here with a 3-ns minimum delay on Low-to-High transitions and a 6.6-ns maximum delay. This delay still gives a 900-ps margin for fast synchronous transfers, judging from worst-case commercial specifications.

Because timing is so tight when doing fast synchronous transfers, take care to avoid destroying any designed-in margins with poor circuit layout. The standard FR4 substrates used for most circuit boards exhibit a dielectric constant of about 5. With this high number, circuit trace delay exceeds 2 ns/foot. To prevent infor-

mation transfer errors, make sure the REQ signal's routing length to the receive data register is never more than 5 in. shorter or longer than, any of the data-path signals.

Once information has been captured in the receive data register, it must be written into the data buffer. The I/O signal in this state indicates that the SCSI bus direction is set for input to the initiator.

With these conditions met and REQ present, a FIFO write operation must occur. For a correct write to occur, the CY7C439 FIFO requires a pulse on the /STBB pin with a minimum width of 30 ns. With SCSI-1 peripherals, you could build a small asynchronous state machine to generate a write FIFO pulse of this minimum width; the state machine could utilize the false state of the REQ signal that occurs after each REQ pulse. If you use this method, you need some external logic to terminate the last write to the memory.

To support SCSI-2 peripherals that use fast synchronous transfers, you need a different method. Because the REQ pulse's transmitted false state for fast synchronous transfers can be as small as 30 ns, a pulse of this same width cannot be guaranteed at the receive end.

You can choose among many methods for generating fixed-width pulses: delay lines, TTL delay elements (74LS31), strings of gates, counter chains, one shots, and standard TTL parts feeding R-C circuits. Each circuit type has its inherent problems. One shots are notorious for not triggering at all or mistriggering, lumped-constant delay lines have high field failure rates, and TTL delay elements have a too-wide margin of variability for a manufacturable design. In this case, however, a new type of reprogrammable CMOS synchronous state machine PLD, the Cypress CY7C361, can easily generate the required pulse.

The CY7C361 is a programmable state machine that allows multiple concurrent and interacting state machines to operate in the same part. Based on a Petri Net or token-passing philosophy, the CY7C361 can contain as many state machines as its state registers, inputs, and outputs support. This part contains 32 separate state registers that can operate at internal frequencies as high as 125 MHz. The CY7C361 also contains an internal clock doubler, which makes it unnecessary to generate and distribute frequencies upwards of 100 MHz in a TTL environment. Because this part is designed for interface operations, it also contains metastable-hardened input structures.

By operating from the same 50-MHz clock used with the REQ/ACK offset counter (doubled internally to 100 MHz), a CY7C361-based 4-state machine can generate a 40-ns pulse to write the information into the FIFO memory.

The state machine must account for the procedure used to govern writes to the FIFO. Although FIFO writes can occur even if the FIFO is half full, as determined by the FIFO status flags, the ACK signal that allows the interface to continue operation is held up until the host reads enough information from the FIFO to bring the FIFO state below half full. This governing procedure is used for asynchronous and synchronous operations. For synchronous operations, data continues to be written into the FIFO even after reaching the half-full state. Although ACK pulses are no longer returned to the target when the FIFO is at or above half full, the FIFO writes are only suspended when the REQ/ACK Offset counter in the target reaches its maximum and stops sending REQ pulses.

Figure 9 shows the simple state diagram for writing information into the FIFO. The diagram includes four active states (1 - 4) and a reset state (0). When in the reset state, the CY7C361 continuously watches for a REQ signal to occur while the SCSI bus's I/O signal is asserted (SCSI bus direction = IN). When this condition occurs, the state machine advances to state 1 and continues through states 2, 3, 4 and back to reset. The CY7C361 implements this state machine using three of the 32 available state registers, labeled here as W0, W1, and W2. State registers W1 and W2 also serve as FIFO strobe-delay states for FIFO read operations.

Figure 10 shows, through three FIFO write cycles, how the CY7C361's state registers change to achieve a fixed 40-ns delay. The outputs of the three state registers are logically ORed together in the CY7C361. Unlike many other register-based state machines, the CY7C361's internal design allows you to OR together adjacent but nonoverlapping state-register outputs to generate a glitch-free output signal.

Next to each state register label in *Figure 10* is either an s, t, or w. These letters represent which of the three possible CY7C361 state register configurations is used for that specific state register. An s (start) specifies that the state register becomes active for exactly one clock cycle each time the required input conditions are met. A t (toggle) specifies that the state register changes state on each clock cycle while the required input conditions are met. The t-type state registers allow very efficient construction of counters. The last type of state register, w (wait for terminate), is set only by a carry in signal generated in the immediately preceding state register; the w-type state register is cleared when its required input conditions are met.

Transmit Control

Transmitting information to the SCSI target is by far the most complex function. The procedure requires controlled interval timing for reading data from the FIFO data buffer, placing the data in the transmit data register and on the SCSI bus, and generating multiple-width ACK pulses.

Because of these operations' controlled timing and concurrency, the CY7C361 is again called into service. The earlier application of this part used three of the 32 available state registers. The transmit function uses many of the part's remaining states to generate the

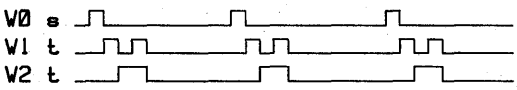


Figure 9. FIFO Write State Register Timing

necessary delays for asynchronous, synchronous, and fast synchronous transfers.

For the SCSI transmit cycles to occur at the maximum rate, the HBA must stage or pipeline data so that the data is immediately available for transmitting. This operation requires that the HBA handle concurrent asynchronous events. As one transfer is occurring on the SCSI bus, the next piece of information must be read out of the FIFO and be available for the next bus transfer. These FIFO read functions operate in two very similar sequences: one for asynchronous SCSI writes and one for synchronous SCSI writes.

Figure 11 shows the state diagram for FIFO read operations. This state diagram has a similar reset state (0) and the same delay states (2, 3, and 4) as the FIFO write state machine. The two entry states are for asynchronous (1) and synchronous (7) SCSI write operations. For asynchronous SCSI writes, the FIFO read starts when synchronous operations are not enabled, data is available in the FIFO, the bus direction is set to out, and a FIFO read is not currently active. For synchronous SCSI writes, the FIFO read starts when the REQ/ACK Offset counter is non-zero, synchronous operations are enabled, data is available in the FIFO, the bus direction is set to out, and a FIFO read is not currently active.

The FIFO read operation uses five more state registers in the CY7C361. The state-register timing diagram in *Figure12* shows these new states:

- R0 starts the FIFO read for asynchronous SCSI writes
- RS0 starts the FIFO read for synchronous SCSI writes

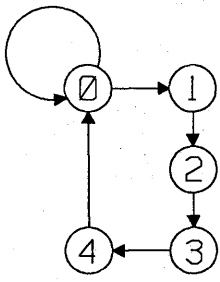


Figure 10. FIFO Load State Diagram

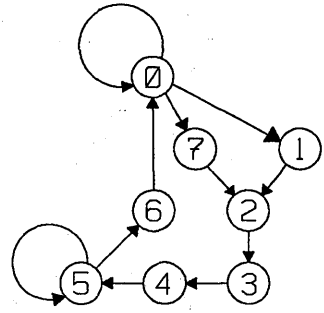


Figure 11. FIFO Read State Diagram

- R1 serves as the FIFO strobe signal (ORed with state registers W0, W1, and W2) and notes internally that a FIFO read is currently active
- ES ends the FIFO read when the minimum delay has passed (delay states 2, 3, and 4) and the transmit data register contains no valid data
- DATA specifies that the transmit data register contains valid data

Figure 12 shows two sequences: R0 starts an asynchronous FIFO read, and RS0 starts a synchronous FIFO read. In normal operation, consecutive FIFO read cycles are of the same type and overlap with data being available in the transmit data register. Because the FIFO output does not change (following the minimum output delay time) until the FIFO strobe is removed, this strobe's trailing edge is used to directly clock the data from the FIFO into the transmit data register.

With the FIFO data now in the transmit data register and driven out onto the SCSI bus, the HBA must generate specific and precise delays to allow the ACK signal to be sent at the proper time. From the time that the data clocks into the transmit data register, a minimum of 60 ns must be timed for asynchronous SCSI writes, 40 ns for fast synchronous SCSI writes, and 90 ns for synchronous SCSI writes.

To create these delays and permit programmable synchronous data rates slower than the maximum allowed, part of the CY7C361 is used to create a loadable delay counter. This counter operates as a hardware subroutine within the CY7C361, providing all the necessary delays for ACK timing.

For asynchronous SCSI writes, the state machine calls the delay routine as soon as information is placed in the transmit data register. When the timer times out (returns to zero), the ACK signal is sent. For synchronous SCSI writes, the state machine calls the delay routine both to set and remove the ACK signal.

The CY7C361 implements the delay hardware as a 4-bit count-up toggle counter, which provides 15 different synchronous timing periods ranging from 100 to 380 ns. *Table 1* lists the values that load into the counter

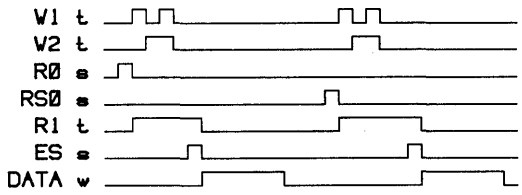


Figure 12. FIFO Read State Register Timing

to provide these periods. The load value for the counter enters the CY7C361 via four input pins. When the delay subroutine is called, the signal levels on these four pins load into four state registers, which in turn load into the counter.

Figure13 shows the state transition diagram for the delay counter. From the reset state (0) the delay counter enters a load state (L). Because the delay counter has 15 possible start points, the load state must have 15 possible exits. When the counter has reached its maximum value (1111), the counter enters an exit state (X) to toggle the ACK signal on or off.

This loadable delay counter uses nine more state registers in the CY7C361. Four of these state registers (CE0, CE1, CE2, and CE3) serve as counter enable bits that load the four toggle state registers (CT0, CT1, CT2, and CT3). The ninth state register is used for the exit state (CTX).

Two count sequences appear in *Figure14*. The first sequence shows the shortest timing interval, created by loading 1111 into the counter. The second sequence shows a longer delay, which results from loading 0101.

Because the delay counter has overhead states, the shortest interval the counter can time is 30 ns. To get the widest range of synchronous transfer periods from the delay counter, a fill state is generated at the start of each ACK cycle to stretch this minimum interval to 40 ns. The 40-ns interval determines the shortest possible ACK Low interval when operating in fast synchronous transfer mode with a 100-ns period. To generate the ACK delay for asynchronous mode, the SCSI specification for writes requires two more delay states to get 60 ns. This added delay is achieved by setting the delay counter inputs to 1101.

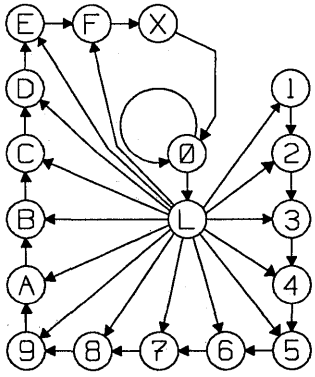


Figure 13. Loadable Delay Counter State Diagram

Figure15 shows the state diagram for asynchronous SCSI write operations. The first active state (1) is the fill state, which the state machine enters as soon as the FIFO read completes and valid data is in the transmit data register. The delay subroutine call appears as a single state (2) that loops until the delay is complete. Once the delay counter times out, the state machine advances to state 3, where ACK is transmitted. The state machine remains in this state until the REQ signal is removed. This clears the ACK signal and returns the state machine to the reset state (0).

The state register timing for this sequence appears in *Figure 16*. This timing diagram shows not only the state registers used for generating the ACK signal, but all the state registers used in the CY7C361. You can therefore see the interaction of the FIFO read, delay counter, and asynchronous ACK control state machines.

Figure 16 shows three transfers. The ES state, which ends the FIFO read operation, starts the ACK delay state machine. As soon as this state machine is started, the next FIFO read is also started. The ACK cycle is terminated by the ATA state register, which monitors the REQ and ACK signals. When the ACK cycle completes, the next FIFO data is clocked into the transmit data register, and another ACK cycle is started.

The state diagram for synchronous transfers appears in *Figure17*. This sequence starts the same as an asynchronous transfer, except that the termination of the first ACK delay starts a second delay to remove the ACK signal. When this second delay times out, the ACK state ends. Meanwhile, the ongoing FIFO read operation has put data into the FIFO. The end of the ACK state prompts the FIFO read to complete and start the next ACK cycle. Two fill states, 4 and 5, are

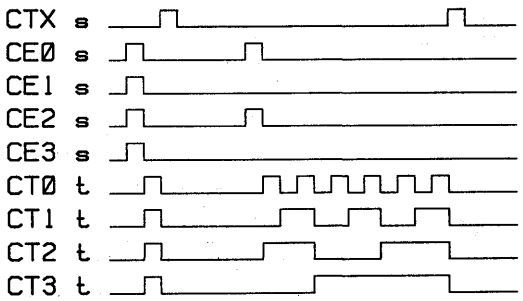


Figure 14. Delay Counter State Register Timing

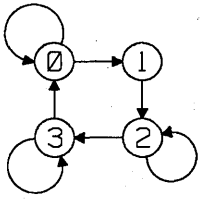


Figure 15. Asynchronous Write State Diagram

ORed with the ACK state to meet the timing requirements of synchronous transfers.

By carefully selecting the data-enable, set-up, hold, and ACK duty cycle, you can use the same state machine for synchronous and fast synchronous transfers. *Figure 18* shows the state register timing for three transfers in fast synchronous mode, with a 100-ns data period. Compare these transfers with *Figure 19*, which shows the state register timing for two synchronous transfers, with a 200-ns data period. The only difference between the two types of transfers is the amount of time spent in the delay counter. Additionally, the FIFO read portion of the waveforms shows that the synchronous FIFO read state register, RS0, starts the FIFO read instead of the R0 state register used with asynchronous SCSI writes.

As configured thus far, the CY7C361-based state machine generates the FIFO strobe signal for FIFO read and write operations and the ACK signal for asynchronous, synchronous, and fast synchronous SCSI writes. As for SCSI read operations, the HBA generates the ACK signal for asynchronous reads by returning the REQ signal as ACK. For synchronous reads, however, the HBA must use a different mechanism.

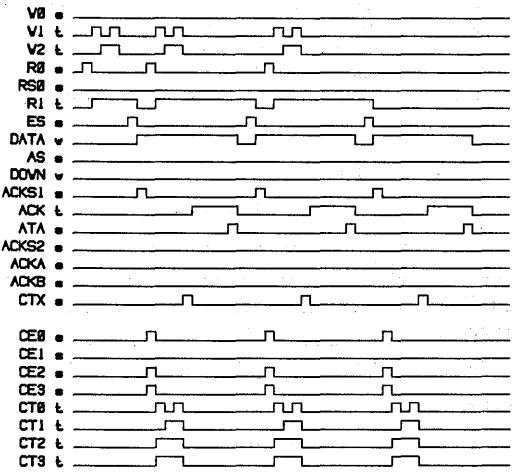


Figure 16. Asynchronous Write State Register Timing

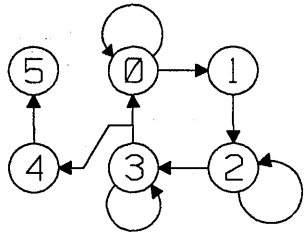


Figure 17. Synchronous Write State Diagram

The ACK sequence needed for synchronous SCSI reads has the same timing as the ACK generated for SCSI writes, except that the initiator places no data on the SCSI bus. Because the CY7C361 outputs do not control the enables of the SCSI transceivers, or the receive and transmit data registers, the same ACK control state sequence used for synchronous SCSI writes can also serve for synchronous SCSI reads.

The return of an ACK on a SCSI read is based on the FIFO having room rather than the HBA having data available. Thus, a new state register must be added to start the ACK cycle. Additionally, a signal is needed to decrement the REQ/ACK Offset counter. Although you might expect to use the output ACK signal for this purpose, it does not occur early enough in the cycle to count down the REQ/ACK Offset counter before the next ACK cycle is ready to start.

Figure 20 shows the state register timing for a fast synchronous SCSI read operation. The DOWN state

Table 1. Synchronous Data Rates

Load Value	Synchronous Period	Data Rate	Data Transfer Mode
1111	100ns	10.0Mt/s	Fast Synchronous
1110	120ns	8.33Mt/s	Fast Synchronous
1101	140ns	7.14Mt/s	Fast Synchronous
1100	160ns	6.25Mt/s	Fast Synchronous
1011	180ns	5.55Mt/s	Fast Synchronous
1010	200ns	5.00Mt/s	Synchronous
1001	220ns	4.54Mt/s	Synchronous
1000	240ns	4.16Mt/s	Synchronous
0111	260ns	3.84Mt/s	Synchronous
0110	280ns	3.57Mt/s	Synchronous
0101	300ns	3.33Mt/s	Synchronous
0100	320ns	3.12Mt/s	Synchronous
0011	340ns	2.94Mt/s	Synchronous
0010	360ns	2.77Mt/s	Synchronous
0001	380ns	2.63Mt/s	Synchronous

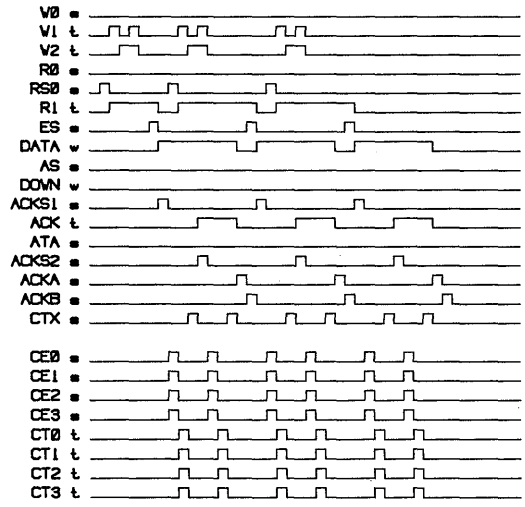


Figure 18. Fast Synchronous Write State Register Timing

register, generated at the start of the ACK delay, counts down the REQ/ACK offset counter. The DOWN state is ORed with the R1 state to generate the count-down signal sent to the REQ/ACK offset counter. The full CY7C361 equation list, which describes this DOWN state and the other configured state registers, appears in *Appendix B*.

External ACK Control

With the CY7C361 handling the time delays for an asynchronous transfer, the transfer sequence so far consists of these steps: information is read from the FIFO, placed in the transmit data register, and delayed for 60 ns, then ACK is generated.

To handle some SCSI possibilities not considered so far, this sequence must be modified. In asynchronous mode, for instance, the information being transmitted can be changed as soon as REQ is removed. The information assertion delay can be started at this point, but the ACK signal sent to the target cannot be transmitted until REQ is received. Although CY7C361 state transitions could handle this type of gating mechanism, the additional delays caused by recognition and metastable-prevent functions would slow the interface. A single AND function of these two signals might appear to correct the problem, but the REQ signal can react very quickly to the returned ACK signal. If the REQ reaction occurs fast enough, the state machine could miss the transition and continue to wait for a signal that has already occurred.

The solution is to construct a small latch external to the CY7C361. The latch allows the ACK signal to be generated as soon as possible, but only transmitted on the SCSI bus after the REQ signal is received. The latch's output prompts the CY7C361 to terminate the current ACK when the CY7C361 sees an external ACK present and REQ not active.

Now another SCSI possibility must be considered. When the HBA receives information on the SCSI bus in asynchronous mode, the ACK signal is just a repeated REQ signal. The repeated REQ must still be justified by the half-full signal from the FIFO. This extra qualifier requires use of another latch to handle the following sequence: If an ACK is returned when the FIFO half-full state is reached, the ACK being sent remains active until REQ is removed, but another ACK is not sent until the half-full flag changes and REQ is present.

This same circuit must also give synchronous transfers a bypass path for generating an ACK pulse that is not tied to REQ. Gating the latch with the SYNC signal, which specifies synchronous operation, it is possible to disable the latch for synchronous operations and enable a different path at the same time.

These complex gating functions are again an excellent fit for a PLD. Because the ACK signal is part of the asynchronous transfers' round-trip path, this application needs a fast part to limit the delays and skew between data and clock. The best choices are probably parts running at 10-ns or faster, such as the members of the PAL18G8, PAL20G10, or PAL22V10C families. Although many of these parts are only available with active-Low outputs, you can correct the signal polarity at the SCSI transceiver by reversing the differential signal lines. *Figure 21* shows the necessary gating function for the ACK signal.

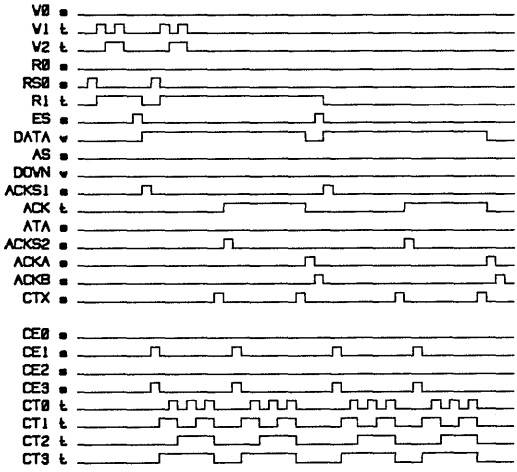


Figure 19. Synchronous Write State Register Timing

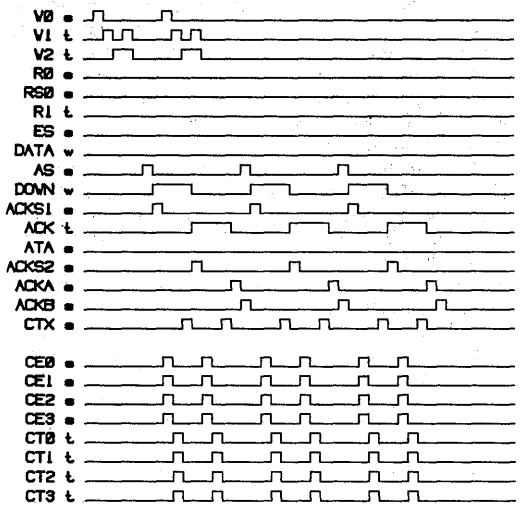


Figure 21. Fast Synchronous Read State Register Timing

Putting It All Together

All the necessary SCSI-2 data-path functions are now accounted for. Interconnecting these pieces as shown in the overall schematic (*Figures22* and *23*) completes the data-path design. The first sheet of this schematic (*Figure22*) details the physical SCSI interface connection and interface transceivers. The second sheet (*Figure 23*) contains the data-path logic functions. Although these two pages form a very compact and fast

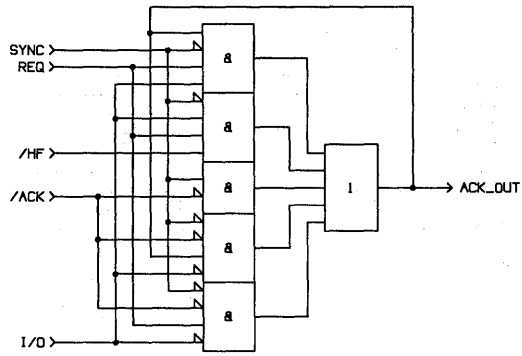


Figure 20. External ACK Gating/Latch

data path, additional functions are needed to complete the host bus adapter. For example, you need control circuitry to operate the transceiver enables; read and write the FIFO on the host bus side; monitor the SCSI bus and change the FIFO direction when necessary; control the selection/reselection sequences; and similar operations.

This data-path design meets its performance goals with a minimal amount of circuitry. Because much of it is implemented in PLD-type devices, you do not have to redesign the HBA to handle almost any change to SCSI-2 or future SCSI versions that affects interface timing; instead, you can simply reprogram the existing parts. This PLD flexibility provides the faster time to market necessary to remain competitive in today's markets.

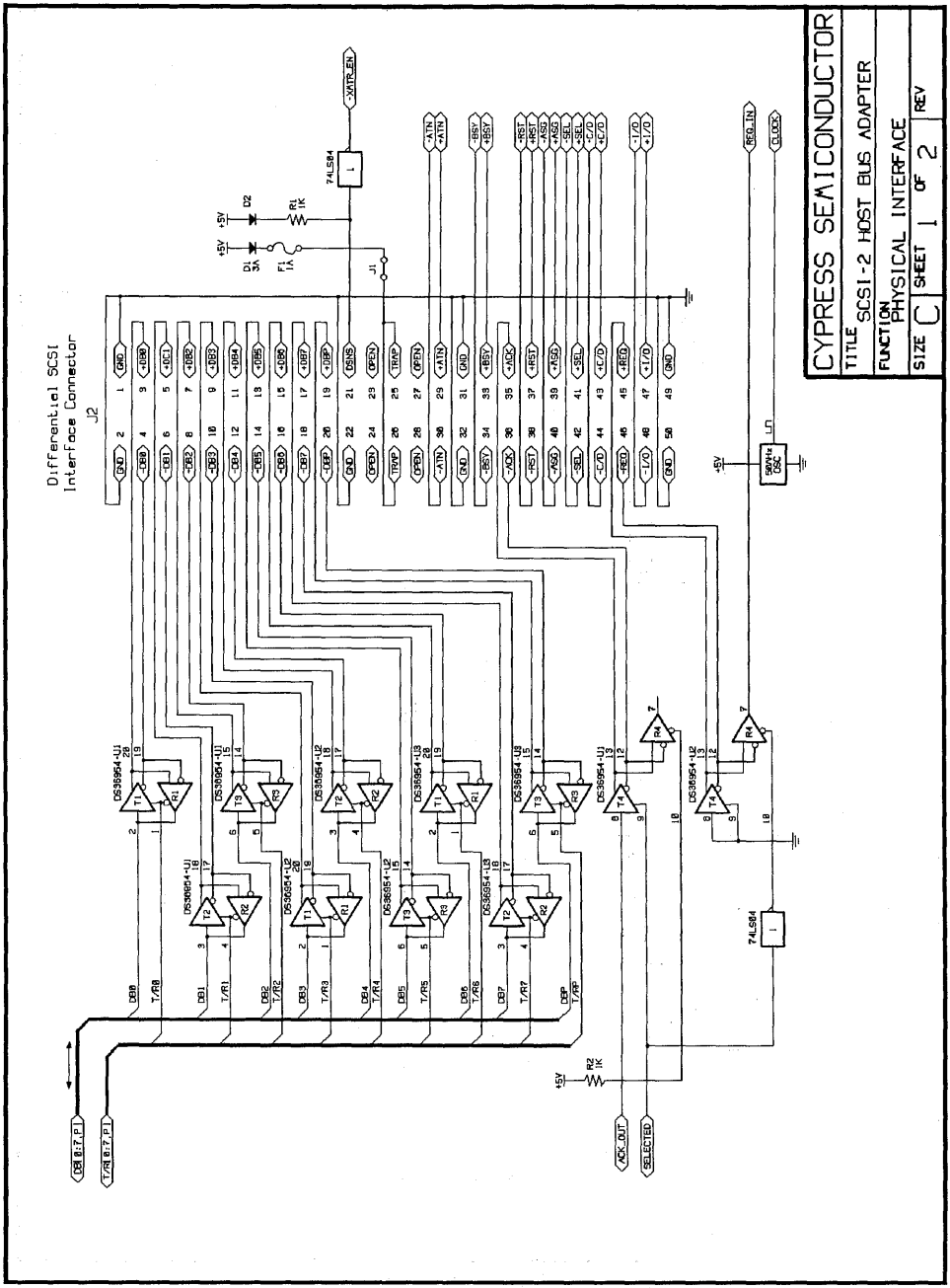


Figure 22. Overall Schematic - Sheet 1

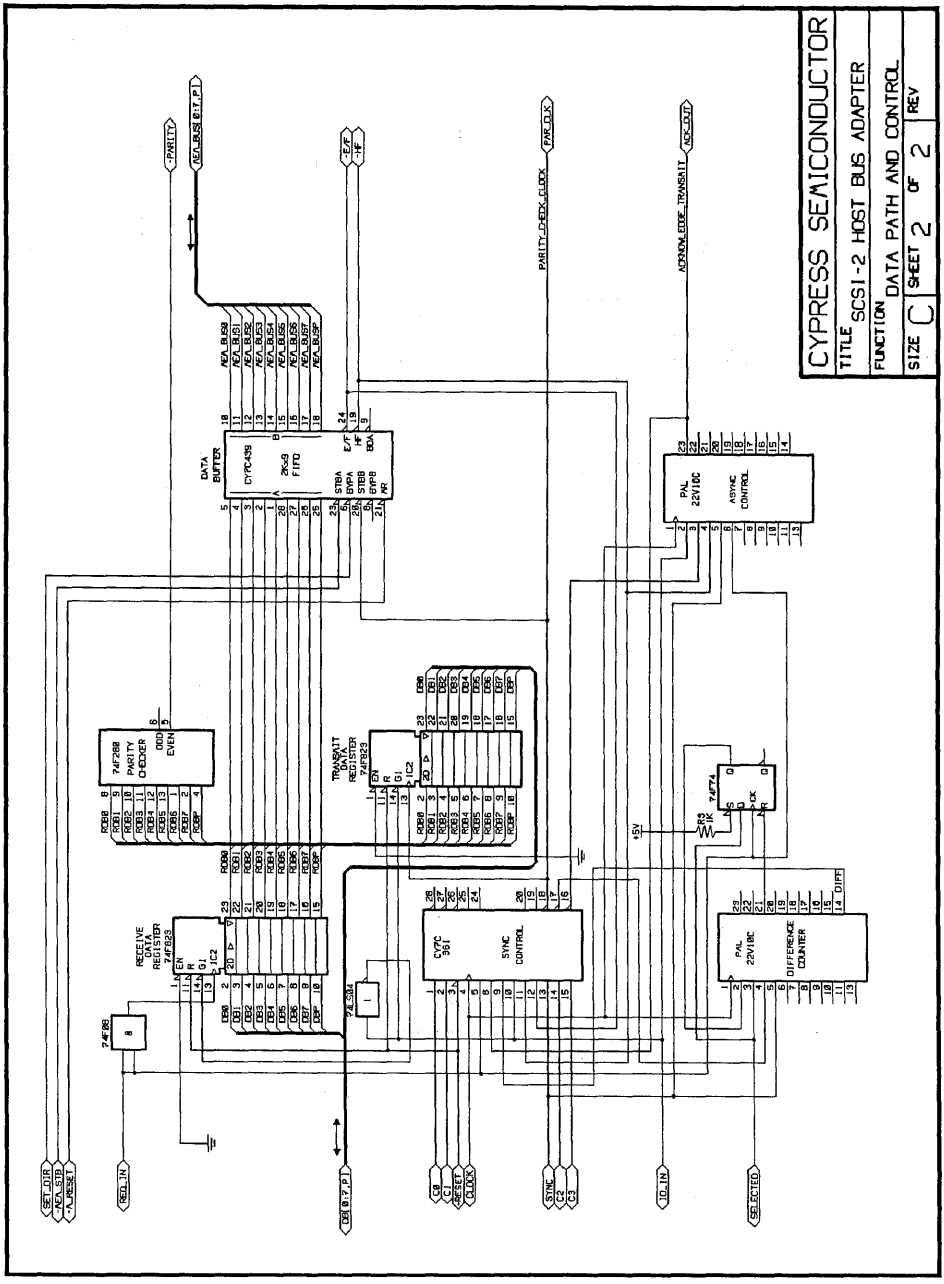


Figure 23. Overall Schematic - Sheet 2

Appendix A. PLD Toolkit Source Code for the REQ/ACK Offset Counter

```
C22V10;

{SCSI2DIF.CYP}

{*************************************************************************
*                                                                        *
*       difference counter  - keeps track of how many REQUEST pulses     *
*       have been received vs. how many ACKNOWLEDGE pulses have been     *
*       sent. The single output DIFF, is used only during synchronous    *
*       data transfers. When DIFF =  1 there exists a received REQUEST    *
*       pulse that has not been responded to by an ACKNOWLEDGE pulse.     *
*       The circuit contains two metastable prevent circuits to          *
*       capture the REQUEST and ACKNOWLEDGE signals. These signals       *
*       are filtered to be enables to a 3 bit up down counter. These     *
*       signals can occur at the same time. If they do the counter       *
*       should not count. Only one count cycle is allowed per enable.    *
*                                                                        *
*************************************************************************}

CONFIGURE;

CLOCK(node=1),                  {50 MHz system clock (20ns period)}
REQ(node=2),                    {SCSI Request signal, used for count up}
SELECTED(node=3),               {used to reset the registers an counter}
!CT_DOWN(node=4),               {down count pulse from CY7C361}
SYNC(node=5),                   {synchronous operation enabled}


{outputs}
DIFF(node=14,noreg,ninv),       {equals 1 if counter is non zero}
Q2(ninv),                       {stage 3 of the counter}
Q1(ninv),                       {stage 2 of the counter}
Q0(ninv),                       {stage 1 of the counter}
DOWN(ninv),                     {qualified ACK or fifo strobe}
DOWN_INH(ninv),                 {inhibit for ACK to limit to one clock}
ACK_IN(ninv),                   {latched ACK signal}
/UP,                            {qualified REQ, clears edge detector}
UP_INH(ninv),                   {inhibit for REQ to limit to one clock}
REQ_IN(ninv),                   {latched REQ signal}


EQUATIONS;

REQ_IN  =       <oe>
                <sum> REQ & SYNC;

UP_INH  =       <oe>
                <sum>   REQ_IN;

UP      =       <oe>
                <sum> REQ_IN & !UP_INH;
```

Appendix A. PLD ToolKit Source Code for the REQ/ACK Offset Counter (continued)

```
ACK_IN  =     <oe>
              <sum>   !CT_DOWN;

DOWN_INH  = <oe>
              <sum>   ACK_IN;

DOWN  =       <oe>
              <sum>  ACK_IN  &  !DOWN_INH;


{3 bit counter}
Q0  =         <oe>
              <sum> SYNC * UP & !DOWN & !Q0
              #   SYNC * DOWN & !UP & !Q0
              #   SYNC * UP & DOWN  & Q0
              #   SYNC * !UP & !DOWN & Q0;

Q1  =         <oe>
              <sum> SYNC * UP & !DOWN & !Q1 & Q0
              #   SYNC * UP & !DOWN & Q1 & !Q0
              #   SYNC * DOWN & !UP & !Q1 & !Q0
              #   SYNC * DOWN & !UP & Q1 & Q0
              #   SYNC * UP & DOWN & Q1
              #   SYNC & !UP & !DOWN & Q1;

Q2  =         <oe>
              <sum> SYNC * UP & !DOWN & !Q2 & Q1 & Q0
              #   SYNC * UP & !DOWN & Q2 & !Q1
              #   SYNC * UP & !DOWN & Q2 & !Q0
              #   SYNC * DOWN & !UP & !Q2 & !Q1 & !Q0
              #   SYNC * DOWN & !UP & Q2 & Q1
              #   SYNC * DOWN & !UP & Q2 & Q0
              #   SYNC * UP & DOWN & Q2
              #   SYNC & !UP & !DOWN & Q2;


DIFF  =       <oe>
              <sum>   Q2
              #   Q1
              #   Q0;
```

Appendix B. PLD ToolKit Source Code for ACK and FIFO Strobe Control

```
CY7C361;
{****************************************************************
*       SCSI2 FIFO and ACK timing controller. Supports asynchronous   *
*       writes and synchronous and fast synchronous reads and writes  *
****************************************************************}

CONFIGURE;
{reset control}
/RESET(node=3,ireg),                    {low asserted reset, single reg}
GLBRST(node=64),                        {global reset control node}

{clock control}
CLKIN(node=4),                          {system clock}
CLKDB(node=74,dbl_clk),                 {enable clock doubler}
IENA(node=29),                          {input clock enable for nodes 3,5,6,9}
IENB(node=30),                          {input clock enable for nodes 10,11,12,13}
IENC(node=31),                          {input clock enable for nodes 1,2,14,15}

{inputs}
ZERO(node=73),                          {internal tie point for enables}
REQ(node=5,iireg),                      {asynchronous SCSI request signal}
ACK_IN(node=6,iireg),                   {gated ACK output signal, latched by REQUEST}
IO_IN(node=10,ireg),                    {SCSI bus set to 0=out, 1=in}
DIFF(node=9,ireg),                      {difference count <> 0}
HF(node=11,iireg),                      {room for data in FIFO - write}
EF(node=12,iireg),                      {data in fifo - read}
SYNC(node=13,ireg),                     {synchronous transfer mode}

{counter inputs}
C0(node=1,ireg),                        {LSB (bit 0) of ACK length counter}
C1(node=2,ireg),                        {bit 1 of ACK length counter}
C2(node=14,ireg),                       {bit 2 of ACK length counter}
C3(node=15,ireg),                       {MSB (bit 3) of ACK length counter}
                                        {all low is an illegal value for C0,C1,C2,C4}

{outputs}
/ACK_OUT(node=16),                      {ACKNOWLEDGE signal, used for asynchronous
                                         SCSI writes and synchronous SCSI reads/writes}
/CT_DOWN(node=17),                      {count down pulse for DIFF counter}
/FIFO_STRB(node=18),                    {FIFO strobe for SCSI writes, FIFO reads}

{state nodes}

{FIFO Write State Machine}
W0(node=32,start),                      {starts FIFO write sequence}
W1(node=33,tog),                        {delay state for FIFO strobe}
W2(node=36,tog),                        {delay state for FIFO strobe}

{FIFO Read State Machine}
RS0(node=34,start),                     {start of sync FIFO read}
R0(node=37,start),                      {start FIFO read strobe}
R1(node=35,tog),                        {stays active until transmit register is marked
                                         as empty, uses delay states from FIFO write machine}
ES(node=38,start),                      {ends FIFO read strobe and sets data
                                         in output latch}
```

Appendix B. PLD ToolKit Source Code for ACK and FIFO Strobe Control (Continued)

```
DATA(node=39,cin,term),                 {data in output latch}
ACKS1(node=43,start),                   {start first ACK delay}
ACK(node=47,tog),                       {ACK active}
ATA(node=42,start),                     {ACK Terminate, Async}
ACKS2(node=47,start),                   {start second ACK delay}
ACKA(node=40,start),                    {synchronous ACK stretch 1}
ACKB(node=41,start),                    {synchronous ACK stretch 2}
AS(node=44,start),                      {start ACK for sync SCSI read}
DOWN(node=45,cin,term),                 {count down pulse for SCSI reads}

{4 bit loadable counter}
CE0(node=54,start),                     {load counter bit 0}
CT0(node=56,tog),                       {counter bit 0}

CE1(node=57,start),                     {load counter bit 1}
CT1(node=58,cin,tog),                   {counter bit 1}

CE2(node=59,start),                     {load counter bit 2}
CT2(node=60,cin,tog),                   {counter bit 2}

CE3(node=61,start),                     {load counter bit 3}
CT3(node=62,cin,tog),                   {counter bit 3}

CTX(node=63,start),                     {terminal count reached (1111)}

EQUATIONS;

{CONTROL}
GLBRST =    <prod>    RESET;            {global reset set to RESET signal}
IENA   =    <inv_sum>   /ZERO;          {allow input clocks}
IENB   =    <inv_sum>   /ZERO;          {allow input clocks}
IENC   =    <inv_sum>   /ZERO;          {allow input clocks}

{STATES}
{start}
W0 =        <prod> IO_IN * REQ;
                    {W0 starts all FIFO write sequences when a REQuest is
                    received with the bus direction set to IN, used as part
                    of the FIFO STBX signal for FIFO writes}

{tog}
W1 =        <prod>
            <inv_prod> /W0 * /W1 * /W2 * /R0 * /RS0;
                    {W1 is triggered by W0 and continues to toggle until W1
                    and W2 return to 0, used as part of the FIFO STBX
                    signal for FIFO writes}

{tog}
W2 =        <prod>   W1;    {W2 is triggered by W1 for two clocks,
                    used as part of the FIFO STBX signal for FIFO writes}

{start}
RS0 =       <prod> /IO_IN * SYNC * EF * DIFF * /R1;
                    {synchronous FIFO read started when the bus is in the proper
                    direction, synchronous mode is active, data is in the FIFO, at
                    least one ACK is pending (DIFF) and a read is not in progress}
```

Appendix B. PLD ToolKit Source Code for ACK and FIFO Strobe Control (Continued)

```
{start}
R0 =        <prod> /IO_IN * /SYNC * EF * /R1;
                    {asynchronous reads are started when the bus is in the proper
                    direction (OUT), synchronous mode is not active, there is data
                    in the FIFO (EF) and a read is not in progress (R1)}
{tog}
R1 =        <prod>
            <inv_prod> /R0 * /RS0 * /ES;
                    {set a read in progress with R0 or RS0, end same with ES when
                    read is complete and no data is in the output latch, used
                    as the FIFO STBX signal for FIFO reads}

{start}
ES =        <prod> R1 * /W1 * /W2 * /DATA;
                    {end read strobe and sets DATA in output latch}
{cin,term}
DATA =      <prod>   ACK
            <inv_prod> /CTX * /ATA;
                    {data in latch set when FIFO read is
                    ended and cleared by end of ACK cycle}
{start}
AS =        <prod> IO_IN * HF * DIFF * /DOWN * /ACK;
                    {start new ACK cycle if DIFF<>0 and
                    cycle not active with room in FIFO}
{cin,term}
DOWN =      <prod>   CTX;               {end counter down count}

{start}
ACKS1 =     <prod>
            <inv_prod> /ES * /AS;       {start the delay counter for the
                    leading edge of the ACK signal}
{tog}
ACK =       <prod>
            <inv_prod> /CTX * /ATA;     {turn ACK on and off}

{start}
ATA =       <prod> ACK_IN * /SYNC * /REQ;
                    {ACK Terminate Async is triggered when an external ACK is
                    present and REQUEST has dropped, this occurs a minimum of 3
                    clocks after ACK is set due to metastable prevent pipeline
                    delays. One more cycle occurs to remove ACK and DATA}

{start}
ACKS2 =     <prod> SYNC * /ACK * CTX;
                    {used only in synchronous modes, starts
                    delay counter for terminate of ACK}

{start}
ACKA =      <prod> CTX * ACK;
                    {lengthen the ACK signal by two clock periods to allow data
                    to change at the trailing edge of output ACK signal}
{start}
ACKB =      <prod>   ACKA;              {lengthen the ACK signal by 2nd clock}
```

Appendix B. PLD ToolKit Source Code for ACK and FIFO Strobe Control (Continued)

```
{start}
CE0 =       <prod>   C0
            <inv_prod> /ACKS1 * /ACKS2;
                        {latch bit 0 of counter for preset}

{start}
CE1 =       <prod>   C1
            <inv_prod> /ACKS1 * /ACKS2;
                        {latch bit 1 of counter for preset}

{start}
CE2 =       <prod>   C2
            <inv_prod> /ACKS1 * /ACKS2;
                        {latch bit 2 of counter for preset}

{start}
CE3 =       <prod>   C3
            <inv_prod> /ACKS1 * /ACKS2;
                        {latch bit 3 of counter for preset}

{tog}
CT0 =       <prod>
            <inv_prod> /CT0 * /CT1 * /CT2 * /CT3 * /CE0;
                        {toggle bit 0 of counter when any bit set}
{cin,tog}
CT1 =       <prod>   CT0;                   {toggle bit 1 of counter when bit 0 is set}

{cin,tog}
CT2 =       <prod> CT0 * CT1;               {toggle bit 2 of counter when bits 1 and 0 are set}

{cin,tog}
CT3 =       <prod> CT0 * CT1 * CT2;         {toggle bit 3 of counter when bits 0, 1, and 2 are set}

{start}
CTX =       <prod> CT0 * CT1 * CT2 * CT3;
                        {counter has completed count up to 1111}

{OUTPUTS}
/C0 =       <inv_oe>;                       {disable output driver to allow C0 as input}

/C1 =       <inv_oe>;                       {disable output driver to allow C1 as input}

/C2 =       <inv_oe>;                       {disable output driver to allow C2 as input}

/C3 =       <inv_oe>;                       {disable output driver to allow C3 as input}

ACK_OUT  =  <inv_sum> /ACK * /ACKA * /ACKB;    {ACKNOWLEDGE signal}

FIFO_STRB  = <inv_sum> /W0 * /W1 * /W2 * /R1;
                        {FIFO read/write strobe}

CT_DOWN  =  <inv_sum> /AS * /DOWN * /R1;
                        {count down input for difference counter}
```

CYPRESS SEMICONDUCTOR

PAL Design Example: A GCR Encoder/Decoder

This application note describes the procedure used to encode/decode serial digital data for recording/reading from one-quarter-inch magnetic tape. The design presented here uses a Cypress CMOS PAL C 16R6 to implement the logic.

Digital data encoding and decoding is often used to increase the reliability of data transmission and storage. One such area is the transformation between data stored on one-quarter inch magnetic tape and serial digital data.

A Little History

The recording format and the Group Code Recording (GCR) code used in this design have been adopted and incorporated in a series of standards. The standards are set by the QIC (Quarter Inch Cartridge) Committee, composed of manufacturers and users of quarter-inch tapes and cartridges. The committee's purpose is to ensure compatibility between manufacturers and reliability to end users.

Quarter-inch tape cartridges are used extensively to backup or archive data from hard disks. Most drives are operated in a continous or streaming mode (for reasons discussed later). Data is recorded at 10,000 FRPI (flux reversals per inch) in a serpentine manner on seven to 14 channels. The tape moves at 30 to 90 ips (inches per second), and the error rates achieved are one in 10^9 or 10^{10}. A cartridge holds 2000 to 3000 feet of 0.001-inch-thick tape and stores 20 to 80 Mbytes of data.

A Typical System

Figure 1 shows a block diagram of a typical tape drive system. The interface with the host (or host adapter) is bidirectional. The interface has a byte-wide data path and 10 to 20 control signals, depending upon the interface standard. Data rates are 300 KBytes/s to 1 MBytes/s.

The formatter or tape controller performs serial/parallel conversion and encoding/decoding as well as error checking; in some cases, the data is also error corrected. Control is usually provided by a state machine, which

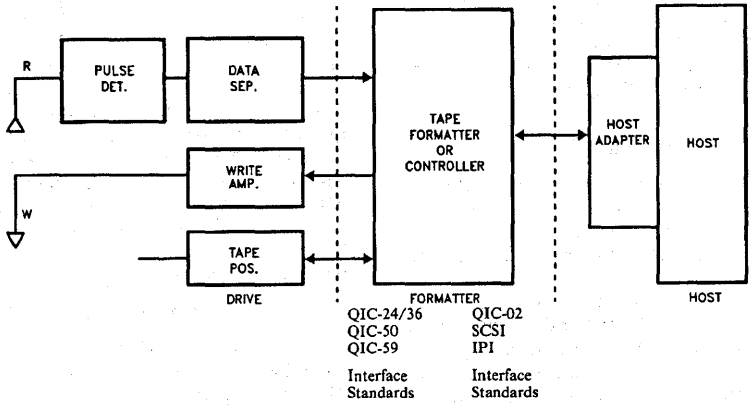


Figure 1. Typical Tape Drive System

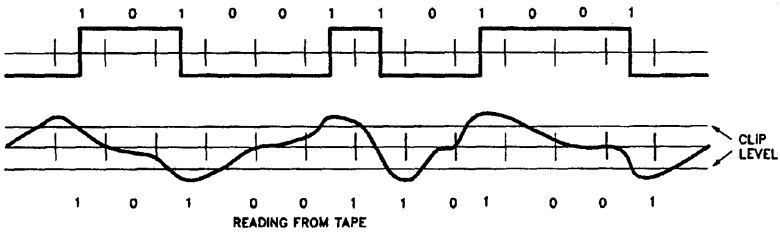


Figure 2. GCR Signal

handles the handshaking with the host as well as control of the tape. Data is written in blocks of various lengths (depending upon the standard), and a read-after-write check is usually performed. Buffer storage of at least two blocks of data is usually provided using static RAMs, FIFOs, or some combination of the two.

The drive electronics include digital signals for controlling and sensing the tape motion and analog signals for the read and write paths. The interface between the drive electronics and the formatter is digital and varies depending on the standard used.

Reading and Writing on Tape

To write on the tape, a current of 100 mA or less is used to change the direction of magnetization. To read from the tape, a coil of wire (the read head) is held against the tape; changes in direction of the tape's magnetic flux induce a voltage (10 mV or less) in the coil.

Recording Codes

All codes used for recording on magnetic mediums are classified as Franaszek Run Length Limited (RLL) codes of the form:

(D, K)

where D = the minimum number of Zeros between consecutive Ones, and K = the maximum number of Zeros between consecutive Ones.

D controls the highest frequency that can be recorded, and K controls the lowest frequency.

Using the Franaszek notation, the GCR code is (1, 2). As illustrated in *Figure 2*, a flux reversal signifies a One, and the absence of a flux reversal signifies a Zero. This is true for all codes.

Peak Detection and Data Separation

GCR recording equipment detects peaks instead of zero crossings because peak-detection circuits are less sensitive to noise. The output of the peak detector goes to the most critical analog circuit in the drive: the data separator.

The data separator provides Ones and Zeros that occur at a precise frequency. The circuit does this using a phase locked loop (PLL). First the data separator synchronizes itself to a crystal-controlled reference clock. Then the circuit attempts to lock itself to the maximum data frequency on the tape. This is done by finding the phase difference between the data separator's own frequency and the peak detector's data output, then adjusting a voltage controlled oscillator (VCO) until the VCO's frequency equals that of the data.

The reference clock's frequency must be at least twice (2f) that of the highest frequency to be read (f). The PLL is synchronized to the 2f reference frequency when not in use.

Before a block of data is recorded, a string of Ones is recorded, which is called the preamble. When the command to read is given, the 2f reference frequency is removed from the data separator, and the signal from the peak detector applied. The PLL then attempts to lock to the preamble — a procedure called getting bit sync.

Just after the preamble, a code violation is recorded so that the formatter can recognize where valid data begins. The detection of the code violation is referred to as obtaining byte sync.

PLLs typically exhibit frequency and phase offsets during preamble acquisition. Phase errors also occur after lock, during the reading of the data field. Differences in tape speed during record and playback (as well as from unit to unit) result in frequency differences between the 2f reference and the data read from the tape. Random phase errors caused by noise, intersymbol interference (bit crowding), timing errors, and other transients might also get the PLL out of lock.

The data separator's PLL is susceptible to these errors because it must satisfy two conflicting conditions: it must lock quickly enough to detect the preamble, but it must not over-correct phase for a single misaligned bit.

Strings of Zeros cause the PLL's phase to shift. If the shift is larger than the bit window, an error occurs. The QIC-24 standard calls for up to a 37-percent bit-shift tolerence, which means that the data separator must be able to recognize a One (flux transversal) that deviates ±18.5 percent from its expected time position without causing a data error. To achieve this performance, a 4-bit binary nibble is encoded into a 5-bit GCR code word, which is written onto the tape.

The Purposes of GCR Code

The 5-bit GCR code format encodes data such that no more than two consecutive Zeros occur in the serial data. This encoding relaxes the performance requirements of the PLL and loop filter, so that the system can achieve the desired performance.

GCR encoding also compensates for the speed variation of the tape due to:

- Mechanical Tolerences in cartridges and tape thickness (±3 percent)
- Tape elasticity and wear
- Motor speed variation
- Temperature and humidity

These static tolerences can result in a (±10-percent tape-speed variation.

In addition to the static tolerences, instantaneous speed variations (ISVs) occur. These result from discontinous tape release at the unwind spool (10 - 20 percent), guide/back stick slip (5 percent), and shuffle ISV (vibration) due to start/stop (5 - 30 percent). The shuffle ISV can be avoided by operating the tape in a continous (streaming) mode. If these dynamic tolerences are added together they can result in (±15-percent speed variation.

The electronics in the tape controller and the drive are designed to compensate for the tape-speed variations due to mechanical tolerences.

The compensation is accomplished by:

- Data encoding and error detection and correction
- PLL design
- Bit-window tolerence

Sequence of operations

During a write operation, the following sequence occurs:

1. Idle (hold)
2. Convert 4-bit parallel input to 5-bit GCR code and load into 5-bit register
3. Shift-out 5 bits to write amplifier.

During a read operation, the following sequence occurs:

1. Idle (same as during write)
2. Shift-in 5 bits
3. Detect sync mark, set/clear invalid flag, convert 5-bit serial input to 4-bit binary value, and load value into register

Note that the read clock and the write clock are not the same. Additionally, the logic must keep up with the tape data rate. Finally, the read and write operations are mutually exclusive. This means that the storage elements (D flip-flops) can be time-shared and that read and write operations require five clocks.

The GCR design requires a total of five states because the idle state is common to both read and write operations. Therefore, the design requires three control lines. It is convenient to designate one control line as an enable line (active Low) and the other two lines as mode-control signals.

This application note does not describe the control of these lines or the required clock synchronization. This is because at the next level of control, you must implement in hardware the responses to error conditions. These response choices tend to be application dependent as well as subjective.

The diagrams in *Figure 3* show the flow of data under control of the ENABLE signal and the M0 and M1 mode-control signals. The GCR code used in this design is part of the QIC-24 Standard and is also the ANSI X3.54 standard (1976). The MSB (leftmost bit) is recorded first. Note that there are a maximum of two consecutive Zeros in the 5-bit code recorded on the tape.

Table 1. GCR Code

Line Number (For Ref.)	4-Bit Code D 3	D 2	D 1	D 0	5-Bit Code Y 3	Y 2	Y 1	Y 0	S o
0	0	0	0	0	1	1	0	0	1
1	0	0	0	1	1	1	0	1	1
2	0	0	1	0	1	0	0	1	0
3	0	0	1	1	1	0	0	1	1
4	0	1	0	0	1	1	1	0	1
5	0	1	0	1	1	0	1	0	1
6	0	1	1	0	1	0	1	1	0
7	0	1	1	1	1	0	1	1	1
8	1	0	0	0	1	1	0	1	0
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	0	1	0	1	0
11	1	0	1	1	0	1	0	1	1
12	1	1	0	0	1	1	1	1	0
13	1	1	0	1	0	1	1	0	1
14	1	1	1	0	0	1	1	1	0
15	1	1	1	1	0	1	1	1	1
	A 3	A 2	A 1	A 0	B 0	B 1	B 2	B 3	B 4

Design Procedure

The procedure for designing the GCR circuits is to map the code conversions using Venn diagrams and write the logic equations as the sum of products, or in minterm form. Because the design requires six flip-flops, the logic is implemented using a CY7C16R6 PAL. Because the PAL has inverting output buffers, the Zeros are mapped instead of the Ones. The D flip-flops require an extra term to hold their states when the ENABLE is HIGH.

For a conventional D flip-flop, for example, the form of the logic equations is:

```
D =   ENABLE 1 (Q)          ; RECIRCULATE
                              PRESENT STATE
    + ENABLE 2 (F2)         ; FUNCTION 2
    + ENABLE 3 (F3)         ; FUNCTION 3
```

where the ENABLE controls are mutually exclusive.

4-bit to 5-bit Conversion for Y3 Output

At the bottom of *Table 1*, the 5-bit code columns are labeled B0 through B4 to help show how the 4-bit code is mapped. In addition, the line numbers are labeled 0 through 15, which correspond to the values of the 4-bit binary code.

Figure 4a shows how the 4-bit binary code is mapped on the Venn diagram. For example, reference line zero, which corresponds to binary value zero, is located in the lower right hand corner.

The Venn diagram in *Figure 4b* shows the conversion for the Y3 output, which is labeled the B0 input to the D flip-flop. Note that the parallel nibble (see *Figure 3*) is reversed end for end so that the MSB is written first when the nibble is shifted out.

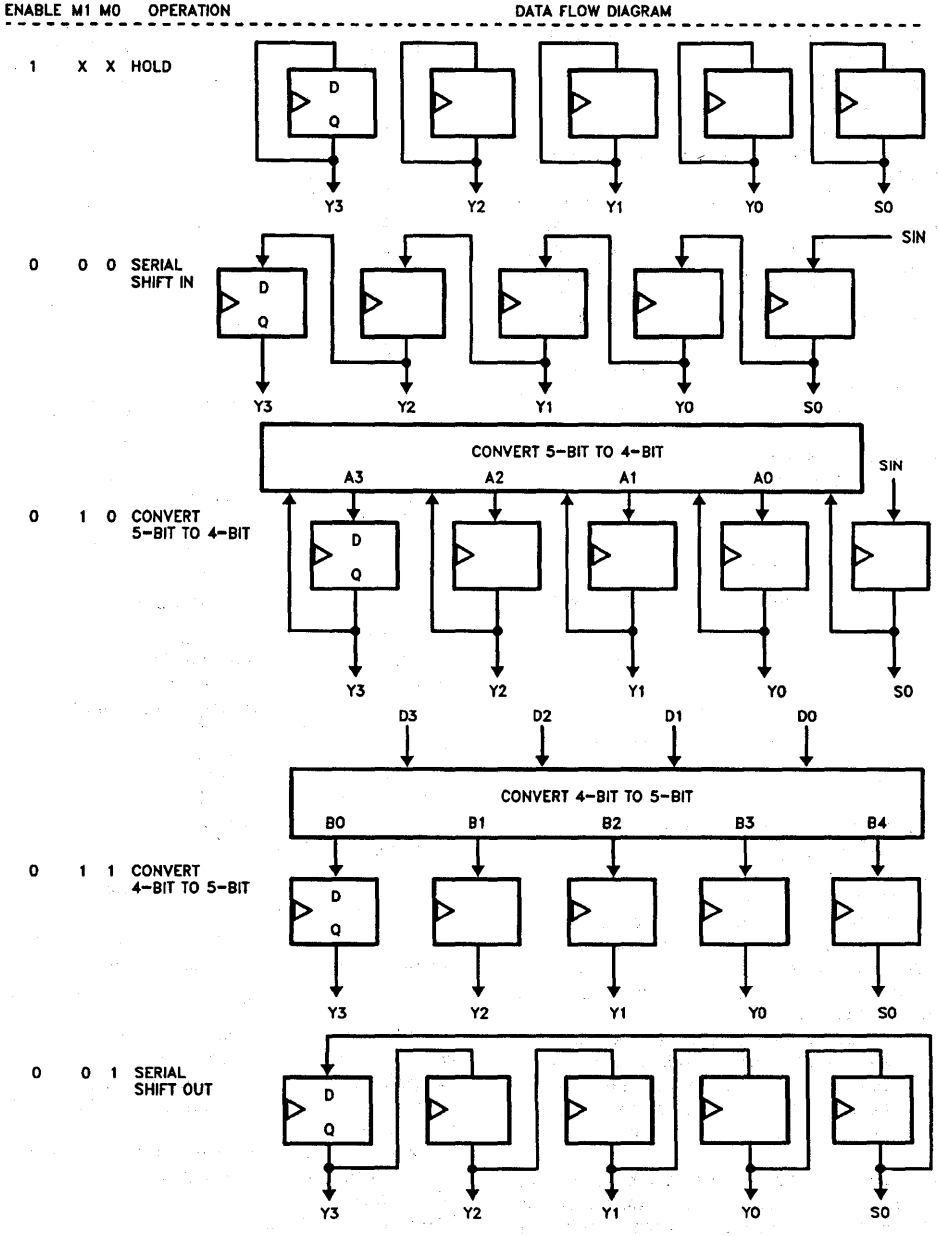


Figure 3. Data Flow Diagrams

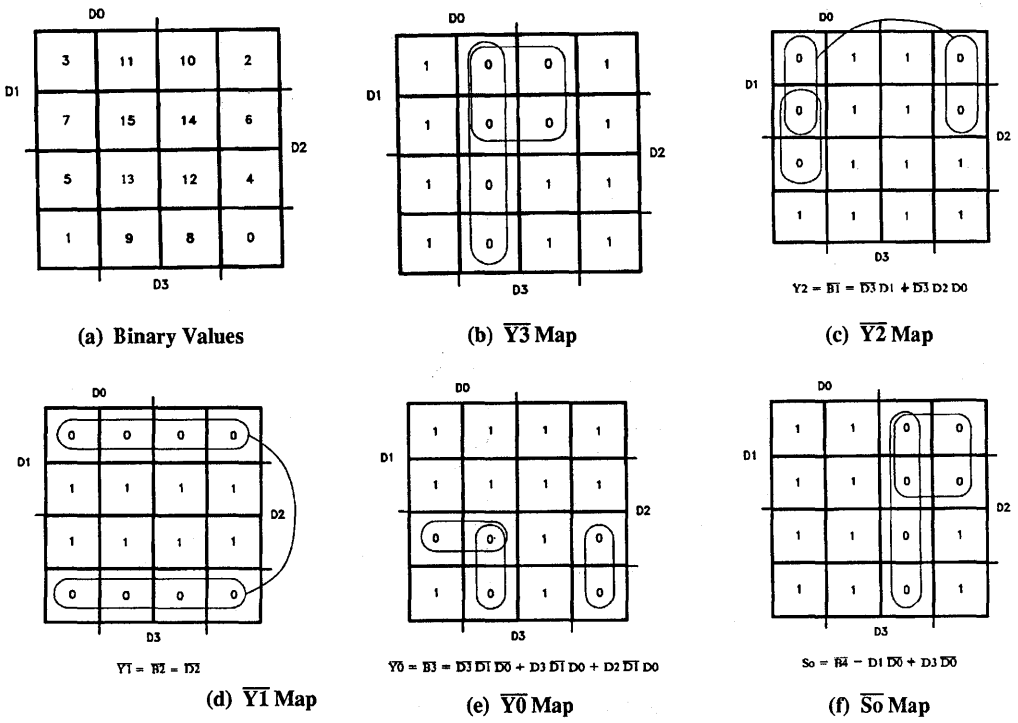


Figure 4. 4- to 5-Bit Conversions

In *Figure 4b*, the Ones and Zeros in column B0 are mapped. For example, reference line zero has the value One in column B0 of *Table 1*. Therefore, a One is placed in the square corresponding to binary value Zero in *Figure 4b*. In a similar manner, reference line 15 has a value of zero in column B0, so a Zero is placed in the square corresponding to binary value fifteen.

Writing the Equation

If the output of the 16R6 PAL were positive-true logic, the equation would include all the Ones on the Venn Diagram. However, because the PAL output is negative logic (active Low), the equation includes all the Zeros. When the PAL inverts the signals, the Zeros are changed to Ones, so that the final outputs are positive-true logic. By inspection:

$\overline{B0} = D3\ D0 + D3\ D1$

or,

$\overline{Y3} = D3\ D0 + D3\ D1$

5- to 4-bit Conversion for $\overline{Y}$ Outputs

A 5- to 4-bit conversion for $\overline{Y}$ outputs requires two 16-square Venn diagrams, because $2^5 = 32$ possible binary values exist. Note in *Table 1*, however, that the 5-bit code columns do not use all 32 possible combinations. The unused combinations are Don't Cares, which are represented by Xs in the Venn diagrams. Don't Cares can be either Ones or Zeros, which further reduces or simplifies the logic equations.

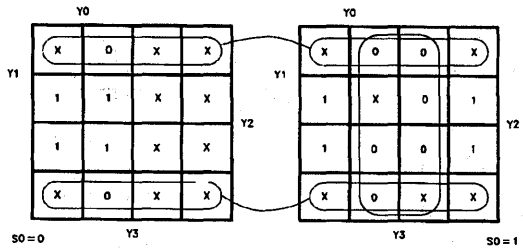
The procedure is to plot the Ones and Zeros, put Xs in the blank squares, and write the equations for the Zeros (*Figure 5*).

Serial Shift In

During serial shift in (both mode control signals Low), the data separator's data output goes to the formatter's input. The signal is called SIN and is applied to the SOUT flip-flop's D input. The SOUT flip-flop's output goes to the Y0 flip-flop's D input, whose output goes to the Y1 flip-flop's input, etc. After five read clocks, the MSB of the 5-bit GCR coded data is in Y3, and the LSB is in SOUT.

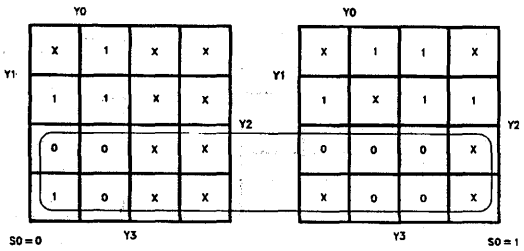
Serial Shift Out

During a write operation, after the 4-bit data is converted to 5-bit data and reversed, the data is shifted out using the write clock and written on tape. The shift direction is opposite to that in serial shift in. Note that the data is right-shifted "end around" (see *Figure 3*) so that after five write clocks the same data appears in the register.



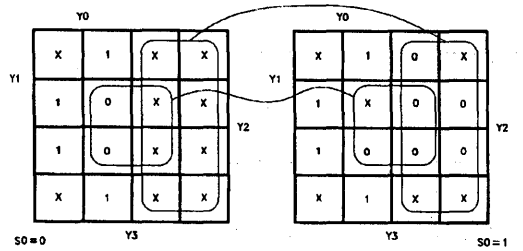
$\overline{Y3} = \overline{A3} = \overline{Y2} + Y3\,So$

(a) $\overline{Y3}$ Map



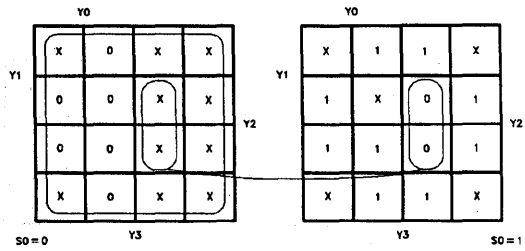
$Y2 = A2 = Y1$

(b) $\overline{Y2}$ Map



$\overline{Y1} = \overline{A1} = \overline{Y0} + Y3\,Y2$

(c) $\overline{Y1}$ Map



$\overline{Y0} = \overline{A0} = Y3\,Y2\,\overline{Y0} + \overline{So}$

(d) $\overline{Y0}$ Map

Figure 5. 5- to 4-Bit Conversions

Invalid Flag (INV Flip-Flop)

The Invalid flip-flop is set to a One when an invalid 5-bit code is read from the tape. This tells the tape formatter that the next data read is the beginning of the data block. Because INV is a negative-true signal, the logic equations are written for Ones on the Venn diagram.

The 16 binary values *not* listed in *Table 1* are plotted as Ones in *Figure 6*. Squares corresponding to valid 5-bit codes contain Zeros; the rest of the squares contain Ones. The equation for the Ones is:

$$INV = Y0\,\overline{SOUT} + Y3\,Y2 + Y3\,Y1\,Y0 + Y3\,Y2\,Y1\,Y0\,S_{out}$$

The Invalid flip-flop is enabled by a signal called C_{IF} (Control Invalid Flag) and reset when C_{IF} is Low.

Synchronization Mark Detection

Bit synchronization is achieved when the illegal 5-bit code of all Ones is read from the tape. This condition is the logical AND of all 5 bits, or

$BS = Y3\,Y2\,Y1\,Y0\,S_{OUT}$.

Implementing the Design

Once the conceptual design is complete, it must be reduced to practice. This process has two main steps: Describe the logic using a high-level language, and Program the PAL

Several design programs that run on the IBM PC (or equivalent) or the VAX computer are available from either semiconductor manufacturers or from third-party software vendors. The first such program, called PALASM (PAL Assembler) was developed by Monolithic Memories. The program enables you to describe the logic in terms of Boolean equations, truth tables, or state diagrams using a language whose syntax is comparable to a microcomputer assembly language.

Appendix A shows the equations for the GCR design, written in the PALASM syntax. This ASCII file was created using Wordstar in the non-document mode.

The PALASM file (GCREX.PAL) is then translated to the syntax of the ABLE design program using the TOABEL program. The format of the command is:

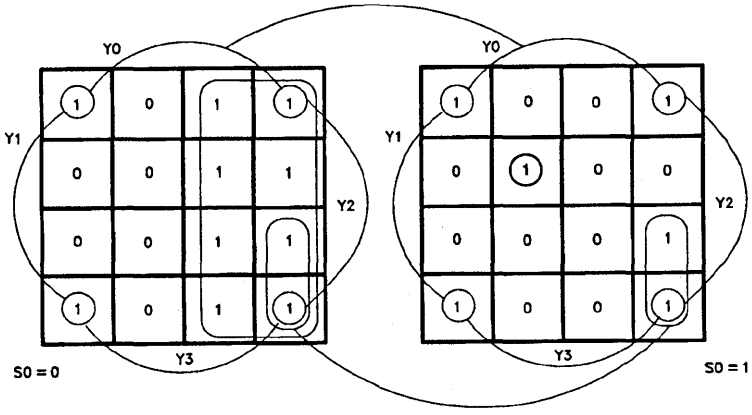
TOABEL -IB:GCREX -OB:GCREXT

The TOABEL program converts the GCREX.PAL file to a file named GCREXT.ABL, whose listing appears in *Appendix B*.

ABEL consists of an executive and several overlay programs that are executed by typing in:

ABEL B:GCREXT

The ABEL program was developed by a programmer manufacturer, Data I/O Corporation. ABEL can simplify a source file (logic reduction), perform logic simulation, and generate test vectors. *Table 2* lists the ABEL programs.



$$INV = \overline{Y0}\,\overline{SOUT} + \overline{Y3}\,\overline{Y2} + \overline{Y3}\,\overline{Y1}\,\overline{Y0} + Y3\,Y2\,Y1\,Y0\,SOUT$$

Figure 6. Binary Values Not Listed in *Table 1*

The ABEL output files for this design based on the PAL C 16R6 (*Figure 7*) are:

- GCREXT.LST
- GCREXT.OUT
- GCREXT.DOC (see *Appendix C*)
- GCREXT.SIM (This design was not simulated.)
- P16R6.JED (see *Appendix D*)

The last file is in JEDEC (JC-42.1-81-62) format and is suitable for loading into a PLD programmer. The listing appears in *Appendix D*. The DOCUMENT program output appears in *Appendix C*. Note that, although the file list includes a simulation file, this design was not simulated.

The CY7C16R6 that implements the design was programmed using the Data I/O model 29B programmer operated in the remote mode to the PC. The design was then verified by testing the device on the bench.

PAL Advantages

This design example illustrates the space-saving advantage of Cypress CMOS PALs. The FUSEMAP program printed out that 40 of the device's 64 available product terms were used.

If the PALASM input equations shown in *Appendix A* are implemented in two-input gates, approximately 30 gates are required for each of the six D flip-flop inputs, or a total of 6 X 30 = 180 two-input gates. The logic equations alone would then require 180/4 = 45 14-pin DIPs. The six flip-flops would require three 14-pin DIPs, for a total of 48 DIPs. Thus, one 20-pin Cypress PAL replaces approximately 50 14-pin DIPs.

This design also illustrates the Cypress PAL's power-saving advantage. The 16R6 PAL's maximum I_{CC} current, under worst-case conditions, is 45 mA. In contrast, the total I_{CC} for 50 TTL packages would be 500 mA, assuming 10 mA for the typical I_{CC} per package. The worst-case I_{CC} for the TTL system could be as high as 20 mA per DIP, which would mean a total of 1A for the system.

The Cypress CMOS PAL reduces system power by a factor of 10 to 15, depending upon whether typical or worst-case numbers are compared.

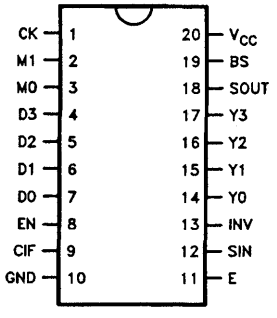


Figure 7. PAL C 16R6

Table 2. ABEL Programs

PROGRAM NAME	FUNCTION
PARSE	Read source file; check syntax; expand macros; act upon assembler directives
TRANSFOR	Convert the description to an intermediate form
REDUCE	Perform logic reduction
FUSEMAP	Create the programmer load (JEDEC) file
SIMULATE	Simulate the operation of a programmed device
DOCUMENT	Create a design documentation file

Appendix A. PALASM Equations

```
PAL16R6              DESIGN EXAMPLE          FILENAME: GCREX.PAL
PAT001                          BRUCE WENNIGER 9/17/85
4B-5B ENCODER/DECODER
CYPRESS SEMICONDUCTOR
 CK M1  M0   D3 D2 D1 D0 /EN  /CIF GND
/E  SIN /INV Y0 Y1 Y2 Y3 SOUT /BS  VCC

/SOUT := EN*/SOUT                    +     ; HOLD/RECIRCULATE
      /EN*/M1*/M0*/SIN               +     ; SERIAL SHIFT IN
        /EN*/M1* M0*/Y0              +     ; SERIAL SHIFT OUT
        /EN* M1*/M0*/SIN             +     ; CONV. SIN & LOAD
        /EN* /M1* M0* D1*/D0         +     ; CONV. PAR. & LOAD
        /EN* /M1* M0* D3*/D0               ; DITTO

/Y0  := EN*/Y0                       +     ; HOLD
        /EN*/M1*/M0*/SOUT            +     ; SERIAL SHIFT IN
        /EN*/M1* M0*/Y1              +     ; SERIAL SHIFT OUT
        /EN* M1*/M0*/SOUT            +     ; CONV. SIN & LOAD
        /EN* M1*/M0* Y3* Y2*/Y0      +     ; DITTO
        /EN* M1* M0* D2*/D1* D0      +     ; CONV. PAR. & LOAD
        /EN* M1* M0* D3*/D1* D0      +     ; DITTO
        /EN* M1* M0*/D3*/D1*/D0            ; DITTO

/Y1  := EN*/Y1                       +     ; HOLD
        /EN*/M1*/M0*/Y0              +     ; SERIAL SHIFT IN
        /EN*/M1* M0*/Y2              +     ; SERIAL SHIFT OUT
        /EN* M1*/M0*/Y0              +     ; CONV. SIN & LOAD
        /EN* M1*/M0* Y3* Y2          +     ; DITTO
        /EN* M1* M0*/D2                    ; CONV. PAR. & LOAD

/Y2  := EN*/Y2                       +     ; HOLD
        /EN*/M1*/M0*/Y1              +     ; SERIAL SHIFT IN
        /EN*/M1* M0*/Y3              +     ; SERIAL SHIFT OUT
        /EN* M1*/M0*/Y1              +     ; CONV. SIN & LOAD
        /EN* M1* M0*/D3* D1          +     ; CONV. PAR. & LOAD
        /EN* M1* M0*/D3* D2* D0            ; DITTO

/Y3  := EN*/Y3                       +     ; HOLD
        /EN*/M1*/M0*/Y2              +     ; SERIAL SHIFT IN
        /EN*/M1* M0*/SOUT            +     ; SERIAL SHIFT OUT
        /EN* M1*/M0* Y3* SOUT        +     ; CONV. SIN & LOAD
        /EN* M1*/M0*/Y2              +     ; DITTO
        /EN* M1* M0* D3* D0          +     ; CONV. PAR. & LOAD
        /EN* M1* M0* D3* D1                ; DITTO

INV  :=/CIF* INV                     +     ; HOLD INV FLAG (ACTIVE LOW)
         CIF* M1*/M0*/Y3*/Y2         +     ; SET IF INVALID
         CIF* M1*/M0*/Y3*/Y1*/Y0     +     ; DITTO
         CIF* M1*/M0*/Y0*/SOUT       +     ; DITTO
         CIF* M1*/M0* Y3* Y2* Y1* Y0* SOUT ; DITTO

BS   = Y3* Y2* Y1* Y0* SOUT                ; BIT SYNC. (ACTIVE LOW)
```

Appendix B. ABEL Listing

```
module _gcrext;          flag '-r0';
title
'PAL16R6                 DESIGN EXAMPLE              FILENAME: GCREX.PAL
PAT001                              BRUCE WENNIGER 9/17/85
4B-5B ENCODER/DECODER
CYPRESS SEMICONDUCTOR
-Translated by TOABEL-';
P16R6 device 'P16R6';

"declarations
        TRUE,FALSE = 1,0;
        H,L = 1,0;
        X,Z,C = .X.,.Z.,.C.;

        GND,VCC
                pin  10,20;

        CK,M1,M0,D3,D2,D1,D0,EN,CIF,E
                pin  1,2,3,4,5,6,7,8,9,11;

        INV,Y0,Y1,Y2,Y3,SOUT
                pin  13,14,15,16,17,18;

        SIN,BS
                pin  12,19;

equations
        !SOUT  := !EN & !SOUT
                # EN & !M1 & !M0 & !SIN
                # EN & !M1 & M0 & !Y0
                # EN & M1 & !M0 & !SIN
                # EN & M1 & M0 & D1 & !D0
                # EN & M1 & M0 & D3 & !D0 ;
        " HOLD/RECIRCULATE
        " SERIAL SHIFT IN
        " SERIAL SHIFT OUT
        " CONV. SIN & LOAD
        " CONV. PAR. & LOAD
        " DITTO

        !Y0    := !EN & !Y0
                # EN & !M1 & !M0 & !SOUT
                # EN & !M1 & M0 & !Y1
                # EN & M1 & !M0 & !SOUT
                # EN & M1 & !M0 & Y3 & Y2 & !Y0
                # EN & M1 & M0 & D2 & !D1 & D0
                # EN & M1 & M0 & D3 & !D1 & D0
                # EN & M1 & M0 & !D3 & !D1 & !D0 ;
        " HOLD
        " SERIAL SHIFT IN
        " SERIAL SHIFT OUT
        " CONV. SIN & LOAD
        " DITTO
        " CONV. PAR. & LOAD
        " DITTO
        " DITTO
```

Appendix B. ABEL Listing (Continued)

```
!Y1     := !EN & !Y1
        # EN & !M1 & !M0 & !Y0
        # EN & !M1 & M0 & !Y2
        # EN & M1 & !M0 & !Y0
        # EN & M1 & !M0 & Y3 & Y2
        # EN & M1 & M0 & !D2 ;
" HOLD
" SERIAL SHIFT IN
" SERIAL SHIFT OUT
" CONV. SIN & LOAD
" DITTO
" CONV. PAR. & LOAD

!Y2     := !EN & !Y2
        # EN & !M1 & !M0 & !Y1
        # EN & !M1 & M0 & !Y3
        # EN & M1 & !M0 & !Y1
        # EN & M1 & M0 & !D3 & D1
        # EN & M1 & M0 & !D3 & D2 & D0 ;
" HOLD
" SERIAL SHIFT IN
" SERIAL SHIFT OUT
" CONV. SIN & LOAD
" CONV. PAR. & LOAD
" DITTO

!Y3     := !EN & !Y3
        # EN & !M1 & !M0 & !Y2
        # EN & !M1 & M0 & !SOUT
        # EN & M1 & !M0 & Y3 & SOUT
        # EN & M1 & !M0 & !Y2
        # EN & M1 & M0 & D3 & D0
        # EN & M1 & M0 & D3 & D1 ;
" HOLD
" SERIAL SHIFT IN
" SERIAL SHIFT OUT
" CONV. SIN & LOAD
" DITTO
" CONV. PAR. & LOAD
" DITTO

!INV    := CIF & !INV
        # !CIF & M1 & !M0 & !Y3 & !Y2
        # !CIF & M1 & !M0 & !Y3 & !Y1 & !Y0
        # !CIF & M1 & !M0 & !Y0 & !SOUT
        # !CIF & M1 & !M0 & Y3 & Y2 & Y1 & Y0
                & SOUT ;
" HOLD INV FLAG
" SET IF INVALID
" DITTO
" DITTO
" DITTO

!BS     = Y3 & Y2 & Y1 & Y0 & SOUT ;
" BIT SYNC.

end _gcrext;
```

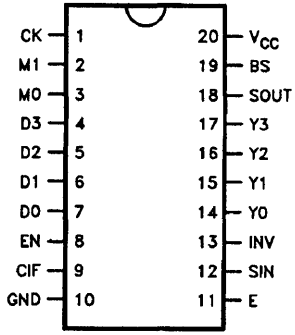
Appendix C. Document File

```
                                                        Page 1
ABEL(tm) Version  1.10  - Document Generator          17-Sept-85 8:30 AM
PAL16R6          DESIGN EXAMPLE        FILENAME: GCREX.PAL
PAT001                        BRUCE WENNIGER 9/17/85
4B-5B ENCODER/DECODER
CYPRESS SEMICONDUCTOR
-Translated by TOABEL-
Equations for Module _gcrext

Device P16R6

   Reduced Equations:

    SOUT := !(!EN & !SOUT
          # EN & !M0 & !M1 & !SIN
          # EN & M0 & !M1 & !Y0
          # EN & !M0 & M1 & !SIN
          # !D0 & D1 & EN & M0 & M1
          # !D0 & D3 & EN & M0 & M1);

    Y0 := !(!EN & !Y0
         # EN & !M0 & !M1 & !SOUT
         # EN & M0 & !M1 & !Y1
         # EN & !M0 & M1 & !SOUT
         # EN & !M0 & M1 & !Y0 & Y2 & Y3
         # D0 & !D1 & D2 & EN & M0 & M1
         # D0 & !D1 & D3 & EN & M0 & M1
         # !D0 & !D1 & !D3 & EN & M0 & M1);

    Y1 := !(!EN & !Y1
         # EN & !M0 & !M1 & !Y0
         # EN & M0 & !M1 & !Y2
         # EN & !M0 & M1 & !Y0
         # EN & !M0 & M1 & Y2 & Y3
         # !D2 & EN & M0 & M1);

    Y2 := !(!EN & !Y2
         # EN & !M0 & !M1 & !Y1
         # EN & M0 & !M1 & !Y3
         # EN & !M0 & M1 & !Y1
         # D1 & !D3 & EN & M0 & M1
         # D0 & D2 & !D3 & EN & M0 & M1);

    Y3 := !(!EN & !Y3
         # EN & !M0 & !M1 & !Y2
         # EN & M0 & !M1 & !SOUT
         # EN & !M0 & M1 & SOUT & Y3
         # EN & !M0 & M1 & !Y2
         # D0 & D3 & EN & M0 & M1
         # D1 & D3 & EN & M0 & M1);

    INV := !(CIF & !INV
```

Appendix C. Document File (Continued)


```
ABEL(tm) Version  1.10  - Document Generator          17 Sept-85 8:30 AM
PAL16R6              DESIGN EXAMPLE          FILENAME: GCREX.PAL
PAT001                        BRUCE WENNIGER 9/17/85
4B-5B ENCODER/DECODER
CYPRESS SEMICONDUCTOR
-Translated by TOABEL-
Equations for Module _gcrext

Device P16R6

        # !CIF & !M0 & M1 & !Y2 & !Y3
        # !CIF & !M0 & M1 & !Y0 & !Y1 & !Y3
        # !CIF & !M0 & M1 & !SOUT & !Y0
        # !CIF & !M0 & M1 & SOUT & Y0 & Y1 & Y2 & Y3);

    BS = !(SOUT & Y0 & Y1 & Y2 & Y3);

Chip diagram for Module _gcrext

Device P16R6
```

PAL C 16R6




```
end of module _gcrext
```

Appendix D. JEDEC File

```
ABEL(tm) Version 1.10  JEDEC file for: P16R6
Created on: 17-Sep-85 8:30 AM
PAL16R6          DESIGN EXAMPLE          FILENAME: GCREX.PAL
PAT001                    BRUCE WENNIGER 9/17/85
4B-5B ENCODER/DECODER
CYPRESS SEMICONDUCTOR
-Translated by TOABEL-*
QP20* QF2048*
L0000
11111111111111111111111111111111
11111101110111011101110111111111
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
11111110111111111111111110111111
10111011111111111111111101111110
10110111111111111111111001111111
01111011111111111111111101111110
01110111111111111111101110111111
01110111011111111111101101111111
00000000000000000000000000000000
00000000000000000000000000000000
11111111111011111111111110111111
10111011111111101111111101111111
10110110111111111111111101111111
01111001110111111111111101111111
01111011111111101111111101111111
01110111011111111111011101111111
01110111011111110111111101111111
00000000000000000000000000000000
00000000000000000000000000000000
11111111111011111111111110111111
10111011111111101111111101111111
10110110111111111111111101111111
01111001110111111111111101111111
01110111111111101111111101111111
01110111011111111111011101111111
01110111011111110111111101111111
00000000000000000000000000000000
00000000000000000000000000000000
11111111111011111111111110111111
10111011111111101111111101111111
10110110111111111111111101111111
01111001110111111111111101111111
01111011111111101111111101111111
01111111111111111111011101111111
01110111011111110111111101111111
00000000000000000000000000000000
11111111111111101111111110111111
10111011111111111111011101111111
10110111111101111111111101111111
01111011111111111111011101111111
01110111011111110111111101111111
```

JEDEC Listing (Continued)

```
01110111101101111011011101111111
00000000000000000000000000000000
00000000000000000000000000000000
11111111111111111111011101111111
10111011111111111111111001111111
10110111111111101111111101111111
01111011111111111111111001111111
01111011110111011111111101111111
01110111111110111111111101111111
00000000000000000000000000000000
00000000000000000000000000000000
11111111111111111111111010111111
10111010111111111111111101111111
10110111111111111111011101111111
01111011111111111111111101111111
01111011101110111111111001111111
01110111111110111101101110111111
01110111011111111011101110111111
01110111011111111011101101111111
11111111111111111111111111100111
01111011111011101111111111111011
01111011111011111110111011111011
01111010111111111111111011111011
01111001110111011101110111111011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000*
C8E51*
D15A
```

CYPRESS SEMICONDUCTOR

T2 Framing Circuitry

This application note describes the design of a T2-based transmission system. This system adds control characters to an image processor's data stream so that the resulting output can be slotted into a T2 channel. DS-2 transmission equipment is then used to relay this information onward.

At receiving locations, the control bits are used to synchronize the site's circuitry to the incoming characters. The data is then restored to its original form, before being routed to its final destination. A block diagram of this system appears in *Figure 1*.

Overview of T1 and T2

Digital transmission systems in North America are hierarchical in structure. Each carrier is multiplexed into higher bandwidth carriers. The lowest level is known as T1. This typically consists of 24 64-Kbit/s pulse code modulation (PCM) telephone channels multiplexed together into frames. A single framing (F) bit precedes every T1 frame to allow for features such as synchronizing channels, sending control characters, and generating cyclic redundancy code (CRC) bits.

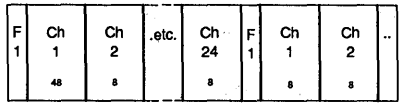
Thus, each frame contains 24 8-bit channels plus an additional framing bit, for a total of 193 bits per frame. The bit rate for a T1 channel equals the rate of a bit in the frame multiplied by the total number of bits in the frame:

$$\frac{24 \times 64{,}000}{192} \times 193 = 1.544 \text{ Mbits/s}$$

The maximum data rate in a T1 channel is therefore

$$1.544 \times \frac{192}{193} = 1.536 \text{ Mbits/s}$$

You can achieve this maximum data rate for T1 transmission when using the Extended Super Frame format. This format dedicates all 8 bits of every channel to user data instead of reserving the eighth bit for channel signaling. *Figure 2* illustrates the composition of a T1 frame.



F = F BIT (one bit)
Channel data = 8 bits
Number of channels = 24

Figure 2. T1 Frame Structure

The next level in the digital communications hierarchy is referred to as T2. Four T1 frames constitute a T2 Multi-frame. These frames are arranged as four sub-frames, each having six blocks of 49 bits. The leading character of every block is used for control purposes, and the following 48 bits consist of data. In total, a Multi-frame comprises 1176 characters.

This format includes three control features:

- *Multi-frame alignment*, provided by a 0111 pattern in each of the four sub-frames. These four bits are referred to as M bits. The fourth M bit location can also serve as an alarm service digit, if required.
- *Frame alignment*, implemented by alternating between logic level 0 and 1. Each sub-frame contains two of these bits, which are referred to as F bits.
- *Justification*: Three bits, referred to as Stuffing Indicator bits (C), are inserted into every sub-frame for justification purposes. Positive, negative, and no justification are possible by inserting the correct code into the relevant locations.

Figure 1. System Overview

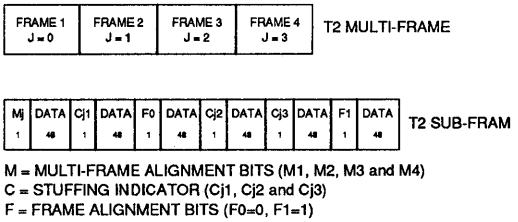


Figure 3. T2 Frame Structure

Figure 3 shows how the data and control bits interleave. *Figure 4* illustrates the sequence in which control bits occur.

The bit rate of T2 information bit rate is 6.312 Mbit/s. The corresponding data rate in a T2 channel is therefore:

$$\frac{6.312 \times 48}{49} = 6.183 \text{ Mbits/s}$$

Further levels exist within the communication hierarchy, but they are not relevant to this design. CCITT G743 provides further details of all the framing structures mentioned here.

Transmitter Site Circuitry

In the example T2 system, the machine from which data originates can operate at frequencies as high as 10 MHz. The data is sourced to the T2 system at 6.183 MHz, which is the data rate of a T2 line. At 10 MHz, stopping and starting artifacts would arise from the disparity between the source and the transmission medium. The output from the transmitter circuitry is maintained at 6.312 MHz to allow the inclusion of control characters into the data stream. Phase-lock-loop design techniques ensure that the clocks in the T2 system and the data source are tightly coupled.

Figure 5 shows the transmitter block diagram. Information feeds into a FIFO, IC1, under control of TXCLKIN (6.183 MHz, the source's clock). TXCLKOUT (6.312 MHz) retrieves data from IC1. IC2 (TXCNTRL PAL) controls the insertion of control bits into the data stream at every 49th time slot. IC4 is a PROM that holds a unique 24-bit control pattern.

A counter, IC3 (PROMADDR PAL), provides the address to the PROM. IC5 (DIVBY49 PAL) is programmed as a counter that increments on successive clock pulses. When this counter reaches its terminal count (49), a carry-

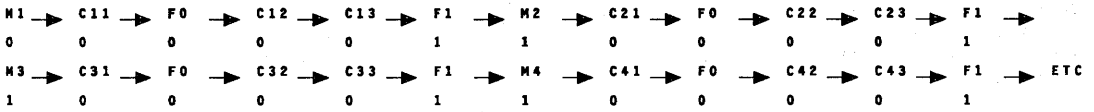


Figure 4. Control Bit Sequence

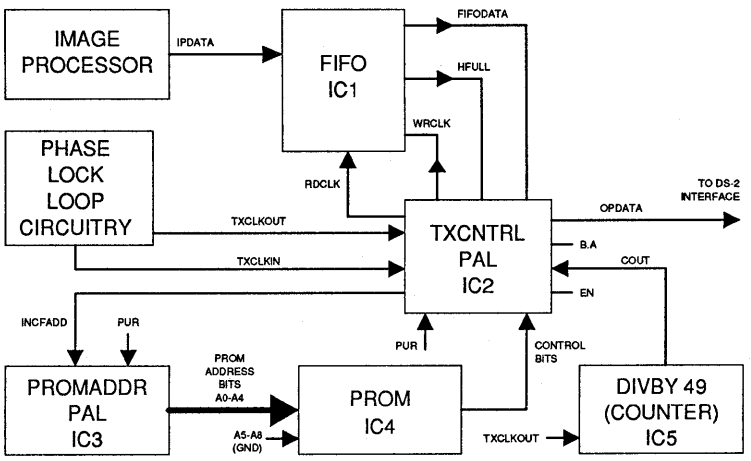


Figure 5. Transmitter Site Circuitry

out signal is produced (FBITLOAD), which serves three purposes:

- It causes the counter to reload its base count (zero)
- It indicates that a control bit has to be inserted into the data stream
- It serves as an input to the state machine in the IC2 PLD, which is the control-bit sequencer that governs when the PROM address generator has to be incremented. A decode of one of the sequencer's states, INCFADD, causes the PROM address to increase by one.

The listings for the design's PALs appear in *Appendices A* through *J*.

Receiver Site Circuitry

The most obvious way to detect a valid pattern of T2 data and control characters is to serially shunt them through a shift register with 1176 stages. Outputs from the first, 50th, 99th, etc. through the 1128th location can then be continuously monitored for the relevant character sequence. This approach is very wasteful in terms of circuitry because monolithic shift registers provide either eight or 16 stages.

Fortunately, you can achieve the same result with one FIFO and two PALs. The principle is to arrange the incoming information so that a pattern recognition circuit periodically samples the most recent, the 50th, the 99th, the 148th, and the 197th bits. This circuitry then compares the information to that expected. When a complete frame of control characters has been detected, the incoming information is frame aligned with the circuitry at the receiver site.

The devices required to implement these tasks appear in *Figure 6*. IC1 (IPFIFO) is a FIFO whose input source is the data and control character stream from the transmit site. The FIFO holds the most recent 196 bits of information entering the receiver circuitry.

IC2 (DATASORT PAL) provides the commands that control this operation and acts as an intermediate buffer stage between the information presented to the FIFO and the characters subsequently read from that device. The outputs of IC3 (CLKGEN PAL) are the Read and Write clocks for the FIFO.

IC4 (ALIGNDET PAL) and IC5 (FRAMCHEK PAL) perform pattern recognition. IC4 compares the expected control bit pattern to the stream of characters appearing at the FIFO's outputs. IC5 interprets the results and sets a flag whenever frame alignment is attained. IC5 also indicates if alignment is subsequently lost.

Frame alignment is declared when four pre-determined bit patterns have been recognized. Thereafter, the circuit makes continuous checks to ensure that alignment is maintained. In total, the circuit seeks 12 bit patterns. If any check yields a negative result, alignment has been lost. A locally generated reset pulse then sets the relevant circuitry to its initial state, and the process of alignment detection begins once again.

For a short period following the application of power, an initialization signal, RESET, is active. This signal ensures that the outputs of IC5 (FRAMCHEK PAL) and IC6 (DSCOUNT PAL) are driven to their initial states and the FIFO (IC1) has all of its internal memory locations and control registers cleared to zero. Once the power-up

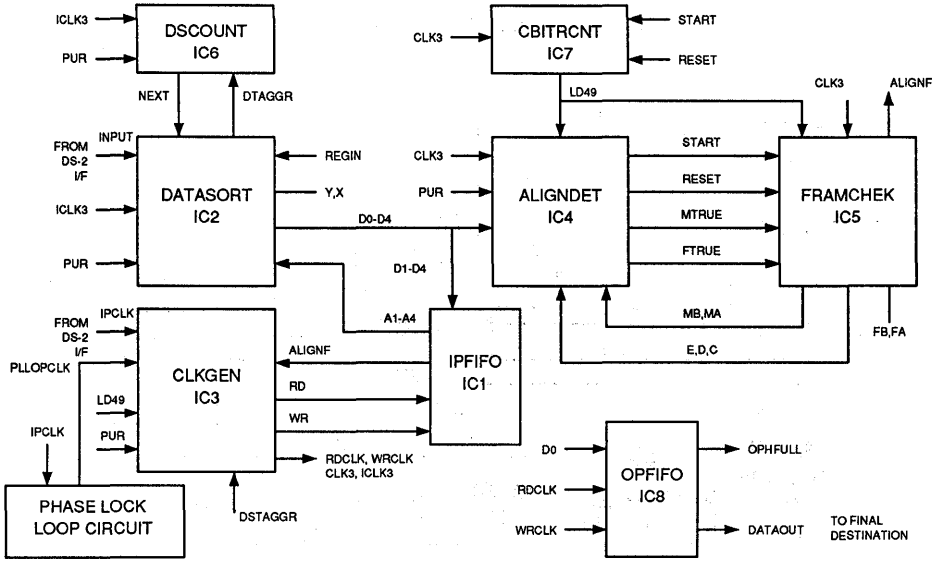


Figure 6. Receiver Site Circuitry

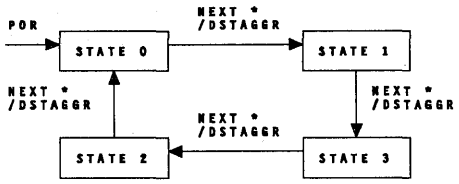


Figure 7. YX Sequencer

routine has completed, the process of writing information into the FIFO commences.

All data entering the receiver is initially fed to the first input stage of the FIFO (D4) via a register in IC2. This ensures that the FIFO's set-up parameter is not violated. Every time a character enters the FIFO, a counter in IC6 increments once. When the terminal count (49) is reached, the counter's carry-out pin (NEXT) goes active. This condition causes the YX sequencer in IC2 to move from its initial state 0 position to state 1 (*Figure 7*). A decode of this state enables the strobe RD, which retrieves stored data from the FIFO. Thereafter, the data from the FIFO's first output stage (A4) is coupled, via IC2, to the FIFO's second input port (D3).

After two further occurrences of NEXT going active, the FIFO's second and third output stages (A3 and A2) are coupled to the third and fourth input ports (D2 and D1), respectively. The YX sequencer goes to state 2. *Figure 8* shows the FIFO's contents when NEXT becomes active for the fourth and final time. At this point, the pattern recognition circuitry can be enabled.

IC2's five data output pins (D4 - D0) effectively perform the same function as a shift register with 197 stages. IC4 monitors this information until it detects the first occurrence of 01000. These control bits are M1/F1/C43/C42/F0, which are the signals present on D4 - D0 of the FIFO after the transmission of 1176 characters. This pattern could correspond to the detection of 01000 in IC4 for the first time. However, it is also quite probable that this sequence could randomly occur in the data stream. Thus, further checks are needed before assuming that the valid recognition pattern has been detected.

As soon as the receiver recognizes the 01000 pattern, a signal labeled START goes active. This term enables a six-stage counter in IC7 (CBITCNTR PAL). The counter counts to 48, then issues a carry-out signal (LD49). A seven-state EDC sequencer (*Figure 9*) in IC5 recognizes every occurrence of this signal and thus always moves to its next stable position (state 1 in this case).

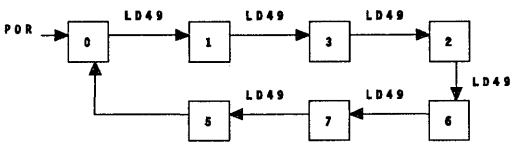


Figure 9. EDC Sequencer

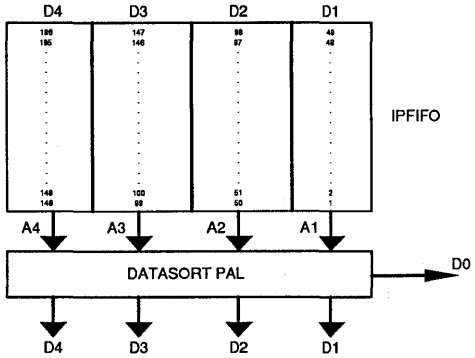


Figure 8. Contents of Receiver Site FIFO

A second check is made in IC4 to determine whether the second valid control bit pattern has been detected. IC4 uses the control bits E, D, C, MA, and MB from IC5's EDC and M sequencer (*Figure 10*) to determine whether the incoming data has been aligned. These control bits represent the state of the sequencers in IC4 and determine the control sequence that should exist on the D4 - D0 inputs.

The second valid control pattern, 10001, is now sought on the bits F1, C43, C42, F0, and C41. If the pattern is not detected, a global reset is issued, and the search for the 01000 pattern recommences. Conversely, if the 10001 pattern is detected, the EDC sequencer assumes state 3. Further, FTRUE becomes true. This signal exists for one clock period and causes the sub-frame detector implemented by the F sequencer (*Figure 11*) to move to its next stable state. A further 147 clocks are allowed to elapse before the next control bit pattern check is carried out.

By this time, the EDC sequencer is in state 6. The occurrence of a 11000 pattern for M4, F1, C33, C32, and F0 provides further proof that alignment has been attained, and the F sequencer moves to its next stable position, state 3. As before, a negative result causes the circuit to issue a global reset. The checking process would then continue with a 10000 pattern for F1, C33, C32, F0, and C31 being sought when a further 49 clock periods had elapsed (EDC sequencer in state 7).

In this case, the occurrence of the correct pattern causes the M sequencer (multi-frame detector) to progress

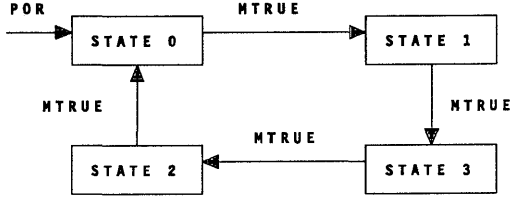


Figure 10. M Sequencer

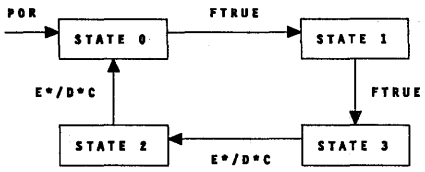


Figure 11. F Sequencer

from its state 0 start position to state 1. The F sequencer's state diagram shows that the sequencer assumes state 2 after the next occurrence of an active LD49 signal, followed one clock period later by a return to the start position, state 0.

As stated previously, the declaration of alignment is made only when four consecutive bit patterns — commencing with the start condition on M0, F1, C3, C3, and F0 — have been sequentially detected. When these criteria have been satisfied, the ALIGNF flag is raised. This flag is held in its active state until one of the ensuing checks produces a negative result. In such an event, the RESET term goes active, thereby forcing certain areas of the receiver's circuitry into the same conditions as occurred at power-up.

Immediately following the receiver's alignment of the incoming data stream, the ensuing information is written into a second FIFO (IC8, OPFIFO). This action is a preface to restoring the data to its original form, i.e., removing the control bits added by the transmitter. Once this operation has been completed, the data can be passed to its final destination. As in the transmitter's design, the receiver's source (IPCLK, 6.312 MHz) and sink (PLLOPCLK, 6.183 MHz) clocks must be locked together. A phase lock loop circuit performs this function.

IC3 provides the control and strobe signals for removing control bits from the data stream. The equations in the source code for this device (*Appendix D*) reveal the following facts:

- Control bits are not written to IC8 (OPFIFO); they coincide with the occurrence of an active LD49 (counter carry-out) signal. Thus, although a data bit is read out of the IPFIFO, the occurrence of LD49 prevents a write strobe (WRCLK) from being generated and the data bit from being written into OPFIFO.
- The process of removing data from IC8 commences as soon as that device is half full, indicated by OPHFULL. This prevents invalid data from being passed to the next stage when the FIFO empties.
- The frequency of the FIFO's write (WRCLK) and read (RDCLK) strobes are 6.312 and 6.183 MHz, respectively.

Other Considerations

The T2 system requires interfaces at both the transmit and receive sites between the hardware described here and the relevant DS-2 equipment. Rockwell's industry-standard DX-33B-4 (CLNS-95-297) and DX-33K-3 (CLNS-95-308) boards suit this task. The latter is fitted with a termination network that matches the receiver's input impedance to that of the transmission medium.

Parts Lists

Transmitter:
IC1 = CY7C433
IC2 = CY7C22V10
IC3 = CY7C22V10
IC4 = CY7C225
IC5 = CY7C22V10

Receiver:
IC1 = CY7C433
IC2 = CY7C22V10
IC3 = CY7C22V10
IC4 = CY7C22V10
IC5 = CY7C22V10
IC6 = CY7C22V10
IC7 = CY7C22V10
IC8 = CY7C433

Appendix A. PAL Equations For TXCNTRL

```
PAL 22V10
T2 TRANSMITTER CONTROLLER (1C2)
CYPRESS SEMICONDUCTOR

/TXCLKIN /PUR  /TXCLKOUT  HFULL FBIT /FBITLOAD FIFODATA  NC8  NC9  NC10 NC11  GND
NC13 WRCLK /RDCLK  /ENREAD OPDATA /EN /B /A /INCFADD NC22 NC23  VCC

EQUATIONS
WRCLK  = TXCLKIN

RDCLK  = TXCLKOUT*ENREAD*/PUR

ENREAD := /ENREAD*HFULL*/PUR
          + ENREAD*/PUR

; TXCLKIN  = 6.183 MHz
; TXCLKOUT = 6.312 MHz
; HFULL = FIFO HALF-FULL FLAG
; PUR = POWER-ON-RESET SIGNAL
; WRCLK = FIFO SHIFT-IN
; RDCLK = FIFO SHIFT-OUT

OPDATA := /OPDATA*FBIT*EN
          + /OPDATA*FIFODATA*/EN
          + OPDATA*/FBIT*EN
          + OPDATA*FIFODATA*/EN

; FBIT = FRAMING BIT FROM PROM
; EN = SELECTS DATA OR FRAMING BIT
; FIFODATA = DATA RETRIEVED FROM FIFO
; OPDATA = DATA PASSED TO DS-2 INTERFACE
; FBITLOAD = FRAMING BIT TO BE INSERTED INTO DATA STREAM
; INCFADD = CAUSES PROM ADDRESS TO BE INCREMENTED
; BA SEQUENCER = CONTROLS SELECTION OF FRAMING BITS

A := /B*/A*FBITLOAD*/PUR
     + /B*A*/PUR

B := /B*A*/PUR
     + B*/A*/PUR

EN = /B*A

INCFADD = B*A

; STATE DIAGRAM FOR BA SEQUENCER
    ;;                          EN                      INCFADD
    ; PUR---- FBITLOAD  ----                 ----          --
    ; ---->| 0 |-------->-------| 1 |------>-----| 3 |------->------| 2  |
    ;        ----                 ----                ----          ----
    ;     |                                                          |
    ;           ---------------------------<--------------------------
```

Appendix B. PAL Equations For DIVBY49

```
PAL 22V10

    DIVIDE BY 49 COUNTER (IC 5)
    CYPRESS SEMICONDUCTOR

    /TXCLKOUT NC2 NC3 NC4 NC5 NC6 NC7 NC8 NC9 NC10 NC11 GND
    NC13 /FBITLOAD Q0 Q1 Q2 Q3 Q4 Q5 NC21 NC22 NC23 VCC

    EQUATIONS

    Q0 := /Q0*/FBITLOAD

    Q1 := /Q1*Q0*/FBITLOAD
        + Q1*/Q0*/FBITLOAD

    Q2 := /Q2*Q1*Q0*/FBITLOAD
        + Q2*/Q1*/FBITLOAD
        + Q2*/Q0*/FBITLOAD

    Q3 := /Q3*Q2*Q1*Q0*/FBITLOAD
        + Q3*/Q2*/FBITLOAD
        + Q3*/Q1*/FBITLOAD
        + Q3*/Q0*/FBITLOAD

    Q4 := /Q4*Q3*Q2*Q1*Q0/FBITLOAD
        + Q4*/Q3*/FBITLOAD
        + Q4*/Q2*/FBITLOAD
        + Q4*/Q1*/FBITLOAD
        + Q4*/Q0*/FBITLOAD

    Q5 := /Q5*Q4*Q3*Q2*Q1*Q0*/FBITLOAD
        + Q5*/Q4*/FBITLOAD
        + Q5*/Q3*/FBITLOAD
        + Q5*/Q2*/FBITLOAD
        + Q5*/Q1*/FBITLOAD
        + Q5*/Q0*/FBITLOAD

    FBITLOAD = Q5*Q4*/Q3*/Q2*/Q1*/Q0

    ; T2CLKOUT = 6.312 MHz
    ; Q0-Q4 = COUNTER OUTPUTS
    ; FBITLOAD = USED TO INSERT FRAMING BITS INTO DATA STREAM
    ;                 (EVERY FORTY-NINTH LOCATION)
```

Appendix C. PROM Equations

PROM
FILENAME:PROM

CONTROL BIT GENERATOR (IC 4)
CYPRESS SEMICONDUCTOR

ADDRESS (HEX)	PROM CONTENTS (HEX)
00	00
01	00
02	00
03	00
04	00
05	01
06	01
07	00
08	00
09	00
0A	00
0B	01
0C	01
0D	00
0E	00
0F	00
10	00
11	01
12	01
13	00
14	00
15	00
16	00
17	01

Appendix D. PAL Equations For PROMADDR

```
PAL 22V10
FILENAME:PROMADDR
    PROM ADDRESS GENERATOR (IC 3)
    CYPRESS SEMICONDUCTOR

    /TXCLKOUT  /PUR  /INCFADD  NC4  NC5  NC6  NC7  NC8  NC9  NC10  NC11  GND
    NC13  A0  A1  A2  A3  A4  /RELOAD  NC20  NC21  NC22  NC23  VCC

    EQUATIONS

    A0 := /A0*INCFADD*/RELOAD
        + A0*/INCFADD*/RELOAD

    A1 := /A1*A0*INCFADD*/RELOAD
        + A1*/A0*/RELOAD
        + A1*/INCFADD*/RELOAD

    A2 := /A2*A1*A0*INCFADD*/RELOAD
        + A2*/A1*/RELOAD
        + A2*/A0*/RELOAD
        + A2*/INCFADD*/RELOAD

    A3 := /A3*A2*A1*A0*INCFADD*/RELOAD
        + A3*/A2*/RELOAD
        + A3*/A1*/RELOAD
        + A3*/A0*/RELOAD
        + A3*/INCFADD*/RELOAD

    A4 := /A4*A3*A2*A1*A0*INCFADD*/RELOAD
        + A4*/A3*/RELOAD
        + A4*/A2*/RELOAD
        + A4*/A1*/RELOAD
        + A4*/A0*/RELOAD
        + A4*/INCFADD*/RELOAD

    RELOAD = PUR
            + Q4*Q3*/Q2*/Q1*/Q0

    ; T2CLKOUT = 6.312 MHz
    ; PUR = POWER-ON-RESET
    ; INCFADD = INCREMENT ADDRESS COUNT
    ; A0-A4  = PROM ADDRESS
    ; RELOAD = LOAD COUNTER WITH BASE COUNT
```

Appendix E. PAL Equations For DATASORT

```
PAL 22V10
FILENAME; DATASORT

    ARRANGE DATA READY FOR PATTERN DETECTOR (IC 2)
    CYPRESS SEMICONDUCTOR

    /ICLK3  A4  A3  A2  A1  INPUT  /NEXT  /PUR  NC9  NC10  NC11  GND
    NC13  /Y  /X  /DSTAGGR  D4  D3  D2  D1  D0  REGIN  NC23  VCC


    EQUATIONS

    X := /X*/Y*NEXT*/DSTAGGR*/PUR
        + X*/Y*/PUR
        + X*/NEXT*/PUR
        + X*DSTAGGR*/PUR

    Y := /Y*X*NEXT*/DSTAGGR*/PUR
        + Y*X*/PUR
        + Y*/NEXT*/PUR
        + Y*DSTAGGR*/PUR

    DSTAGGR := /DSTAGGR*Y*/X*NEXT*/PUR
                    + DSTAGGR*/PUR

    ; STATE DIAGRAM FOR YX SEQUENCER
    ;
    ;PUR  ---- NEXT*/DSTAGGR  ---- NEXT*/DSTAGGR ---- NEXT*/DSTAGGR ----
    ;  ---->| 0 |------------>-------------| 1 |------------>-------------| 3 |------------>-------------| 2 |
    ;     ----                              ----                            ----                            ----
    ;           |                                    NEXT*/DSTAGGR                                        |
    ;           ----------------------------------------------------<-----------------------------------------

    ; YX SEQUENCER = CONTROLS ARRANGEMENT OF DATA IN FIFO
    ; DSTAGGR = INDICATES WHEN DATA READY FOR PATTERN RECOGNITION
    ; PUR = POWER-ON-RESET
    ; NEXT = COUNTER O/P, CONTROLS DATA ORGANISATION INTO/OUT OF FIFO

    REGIN := INPUT

    D0 := /DSTAGGR
        + /D0*A1*DSTAGGR*/Y*/X
        + D0*Y
        + D0*X
        + D0*A1

    D1 := /Y*/DSTAGGR
        + /D1*Y*X*/DSTAGGR
        + /D1*Y*/X*A2*/DSTAGGR
        + /D1*/Y*/X*A2*DSTAGGR
        + D1*A2
        + D1*X
        + D1*Y*DSTAGGR
```

Appendix E. PAL Equations For DATASORT (cont.)

```
D2 := /Y*/DSTAGGR
      + /D2*Y*/X*A3*/DSTAGGR
      + /D2*/Y*/X*A3*DSTAGGR
      + D2*A3
      + D2*Y*DSTAGGR
      + D2*X*DSTAGGR

D3 := /Y*/X*/DSTAGGR
      + /D3*X*A4*/DSTAGGR
      + /D3*Y*/X*A4*/DSTAGGR
      + /D3*/Y*/X*A4*DSTAGGR
      + D3*A4
      + D3*X*DSTAGGR
      + D3*Y*DSTAGGR

D4 := REGIN

; D0-D4 = OUTPUTS TO PATTERN RECOGNITION CIRCUITRY, ALSO
;             REGISTERED DATA BEING FED BACK INTO FIFO I/P STAGES
; A0-A4 = FIFO OUTPUTS BEING FED TO REGISTER
; INPUT = SERIAL DATA STREAM FROM RECEIVER I/P STAGE
; REGIN = REGISTERED I/P DATA
```

Appendix F. PAL Equations for CLKGEN

```
PAL 22V10
FILENAME;CLKGEN

CLOCK GENERATOR FOR DATA SORTING CIRCUITRY AND OPFIFO (IC 3)
CYPRESS SEMICONDUCTOR

/IPCLK /Y /X /DSTAGGR PLLOPCLK /PUR OPHFULL /LD49 /ALIGNF NC10 NC11 GND
NC13 /CLK3 /ICLK3 /WR /RD /CLK4 /ICLK4 /ENREAD RDCLK WRCLK NC23 VCC

EQUATIONS

CLK3 = IPCLK

ICLK3 = /IPCLK

CLK4 = PLLOPCLK

ICLK4 = /PLLOPCLK

WR = /ICLK3

RD = /Y*X*/DSTAGGR*ICLK4
     + Y*/DSTAGGR*ICLK4
     + /Y*/X*DSTAGGR*ICLK4

; IPCLK = MASTER CLOCK FROM DS-2 INTERFACE (6.312MHz)
; YX SEQUENCER = USED TO CONTROL WHEN FIFODATA RETRIEVED
; DSTAGGR = USED TO CONTROL WHEN FIFO DATA RETRIEVED
; PLLOPCLK = O/P FROM PHASE LOCK LOOP CIRCUIT (6.183 MHz),
;                 DERIVED FROM 6.312 MHz MASTER CLOCK
; CLK3/ICLK3 = DERIVATION OF MASTER CLOCK (6.312 MHz)
; CLK4/ICLK4 = DERIVATION OF PHASE LOCKED O/P (6.183MHz)
; WR = I/P STAGE FIFO SHIFT-IN
; RD = I/P STAGE FIFO SHIFT-OUT

WRCLK = ALIGNF*/LD49*/CLK3

RDCLK = ENREAD*CLK4

ENREAD := /ENREAD*OPHFULL*/PUR
          + ENREAD*/PUR


; WRCLK = SHIFT-IN SIGNAL TO O/P STAGE FIFO
; RDCLK = SHIFT-OUT SIGNAL TO O/P STAGE FIFO
; ENREAD = CONTROLS WHEN DATA CAN BE READ FROM O/P STAGE FIFO
; ALIGNF = "ALIGNMENT" INDICATOR
; LD49 = O/P STAGE FIFO SHIFT-IN DISABLE TERM
; OPHFULL = INDICATES WHEN O/P STAGE FIFO IS HALF FULL
; PUR = POWER-ON-RESET
```

Appendix G. PAL Equations For DSCOUNT

```
PAL 22V10
FILENAME; DSCOUNT

DIVIDE-BY-49 COUNTER FOR DATA SORTING PROCESS (IC 6)
CYPRESS SEMICONDUCTOR

/ICLK3  /PUR  /DSTAGGER  NC4  NC5  NC6  NC7  NC8  NC9  NC10  NC11  GND
NC13  Q0  Q1  Q2  Q3  Q4  Q5  /NEXT  NC21  NC22  NC23  VCC

EQUATIONS

Q0 := /Q0*/DSTAGGR*/NEXT
    + Q0*DSTAGGR*/NEXT

Q1 := /Q1*Q0*/DSTAGGR*/NEXT
    + Q1*/Q0*/NEXT
    + Q1*DSTAGGR*/NEXT

Q2 := /Q2*Q1*Q0*/DSTAGGR*/NEXT
    + Q2*/Q1*/NEXT
    + Q2*/Q0*/NEXT
    + Q2*DSTAGGR*/NEXT

Q3 := /Q3*Q2*Q1*Q0*/DSTAGGR*/NEXT
    + Q3*/Q2*/NEXT
    + Q3*/Q1*/NEXT
    + Q3*/Q0*/NEXT
    + Q3*DSTAGGR*/NEXT

Q4 := /Q4*Q3*Q2*Q1*Q0*/DSTAGGR*/NEXT
    + Q4*/Q3*/NEXT
    + Q4*/Q2*/NEXT
    + Q4*/Q1*/NEXT
    + Q4*/Q0*/NEXT
    + Q4*DSTAGGR*/NEXT

Q5 := /Q5*Q4*Q3*Q2*Q1*Q0*/DSTAGGR*/NEXT
    + Q5*/Q4*/NEXT
    + Q5*/Q3*/NEXT
    + Q5*/Q2*/NEXT
    + Q5*/Q1*/NEXT
    + Q5*/Q0*/NEXT
    + Q5*DSTAGGR*/NEXT

NEXT = PUR
      + Q5*Q4*/Q3*/Q2*/Q1*/Q0*/DSTAGGR

; ICLK3 = 6.312 MHz CLOCK DERIVED FROM DS-2 INTERFACE
; DSTAGGR = INDICATES WHEN DATA IS READY TO BE INTERROGATED BY
;                PATTERN RECOGNITION CIRCUITRY
; PUR = POWER-ON-RESET
; Q0-Q5 = O/P STAGES OF COUNTER
; NEXT = LOAD-ALL-ZEROES COMMAND TO COUNTER
```

Appendix H. PAL Equations For CBITRCNT

```
PAL 22V10
FILENAME; CBITRCNT

CONTROL BIT REMOVAL INDICATOR/COUNTER (IC 7)
CYPRESS SEMICONDUCTOR

/CLK3  /RESET  /START  NC4  NC5  NC6  NC7  NC8  NC9  NC10  NC11  GND
NC13  Q0  Q1  Q2  Q3  Q4  Q5  /LD49  NC21  NC22  NC23  VCC

EQUATIONS

Q0 := /Q0*START*/LD49
    + Q0*/START*/LD49

Q1 := /Q1*Q0*START*/LD49
    + Q1*/Q0*/LD49
    + Q1*/START*/LD49

Q2 := /Q2*Q1*Q0*START*/LD49
    + Q2*/Q1*/LD49
    + Q2*/Q0*/LD49
    + Q2*/START*/LD49

Q3 := /Q3*Q2*Q1*Q0*START*/LD49
    + Q3*/Q2*/LD49
    + Q3*/Q1*/LD49
    + Q3*/Q0*/LD49
    + Q3*/START*/LD49

Q4 := /Q4*Q3*Q2*Q1*Q0*START*/LD49
    + Q4*/Q3*/LD49
    + Q4*/Q2*/LD49
    + Q4*/Q1*/LD49
    + Q4*/Q0*/LD49
    + Q4+/START*/LD49

Q5 := /Q5*Q4*Q3*Q2*Q1*Q0*START*/LD49
    + Q5*/Q4*/LD49
    + Q5*/Q3*/LD49
    + Q5*/Q2*/LD49
    + Q5*/Q1*/LD49
    + Q5*/Q0*/LD49
    + Q5*/START*/LD49

LD49 = Q5*Q4*/Q3*/Q2*/Q1*/Q0
      + RESET

; CLK3 = 6.312 MHz CLOCK DERIVED FROM THE DS-2 INTERFACE
; RESET = LOCALISED RESET GENERATED WHEN "ALIGNMENT" IS LOST
; START = INDICATES THAT THE FIRST CONTROL BIT SEQUENCE (01000)
;             HAS BEEN DETECTED
; Q0-Q5 = COUNTER O/P STAGES
; LD49 = LOAD-ALL-ZEROES COMMAND
```

Appendix I. PAL Equations For ALIGNDET

```
PAL 22V10
FILENAME; ALIGNDET

FRAME ALIGNMENT DETECTOR (IC 4)
CYPRESS SEMICONDUCTOR

/CLK3  D0  D1  D2  D3  D4  /PUR  /E  /D  /C  /LD49  GND
NC13  NC14  /MTRUE  /FTRUE  /START  /RESET  NC19  NC20  NC21  /MB  /MA  VCC

EQUATIONS

START := /START*/D4*D3*/D2*/D1*/D0
         + START*/RESET

FTRUE = /E*/D*C*LD49*/D4*/D1*START
        + E*D*/C*LD49*D4*/D1*START

MTRUE = E*D*C*LD49*D4*D3*/D0*START*/MB
        + E*D*C*LD49*D4*D3*/D0*START*MB*MA
        + E*D*C*LD49*/D4*D3*/D0*START*MB*/MA

RESET = PUR
        + E*D*C*LD49*/D4*START*/MB
        + E*D*C*LD49*/D3*START*/MB
        + E*D*C*LD49*D0*START*/MB
        + E*D*C*LD49*/D4*START*MB*MA
        + E*D*C*LD49*/D3*START*MB
        + E*D*C*LD49*D0*START*MB
        + E*D*C*LD49*D4*START*MB*/MA
        + /E*/D*C*LD49*D4*START
        + /E*/D*C*LD49*/D1*START
        + E*D*/C*LD49*/D4*START
        + E*D*/C*LD49*D1*START

; CLK3 = 6.312 MHz CLOCK DERIVED FROM DS-2 INTERFACE
; D0-D4 = DATA CHANNELS ON WHICH CONTROL-BIT-PATTERN-RECOGNITION
;           IS CARRIED OUT
; PUR = POWER-ON-RESET
; EDC = SEQUENCER USED WHEN SEEKING "ALIGNMENT"
; LD49 = INDICATES WHEN COMPARISON BETWEEN DATA CHANNELS
;        AND EXPECTED PATTERN SHOULD BE CARRIED OUT
; MTRUE = MULTI-FRAME DETECTION INDICATOR
; FTRUE = SUB-FRAME DETECTION INDICATOR
; START = INDICATES THAT THE FIRST CONTROL BIT PATTERN HAS BEEN
;           DETECTED
; RESET = ASSERTED WHEN ACTUAL AND EXPECTED CONTROL BIT PATTERNS
;           ARE NOT IN AGREEMENT
; MBMA = SEQUENCER ASSOCIATED WITH MULTI-FRAME DETECTION
```

Appendix J. PAL Equations For FRAMCHEK

```
PAL 22V10
FILENAME; FRAMCHEK

FRAME ALIGNMENT CHECKER AND OPFIFO WRITE CONTROLLER (IC 5)
CYPRESS SEMICONDUCTOR

/CLK3  /RESET  /MTRUE  /FTRUE  /LD49  NC6  NC7  NC8  NC9  NC10  NC11  GND
NC13  /MB  /MA  /FB  /FA  /E  /D  /C  /ALIGNF  NC22  NC23  VCC


EQUATIONS

MB := /MB*MA*MTRUE*/RESET
      + MB*MA*/RESET
      + MB*/MTRUE*/RESET

MA := /MA*/MB*MTRUE*/RESET
      + MA*/MB*/RESET
      + MA*/MTRUE*/RESET


; M SEQUENCER STATE DIAGRAM
;
; RESET   --- MTRUE     --- MTRUE      --- MTRUE   ---
; ---------->| 0 |------->-------| 1 |-------->-------| 3 |------->--------| 2 |
;            ---              ---                ---              ---
;            |                      MTRUE                          |
;              ------------------------------<------------------------------

FB := /FB*FA*FTRUE*/RESET
      + FB*/FA*/RESET
      + FB*/E*/RESET
      + FB*D*/RESET
      + FB*/C*/RESET

FA := /FA*/FB*FTRUE*/RESET
      + FA*/FB*/RESET
      + FA*/E*/RESET
      + FA*D*/RESET
      + FA*/C*/RESET

; F SEQUENCER STATE DIAGRAM
;
; RESET --- FTRUE   ---  FTRUE  ---  E*/D*C ---
;   ------>| 0 |----->-----| 1 |----->------| 3 |----->-----| 2 |
;          ---         ---          ---          ---
    ;  |                  E*/D*C                 |
    ;          ----------------------<------------------


E := /E*D*/C*LD49*/RESET
     + E*D*/RESET
     + E*/C*/RESET
```

Appendix J. PAL Equations For FRAMCHEK

```
D := /D*/E*C*LD49*/RESET
     + D*/E*/RESET
     + D*/C*/RESET
     + D*/LD49*/RESET

C := /E*/D*/C*LD49*/RESET
    + E*D*/C*LD49*/RESET
    + /E*/D*C*/RESET
    + E*D*C*/RESET
    + /E*C*/LD49*/RESET

; EDC SEQUENCER STATE DIAGRAM

;  RESET--- LD49  --- LD49 --- LD49 --- LD49 --- LD49 --- LD49  ---
;   ------->| 0 |--->---| 1 |--->---| 3 |---->--| 2 |-->---| 6 |--->---| 7 |--->---| 5 |
;           ---           ---         ---         ---         ---         ---         ---
;    |                                                                                 |
;              ------------------------------------<-----------------------------------

ALIGNF := /ALIGNF*E*/D*C*/RESET
              + ALIGNF*/RESET

; ALIGNF STATE DIAGRAM
;                                ALIGNF
; RESET   ---  E*/D*C   ---
; --------->| 0 |------>-----| 1 |
;            ---                ---


; SEQUENCE OF EVENTS PRIOR TO ALIGNMENT DECLARATION:
;
; START-LD49-STRUE-LD49-LD49-LD49-STRUE-LD49-LD49-MTRUE


; CLK3 = 6.312 MHz CLOCK DERIVED FROM DS-2 INTERFACE
; RESET = ISSUED IF ACTUAL AND EXPECTED CONTROL BIT PATTERNS DO
;              NOT AGREE
; MTRUE = MULTI-FRAME DETECTION INDICATOR
; FTRUE = SUB-FRAME DETECTION INDICATOR
; LD49 = INDICATES WHEN COMARISON BETWEEN ACTUAL AND EXPECTED
;            CONTROL BIT PATTERNS SHOULD TAKE PLACE
; MBMA = SEQUENCER ASSOCIATED WITH MULTI-FRAME DETECTION
; FBFA = SEQUENCER ASSOCIATED WITH SUB-FRAME DETECTION
; EDC = SEQUENCER USED IN DETERMINATION OF "ALIGNMENT"
; ALIGNF = WHEN TRUE INDICATES "ALIGNMENT" HAS BEEN ATTAINED
```

CYPRESS SEMICONDUCTOR

Using CUPL With Cypress PLDs

This application note covers the following topics:

- CUPL package components
- CUPL programming language syntax
- CUPL examples, using Cypress PLDs
- CUPL compiling

A high-level universal language for programmable logic devices (PLDs), CUPL works with schematic capture packages such as SCHEMA and OrCAD-SDT and can port to UNIX-based systems.

CUPL Package Components

The CUPL package consists of CUPL (Universal Compiler for Programmable Logic), CSIM (CUPL Simulator), CBLD (CUPL Build), and PTOC (PAL ASM to CUPL Translator).

CUPL

The major component of the CUPL package is the CUPL program. This file allows you to compile logic description files that can be downloaded to a device programmer. CUPL supports Cypress's entire 20-pin PAL family, the PAL C 22V10, the PAL C 20G10, and the CY7C33x family of parts.

In addition to providing a programming syntax similar to that of other PLD programming packages, CUPL helps implement lists, address ranges, and bit fields efficiently. CUPL includes state machine syntax (SMS) and truth-table input capability, allowing you to enter complex designs easily into Cypress's PLDs. CUPL also has four levels of minimization for logic reduction.

CUPL comes with a menu-driven interface and a DOS command-line interface (the latter is explained in the last section of this application note). The menu interface integrates all the features necessary for efficient design implementation, including a program and JEDEC file editor, compiler, and simulator (*Figure1*).

CSIM

CSIM, the file simulator for CUPL, takes an ASCII file as input (filename.SI) and outputs a file called filename.SO. The input file functionally describes the part by specifying the device's input and expected output. The output file contains a comparison of the device's expected output with its actual output; this is based on a file created by CUPL during compilation called the absolute file, filename.ABS. The comparison file contains the original header information found in filename.SI, all vectors that compared positively, and all discrepancies. CSIM flags the discrepancies with the values determined from the original logic equations.

The CSIM command line is shown in *Figure 2*. When running CSIM with the -w or -d flag, you can change the view of the waveform by using the keys shown in *Figure3*.

CBLD

The CBLD program allows you to maintain and personalize CUPL device libraries. *Figure 4* shows the CBLD command line. You can use CBLD to create custom library files consisting, for example, of only the parts you currently use. The structure of this ASCII text file appears in *Figure5*.

CBLD also checks to see if the current CUPL version matches the current version of the device library. If the key in the library does not match the CUPL version,

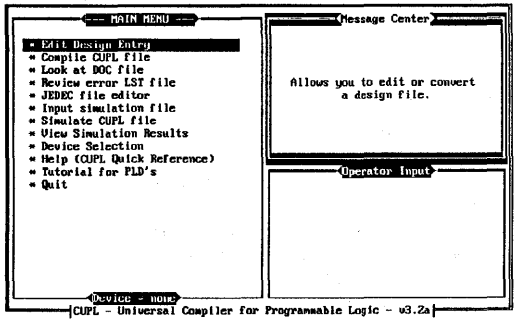


Figure 1. Menu Interface Screen

```
CSIM [flags] [library] source

Where:
[-flags] may have the following values

   -l    create listing file
   -j    append test vectors to JEDEC
         file
   -v    display simulation to screen
   -u    use specified library
   -w    (MS-DOS only) create listing
         file and display waveforms
   -d    (MS-DOS only) display an
         existing simulation output
         file in waveform format

[library] is the name of the library
that contains the device which was
used when CUPL compiled the original
source file.

source is the name of the ASCII
source file
```

Figure 2. CSIM Command Line

```
→     Scroll Right
←     Scroll Left
↑     Scroll Up
↓     Scroll Down
F1    Decrease scale horizontally
F2    Enlarge scale horizontally
F3    Grid on/off
F4    Exit to DOS
F5    Shift screen left
F6    Shift screen right
F9    Create waveform hardcopy
F10   Waveform legend
```

Figure 3. CSIM Waveform Viewing Commands

CBLD generates an error message, and compilation is aborted. The file CUPL.DL contains a description of all devices supported by the current version of CUPL.

CUPL Programming Language Elements

The CUPL programming language's elements and syntax are very similar to those of other languages. Reserved words that cannot be used as variable names are listed in *Figure6*.

You can use alternate number bases in CUPL by putting the base's name within single quotes immediately before the number. The designations for the supported number bases appear in *Table 1*. For example, to assign the hexadecimal value 16 to the variable "A," write:

A = 'h'16

```
CBLD [flags] [build] [library][devices]

Where:
[flags] may have the following values

   -b    generate library using build
         file
   -l    list long contents of library
   -m    list allowable macros by pin
   -t    list short contents of library
   -u    use specified library
   -e    list allowable extensions for
         devices

[Build] is the name of the build file
to be used with the -b option flag

[Library] is a device library name
and path name to be used with the -u
option

[Devices] is one or more device names
to be used with the -t or -l option
```

Figure 4. CBLD Command Line

You can place an "X" within any number to indicate a Don't Care value. *Appendix A* shows an example of using the Don't Care specification within truth tables.

Comments are delimited with /* and */. The CUPL compiler ignores everything between these characters. For example, to put a paragraph of explanation within a program, enclose the entire paragraph in a set of comment delimiters. You do not have to put delimiters on every line, as in some packages.

CUPL also supports list notation. Enclose all items in the list in square brackets:

[variable, variable, variable,...]

When using sequentially numbered lists, you can abbreviate the format to

[variablem..n]

CUPL's format can be considered in three major parts: the header, pin/node definition, and equations sections. The header section contains general information about the design. The pin/node section assigns variable names to the device's pins and nodes. The equations section declares the device's function and can include truth tables, state machine syntax, Boolean equations, or a combination of these three. (Sample CUPL programs are listed in the appendices and are described later in this application note.)

Header Section

Figure 7 shows the header format. The NAME descriptor must be followed by the name for the JEDEC map output, and the DEVICE descriptor must

```
TARGET library
SOURCE library1
   devices | *
   SOURCE library2
   devices | *

Where:
TARGET identifies the new library.

SOURCE identifies the source
libraries.

library indicates the target library
name.

library1 and library2 indicate source
library names

devices describes devices that are
contained in the     libraries

* is used to describe all devices in
a library
```

Figure 5. CBLD Custom Library Build File Format

specify the device library for use during compilation. If you specify a different device file on the command line when you invoke the compiler, this file overrides the name found after DEVICE in the programming file.

Pin/Node Section

The pin declaration assigns specific pins to variable names using the format

PIN pin_n = [!]var;

Both pin_n and var can be lists. Use the "!" with inputs to indicate an active Low. The compiler chooses the signal's inverted sense when it is indicated as active in the logic equations. Use the "!" with outputs to indicate an active-Low output, and write the equations in a logically true form. In this case, the compiler performs DeMorgan 's Theorem on the output variable to ensure that the output is a Low-asserted signal.

APPEND	FORMAT	PIN
ASSEMBLY	FUNCTION	PINNODE
ASSY	IF	PRESENT
COMPANY	JUMP	REV
CONDITION	LOC	REVISION
DATE	LOCATION	SEQUENCE
DEFAULT	MACRO	SEQUENCED
DESIGNER	MIN	SEQUENCEJK
DEVICE	NAME	SEQUENCERS
ELSE	NODE	SEQUENCET
FIELD	OUT	TABLE
FLD	PARTNO	

Figure 6. CUPL Reserved Words

Table 1. Number Base Representation

Base Name	Base	Prefix
Binary	2	'b'
Octal	8	'o'
Decimal	10	'd'
Hexadecimal	16	'h'

Table 2. CUPL Logical Operators

Operator	Example	Description
!	!A	NOT
&	A&B	AND
#	A#B	OR
$	A$B	XOR

The NODE declaration statement tells the compiler that a variable is needed to hold some kind of state information within the device. This variable's outputs are not assigned to any output pin. You can use the NODE statement to assign variable names—and thus functions—to the buried registers in the CY7C330. Or you might use the NODE statement to arbitrarily assign a variable name to any unused macrocell in a PAL C 22V10. This statement has the form

NODE [!]var;

Because the NODE statement arbitrarily assigns a register to the specified variable name, it might be more desirable to force the assignment of a variable to a specific node. You can do this with the PINNODE statement:

PINNODE node_n = ![var]

The FIELD assignment assigns a group of signals to one variable name. This feature is useful for address decoding and with truth tables, as shown in *Appendix A*. The FIELD statement has the form:

FIELD var = [var,var,...,var]

The MIN declaration overrides the minimization level for a specific variable. This is useful, for example, in designs where a portion of the design should not be minimized. The MIN declaration has the form

MIN var[.ext] = level;

```
NAME;
PARTNO;
REVISION;
DATE;
DESIGNER;
COMPANY;
ASSEMBLY;
LOCATION;
DEVICE;
FORMAT;
```

Figure 7. CUPL Header Format

CUPL also contains several preprocessor commands that operate on the source file before the file is passed on to the parser. These commands perform functions such as string substitution, file inclusion, and conditional compilation. The commands allow you to develop general-purpose descriptions or modular portions of descriptions and customize them for different applications. *Appendix D* shows how to use the preprocessor command $DEFINE to assign numbers to state variables.

Ext	Side	Description
.D	L	D input of D flip-flop
.L	L	D input of latch
.J	L	J input of JK flip-flop
.K	L	K input of JK flip-flop
.S	L	S input of SR flip-flop
.R	L	R input of SR flip-flop
.T	L	T input of T flip-flop
.DQ	R	Q output of D flip-flop
.LQ	R	Q output of a latch
.AP	L	Asynch preset of flip-flop
.AR	L	Asynch reset of flip-flop
.SP	L	Synch preset of flip-flop
.SR	L	Synch reset of flip-flop
.CK	L	Programmable clock of flip-flop
.OE	L	Programmable OE
.CA	L	Complement array
.PR	L	Programmable preload
.CE	L	CE input of enabled D-CE type flip-flop
.LE	L	Programmable latch enable
.OBS	L	Programmable observability of buried nodes
.BYP	L	Register bypass
.DFB	R	D feedback selection
.LFB	R	Latch feedback selection
.TFB	R	T feedback selection
.IO	R	Pin feedback selection
.INT	R	Internal feedback selection
.CKMUX	L	Clock MUX selection
.OEMUX	L	Tri-state MUX selection
.TEC	L	Technology-dependent fuse selection
.IMUX	L	Input MUX selection of two pins
.T1	L	T1 input of 2-T flip-flop
.T2	L	T2 input of 2-T flip-flop
.IOD	R	Pin feedback path through D register
.IOL	R	Pin feedback through Latch
.IOCK	L	Clock for pin feedback register
.IOAR	L	Asynchronous reset for pin feedback register
.IOAP	L	Asynchronous preset for pin feedback register
.IOSR	L	Synchronous reset for pin feedback register
.IOSP	L	Synchronous preset for pin feedback register
.ARMUX	L	Asynchronous reset MUX selection
.APMUX		Asynchronous preset MUX selection
.LEMUX	L	Latch enable MUX selection

Figure 8. CUPL Variable Extensions

CUPL Programming Language Syntax

This section focuses on CUPL's equation section. The program's logical and arithmetic operators (*Tables 2* and *3*, respectively) resemble those used in other programming languages.

A variable's function depends on the extension added to it in the logic equation. These extensions define such capabilities as flip-flop descriptions and programmable three-state enables. The first column of *Figure 8* lists the extension that is used after the variable name. The second column indicates the side of the equation on which the extension is used. The third column briefly describes the extension's function. For example, the .OE extension controls the output-enable function for all Cypress PLDs with I/O pins; the .CKMUX extension selects the source for the input-register clock in the CY7C330 and CY7C332; and .D selects registered output on devices that have both combinatorial and registered outputs.

To see the extensions you can use with a specific Cypress part, use the CBLD program. To see all the possible extensions for use when programming the PAL C 22V10, for example, the command line is

CBLD -e CUPL P22V10

You can use the APPEND statement to assign more than one expression to a variable. This is the same as logically ORing the variable's present state with the expression that follows the APPEND statement. The latter has the form

APPEND [!]var[.ext] = expr;

CUPL also has several powerful set operations that you can use to increase code readability and decrease the amount of equation input. These set operations serve in the equations section to simplify equation input. For example,

[var1, var2, var3] & var4;

equates to

[var1&var4, var2&var4, var3&var4]

Table 3. CUPL Arithmetic Operators

Operator	Example	Operation	Priority
+	A+B	Add	1
-	A-B	Subtract	1
*	A*B	Multiply	2
/	A/B	Divide	2
%	A%B	Modulus	2
**	A**B	Exponent	3

```
TABLE var_list_1      =>    var_list_2
{
   input_1            =>    output_1
   input_2            =>    output_n
}

Where:
var_list_1 are the input variables
var_list_2 are then output variables
input_n is the value of the inputs
          (hex by default)
output_n is the value of the outputs
```

Figure 9. Truth Table Entry Format

Use set operations such as this with caution to ensure that when CUPL expands an expression, the result represents the minimum amount of logic needed to completely specify the desired operation. To see if a set of variables equals a constant, type

[var1, var2, var3]:constant

Or to check whether a set of variables lies between a range of constants, type

[var1, var2, var3]:[constant_lo..constant_hi]

CUPL supports truth tables with the format shown in *Figure 9*. Truth tables are one of the easiest ways to express device function, and they are among the most easily modified methods of design entry. You specify the input and output variable lists, then specify a one-to-one assignment from the value of the input variable list to the value of the output variable list. You can use Don't Care values in the input specifications to make design entry easier. An example of truth tables with Don't Care values is shown in *Appendix A*.

The state machine syntax of CUPL has the general form of

```
SEQUENCE  state_var_list
{       PRESENT state_1
                statements;
                .
        PRESENT state_n
                statements;
}
```

where SEQUENCE is the state space, and PRESENT indicates the device's present state and the function the machine should perform based on that state.

The state machine syntax can be divided into six parts:

1. Unconditional Next Statement (*Figure 10*): If the machine is in state_n, then transition to state_m.

```
PRESENT state_n
NEXT state_m;
```

2. Conditional Next Statement (*Figure 11*): If the machine is in state_n and if expr_1 is true, then transition to state_m, else if expr_n is true, transition to state_y, else transition to state_z.

```
PRESENT state_n
        IF expr_1 NEXT state_m;
                .
                .
        IF expr_n NEXT state_y;
        [DEFAULT NEXT state_z;]
```

3. Unconditional Synchronous Output Statement (*Figure 12*): This statement describes a transition from the present state to a next state with a synchronous output accompanying the transition.

```
PRESENT state_n
        NEXT state_n OUT [!]var...OUT[!]var;
```

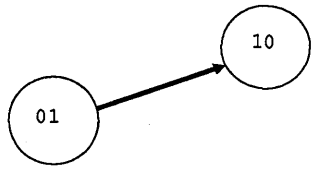
4. Conditional Synchronous Output Statement (*Figure 13*): This statement describes a condition transition with its associated synchronous outputs.

```
PRESENT state_n
        IF expr NEXT state_1 OUT [!]var...OUT [!]var;
                .
                .
        IF expr NEXT state_n OUT [!]var...OUT [!]var;
        [DEFAULT NEXT state_m OUT [!]var;]
```

5. Unconditional Asynchronous Output Statement (*Figure 14*): This statement describes the asynchronous outputs associated with a specific state.

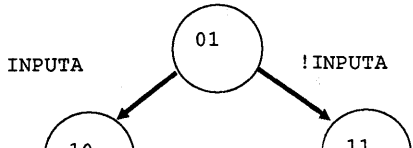
```
PRESENT state_n
        OUT [!]var...OUT [!]var;
```

6. Conditional Asynchronous Output Statement (*Figure 15*): This statement describes a conditional



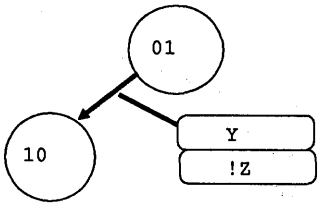

```
PRESENT 'b'01
NEXT 'b'10;
```

Figure 10. Unconditional Next State Diagram




```
PRESENT 'b'01
   IF INPUTA NEXT 'B'10;
   IF !INPUTA NEXT 'B'11;
```

Figure 11. Conditional Next Statement Diagram




```
PRESENT 'b'01
   NEXT  'B'10 OUT Y OUT !Z;
```

Figure 12. Unconditional Synchronous Output Diagram

asynchronous output associated with a specific state and a specific input.

```
PRESENT state_n
      IF expr OUT [!]var...OUT [!]var;
         .
      IF expr OUT [!]var...OUT [!]var;
      [DEFAULT OUT [!]var...OUT [!]var;]
```

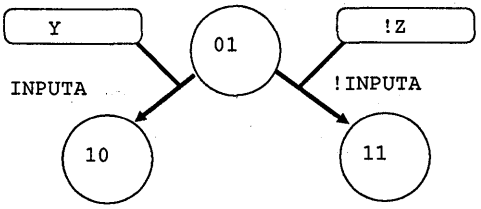
CUPL Examples Using Cypress PLDs

The two examples described here both implement the functions of a Thunderbird's (T-Bird's) tail lights—including the sequentially flashing directional signals. The examples present this function in both the truth table and state machine formats to give you models of these CUPL syntax structures.

Truth Table Example

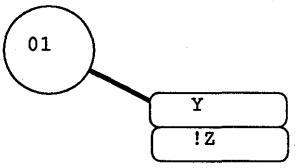
The first example shows how to configure a 22V10 so that it makes two three-segment T-Bird tail lights perform flashing, braking, left turn, right turn, and a combination of these functions.

Consider the truth table example first. This example illustrates both the Truth Table syntax and CUPL's pin declarations. Note that when you use a truth table, you must assign all inputs to a variable name




```
PRESENT 'b'01
   IF INPUTA NEXT 'B'10 OUT Y;
   IF !INPUTA NEXT 'B'11 OUT !Z;
```

Figure 13. Conditional Synchronous Output Diagram




```
PRESENT 'b'01
   OUT Y OUT !Z;
```

Figure 14. Unconditional Asynchronous Output Diagram

using the FIELD statement. Similarly, you must assign all outputs to a variable name. All the inputs and outputs in the body of the truth table must be specified without commas, brackets, or variables. The CUPL 3.2 source code for this example is shown in *Appendix A*.

CUPL's simulator verifies that this truth table operates correctly. When compiling the source code, you must use the -A flag to produce an absolute file for the simulator's use. The simulator also needs an input file, filename.SI, which contains the test vectors. To simulate a design with output going to both the screen and a listing file, filename.SO, type

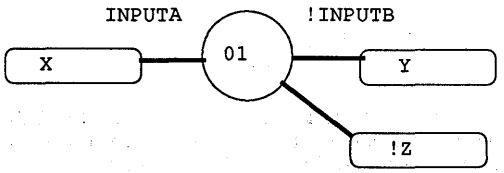
```
CSIM -L -V FILENAME
```

Appendices B and *C* list the input and output simulation files, respectively.

State Machine Examples with the CY7C330

The second example performs the same function as the first, but is coded in CUPL's state machine syntax instead of truth tables. This second example also differs in that it employs Cypress's CY7C330.

The CY7C330 is a high-performance, erasable programmable logic device (EPLD). Through the use of the user-configurable output macrocell, bidirectional I/O capability, input registers, and three separate




```
PRESENT 'b'01
   IF INPUTA OUT X;
   IF !INPUTB OUT Y;
   DEFAULT OUT Z;
```

Figure 15. Conditional Asynchronous Output With Default

clocks, Cypress has tailored the CY7C330's architecture to implement high-performance state machines.

This 28-pin device contains 11 dedicated input macrocells, whose input registers can be controlled by either of two input-register clocks. The 12 I/O macrocells (see *Figure 1* in "Using ABEL to Program the CY7C330") contain an output register that is controlled by a dedicated state-register clock, output-enable control, an exclusive-or product term, an input register, and feedback selection. Each macrocell has between nine and 19 product terms you can use for design implementation. Each pair of macrocells also has a shared input multiplexer, which allows you to bury an output register while still utilizing the I/O pin as a device input. The CY7C330's output enable can be controlled by either pin 14 or a product term. The device also provides four buried registers that can hold state information.

The T-Bird design requires only four flip-flops [Q0..3] to specify all possible tail-light combinations. Note that assignments such as LEFT.D = 'b'001 are not allowed in the main body of the state machine structure. Instead, all outputs must be handled individually with the OUT command. The source code for this example appears in *Appendix D*.

An additional CY7C330 example shows the extended function of this PLD family. The CY7C330, unlike the PAL C 22V10, has more nodes than pins. Thus, the additional nodes must be assigned node numbers so that they can be referenced in the design. *Table 4* lists the node names. Numbers 33 to 44 refer to the output register associated with each pin. IMUX1 refers to the shared input multiplexer between pins 28 and 27.

The second CY7C330 design example is an up/down counter with preloadable limits. The lower limits are loaded the dedicated input registers on the rising edge of the lower-limit clock (LLC), and the upper limits are loaded the I/O macrocells' input registers on the rising edge of the upper-limit clock (ULC). The waveforms for preloading the upper limits and lower limits are shown in *Figure16*.

When preloading is done, the counter counts upward from the last loaded limit until the other limit is reached. The counter then counts in the opposite direction until reaching the other limit. The waveforms for counting between the preloaded limits of 4 and 8 are shown in *Figure17*. If the input register on a specific pin is not being used, you can reference the output register by referring to the I/O pin name. This is shown on pins 20 and 23.

The CY7C330's shared input multiplexer is used to select an additional input into the product term array from either of a macrocell pair's input registers (and thus either macrocell's I/O pin). When referencing this input-signal name in the equations section, you must use the MUX name instead of the actual input signal name.

Another important CY7C330 feature is the XOR product term. During DeMorgan minimization, CUPL uses the XOR term to invert an equation's polarity when an active-Low output signal is specified. Using the XOR term in this example greatly reduces the number of product terms needed to specify the design. By connecting the signal name to the XOR product term, as shown in the equations, the equations represent a T flip-flop.

For example, the equations for CNT2 specify that the flip-flop toggles (a) when preloading the lower limit, for CNT2 not equal to LL2, (b) when preloading the upper limit, for CNT2 not equal to UL2, (c) when counting UP, for CNT0 and CNT1 High, and (d) when counting DOWN, for CNT0 and CNT1 Low. It is important to keep in mind that UP, UEQUAL, and LEQUAL are Low-asserted internal signals.

The part utilization for this design is shown in *Appendix E*. The CUPL design file appears in *Appendix F*.

CY7C332 State Machine Example

The last example uses the Cypress CY7C332. This versatile combinatorial PLD has 25 array inputs: 13 dedicated inputs and 12 I/O inputs. Each input has a macrocell that you can configure as a register, latch, or simple buffer. Outputs have polarity and three-state-

Table 4. Cypress CY7C330 Node Assignments

PIN	NODE
BR0	29
BR1	30
BR2	31
BR3	32
28	33
27	34
26	35
25	36
24	37
23	38
20	39
19	40
18	41
17	42
16	43
15	44
IMUX1	45
IMUX2	46
IMUX3	47
IMUX4	48
IMUX5	49
IMUX6	50

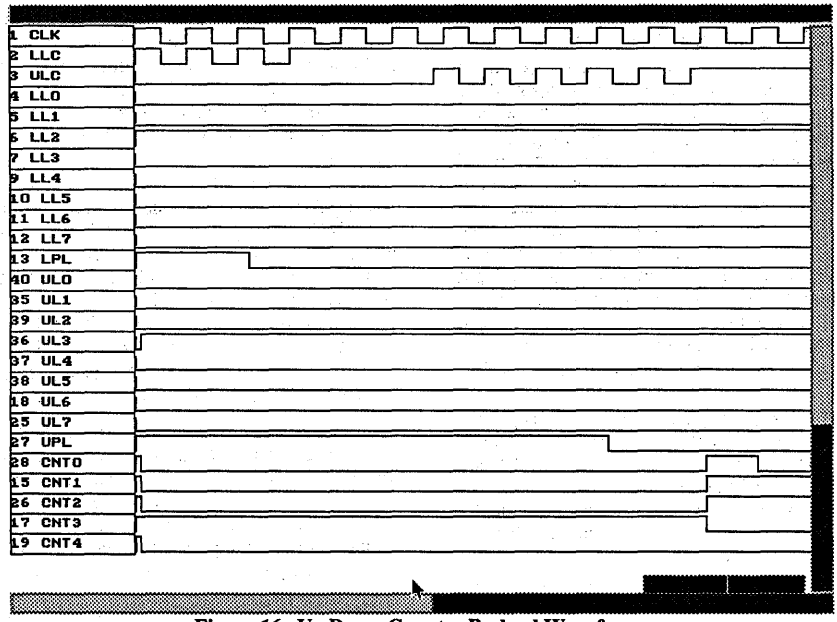


Figure 16. Up Down Counter Preload Waveforms

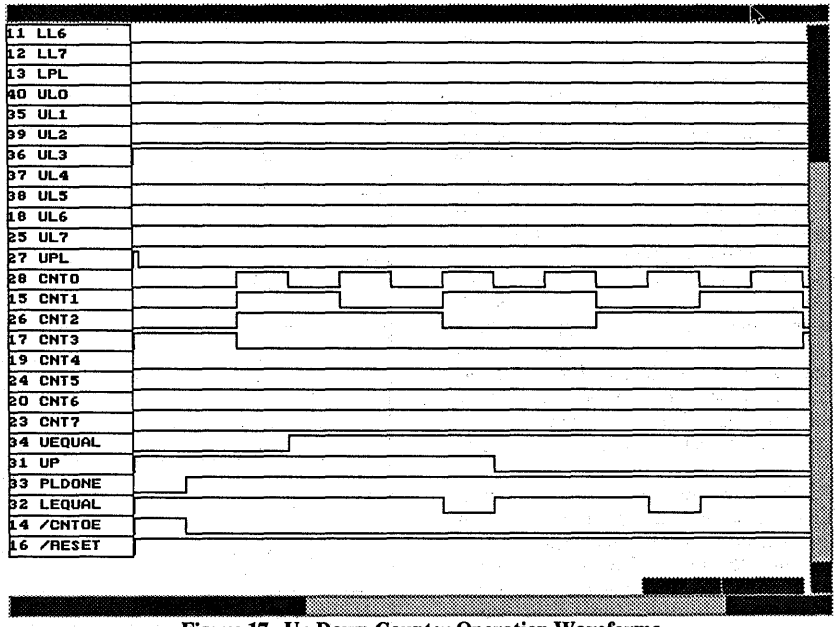


Figure 17. Up Down Counter Operation Waveforms

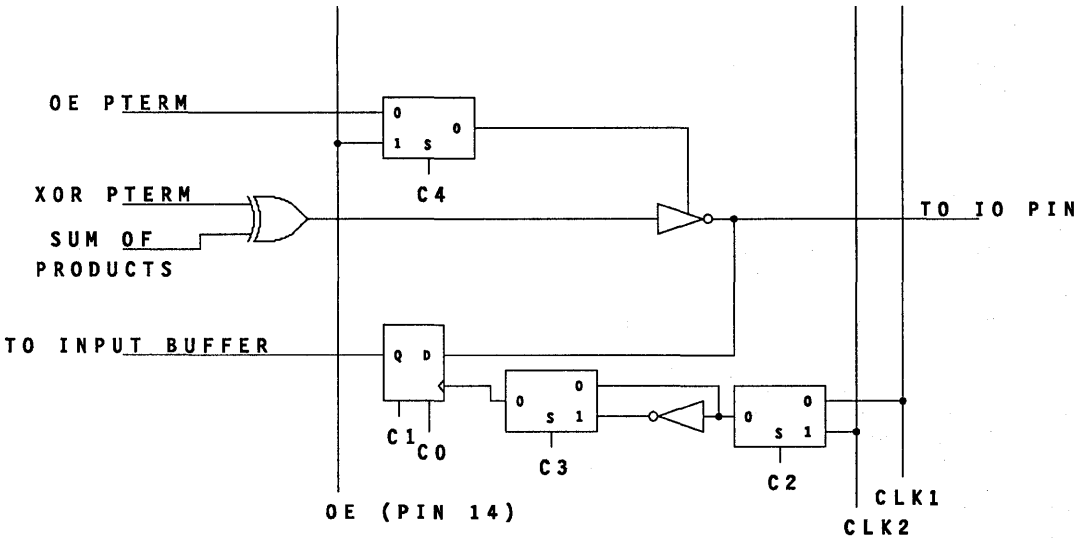


Figure 19. The CY7C332 I/O Macrocell

control product terms. *Figure 18* shows the I/O macrocell. Each macrocell has up to 19 product terms to accommodate complex applications.

In this example, the CY7C332 serves as a simple decoder (*Appendix G*). The device decodes a group of address lines to select one of four "windows" in memory. Inputs are implemented in each of the possible macrocell configurations. When reviewing the example code, it is important to note the use of the .CKMUX, .LEMUX, .DQ, and .LQ extensions.

CUPL Compilation

The input to the CUPL system is an ASCII text file with extension .PLD. The various outputs include a .JED file for programming, a .LST error listing, and a .DOC equations-and-utilization file. You can compile a file either from the DOS command line, or from the CUPL menu structure. The compilation command and its description are shown in *Figure 19*.

References

This *Application Handbook* provides a more detailed explanation of the up/down counter example using the CY7C330 in "Understanding the CY7C330 Synchronous EPLD." More information on the CY7C33x family can be found in Cypress's *BiCMOS/CMOS Data Book*.

```
cupl [-flags] [library] [device]
source

Where -flags is the following set of
compiler options

-j    JEDEC download format
-n    use source filename as JEDEC
      filename
-h    ASCII-HEX download format
-i    HL download format
-a    create absolute file (for
      simulation purposes)
-l    create listing file
-x    create expanded product-terms
      in documentation file
-f    create fuse plot/chip diagram
      in documentation file
-p    create PDEF database
      interchange format file
-b    create Berkeley PLA format file
-d    deactivate unused OR terms
-r    disable product term merging
-g    program security fuse
-u    use specified library for
      compilation
-s    perform logic simulation after
      compile
-e    create expanded macro
      definition file
-x    generates a part usage
      documentation file (filename.DOC)
-w    perform simulation with
      waveform output (PC only)
-m0   no minimization
-m1   quick minimization (default)
-m2   minimization level 2 (Quine-
      McCluskey)
-m3   minimization level 3 (Presto)
-m4   minimization level 4 (Espresso)
-c    create PALASM format file

Library is the library name including
the path that should be used other
than the default library.  This op-
tion is used in conjunction with the -
u flag.

Device is the CUPL mnemonic name of
the device which should be used when
compiling the source file.  This op-
tion over rides the name used in the
CUPL source file.

Source is the user-created ASCII
logic description file (filename.
PLD).
```

Figure 20. CUPL Compilation

Appendix A. T-Bird Truth-Table CUPL Code for PALC22V10

```
Name         TBIRD_TT.PLD;
Partno       PALC22V10;
Revision     01;
Date         04-08-90;
Designer     Joe Designer;
Company      Cypress Semiconductor;
Location     U1;
Assembly     Test;
Device       P22V10;


/*
   This program implements the control signals for the tail lights
   of a Thunderbird.  The lights have three segments for both the
   left and right tail light.  The control signal into the device
   include a Left and Right signal, a Flash signal (Hazard), a brake
   signal, and a ignition signal (IGN).  The outputs of the device
   are the six separate tail light segments.  A Truth Table is used
   to specify the control logic.
*/


PIN     1        =       CLK;             /*  Clock for Device     */
PIN     4        =       LT;              /*  Left turn signal     */
PIN     5        =       RT;              /*  Right turn signal    */
PIN     6        =       BRAKE;           /*  Brake signal         */
PIN     7        =       FLASH;           /*  Hazard flash singal  */
PIN     8        =       IGN;             /*  Ignition input       */

PIN     21       =       RI;              /*  Right inside tail light */
PIN     22       =       RM;              /*  Right middle  */
PIN     23       =       RO;              /*  Right outside */
PIN     16       =       LI;              /*  Left inside   */
PIN     15       =       LM;              /*  Left middle   */
PIN     14       =       LO;              /*  Left outside  */
PIN     [17..20]=       [Q0..3];        /*  State variable holders  */

FIELD  INPUTS       =       [IGN,FLASH,LT,RT,BRAKE,LO,LM,LI,RI,RM,RO];
FIELD  OUTPUTS      =       [LO.D,LM.D,LI.D,RI.D,RM.D,RO.D];

TABLE   INPUTS       =>         OUTPUTS
{

/* Quiescent state */

         'B'11000XXXXXX   =>       'B'0;
         'B'01XX0XXXXXX   =>       'B'0;

/* Flash */

         'B'X0XXX111111   =>       'B'0;
         'B'X0XXX000000   =>       'B'111111;
```

Appendix A. T-Bird Truth-Table CUPL Code (cont)

```
/* Brake */

        'B'X1001XXXXXX  =>      'B'111111;


/* Left turn */

        'B'11100000XXX  =>      'B'001000;
        'B'11100001XXX  =>      'B'011000;
        'B'11100011XXX  =>      'B'111000;
        'B'11100111XXX  =>      'B'0;

/* Right turn */

        'B'11010XXX000  =>      'B'000100;
        'B'11010XXX100  =>      'B'000110;
        'B'11010XXX110  =>      'B'000111;
        'B'11010XXX111  =>      'B'0;

/* Left turn and brake */

        'B'11101000XXX  =>      'B'001111;
        'B'11101001XXX  =>      'B'011111;
        'B'11101011XXX  =>      'B'111111;
        'B'11101111XXX  =>      'B'000111;

/* Right turn and brake */

        'B'11011XXX000  =>      'B'111100;
        'B'11011XXX100  =>      'B'111110;
        'B'11011XXX110  =>      'B'111111;
        'B'11011XXX111  =>      'B'111000;

/* Both turn - light flash in reverse sequence */

        'B'11110000000  =>      'B'111111;
        'B'11110111111  =>      'B'011110;
        'B'11110011110  =>      'B'001100;
        'B'11110001100  =>      'B'0;

/* Illegal condition - All on */

        'B'11111000000  =>      'B'100001;
        'B'11111100001  =>      'B'010010;
        'B'11111010010  =>      'B'001100;
        'B'11111001100  =>      'B'0;
}
```

Appendix B. T-Bird Simulator Input

```
Name            TBIRD_TT.PLD;
Partno          PALC22V10;
Revision        01;
Date            04-08-90;
Designer        Joe Designer;
Company         Cypress Semiconductor;
Location        U1;
Assembly        Test;
Device          P22V10;


ORDER: "INPUTS-   ",  CLK, IGN, FLASH, LT, RT, BRAKE,
       "    OUTPUTS-   ", LO, LM, LI, RI, RM, RO;

VECTORS:

          $MSG " QUIESCENT STATE - 1";
          C 1 1 0 0 0     L L L L L L
          $MSG " QUIESCENT STATE - 1";
          C 0 1 X X 0     L L L L L L

          $MSG "";
          $MSG " FLASH HIGH";
          C 0 0 X X X     H H H H H H
          $MSG " FLASH LOW";
          C 0 0 X X X     L L L L L L
          $MSG " FLASH HIGH";
          C 0 0 X X X     H H H H H H

          $MSG "";
          $MSG " BRAKE";
          C X 1 0 0 1     H H H H H H

          $MSG "";
          $MSG " LEFT TURN OFF";
          C 1 1 1 0 0     L L L L L L
          $MSG " LEFT TURN 1";
          C 1 1 1 0 0     L L H L L L
          $MSG " LEFT TURN 2";
          C 1 1 1 0 0     L H H L L L
          $MSG " LEFT TURN 3";
          C 1 1 1 0 0     H H H L L L
          $MSG " LEFT TURN OFF";
          C 1 1 1 0 0     L L L L L L

          $MSG "";
          $MSG " RIGHT TURN 1";
          C 1 1 0 1 0     L L L H L L
          $MSG " RIGHT TURN 2";
          C 1 1 0 1 0     L L L H H L
          $MSG " RIGHT TURN 3";
          C 1 1 0 1 0     L L L H H H
          $MSG " RIGHT TURN OFF";
          C 1 1 0 1 0     L L L L L L
```

Appendix B. T-Bird Simulator Input (cont)

```
$MSG "";
$MSG " BRAKE AND LEFT TURN 1";
C 1 1 1 0 1     L L H H H H
$MSG " BRAKE AND LEFT TURN 2";
C 1 1 1 0 1     L H H H H H
$MSG " BRAKE AND LEFT TURN 3";
C 1 1 1 0 1     H H H H H H
$MSG " BRAKE AND LEFT TURN OFF";
C 1 1 1 0 1     L L L H H H

$MSG "";
$MSG " BRAKE AND RIGHT TURN OFF";
C 1 1 0 1 1     H H H L L L
$MSG " BRAKE AND RIGHT TURN 1";
C 1 1 0 1 1     H H H H L L
$MSG " BRAKE AND RIGHT TURN 2";
C 1 1 0 1 1     H H H H H L
$MSG " BRAKE AND RIGHT TURN 3";
C 1 1 0 1 1     H H H H H H
```

Appendix C. T-Bird Simulator Output

```
   CSIM: CUPL Simulation Program
Version 3.2a Serial# MD-32A-6295
Copyright (C)  1983,1989 Logical Devices, Inc.
CREATED Mon Apr 09 09:32:04 1990

LISTING FOR SIMULATION FILE: tbird_tt.si

    1: Name              TBIRD_TT.PLD;
    2: Partno            PALC22V10;
    3: Revision          01;
    4: Date              04-08-90;
    5: Designer          Joe Designer;
    6: Company           Cypress Semiconductor;
    7: Location          U1;
    8: Assembly          Test;
    9: Device            P22V10;
   10:
   11:
   12: ORDER: "INPUTS-   ",  CLK, IGN, FLASH, LT, RT, BRAKE,
   13:          "    OUTPUTS-   ", LO, LM, LI, RI, RM, RO;
   14:

================================================================================
                                  Simulation Results
================================================================================
 QUIESCENT STATE - 1
0001: INPUTS-  C11000    OUTPUTS-  LLLLLL
 QUIESCENT STATE - 1
0002: INPUTS-  C01XX0    OUTPUTS-  LLLLLL

 FLASH HIGH
0003: INPUTS-  C00XXX    OUTPUTS-  HHHHHH
 FLASH LOW
0004: INPUTS-  C00XXX    OUTPUTS-  LLLLLL
 FLASH HIGH
0005: INPUTS-  C00XXX    OUTPUTS-  HHHHHH

 BRAKE
0006: INPUTS-  CX1001    OUTPUTS-  HHHHHH

 LEFT TURN OFF
0007: INPUTS-  C11100    OUTPUTS-  LLLLLL
 LEFT TURN 1
0008: INPUTS-  C11100    OUTPUTS-  LLHLLL
 LEFT TURN 2
0009: INPUTS-  C11100    OUTPUTS-  LHHLLL
 LEFT TURN 3
0010: INPUTS-  C11100    OUTPUTS-  HHHLLL
 LEFT TURN OFF
0011: INPUTS-  C11100    OUTPUTS-  LLLLLL
```

Appendix C. T-Bird Simulator Output (cont)

```
 RIGHT TURN 1
0012: INPUTS-   C11010    OUTPUTS-   LLLHLL
 RIGHT TURN 2
0013: INPUTS-   C11010    OUTPUTS-   LLLHHL
 RIGHT TURN 3
0014: INPUTS-   C11010    OUTPUTS-   LLLHHH
 RIGHT TURN OFF
0015: INPUTS-   C11010    OUTPUTS-   LLLLLL

 BRAKE AND LEFT TURN 1
0016: INPUTS-   C11101    OUTPUTS-   LLHHHH
 BRAKE AND LEFT TURN 2
0017: INPUTS-   C11101    OUTPUTS-   LHHHHH
 BRAKE AND LEFT TURN 3
0018: INPUTS-   C11101    OUTPUTS-   HHHHHH
 BRAKE AND LEFT TURN OFF
0019: INPUTS-   C11101    OUTPUTS-   LLLHHH

 BRAKE AND RIGHT TURN OFF
0020: INPUTS-   C11011    OUTPUTS-   HHHLLL
 BRAKE AND RIGHT TURN 1
0021: INPUTS-   C11011    OUTPUTS-   HHHHLL
 BRAKE AND RIGHT TURN 2
0022: INPUTS-   C11011    OUTPUTS-   HHHHHL
 BRAKE AND RIGHT TURN 3
0023: INPUTS-   C11011    OUTPUTS-   HHHHHH
```

Appendix D. T-Bird State-Machine CUPL Code For CY7C330

```
Name            TBIRD_SM.PLD;
Partno          CY7C330;
Revision        01;
Date            04-07-90;
Designer        Joe Designer;
Company         Cypress Semiconductor;
Location        U1;
Assembly        Test;
Device          P7C330;


/*
    This program implements the control signals for the tail lights
    of a Thunderbird.  The lights have three segments for both the
    left and right tail light.  The control signal into the device
    include a Left and Right signal, a Flash signal (Hazard), a brake
    signal, and a ignition signal (IGN).  The outputs of the device
    are the six separate tail light segments.  A State Machine is used
    to specify the control logic.
*/


PIN     1       =       CLK;                /*  Clock for Device     */
PIN     2       =       INCLK;              /*  Clock for Inputs     */
PIN     4       =       LT;                 /*  Left turn signal     */
PIN     5       =       RT;                 /*  Right turn signal    */
PIN     6       =       BRAKE;              /*  Brake signal         */
PIN     7       =       FLASH;              /*  Hazard flash singal  */
PIN     9       =       IGN;                /*  Ignition input       */

PIN     28      =       RI;                 /*  Right inside tail light */
PIN     27      =       RM;                 /*  Right middle   */
PIN     26      =       RO;                 /*  Right outside  */
PIN     25      =       LI;                 /*  Left inside    */
PIN     24      =       LM;                 /*  Left middle    */
PIN     23      =       LO;                 /*  Left outside   */
PINNODE [29..32]=    [Q0..3];           /*  State variable holders   */

FIELD OUTPUTS    =         [LO,LM,LI,RI,RM,RO];
OUTPUTS.OE       =         'B'1;
OUTPUTS.SR       =         'B'0;
OUTPUTS.SP       =         'B'0;

/* Using the $DEFINE statement to assign variable name to state values */

$DEFINE S0       'B'0000
$DEFINE S1       'B'0001
$DEFINE S2       'B'0010
$DEFINE S3       'B'0011
$DEFINE S4       'B'0100
$DEFINE S5       'B'0101
$DEFINE S6       'B'0110
$DEFINE S7       'B'0111
$DEFINE S8       'B'1000
```

Appendix D. T-Bird State-Machine CUPL Code (cont)

```
$DEFINE S9        'B'1001
$DEFINE S10       'B'1010
$DEFINE S11       'B'1011
$DEFINE S12       'B'1100
$DEFINE S13       'B'1101
$DEFINE S14       'B'1110
$DEFINE S15       'B'1111

/* The state machine construct where Q0..3 are the state variables */

SEQUENCE [Q0..3]
{

/* Initial state all lights off */

PRESENT S0
          OUT !LO.D OUT !LM.D OUT !LI.D OUT !RI.D OUT !RM.D OUT !RO.D;
          IF (FLASH) NEXT S15;
          IF (BRAKE & !(LT # RT)) NEXT S15;
          IF (IGN & LT & !BRAKE) NEXT S1;
          IF (IGN & RT & !BRAKE) NEXT S4;
          IF (IGN & LT & BRAKE) NEXT S7;
          IF (IGN & RT & BRAKE) NEXT S11;
          DEFAULT NEXT S0;

/* Left turn */

PRESENT S1
          OUT !LO.D OUT !LM.D OUT LI.D OUT !RI.D OUT !RM.D OUT !RO.D;
          IF (IGN & LT) NEXT S2;
          DEFAULT NEXT S0;

PRESENT S2
          OUT !LO.D OUT LM.D OUT LI.D OUT !RI.D OUT !RM.D OUT !RO.D;
          IF (IGN & LT) NEXT S3;
          DEFAULT NEXT S0;

PRESENT S3
          OUT LO.D OUT LM.D OUT LI.D  OUT !RI.D  OUT !RM.D  OUT !RO.D ;
          NEXT S0;

/* Right Turn */

PRESENT S4
          OUT !LO.D  OUT !LM.D  OUT !LI.D  OUT RI.D  OUT !RM.D OUT !RO.D;
          IF (IGN & RT) NEXT S5;
          DEFAULT NEXT S0;

PRESENT S5
          OUT !LO.D OUT !LM.D OUT !LI.D OUT RI.D OUT RM.D OUT !RO.D;
          IF (IGN & RT) NEXT S6;
          DEFAULT NEXT S0;
```

Appendix D. T-Bird State-Machine Code (cont)

```
PRESENT S6
        OUT !LO.D OUT !LM.D OUT !LI.D OUT RI.D OUT RM.D OUT RO.D;
        NEXT S0;

/* Brake and Left Turn */

PRESENT S7
        OUT !LO.D OUT !LM.D OUT LI.D OUT RI.D OUT RM.D OUT RO.D;
        IF (IGN & LT) NEXT S8;
        DEFAULT NEXT S0;

PRESENT S8
        OUT !LO.D OUT LM.D OUT LI.D OUT RI.D OUT RM.D OUT RO.D;
        IF (IGN & LT) NEXT S9;
        DEFAULT NEXT S0;

PRESENT S9
        OUT LO.D OUT LM.D OUT LI.D OUT RI.D OUT RM.D OUT RO.D;
        IF (IGN & LT) NEXT S10;
        DEFAULT NEXT S0;

PRESENT S10
        OUT !LO.D OUT !LM.D OUT !LI.D OUT RI.D OUT RM.D OUT RO.D;
        IF (IGN & LT) NEXT S7;
        DEFAULT NEXT S0;

/* Brake and Right Turn */

PRESENT S11
        OUT LO.D OUT LM.D OUT LI.D OUT RI.D OUT !RM.D OUT !RO.D;
        IF (IGN & RT) NEXT S12;
        DEFAULT NEXT S0;

PRESENT S12
        OUT LO.D OUT LM.D OUT LI.D OUT RI.D OUT RM.D OUT !RO.D;
        IF (IGN & RT) NEXT S13;
        DEFAULT NEXT S0;

PRESENT S13
        OUT LO.D OUT LM.D OUT LI.D OUT RI.D OUT RM.D OUT RO.D;
        IF (IGN & RT) NEXT S14;
        DEFAULT NEXT S0;

PRESENT S14
        OUT LO.D OUT LM.D OUT LI.D OUT !RI.D OUT !RM.D OUT !RO.D;
        IF (IGN & RT) NEXT S11;
        DEFAULT NEXT S11;

/* Brake and/or flash tail lights on */

PRESENT S15
        OUT LO.D OUT LM.D OUT LI.D OUT RI.D OUT RM.D OUT RO.D;
        IF (BRAKE & !(RT # LT)) NEXT S15;
        DEFAULT NEXT S0;

}
```

Appendix E. Up/Down Counter Part Utilization

CY7C330 Resources Planning Sheet
Project : Up/Down Counter with Limits

Pin	Input Register Function	Input Register Clock	Register Function	Output Enable	# of PTerms
1	State Clk				
2	Clk 1				
3	Clk 2				
4	LL0	1			
5	LL1	1			
6	LL2	1			
7	LL3	1			
8	VSS				
9	LL4	1			
10	LL5	1			
11	LL6	1			
12	LL7	1			
13	PRELOAD LOW	1			
14	COUNTER OE	-			
15	UL1	2	CNT1	Pin 14	9
16	Reset	1	-	Z	19
17	UL3	2	CNT3	Pin 14	11
18	UL6	2	-	Z	17
19	UL4	2	CNT4	Pin 14	13
20	-	-	CNT6	Pin 14	15
21	VSS				
22	VCC				
23	-	-	CNT7	Pin 14	15
24	UL5	2	CNT5	Pin 14	13
25	UL7	2	-	Z	17
26	UL2	2	CNT2	Pin 14	11
27	PRELOAD HIGH	2	-	Z	19
28	UL0	2	CNT0	Pin 14	9
H1	None	-	Up Equals	None	19
H2	None	-	L/H Prel'Done	None	11
H3	None	-	Down Equals	None	17
H4	None	-	Up Count	None	13

Notes :Input Register Clock #1 is pin 2
#2 is pin 3
See the Application Note for the meaning of the pin names.
Output Enable = 14 means the asynchronous pin 14 direct enable.
Z means the pin is never active

Appendix F. Up/Down Counter CUPL Code for the CY7C330

```
Name            COUNTER.PLD;
Partno          PALC22V10;
Revision        01;
Date            02-25-90;
Designer        Joe Designer;
Company         Cypress Semiconductor;
Location        U1;
Assembly        COUNTER;
Device          P7C330;

/*
   This design is an up/down counter with prelaodable limits.  The Lower limits
   are loaded into the dedicated input registers on the rising edge of LLC and
   the upper limits are loaded into the input registers found in the I/O macrocells
   on the rising edge of ULC.  The counter begins counting, when preloading is done
   upwards until the upper limit is reached, and then, begins counting downward.
   This design, because the equations are already minimized and in sum of products
   form, should be compiled with the -M0 flag (no minimization).
*/

PIN     1       =       CLK;            /* Clock used for counting */
PIN     2       =       LLC;            /* Clock for preloading lower limit */
PIN     3       =       ULC;            /* Clock for preloading upper limit */

PIN     [4..7] =       [LL0..3];      /* Lower limit hold registers */
PIN     [9..12]=       [LL4..7];
PIN     13      =       LPL;            /* Lower limit preload indications */


/*
   Counter output registers.  Pin assignments are based on the number of
   product terms are available on that pin.
*/

PIN     28      =       CNT0;           /* Also used for Upper limit loading */
PIN     15      =       CNT1;           /* Also used for Upper limit loading */
PIN     26      =       CNT2;           /* Also used for Upper limit loading */
PIN     17      =       CNT3;           /* Also used for Upper limit loading */
PIN     19      =       CNT4;           /* Also used for Upper limit loading */
PIN     24      =       CNT5;           /* Also used for Upper limit loading */
PIN     20      =       CNT6;
PIN     23      =       CNT7;
PIN     18      =       UL6;           /* Used for Upper limit loading */
PIN     25      =       UL7;           /* Used for Upper limit loading */
PIN     27      =       UPL;


PINNODE 29    =       UEQUAL;         /* Upper limit has been reached */
PINNODE 30    =       PLDONE;         /* Preloading has finished */
PINNODE 31    =       LEQUAL;         /* Lower limit has been reached */
PINNODE 32    =       UP;              /* Count direction */

PIN     16     =      !RESET;         /* Reset signal clears all registers */
PIN     14     =      !CNTOE;         /* I/O pin OE used for loading upper limit */
```

Appendix F. Up/Down Counter Code for CUPL (cont)

```
PINNODE 45      =       UL0;            /* Shared input MUX definition */
PINNODE 46      =       UL2;            /* Shared input MUX definition */
PINNODE 47      =       UL5;            /* Shared input MUX definition */
PINNODE 48      =       UL4;            /* Shared input MUX definition */
PINNODE 49      =       UL3;            /* Shared input MUX definition */
PINNODE 50      =       UL1;            /* Shared input MUX definition */

UL0.IMUX        =      CNT0.IOD;        /* These definitions are used to */
UL2.IMUX        =      CNT2.IOD;        /* indicate which pin will be fed */
UL5.IMUX        =      CNT5.IOD;        /* through the share feedback MUX.*/
UL4.IMUX        =      CNT4.IOD;
UL3.IMUX        =      CNT3.IOD;
UL1.IMUX        =      CNT1.IOD;

UL0             =       CNT0.IOD;       /* These definitions are used to */
UL2             =       CNT2.IOD;       /* Simulate the design properly  */
UL5             =       CNT5.IOD;
UL4             =       CNT4.IOD;
UL3             =       CNT3.IOD;
UL1             =       CNT1.IOD;

UPL.CKMUX       =      ULC;
LPL.CKMUX       =      LLC;
RESET.CKMUX     =      LLC;
[CNT0..5].CKMUX   =   ULC;            /* Pin 3 will be used for upper preload */
[UL6..7].CKMUX    =   ULC;            /* Pin 3 will be used for upper preload */
[LL0..7].CKMUX    =   LLC;            /* Pin 2 will be used for lower preload */

[CNT0..7].SR      =   RESET.DQ;       /* Count register will be reset by pin 16 */

[CNT0..7].OEMUX   =   CNTOE;          /* OE will be controlled by pin 14 */

/*
   Count equations.  Note how the use of the XOR terms significantly reduces the
   number of product terms that are needed.  This allows this complex design to fit
   fit into the device.
*/

CNT0.D =   CNT0
      $       PLDONE
      #       !LL0.DQ   & LPL.DQ   & CNT0
      #       !CNT0.IOD& UL0 & UPL.DQ
      #       LL0.DQ   & LPL.DQ   & !CNT0
      #       CNT0.IOD& !UL0 & UPL.DQ ;


CNT1.D =   CNT1
      $       !LL1.DQ & LPL.DQ & !PLDONE & CNT1
      #       LL1.DQ & LPL.DQ & !PLDONE & !CNT1
      #       UPL.DQ & !PLDONE & !UL1 & CNT1
      #       UPL.DQ & !PLDONE & UL1 & !CNT1
      #       CNT0.IOD& PLDONE & !UP
      #       !CNT0.IOD& PLDONE & UP ;
```

Appendix F. Up/Down Counter Code for CUPL (cont)

```
CNT2.D =   CNT2
       $      !LL2.DQ   & LPL.DQ   & CNT2 & !PLDONE
       #      LL2.DQ   & LPL.DQ   & !CNT2 & !PLDONE
       #      UPL.DQ & CNT2 & !UL2 & !PLDONE
       #      UPL.DQ & !CNT2 & UL2 & !PLDONE
       #      CNT0.IOD& PLDONE & !UP & CNT1
       #      !CNT0.IOD& PLDONE & UP & !CNT1;

CNT3.D =   CNT3
       $      !LL3.DQ & LPL.DQ & !PLDONE & CNT3
       #      LL3.DQ & LPL.DQ & !PLDONE & !CNT3
       #      UPL.DQ & !PLDONE & !UL3 & CNT3
       #      UPL.DQ & !PLDONE & UL3 & !CNT3
       #      CNT0.IOD& CNT2 & PLDONE & !UP & CNT1
       #      !CNT0.IOD& !CNT2 & PLDONE & UP & !CNT1;

CNT4.D =   CNT4
       $      !LL4.DQ & LPL.DQ & !PLDONE & CNT4
       #      LL4.DQ & LPL.DQ & !PLDONE & !CNT4
       #      UPL.DQ & !PLDONE & !UL4 & CNT4
       #      UPL.DQ & !PLDONE & UL4 & !CNT4
       #      CNT0.IOD& CNT2 & PLDONE & !UP & CNT3 & CNT1
       #      !CNT0.IOD& !CNT2 & PLDONE & UP & !CNT3 & !CNT1;

CNT5.D =   CNT5
       $      !LL5.DQ   & LPL.DQ   & CNT5 & !PLDONE
       #      LL5.DQ   & LPL.DQ   & !CNT5 & !PLDONE
       #      UPL.DQ & CNT5 & !UL5 & !PLDONE
       #      UPL.DQ & !CNT5 & UL5 & !PLDONE
       #      CNT0.IOD& CNT2 & PLDONE & CNT4 & !UP & CNT3 & CNT1
       #      !CNT0.IOD& !CNT2 & PLDONE & !CNT4 & UP & !CNT3 & !CNT1;

CNT6.D =   CNT6
       $      !LL6.DQ   & LPL.DQ   & !PLDONE & CNT6
       #      LL6.DQ   & LPL.DQ   & !PLDONE & !CNT6
       #      UPL.DQ & !PLDONE & CNT6 & !UL6.DQ
       #      UPL.DQ & !PLDONE & !CNT6 & UL6.DQ
       #      CNT0.IOD& CNT2 & CNT5 & PLDONE & CNT4 & !UP & CNT3 & CNT1
       #      !CNT0.IOD& !CNT2 & !CNT5 & PLDONE & !CNT4 & UP & !CNT3 & !CNT1;

CNT7.D =   CNT7
       $      !LL7.DQ   & LPL.DQ   & CNT7 & !PLDONE
       #      LL7.DQ   & LPL.DQ   & !CNT7 & !PLDONE
       #      UPL.DQ & !UL7.DQ & CNT7 & !PLDONE
       #      UPL.DQ & UL7.DQ & !CNT7 & !PLDONE
       #      CNT0.IOD& CNT2 & CNT5 & PLDONE & CNT6 & CNT4 & !UP & CNT3 & CNT1
       #      !CNT0.IOD& !CNT2 & !CNT5 & PLDONE & !CNT6 & !CNT4 & UP & !CNT3 &
!CNT1;
```

Appendix F. Up/Down Counter Code for CUPL (cont)

```
/* Direction of count */


UP.D   =    UP
      $       !UEQUAL & !UP & PLDONE
      #       !LEQUAL & UP & PLDONE
      #       UPL.DQ & !PLDONE & !UP
      #       LPL.DQ & !PLDONE & UP;

/* Has the lower limit been reached */

LEQUAL.D =  LL6.DQ & !CNT6
           # !LL7.DQ &  CNT7
           #  LL7.DQ & !CNT7
           #  LL3.DQ & !CNT3
           # !LL5.DQ &  CNT5
           #  LL5.DQ & !CNT5
           # !LL1.DQ &  CNT1
           #  LL0.DQ & !CNT0
           # !LL2.DQ &  CNT2
           # !LL4.DQ &  CNT4
           #  LL4.DQ & !CNT4
           # !LL0.DQ &  CNT0
           #  LL1.DQ & !CNT1
           # !LL6.DQ &  CNT6
           # !LL3.DQ &  CNT3
           #  LL2.DQ & !CNT2;

/* Has preloading finished */

PLDONE.D = !LPL.DQ  & !UPL.DQ ;

/* Has the upper limit been reached */

UEQUAL.D = !CNT6    &  UL6.DQ
           # !UL7.DQ & CNT7
           # UL7.DQ  & !CNT7
           # UL3      & !CNT3
           # CNT5     & !UL5
           # !CNT5    & UL5
           # !UL1     & CNT1
           # !CNT0.IOD  & UL0
           # CNT2     & !UL2
           # !UL4     & CNT4
           # UL4      & !CNT4
           # CNT0.IOD   & !UL0
           #  UL1     & !CNT1
           # CNT6     & !UL6.DQ
           # !UL3     & CNT3
           # !CNT2    & UL2;
```

Appendix G. Decoder CUPL Code

```
Name            DCOLUMNS332.PLD;
Partno          P7C332;
Revision        01;
Date            10-09-90;
Designer        Joe Designer;
Company         Cypress Semiconductor;
Location        332 DCOLUMNSR;
Assembly;
Device          P7C332;

/*
    This design is a simple decoder.  Agroup of address lines are decoded
    to select one of 4 "windows" in memory.  The inputs have been configured
    in each of their possible configurations.  Although this application would
    not be used in a real design, this example shows how to configure the
    input registers in each of their possible modes.
*/


PIN       1             =     CLK;              /* Clock pin */
PIN       2             =     LTCHEN;           /* Latch enable pin */
PIN       [3..7]        =     AD16..20];        /*Address lines */
PIN       [9..13]       =     [AD21..25];
PIN       14            =     !COE;             /* Output enable */
PIN       [15..20]      =     [AD26..31];
PIN       [23..26]      =     ![WINDOW0..3];    /* Window selection output */
PIN       27            =     !NOTME;           /* No window selected */
PIN       28            =     !DCDEN;           /* Decode enable */

[!WINDOW0..3].OEMUX  =     COE;              /* OE controlled by pin 14 */
NOTME.OE             =     'b'1;             /* Notme always on bus */
[AD16..19].CKMUX     =     CLK;              /* Clocked on rising edge */
[AD20..23].CKMUX     =     !CLK;             /* Clocked on falling edge */
[AD24..27].LEMUX     =     LTCHEN;           /* Latched when high */
[AD28..31].LEMUX     =     !LTCHEN;          /* Latched when low */

/* Window selection Equations */

WINDOW0          =         DCDEN.DQ & AD31.LQ & AD30.LQ & AD29.LQ & AD28.LQ &
                           AD27.LQ & AD26.LQ & AD25.LQ & AD24.LQ &
                           AD23.DQ & AD22.DQ & AD21.DQ & AD20.DQ &
                           !AD19.DQ & !AD18.DQ & !AD17.DQ & !AD16.DQ;

WINDOW1          =         DCDEN.DQ & AD31.LQ & AD30.LQ & AD29.LQ & AD28.LQ &
                           AD27.LQ & AD26.LQ & AD25.LQ & AD24.LQ &
                           AD23.DQ & AD22.DQ & AD21.DQ & AD20.DQ &
                           !AD19.DQ & AD18.DQ & !AD17.DQ & !AD16.DQ;

WINDOW2          =         DCDEN.DQ & AD31.LQ & AD30.LQ & AD29.LQ & AD28.LQ &
                           AD27.LQ & AD26.LQ & AD25.LQ & AD24.LQ &
                           AD23.DQ & AD22.DQ & AD21.DQ & AD20.DQ &
                           AD19.DQ & !AD18.DQ & !AD17.DQ & !AD16.DQ;
```

Appendix G. Decoder CUPL Code (cont)

```
WINDOW3         =       DCDEN.DQ & AD31.LQ & AD30.LQ & AD29.LQ & AD28.LQ &
                                AD27.LQ & AD26.LQ & AD25.LQ & AD24.LQ &
                                AD23.DQ & AD22.DQ & AD21.DQ & AD20.DQ &
                                AD19.DQ & AD18.DQ & !AD17.DQ & !AD16.DQ;

NOTME           =       'B'1
                  $      DCDEN.DQ & AD16.DQ
                  #      DCDEN.DQ & !AD16.DQ & AD17.DQ
                  #      DCDEN.DQ & !AD31.LQ
                  #      DCDEN.DQ & !AD30.LQ
                  #      DCDEN.DQ & !AD29.LQ
                  #      DCDEN.DQ & !AD28.LQ
                  #      DCDEN.DQ & !AD27.LQ
                  #      DCDEN.DQ & !AD26.LQ
                  #      DCDEN.DQ & !AD25.LQ
                  #      DCDEN.DQ & !AD24.LQ
                  #      DCDEN.DQ & !AD23.DQ
                  #      DCDEN.DQ & !AD22.DQ
                  #      DCDEN.DQ & !AD21.DQ
                  #      DCDEN.DQ & !AD20.DQ;
```

Using ABEL to Program the Cypress 22V10

Introduction

This application note presents a compilation of examples using the popular PALC22V10 programmable logic device. The examples demonstrate the 22V10's advanced features and some of the high-level logic description techniques of the ABEL programming langauge.

Each of the first seven examples illustrates a specific 22V10 feature and lists the ABEL programming language statements necessary to implement the feature. The ABEL files also contain test vectors that exercise the feature. The remaining examples describe complete 22V10 designs that combine many of the individual features. All the examples have been tested, and you can obtain the code for them on floppy disk from Cypress Semiconductor. The design examples provided are:

- Asynchronous reset/synchronous preset from single inputs
- Asynchronous reset/synchronous preset from product terms
- Asynchronous reset/synchronous preset used to load predetermined non-zero values, employing is-type statements
- Output-enable control from a single input
- Output-enable control from product terms
- Using 16 product terms—an 8-bit identity comparitor
- Using feedback to realize more than 16 product terms in a 9-bit single-output identity comparitor
- Bidirectional I/O—bus interface with answer-back
- 10-bit address generator/multiplexer
- Three state machines in one 22V10

You can use these examples as a design reference. They are excellent tools for designers new to programmable logic as well as for veteran PLD users. Add the files to your ABEL source-file library, and include any part of the files in your own designs. You can use the files as a template by editing them using any text editor in the non-document mode. Conversion to the CUPL or PLD ToolKit ToolKit programming language is easily accomplished due to these languages' syntactical similarity. For conversion to other languages, consult your user's guide.

Notes on the ABEL Programming Language

Before examining the application examples, consider an introduction to the structure and syntax of the ABEL programming language. A rudimentary understanding of the ABEL language is necessary to fully appreciate the example files included here.

An ABEL source file provides the information necessary to describe a PLD design's logical operation. You can see these files' keywords and structure in any of the examples. The ABEL language processor processes source files to generate a JEDEC programming file and design documentation. The language processor also uses test vectors, which you generate as part of the source file, to test the design's function.

ABEL Design Entry Methods

The ABEL programming language offers three methods for defining the logical operation of a given design. These methods are:

- Boolean Equation
- Truth table
- State diagram

A source file can include any or all of these design entry methods. The following sections describe the Boolean equation, truth table, and state diagram entry methods as well as the operators and notation conventions used in the source files.

ABEL Operators and Notation Conventions

In addition to the standard AND and OR logical operators, ABEL supports several high-level logic definitions. ABEL interprets "+" and "*" signs—which in standard Boolean notation stand for OR and AND operations, respectively—to indicate arithmetic addition and multiplication. This convention greatly simplifies the design of counters and ALU logic. *Table 1* shows the logical operators ABEL supports. The labels A, B, and C in the examples can be either individual pins or a set of pins, as defined in the source file.

Note that you can use these operators with operands of more than one bit on a bit-by-bit basis. For example, logically ORing hexidecimal values of 8 and 2 yields hexidecimal value A:

```
^h08 # ^h02 = ^h0A
```

Specifying Alternate Number Bases

The ^h symbols in the example above instruct the language processor to interpret the value following the symbol as base 16 (hex). The default number base in ABEL is decimal, but you can change the base for individual expressions with ^b for binary, ^o for octal, ^d for decimal, or ^h for hexidecimal. You can also use the "@ radix" command to change the default number base to binary, octal, decimal, or hexidecimal for all subsequent statements in a source document. All the source files in this application note include the command "@ radix 16" to set the number base to hexidecimal.

Table 1. ABEL Logical Operators

Operator	Definition	Example
!	NOT: ones compliment	C = !A;
&	AND	C = A & B;
#	OR	C = A # B;
$	XOR: exclusive OR	C = A $ B;
!$	XNOR: exclusive NOR	C = A !$ B;

Arithmetic Operators

ABEL provides arithmetic operators to allow for easy implementation of math and shifting functions. *Table 2* lists the arithmetic operators supported by ABEL.

Shifting operations are unsigned, and zeros are shifted into the side of the expression opposite the direction of the shift. Also note that ABEL interprets the symbol "/" as an unsigned division operation. Other programmable logic languages use this symbol to indicate inversion. The symbol "%" gives the remainder of the division operation performed by "/".

Relational Operators

Relational operators perform various comparisons of elements in an expression and yield a Boolean true or false based on the result of the comparison. These operators greatly simplify the description of magnitude comparisons and reduce an identity comparison to a single statement. All relational operations are unsigned; take care when you represent negative numbers in twos compliment. *Table 3* lists the relational operators.

Relational operators are frequently used where ranges of values cause a given output. For example, if you want to decode an active-low chip-select line (CS1) for any address from ^h2000 to ^h2FFF, you can write the logic for this output in a single line:

```
!CS1 = (ADD >= ^h2000) & (ADD<=^h2FFF);
```

Assignment Operators

Note that all example operations shown so far are for purely combinatorial outputs. The structure for combinatorial equations is:

```
OUTPUT(s) = Expression(s) and/or Condition(s);
```

Table 2. ABEL Arithmetic Operators

Operator	Definition	Example
-	2s complement	C = -A;
-	subtraction	C = A - B;
+	addition	C = A + B;
*	multiplication	C = A * B;
/	integer division	C = A / B;
%	remainder	C = A % B;
<	shift left	C = A < 2; (shift left 2 bits)

Table 3. ABEL Relational Operators

Operator	Definition	Example
= =	equal	C=(A==B);
!=	not equal	C=(A!=B);
<	less than	C=(A<B);
>	greater than	C=(A>B);
< =	less than or equal	C=(A<=B);

The assignment operator is the "=" sign, meaning that OUTPUT(s) combinatorially follow the evaluation of the expressions and conditions. If an output or set of outputs is registered (changing synchronously with the clock's rising edge), use the assignment operator ":=". The structure of a registered equation, shown below, is essentially the same as a combinatorial equation but with this assignment operator:

OUTPUT(s) := Expression(s) and/or Condition(s);

Operator Priority

Operators in an expression are evaluated using a priority hierarchy. If two or more operators with equal priority appear in a single expression, they are evaluated in the order listed, from left to right within the expression. *Table 4* lists the priority of all operators.

You can use parentheses as in normal mathematics to alter the order of evaluation. ABEL performs the operation in the innermost parentheses first.

Special Constants

ABEL supports several special constants that ease the writing of equations and test vectors. *Table 5* lists these special constants and their functions.

To use several of these constants in an abbreviated form and enable the symbols H and L to represent binary Ones and Zeros, place the following statement in the labels section of the source document, as in the examples in this application note:

H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

Logic Reduction Levels

At the beginning of every source file in this brief appears the statement

flag '-r4'

Table 4. ABEL Operator Priority

HighestPriority

- Twos compliment, not subtraction

! Inversion, ones compliment

Second Highest Priority

< Shift left

> Shift right

* Multiply

/ Unsigned division

% Remainder from division

Third Highest Priority

+ Add

- Subtract

OR

$ XOR

!$ XNOR

Lowest Priority

All Relational Operators (==, !=, <, >, <=, >=)

This statement signals the language processor to use logic reduction level 4. In cases where you need propagation delays of a specific length, use the statement

flag '-r0'

Table 5. ABEL Special Constants

Special Constant	Definition
.C.	Clock: causes a low-high-low transition at a selected input for testing.
.F.	Floating input or output
.K.	Same as .C., but high-low-high
.P.	Register preload
.X.	Don't care condition
.Z.	Tests input or output for high impedance

which tells the language processor to use no reduction. ABEL provides four logic reduction levels, as listed in *Table 6*.

ABEL Design Entry: Boolean Equations

Boolean equations are the most common method of design entry. To use them, you give a name to each pin required for the application. If a design requires the special functions available in many devices (i.e., reset and preset), you also identify and name the nodes that control these functions. (The 22V10 has two such nodes: asynchronous reset at node 25 and synchronous preset at node 26.) Groups of pins and/or frequently used constants can also be given labels to facilitate writing equations.

Following the keyword EQUATIONS in the source file, you describe the required logic with Boolean equations that use the pin, node, and/or label names.

If an output has an output-enable term associated with it, you can write an equation for that term by using the pin name with the extension ".OE" followed by the equation for the term. An example of this is:

```
OUT1.OE = !RD & (INPUTS == 0);
```

Table 6. ABEL Logic Reduction Levels

Level	Statement	Description
0	flag'-r0'	No reduction. All equations must be in sum-of-products form.
1	flag'-r1'	Equations are expanded to sum-of-products form and reduced with standard Boolean algebra. This is the default.
2	flag'-r2'	Includes level 1 reduction plus the PRESTO algorithm. This process is iterative, so processing time is increased significantly.
3	flag'-r3'	The PRESTO algorithm is performed on a pin-by-pin basis. This is faster than standard PRESTO reduction.
4	flag'-r4'	This reduction level uses the ESPRESSO reduction algorithm.

This statement enables OUT1 if pin RD is Low and the group of pins (can be any number of pins) labeled INPUTS are all Low. If these conditions are not met, the output remains three-stated.

The 22V10 has a separate combinatorial output-enable product term for each I/O pin. The output enable is therefore easily controlled by either a single selectable pin or from a product term. To make an output enable synchronous or to expand the number of product terms available, you can dedicate an I/O macrocell to realize the appropriate logic; the macrocell's output feeds back to control the output-enable product term. This method causes additional propagation delay, however, due to the extra pass through the AND/OR array.

The use of the enable equations is purely optional; in the absence of these equations, the ABEL language processor automatically enables any I/O pin defined in the Boolean equations as an output and disables any I/O specified as an input. The outputs appear on the left side of the equations.

This application note outlines the operators and syntax of all Boolean equations. You can find additional information in the *ABEL Language Reference* and *User's Guide* supplied with the ABEL software.

ABEL Design Entry: Truth Tables

A truth table is a list of input combinations and the resulting outputs. Normally, the inputs are listed in ascending binary order from the minimum value to the maximum value. This format takes all possible input situations into account and prevents any undefined input combinations from producing undesirable outputs.

The keyword TRUTH_TABLE marks the beginning of the table within the source file. Immediately following the keyword, you list the input(s) and output(s) labels in parentheses with an arrow (a minus sign and a greater than sign "->") between the inputs and outputs. If you specify more than one input or output, you must enclose the set in square brackets "[ ]".

Figure 1 shows the statements required to implement a 3-to-8-line decoder. Note the use of the set identifier Q7..Q0. This can be written out as Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0.

The main advantage of the truth table entry method lies in writing test vectors. You can block-copy the entire truth table to the source file's test-vector section.

Any design specified by a truth table can also be entered as Boolean equations. For example, the output

Q6 in the above example could be represented by the Boolean equation:

Q6 = I2 & I1 & !I0;

ABEL Design Entry: State Diagrams

One of the most powerful features of the ABEL programming language is its ability to compile state diagrams directly. By allowing direct state-diagram entry, ABEL frees you from the tedious task of generating Boolean equations with the expressions and conditions that cause each possible transition for each individual state register.

You can implement several state machines in a single device, and you might have a set of outputs for each state machine. The state diagram for each set of outputs begins with the keyword STATE_DIAGRAM, followed by the pin names or labels that make up the state outputs. You then list each state, followed by any operations to be performed while in that state and at least one transition statement. A transition statement can be in any of three forms:

- GOTO, for unconditional transitions to the next state
- IF..THEN..ELSE , for two-way branching
- CASE..ENDCASE , for N-way branching

You can chain IF..THEN..ELSE statements to achieve n-way branching, but the CASE..ENDCASE construct accomplishes the same objective with less typing. By using labels for state outputs and condition inputs, you can implement even the most complex designs with ease.

```
truth_table
([I2,I1,I0] -> [Q7..Q0])
 [0,0,0] -> [0,0,0,0,0,0,0,1];
 [0,0,1] -> [0,0,0,0,0,0,1,0];
 [0,1,0] -> [0,0,0,0,0,1,0,0];
 [0,1,1] -> [0,0,0,0,1,0,0,0];
 [1,0,0] -> [0,0,0,1,0,0,0,0];
 [1,0,1] -> [0,0,1,0,0,0,0,0];
 [1,1,0] -> [0,1,0,0,0,0,0,0];
 [1,1,1] -> [1,0,0,0,0,0,0,0];
```

Figure 1. Truth Table for 3:8 Line Decoder

As an example, consider a bidirectional, 3-bit counter with inputs UP and DOWN and outputs Q2, Q1, and Q0. If UP or DOWN is High, the counter counts in the direction specified. If both UP and DOWN are High, the counter holds the current count. If both UP and DOWN are Low, the counter resets to zero. In addition, output MAX is High if the counter is in the UP mode and the count equals 7 or if the counter is in the DOWN mode and the count equals zero. Convenient labels for implementing this design appear in *Figure 2*, and *Figure3* lists the source code for the state diagram.

You can add another statement, WITH..ENDWITH, to any transition statement to set additional outputs to any given state when the transition preceding the WITH..ENDWITH statement is executed. In the previous state diagram, for example, assume the transition from state S5 to S6 is to set a pin called FLAG. To achieve this result, the S5 diagram is modified as shown in *Figure4*.

PALC22V10 Design Examples

The design examples presented here exploit the various features of the 22V10 PLD. The first seven designs focus on specific features and illustrate the techniques for using and testing these features. The last three designs combine several of the features to demonstrate the device's versatility. It is the 22V10's tremendous versatility that has made it the most popular of all Cypress PLDs. Each of the last three designs, if implemented in SSI and MSI TTL, would require from seven to 13 packages.

Asynchronous Reset/Synchronous Preset

As shown in *Figure5*, this example defines pins 2 and 3 to be the asynchronous reset and synchronous preset inputs, respectively. Eight inputs defined as INPUT7..INPUT0 are given the label INPUTS. Eight

```
"labels
OUTS = [Q2..Q0];
MODE = [UP,DOWN];
CNTUP = ^b10; CNTDWN = ^b01;
RST = ^ b00; HOLD = ^ b11;
S0 = ^b000; S1 = ^b001; S2 = ^b010;
S3 = ^b011; S4 = ^b100; S5 = ^b101;
S6 = ^ b110; S7 = ^ b111;
```

Figure 2. State Machine Labels for Counter Example

```
state_diagram OUT

state S0: MAX = (MODE == CNTDWN);
        case (MODE = = CNTUP) : S1;
             (MODE = = CNTDWN) : S7;
             (MODE = = HOLD) : S0;
             (MODE = = RST)  : S0;
        endcase;

state S1 : MAX = 0;
        case (MODE = = CNTUP) : S2;
             (MODE = = CNTDWN) :S0;
             (MODE = = HOLD) : S1;
             (MODE = = RST)  : S0;
        endcase;

state S2 : MAX = 0;
        case (MODE = = CNTUP) : S3;
             (MODE = = CNTDWN):S1;
             (MODE = = HOLD) : S2;
             (MODE = = RST) : S0;
        endcase;

state S3 : MAX = 0;
        case (MODE = = CNTUP) : S4;
             (MODE = = CNTDWN) :S2;
             (MODE = = HOLD) : S3;
             (MODE = = RST) : S0;
        endcase;

state S4 : MAX = 0;
        case (MODE = = CNTUP) : S5;
             (MODE = = CNTDWN): S3;
             (MODE = = HOLD) : S4;
             (MODE = = RST)  : S0;
        endcase;

state S5 : MAX = 0;
        case (MODE = = CNTUP) : S6;
             (MODE = = CNTDWN) : S4;
             (MODE = = HOLD) : S5;
             (MODE = = RST)  : S0;
        endcase;

state S6 : MAX = 0;
        case (MODE = = CNTUP) : S7;
             (MODE = = CNTDWN) : S5;
             (MODE = = HOLD) : S6;
             (MODE = = RST)  : S0;
        endcase;

state S7 :MAX = (MODE == CNTDWN);
        case (MODE = = CNTUP) : S0;
             (MODE = = CNTDWN): S6;
             (MODE = = HOLD) : S7;
             (MODE = = RST)  : S0;
        endcase;
```

Figure 3. ABEL Source Code for Counter Example

corresponding outputs, OUTPUT7..OUTPUT0, are labeled OUTPUTS. Note how the use of labels enables the logic for all eight outputs to be written in a single equation. The equation:

```
OUTPUTS := INPUTS;
```

causes the data at INPUTS to be registered in OUTPUTS on the rising edge of CLK. The assignment operator ":=" indicates that the operation is clocked (registered). The 22V10 clock input is, by definition, pin 1.

The pin assignments section identifies the predefined node numbers for the reset and preset functions. The equations for the nodes, in terms of the selected pins, are then written in the file's equations section.

Asynch. Reset and Synch. Preset from Product Terms

This example (*Figure 6*) implements an asynchronous reset and synchronous preset, as does the example in *Figure 5*. In this case, however, product terms activate the reset and preset nodes. Specifically, the reset node is High (active) only when INPUTS equal 55 hex. Similarly, INPUTS equaling AA hex control the preset term. Note how the test vectors distinguish and test the synchronous versus the asynchronous operations.

Reset and Preset Load Predetermined Values

The examples in *Figures 5* and *6* use the macrocells' positive, registered output for the pins represented by OUTPUTS. Under this arrangement, the asynchronous reset causes all outputs to go Low and the synchronous preset causes them to go High.

This example demonstrates how you can use istype statements in the pin assignments section to set any pattern of Ones and Zeros, either asynchronously with reset or synchronously with preset. To understand this operation, note in *Figure7* that the 22V10 provides four

```
state S5 : MAX = 0;
        case (MODE = = CNTUP) : S6
                with FLAG := 1;
                endwith

             (MODE = = CNTDWN): S0;
             (MODE = = HOLD) : S5;
             (MODE = = RST)  : S0;
        endcase;
```

Figure 4. WITH..ENDWITH Example

```
                                                        "Cypress Semiconductor Corp. 11/10/1987

module Rst_Pre1                                         "Module name test

    flag '-r3'                                          "Logic Reduction level r3, fast PRESTO
    title 'Asynchronous Reset / Synchronous Preset Control From A Single Input

                                                        "Device designator and type
    U1 device 'P22V10';

                                                        "Pin assignments
    CLK                                 pin 1;          "Clock input
    RST                                 pin 2;          "Defines async reset pin
    PRE                                 pin 3;          "Defines sync preset pin
    INPUT7,INPUT6,INPUT5,INPUT4         pin 4,5,6,7;
    INPUT3,INPUT2,INPUT1,INPUT0         pin 8,9,10,11;
    OUTPUT7,OUTPUT6,OUTPUT5,OUTPUT4     pin 23,22,21,20;
    OUTPUT3,OUTPUT2,OUTPUT1,OUTPUT0     pin 19,18,17,16;
    reset,preset                        node 25,26;     "Pre-assigned node #s

                                                        "Labels
    H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
    INPUTS          =       [INPUT7..INPUT0];
    OUTPUTS         =       [OUTPUT7..OUTPUT0];

    @radix 16;                                          "This command forces the default
                                                        "number base to HEX.
    equations
    reset           =       !RST;                       "Async reset when pin RST low
    preset          =       PRE;        "Sync preset if pin PRE is high during the rising edge of CLK
    OUTPUTS         :=      INPUTS;     "The := indicates that this a clocked (synchronous) operation
    test_vectors
                                                        "Test reset and preset
    ([CLK,RST,PRE,INPUTS]   ->      OUTPUTS)
    [C,H,L,55]      ->      55;                         "Test outputs by clocking in 55
    [L,H,L,0AA]     ->      55;                         "Test registers hold old data (55)
    [C,H,L,0AA]     ->      0AA;                        "Clock AA (leading zero necessary for hex digits A-F)
    [C,H,L,0FF]     ->      0FF;                        "Set all outputs high (FF)
    [L,L,L,0FF]     ->      0;                          "RST low asynchronously
    [C,H,H,0]       ->      0FF;                        "PRE high synchronously
end Rst_Pre1
```

Figure 5. Reset/Preset from Single Pins

```
                                                "Cypress Semiconductor Corporation, 11/10/1987
module Rst_Pre2                                 "Module name test
       flag '-r3'                               "Logic Reduction level r3, PRESTO algorithm by pin
       title 'Asynchronous Reset / Synchronous Preset Example 2, Reset and Preset generated from Product terms'
                                                "*************************************************************
                                                "* This Example will Asynchronously Reset all registers when the inputs
                                                "* Synchronously Set all registers when the inputs equal AA
                                                "*************************************************************
                                                "Device designator and type
       U1 device 'P22V10';
                                                                        "Pin assignments
       CLK                                      pin 1;                  "Clock input
       INPUT7,INPUT6,INPUT5,INPUT4              pin 4,5,6,7;
       INPUT3,INPUT2,INPUT1,INPUT0              pin 8,9,10,11;
       OUTPUT7,OUTPUT6,OUTPUT5,OUTPUT4          pin 23,22,21,20;
       OUTPUT3,OUTPUT2,OUTPUT1,OUTPUT0          pin 19,18,17,16;
       reset,preset                             node 25,26;             "Pre-assigned node #s

                                                                        "Labels
       H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
       INPUTS                  =       [INPUT7..INPUT0];
       OUTPUTS         =       [OUTPUT7..OUTPUT0];

       @radix 16 ; "command forces the default number base to be HEX

       equations
       reset           =       (INPUTS==55);           "Async reset when input = 55
       preset          =       (INPUTS==0AA);          "Sync preset if inputs = AA during the rising edge of CLK
       OUTPUTS         :=      INPUTS;                 "The := indicates that this a clocked (synchronous) operation

       test_vectors                             "Test reset and preset
       ([CLK,INPUTS]  ->  OUTPUTS)
       [C,0]   ->      0;                      "Test outputs by clocking in 0
       [L,0FF] ->      0;                      "Test registers hold old data (0)
       [C,0FF] ->      0FF;                    "Clock in FF (note leading zero for hex digits A thru F)
       [L,55]  ->      0;                      "RST low asynchronously on inputs = 55
       [L,0AA]->       0;                      "No change, PRE is synchronous
       [C,0AA]->       0FF;                    "PRE acts synchronously on inputs = AA
end Rst_Pre2
```

Figure 6 . Reset / Preset From Product Terms

paths from the macrocells to the I/O pins: the Q and Q\ outputs of the macrocell's register and the true and inverted combinatorial terms that bypass the register. All these paths pass through a 4:1 multiplexer, which is controlled by architecture bits C0 and C1.

The istype statements allow you to select which channel of the multiplexer is routed to the I/O pin. *Table 8* shows the choices available.

An additional parameter in the istype statement allows you to select feedback paths. The choices are feed_term, feed_reg, and feed_pin. An example showing this parameter is:

```
OUTPUT6 istype 'pos,com,feed_pin';
```

Specifying a feedback path for the 22V10 is redundant, however. This is because the 22V10 selects a feedback path using the same architecture bit (C1) that controls the selection of registered or combinatorial outputs. The 22V10 does not offer a feedback path from product terms.

Table 8. Macrocell Configuration Selections

C1	C0	Configuration	istype Values
0	0	Reg,Active Low	'neg, reg'
0	1	Reg,Active High	'pos, reg'
1	0	Comb,Active Low	'neg, com'
1	1	Comb,Active High	'pos, com'

Note from the test vectors in *Figure 8* that the use of istype statements does not affect the outputs' polarity as described by the Boolean equations. Conversely, if you define an output as active Low through a Boolean equation, as in:

```
!OUTPUT6 := INPUT6;
```

the state of the register is inverted for normal operation and for reset and preset conditions.

A final note on using istype statements in conjunction with the reset node: The 22V10 resets when V_{CC} is first applied to the chip. Istype statements and active-Low Boolean equations give you the opportunity to force the device's outputs to any desired state upon power up.

Output Enable Controlled by One Pin

The example in *Figure 9* defines pin 2 as the output enable pin for all outputs. Note the use of special constant ".Z." which is redefined as simply "Z" in the file's labels section. The constant is used in the test vectors to verify that the outputs are three-stated (high-Z) under the appropriate conditions.

Product-Term-Based Output Enable

While *Figure 9* illustrates gang control of all output enables via an input pin, *Figure10* shows several outputs with individual output enables generated from separate product terms.

As with reset and preset, you can make output enables synchronous or extend the number of product terms by using a macrocell to generate the necessary logic and

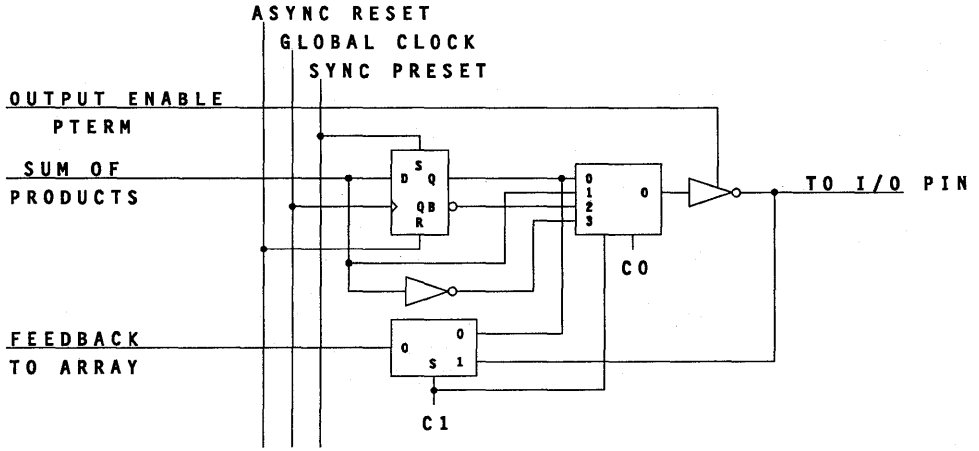


Figure 7. The PALC22V10 Macrocell

```
                                                    "Cypress Semiconductor Corporation, 11/10/1987
module Rst_Pre3                                     "Module name test
       flag '-r3'                                   "Logic Reduction level r3, PRESTO algorithm by pin

       title 'Asynchronous Reset/Synchronous Preset Example 3, Using Reset and Preset to Load to Predetermined States
       "*************************************************************************
       "* This Example will Asynchronously Load a Value of 55 and Synchronously Load    *
       "* Value of AA by using 'istype' statements to invert alternating output registers    *
       "*************************************************************************
                                            "Device designator and type
       U1 device 'P22V10';
                                                                    "Pin assignments
       CLK                                  pin 1;                 "Clock input
       RST                                  pin 2;                 "Defines async reset pin
       PRE                                  pin 3;                 "Defines sync preset pin
       INPUT7,INPUT6,INPUT5,INPUT4          pin 4,5,6,7;
       INPUT3,INPUT2,INPUT1,INPUT0          pin 8,9,10,11;
       OUTPUT7,OUTPUT6,OUTPUT5,OUTPUT4      pin 23,22,21,20;
       OUTPUT3,OUTPUT2,OUTPUT1,OUTPUT0      pin 19,18,17,16;
       OUTPUT7,OUTPUT5,OUTPUT3,OUTPUT1      istype 'pos,reg';  "Odd regs positive logic
       OUTPUT6,OUTPUT4,OUTPUT2,OUTPUT0      istype 'neg,reg';  "Even regs negative
       reset,preset                         node 25,26;        "Pre-assigned node #s
                                            "Labels
       H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
       INPUTS          =       [INPUT7..INPUT0];
       OUTPUTS         =       [OUTPUT7..OUTPUT0];

       @radix 16;                           "command forces the default number base to be HEX

       equations
       reset           =       !RST;        "Async reset when pin RST low
       preset          =       PRE;         "Sync preset if pin PRE is high during the rising edge of CLK
       OUTPUTS         :=      INPUTS;      "The := indicatese that this a clocked (synchronous) operation
       test_vectors
       ([CLK,RST,PRE,INPUTS] -> OUTPUTS)  "Test Reset and Preset
       [C,H,L,55]      ->      55;          "Test outputs by clocking in 55
       [L,H,L,0AA]     ->      55;          "Test registers hold old data (55)
       [C,H,L,0AA]     ->      0AA;         "Clock in AA (note the leading zero necessary for hex digits A thru F)
       [C,H,L,0FF]     ->      0FF;         "Set all outputs high (FF)
       [L,L,L,0FF]     ->      55;          "RST low asynchronously (bits 6,4,2,0 inverted)
       [C,H,H,0]               ->      0AA;         "PRE high synchronously (bits 6,4,2,0 inverted)

end Rst_Pre3
```

Figure 8. Resetting and Presetting to Predetermined Values

```
                                        "Cypress Semiconductor Corporation November 10, 1987
module Out_Enable1                      "Module name
       flag '-r3'                       "Logic Reduction level r3
       title 'Output Enable from Single Input Example '
                                        "*********************************************
                                        "* This example demonstrates the Output Enable,     *
                                        "* Function being controlled by a single input      *
                                        "*********************************************
       U1 device 'P22V10';              "Device designator and type
                                        "Pin assignments
       CLK                                  pin 1;          "Clock input
       OE                                   pin 2;          "Output enable input
       INPUT7,INPUT6,INPUT5,INPUT4          pin 4,5,6,7;
       INPUT3,INPUT2,INPUT1,INPUT0          pin 8,9,10,11;
       OUTPUT7,OUTPUT6,OUTPUT5,OUTPUT4      pin 23,22,21,20;
       OUTPUT3,OUTPUT2,OUTPUT1,OUTPUT0      pin 19,18,17,16;
       reset,preset                         node 25,26;     "Pre-assigned node #s
                                        "Labels
       H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
       INPUTS          =       [INPUT7..INPUT0];
       OUTPUTS         =       [OUTPUT7..OUTPUT0];
       OUTENA          =       [OUTPUT7.OE,OUTPUT6.OE,OUTPUT5.OE,OUTPUT4.OE];
       OUTENB          =       [OUTPUT3.OE,OUTPUT2.OE,OUTPUT1.OE,OUTPUT0.OE];

       @radix 16;                       "This command forces the default number base to be HEX

       equations
       OUTENA          =       !OE;    "Outputs enabled only if pin OE is low
       OUTENB          =       !OE;
       OUTPUTS         :=      INPUTS;

       test_vectors                     "Test output enables

       ([CLK,OE,INPUTS]        ->      OUTPUTS)
       [C,L,55]        ->      55;     "Test outputs by clocking in 55 (outputs enabled)
       [L,H,0AA]       ->      Z;      "Test outputs go to high-Z state on OE high
       [L,L,0AA]       ->      55;     "Test registers hold old data (55)
       [C,L,0AA]       ->      0AA;    "Clock in AA (note the leading zero necessary for hex digits A thru F)
       [C,H,0FF]       ->      Z;      "Set all outputs high (FF) but tri-stated
       [L,L,X]         ->      0FF;    "Turn outputs on and read FF
end  Out_Enable1
```

Figure 9. Output Enable Controlled by a Single Input

```
                                        "Cypress Semiconductor Corp.  11/10/1987
module Out_Enable2                      "Module name
       flag '-r3'                       "Logic Reduction level r3
       title 'Output Enable From a Product Term Example'
                                        "*********************************************
                                        "* This example demonstrates the Output Enable  *
                                        "* Function being controlled by a product term  *
                                        "*********************************************
       U1 device 'P22V10';                      "Device designator and type
                                                "Pin assignments
       CLK, OE                          pin 1, 2;       "Clock and Output Enable inputs
       INPUT7,INPUT6,INPUT5,INPUT4      pin 4,5,6,7;
       INPUT3,INPUT2,INPUT1,INPUT0      pin 8,9,10,11;
       OUTPUT7,OUTPUT6,OUTPUT5,OUTPUT4  pin 23,22,21,20;
       OUTPUT3,OUTPUT2,OUTPUT1,OUTPUT0  pin 19,18,17,16;
       reset,preset                     node 25,26;     "Pre-assigned node #s

       H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;                "Labels
       INPUTS          =       [INPUT7..INPUT0];
       OUTPUTS         =       [OUTPUT7..OUTPUT0];

       @radix 16;              "This command forces the default number base to be HEX

       equations               "Each Output individually enabled if the corresponding digital code is applied at
                               "inputs and OE is low
       OUTPUT0.OE = (INPUTS == 0) & !OE;       OUTPUT1.OE = (INPUTS == 1) & !OE;
       OUTPUT2.OE = (INPUTS == 2) & !OE;       OUTPUT3.OE = (INPUTS == 3) & !OE;
       OUTPUT4.OE = (INPUTS == 4) & !OE;       OUTPUT5.OE = (INPUTS == 5) & !OE;
       OUTPUT6.OE = (INPUTS == 6) & !OE;       OUTPUT7.OE = (INPUTS == 7) & !OE;
       OUTPUTS := INPUTS;
       test_vectors
       ([CLK,OE,INPUTS]  ->  [OUTPUT7..OUTPUT0])
       [C,H,55]        ->      [Z,Z,Z,Z,Z,Z,Z,Z];
       [L,H,0]         ->      [Z,Z,Z,Z,Z,Z,Z,Z];
       [L,L,0]         ->      [Z,Z,Z,Z,Z,Z,Z,1];
       [L,L,1]         ->      [Z,Z,Z,Z,Z,Z,0,Z];      "Loads 55, checks OE high overrides
       [L,L,2]         ->      [Z,Z,Z,Z,Z,1,Z,Z];      "all enable terms, then enables and
       [L,L,3]         ->      [Z,Z,Z,Z,0,Z,Z,Z];      "checks all outputs one at a time
       [L,L,4]         ->      [Z,Z,Z,1,Z,Z,Z,Z];
       [L,L,5]         ->      [Z,Z,0,Z,Z,Z,Z,Z];
       [L,L,6]         ->      [Z,1,Z,Z,Z,Z,Z,Z];
       [L,L,7]         ->      [0,Z,Z,Z,Z,Z,Z,Z];

end  Out_Enable2
```

Figure 10. Separate Output Enables Controlled by Product Terms

looping back the term via a feedback path. This method incurs additional propagation delay due to passing through the AND/OR array twice, however.

The special constant ".Z." is used in the test vectors for this design to verify the operation of outputs in the three-stated (high-Z) mode.

An 8-Bit Identity Comparitor

This example (*Figure 11*) points out how the 22V10's variable-product-term architecture permits you to directly implement logic that would otherwise require multiple feedback terms in standard PLDs. The 22V10 offers 16 product terms maximum, compared to only eight product terms per output for standard 20-pin PLDs.

An n-bit comparitor requires 2^n product terms to implement. This example achieves 8-bit comparison by decomposing the 8 bits into two 4-bit comparisons and using I/O pins 18 and 19 for each 4-bit comparison. These pins have 16 product terms each. The results of each 4-bit comparison are available at the pins one t_{pd} after a match is detected

Note in *Figure 11* how the inputs and outputs are used in more than one label. This practice facilitates writing equations and test vectors for the individual 4-bit fields and the complete 8-bit fields.

Single-Output, 9-Bit Identity Comparitor

This example is very similar to the example in *Figure 11*, except this example rearranges the DATA inputs to AND the two 4-bit comparitor outputs with the result of the single, 9th-bit compare. The result is a single DATA = INPUTS output called INEQDATA. The source code for this example appears in *Figure12*.

The disadvantage of this implementation is that it incurs an additional t_{pd} by feeding the individual 4-bit comparitor outputs back through the AND/OR array. Note that although the terms fed back to INEQDATA represent 34 (16 + 16 + 2) product terms, only three of the eight product terms available at I/O pin 23 are used; each of the three individual compares have already been reduced to single signals by the time they reach the AND/OR array for pin 23. You can also use the extra product terms along with a separately defined input for cascading the design to n-bit length.

Bus Interface Data Trap with Answer-back

This example demonstrates the 22V10's bidirectional I/O capabilities (*Figure 13*). In this example, an 8-bit pattern is supplied to INPUTS and is continuously compared to the data on DATA7..DATA0.

This design is intended for an application in which DATA7..DATA0 is a Z80 microprocessor's data bus. If the interrupt is enabled (pin INTRENBL is High), the 8-bit comparitor output drives pin INTR active (Low). In response, the Z80 drives pin IDREQ High. This action asks the device that initiated the interrupt to place its 8-bit ID code on the data bus. In this example, the ID code used is ^h55. You can use any code by modifying the equation for DATA in the source file.

Counter/Address Generator/Multiplexer

This 10-bit counter, address generator, and multiplexer example (*Figure14*) implements the address-generation circuitry for the front end of a high-speed data-acquisition module. The design requires two modes of operation: In ACQUIRE mode, counters generate the ten address lines. In READ mode, a microprocessor's address lines generate the same addresses.

A discrete version of this application employs quad 2:1 multiplexers to select whether the counters or microprocessor provide the address information. The entire discrete circuit, excluding the SRAM being addressed, consists of 11 SSI and MSI TTL components. The example given here implements the equivalent circuitry in a single 22V10.

Note how the MODE pin in the equations for the AOUT outputs controls the source of the addresses. Also note the use of the asynchronous reset node: the reset term is generated when the MODE is set for microprocessor access (Low) and the processor address itself is zero. Although the effect at the outputs (all outputs = zero) is the same as if the reset term were not included, the asynchronous reset gives the processor a way to reset all the registers to a known state before allowing the counters to free-run again.

Timing Diagram

One of the more interesting features of the ABEL SIMULATE program is its ability to generate timing diagrams for specified pins based on the test vectors in a source file. Although a timing diagramdoes not show propagation delays, it can help you verify a device's in-circuit operation with a logic analyzer. The SIMULATE output file shown in *Figure 15* is generated with the command line:

```
simulate -iaddmux.out -oaddmux.sim -t4 -
    w1,2,3,4,5,13,14,15,16,17,18
```

```
                                        "Cypress Semiconductor Corporation      November 10, 1987
module AllTerms                         "Module name
    flag '-r3'                          "Logic Reduction level r3, PRESTO algorithm by pin
    title 'Using 16 Product Terms; An 8-bit Identity Comparitor '
                    "*****************************************************************************
                    "* In this design, an 8-bit word is presented at I/O pins 23,22,21,20,17,16,15 and 14.
                    "* These pins are used for inputs only in this example. The 8-bit word is compared, 4 bits
                    "* at a time, to inputs INPUT7..0. Combinatorial outputs COMPHI and COMPLO show
                    "* the result of each 4-bit comparison. Pins 19 and 18 are used as the comparitor outputs
                    "* since these pins have enough Product Terms (16) for the required 4-bit comparisons.
                    "*****************************************************************************
    U1 device 'P22V10';                 "Device designator and type
                                                        "Pin assignments
    CLK                                 pin 1;          "Clock input (NOT used)
    INPUT7,INPUT6,INPUT5,INPUT4         pin 4,5,6,7;
    INPUT3,INPUT2,INPUT1,INPUT0         pin 8,9,10,11;
    DATA7,DATA6,DATA5,DATA4             pin 23,22,21,20;
    DATA3,DATA2,DATA1,DATA0             pin 17,16,15,14;
    COMPHI,COMPLO                       pin 19,18;      "Comparator outputs
    reset,preset                        node 25,26;     "Pre-assigned node #s
    H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
    INPUTSH         =       [INPUT7..INPUT4];           "High-order nibble
    DATAH           =       [DATA7..DATA4];
    INPUTSL         =       [INPUT3..INPUT0];           "Low-order nibble
    DATAL           =       [DATA3..DATA0];
    DATA            =       [DATA7..DATA0];             "All 8 bits
    INPUTS          =       [INPUT7..INPUT0];

    @radix 16;

    equations
    COMPHI = (INPUTSH == DATAH);                        "High-order nibble compare
    COMPLO = (INPUTSL == DATAL);                        "Low-order nibble compare

    test_vectors
    ([DATA,INPUTS]  ->  [COMPHI,COMPLO])
    [0,0]   ->  [H,H];      [1,1]   ->  [H,H];      [2,2]   -> [H,H];
    [4,4]   ->  [H,H];      [8,8]   ->  [H,H];      [0F,0F] -> [H,H];
    [0E,0E] ->  [H,H];      [0D,0D] ->  [H,H];      [0B,0B] -> [H,H];
    [7,7]   ->  [H,H];      [0,0F]  ->  [H,L];      [0F0,0F]-> [L,L];
    [0F0,0] ->  [L,H];      [0F0,0FF]-> [H,L];

end AllTerms
```

Figure 11. Using 16 Product Terms : An 8-Bit Identity Comparitor

```
                                    "Cypress Semiconductor Corporation November 10, 1987

module CompFB                       "Module name
    flag '-r3'                              "Logic Reduction level r3, PRESTO algorithm by pin
    title 'Using Feedback to Realize more than 16 Product Terms; A Single Output, 9-bit Identity Comparitor  '
                "*****************************************************************************
                "* In this design, an 9-bit word is presented at pins 23,22,21,20,17,16,11,10 and 9.      *
                "* These pins are used for inputs only in this example. The 8 LSBs of the 9-bit word are  *
                "* compared, 4 bits at a time, to inputs INPUT7..0. Combinatorial outputs COMPHI and      *
                "* COMPLO show the results of each 4-bit comparison. Pins 19 and 18 are used as the      *
                "* comparitor outputs since these pins have enough Product Terms (16) for the required   *
                "* 4-bit comparison. The MSBs (bit 8) of DATA and are compared at output COMPMSB.  *
                "* Outputs COMPMSB, COMPHI, and COMPLO are ANDED together to form output   *
                "* INEQDATA.                                                                  *
                "*****************************************************************************
    U1 device 'P22V10';                                         "Device designator and type
                                                                "Pin assignments
    INPUT8,INPUT7,INPUT6,INPUT5,INPUT4      pin 1,2,3,4,5;
    INPUT3,INPUT2,INPUT1,INPUT0             pin 6,7,8,9;
    DATA8,DATA7,DATA6,DATA5,DATA4           pin 10,11,13,14,15;
    DATA3,DATA2,DATA1,DATA0                 pin 16,17,20,21;
    COMPH,COMPL,COMPMSB,INEQDATA            pin 19,18,22,23;  "Comparator outputs
    reset,preset                            node 25,26;       "Pre-assigned node #s
    H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
    INPUTSH         =       [INPUT7..INPUT4];                 "High-order nibble
    DATAH           =       [DATA7..DATA4];
    INPUTSL         =       [INPUT3..INPUT0];                 "Low-order nibble
    DATAL           =       [DATA3..DATA0];
    DATA            =       [DATA8..DATA0];                   "All nine bits
    INPUTS          =       [INPUT8..INPUT0];
    @radix 16;
    equations
    COMPH           =       (INPUTSH == DATAH);               "High-order nibble compare
    COMPL           =       (INPUTSL == DATAL);               "Low-order nibble compare
    COMPMSB         =       (INPUT8 == DATA8);                "MSB compare
    INEQDATA        =       COMPH & COMPL & COMPMSB;          "Logical AND of all comparisons
    test_vectors
    ([DATA,INPUTS]  ->  [COMPH,COMPL,COMPMSB,INEQDATA])
    [0,0]   ->      [H,H,H,H];      [111,111]  ->   [H,H,H,H];
    [22,22] ->      [H,H,H,H];      [44,44]    ->   [H,H,H,H];
    [88,88] ->      [H,H,H,H];      [1FF,1FF]  ->   [H,H,H,H];
    [0,100] ->      [H,H,L,L];      [1FF,0FF]  ->   [H,H,L,L];
    [1FE,1FF]  ->   [H,L,H,L];      [1FE,1EE]  ->   [L,H,H,L];
end CompFB
```

Figure 12. Realizing More Than 16 Product Terms Through Feedback: A 9-Bit, Single-Output Identity Comparitor

```
                                        "Cypress Semiconductor Corp., 11/10/1987
module BiDirect                         "Module name test
    flag '-r3'                          "Logic Reduction level r3, PRESTO algorithm by pin
    title 'Bi-Directional I/O A Bus Interface Data Trap with Answer-Back'
                "*****************************************************************************
                "* This example compares the pattern at pins INPUTS to the data on data bus pins      *
                "* DATA7..DATA0. Pin INTR is driven low if they match and INTRENBL (interrupt      *
                "* enable) is high. Input IDREQ is then driven high, requesting ID code (^ h55 in   *
                "* this example) to be put on the data bus                                          *
                "*****************************************************************************
    U1 device 'P22V10';

    IDREQ,  INTRENBL                    pin 2,3;        ", Output Enable, Interrupt Enable
    COMPL,INTR                          pin 19,18;      "Used in comparision of 4 LSBs
    INPUT7,INPUT6,INPUT5,INPUT4         pin 4,5,6,7;
    INPUT3,INPUT2,INPUT1,INPUT0         pin 8,9,10,11;
    DATA7,DATA6,DATA5,DATA4             pin 23,22,21,20;
    DATA3,DATA2,DATA1,DATA0             pin 17,16,15,14;
    reset,preset                        node 25,26;             "Pre-assigned node #s
    H,L,X,C,Z       = 1,0,.X.,.C.,.Z.;
    INPUTS = [INPUT7..INPUT0];                                  "All inputs
    INPUTH = [INPUT7..INPUT4];                                  "High order nibble of INPUTS
    INPUTL = [INPUT3..INPUT0];                                  "Low order nibble of INPUTS
    DATA = [DATA7..DATA0];                                      "All data I/Os
    DATAH = [DATA7..DATA4];                                     "High order nibble of DATA
    DATAL = [DATA3..DATA0];                                     "Low order nibble of DATA
    DATAOEA = [DATA7.OE,DATA6.OE,DATA5.OE,DATA4.OE];
    DATAOEB     = [DATA3.OE,DATA2.OE,DATA1.OE,DATA0.OE];
    IDCODE = ^h55;                                              "Identification code

    equations
    DATAOEA=    IDREQ;                          "Enables ID output onto data bus
    DATAOEB=    IDREQ;
    DATA =      IDCODE;                         "Identification code for device (^h55)
    COMPL=      (DATAL == INPUTL);              "4 LSBs compare
    !INTR =     (DATAH = = INPUTH) & COMPL & INTRENBL; "INTR active low, All bits equal and
                                                    "interrupt enabled (INTRENBL high)
    test_vectors
    ([IDREQ,INTRENBL,DATA,INPUTS] -> [COMPL,INTR,DATA])
    [L,H,^h0F,^h1F]         ->      [H,H,X];        "Low nibble equal,high not equal
    [L,H,^h0F0,^h0F1]       ->      [L,H,X];        "High nibble equal, low not equal
    [L,L,^h0AA,^h0AA]       ->      [H,H,X];        "Test Interrupt Enable
    [L,H,^h0AA,^h0AA]       ->      [H,L,X];        "DATA = INPUTS, INTR goes active (low)
    [L,H,^h55,^h55]         ->      [H,L,X];
    [H,H,Z,X]               ->      [X,X,IDCODE];   "DATA pins output IDCODE (^h55)

end BiDirect
```

Figure 13. BiDirectional I/O : Bus Interface Data

```
module AddGenMux flag '-r3'                    "Cypress Semiconductor Corporation November 10, 1987
     title '10-bit Address Generation / Multiplexer IC'
                                    "*********************************************************
                                    "* This PLD design generates Address signals A0-A9.      *
                                    "* If Control signal MODE is high, the address signals   *
                                    "* are the output of a 10-bit counter. If MODE is low    *
                                    "* the device passes uP Address lines UPADD0-UPADD9      *
                                    "*********************************************************
     AdrsGen device 'p22v10';
     CLK                                        pin 1;          "System Master Clock
     A0,A1,A2,A3,A4,A5,A6,A7,A8,A9              pin14,15,16,17,18,19,23,22,21,20;
     UPADD0,UPADD1,UPADD2,UPADD3                pin 2,3,4,5;
     UPADD4,UPADD5,UPADD6,UPADD7                pin 6,7,8,9;
     UPADD8,UPADD9                              pin 10,11;
     MODE                                       pin 13;
     reset,preset                               node 25,26;
     H,L,X,C,Z      =       1,0,.X.,.C.,.Z.;
     AOUT           =       [A9..A0];                       "Address Outputs
     UPADD                  =       [UPADD9..UPADD0];       "uP Address Lines
     @radix 16;
     equations                                              "Boolean equations
     reset          =       (UPADD == 0) & !MODE;           "Reset if uP Address = 00 and MODE is low
     AOUT           :=      ((AOUT + 1) & MODE)             "Count up if MODE high or
                            # (UPADD & !MODE);              "Pass UPADD if MODE low
     test_vectors                                           "Check Operation
     ([CLK,UPADD,MODE]  ->  AOUT)
     [X,0,L] ->     0;                                      "Checks Reset Function
     [C,X,H] ->     1;      [C,X,H]  ->     2;      [C,X,H] ->      3;      [C,X,H]-> 4;
     [C,X,H] ->     5;      [C,X,H]  ->     6;      [C,X,H] ->      7;      [C,X,H]-> 8;
     [C,X,H] ->     9;      [C,X,H]  ->     0A;     [C,X,H] ->      0B;     [C,X,H]-> 0C;
     [C,X,H] ->     0D;     [C,X,H]  ->     0E;     [C,X,H] ->      0F;     [C,X,H]-> 10;
     [C,111,L]->    111;    [C,222,L]->     222;    [C,44,L] ->     44;     [C,88,L]-> 88;
     [C,2EE,L]->    2EE;    [C,1DD,L]  ->   1DD;    [C,3BB,L]->     3BB;    [C,377,L]-> 377;
     [C,155,L]->    155;    [C,2AA,L]  ->   2AA;    [C,3FF,L]->     3FF;    [C,222,H]-> 00;
     [C,0FF,L]->    0FF;    [C,X,H] ->      100;
     [C,1FF,L]->    1FF;    [C,X,H] ->      200;                    "Load to states where all 8 LSBs
     [C,2FF,L]->    2FF;    [C,X,H] ->      300;                    "are high (uP mode), then toggle in
     [C,3FF,L]->    3FF;    [C,X,H] ->      0;                      "counter mode
end  AddGenMux
```

Figure 14. 10-Bit Address Generator/Multiplexer

The "-i" indicates the input file, which in this case is the intermediate output file created by ABEL's FUSEMAP program. The "-o" tells SIMULATE which file to write the results into. The "-t4" specifies the trace level where waveforms are displayed, and the "-w1..18" indicates which pins to show in the waveform output.

You can find more information on SIMULATE in the *ABEL User's Guide* and *Language Reference* supplied with the ABEL software from DataI/O.

Three State Machines in One 22V10

This final example demonstrates the power of the 22V10 when used as a synchronous state machine. The application involves the redesign of a radar system's timing circuitry. The system performs 12 discrete Fourier transforms on each set of quadrature data returned in three antenna beams that are gated for nine ranges. The nonbinary nature of these numbers (three beams, nine ranges, and 12 speed bins) make generating the timing signals with counter circuits cumbersome.

This example creates three state machines in a single 22V10. As you can see from the state diagrams (*Figure 16*), the filter state machine is free running. The beam state machine only changes states when the filter outputs are in their maximum condition. Similarly the gate information changes only if both the filter and beam outputs are at their maximum values.

Note the combined use of Boolean equations and state diagrams. A separate state diagram describes each state machine, but the transitions depend upon the condition of the other state outputs. Also of note is the extreme use of labels for pins, groups of pins, and the state outputs. This approach greatly simplifies the writing of the state diagrams and test vectors.

When this design was first compiled, the ABEL FUSEMAP routine indicated several outputs that had too many terms for the physical array of the corresponding I/O pin. The design was made to fit by carefully arranging the I/Os. The flag "-r3" reduction statement made the fit possible without the tedium of generating and manually reducing Boolean equations from the state diagrams.

The test vectors for this design are of particular interest. Note how the @REPEAT command cycles through 35 states to make the gate state outputs toggle. This powerful command helps describe 325 test vectors in a concise and manageable manner.

```
ABEL Version 2.00b  Data I/O Corp.
Address Generation / Multiplexer IC
Simulate device AdrsGen, type 'P22V10'

          U  U  U  u
          P  P  P  P
          A  A  A  A  M
       C  D  D  D  D  O
       L  D  D  D  D  D  A  A  A  A
       K  0  1  2  3  E  0  1  2  3

V 0001 |  |  |  |  |  |_ |_ |  |  |
V 0002 C  |  |  |  |     |  _| |_ |  |
V 0003 C  |  |  |  |     |  |_ |  |  |
V 0004 C  |  |  |  |     |  _| _| |_ |
V 0005 C  |  |  |  |     |  |_ |    |  |
V 0006 C  |  |  |  |     |  _| |_ |  |
V 0007 C  |  |  |  |     |  |_ |  |  |
V 0008 C  |  |  |  |     |  _| _| _| |_
V 0009 C  |  |  |  |     |  |_ |  |  |
V 0010 C  |  |  |  |     |  _| |_ |  |
V 0011 C  |  |  |  |     |  |_ |  |  |
V 0012 C  |  |  |  |     |  _| _| |_ |
V 0013 C  |  |  |  |     |  |_ |    |  |
V 0014 C  |  |  |  |     |  _| |_ |  |
V 0015 C  |  |  |  |     |  |_ |  |  |
V 0016 C  |  |  |  |     |  _| _| _| _|
V 0017 C  |_ |  |  |     _| |_ |  |  |
V 0018 C     _| |_ |  |  |    _| |_ |  |
V 0019 C  |     _| |_ |  |  |    _| |_ |
V 0020 C  |  |     _| |_ |  |  |    _| |_
V 0021 C  |  |_ |_ |    |  |  |  |_ |_ |
V 0022 C  |_    _|  |  |  |  |_    _|  |  |
```

Figure 15. ABEL Simulated Waveform

```
                                                "Cypress Semiconductor Corporation November 10, 1987
module Statexam flag '-r3'
        title 'Timing Generation TRIPLE State Machine for DFT Processor using a Cypress Semiconductor PAL C22V10'
                "*************************************************************************
                "*  BEAM STATES - 0, 1, 2 (3 not used), GATE STATES - 0, 1, 2, 4, 5, 6, 8, 9, A
                "*  (3,7,B,C,D,E,F not used), FILTER STATES - 0, 1, 2, 4, 5, 6, 8, 9, A, C, D, E
                "*  (3,7,B,F not used)
                "*************************************************************************
        U1 device 'P22V10';
        SYSCLK                  pin 1;
        START                   pin 2;              "Used for reset/power-up
        AB0,AB1,AB2,AB3,AB4     pin 23,14,22,15,21; "Pins are non-sequential to take advantage of
        AB5,AB6,AB7,AB8,AB9     pin 16,18,19,20,17; "The variable number of product terms in the 22V10
        reset,preset            node 25,26;         "Pre-assigned node #s
        AB0,AB1,AB2,AB3,AB4     istype  'pos,reg';  "Unnecessary because ABEL will set architecture bits
        AB5,AB6,AB7,AB8,AB9     istype  'pos,reg';  "automatically - shown for example purposes only
        H,L,X,C,Z       =       1,0,.X.,.C.,.Z.;
        ABall           =       [AB9..AB0];
        FILT            =       [AB3..AB0];
        BEAM            =       [AB5,AB4];
        GATE            =       [AB9..AB6];
        @radix 16;
                                "Filter States - note missing states
        F0 =  00; F1 =  01; F2 =  02; F3 =  04; F4 =  05; F5 =  06; F6 =  08;
        F7 = 09; F8 = 0A; F9 = 0C; F10 = 0D; F11 = 0E;
                                        "Beam States
        B0 = 00; B1 = 01; B2 = 02;
                                        "Gate States
        G0 = 00; G1 = 01; G2 = 02; G3 = 04; G4 = 05; G5 = 06; G6 = 08; G7 = 09; G8 = 0A;
        equations
        reset = START;                  "Initialize to all lows on START
        state_diagram FILT
        State F0: GOTO F1; State F1: GOTO F2; State F2: GOTO F3; State F3: GOTO F4;
        State F4: GOTO F5; State F5: GOTO F6; State F6: GOTO F7; State F7: GOTO F8;
        State F8: GOTO F9; State F9: GOTO F10; State F10: GOTO F11; State F11: GOTO F0;
        state_diagram BEAM
        State B0: case  (FILT  ==  ^b1110)    : B1;
                        (FILT  !=  ^b1110)    : B0;
                endcase;
        State B1: case  (FILT  ==  ^b1110)    : B2;   "Increment ONLY if
                        (FILT  !=  ^b1110)    : B1;   "FILT is at max (0E)
                endcase;
        State B2: case  (FILT  ==  ^b1110)    : B0;
                        (FILT  !=  ^b1110)    : B2;
                endcase;
```

Figure 16. Triple State Machine (part1)

```
state_diagram GATE                    "Increments ONLY if BEAM and FILT are at max
State G0: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G1;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G0;
        endcase;
State G1: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G2;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G1;
        endcase;
State G2: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G3;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G2;
                endcase;
State G3: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G4;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G3;
        endcase;
State G4: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G5;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G4;
        endcase;
State G5: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G6;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G5;
        endcase;
State G6: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G7;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G6;
        endcase;
State G7: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G8;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G7;
        endcase;
State G8: case  ((BEAM == ^b10) & (FILT == ^b1110))   : G0;
                ((BEAM != ^b10) # (FILT != ^b1110))   : G8;
        endcase;
test_vectors                "Verifies devices operation
([SYSCLK,START]  ->  [GATE,BEAM,FILT])
[X,H] -> [G0,B0,F0];    [C,L] -> [G0,B0,F1];    [C,L] -> [G0,B0,F2];[C,L] -> [G0,B0,F3];
[C,L] -> [G0,B0,F4];    [C,L] -> [G0,B0,F5];    [C,L] -> [G0,B0,F6];[C,L] -> [G0,B0,F7];
[C,L] -> [G0,B0,F8];    [C,L] -> [G0,B0,F9];    [C,L] -> [G0,B0,F10];[C,L] -> [G0,B0,F11];
[C,L] -> [G0,B1,F0];    [C,L] -> [G0,B1,F1];    [C,L] -> [G0,B1,F2];[C,L] -> [G0,B1,F3];
[C,L] -> [G0,B1,F4];    [C,L] -> [G0,B1,F5];    [C,L] -> [G0,B1,F6];[C,L] -> [G0,B1,F7];
[C,L] -> [G0,B1,F8];    [C,L] -> [G0,B1,F9];    [C,L] -> [G0,B1,F10];[C,L] -> [G0,B1,F11];
[C,L] -> [G0,B2,F0];    [C,L] -> [G0,B2,F1];    [C,L] -> [G0,B2,F2];[C,L] -> [G0,B2,F3];
[C,L] -> [G0,B2,F4];    [C,L] -> [G0,B2,F5];    [C,L] -> [G0,B2,F6];[C,L] -> [G0,B2,F7];
[C,L] -> [G0,B2,F8];    [C,L] -> [G0,B2,F9];    [C,L] -> [G0,B2,F10];[C,L] -> [G0,B2,F11];
[C,L] -> [G1,B0,F0];            "Gate output changes state here
@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G2,B0,F0];@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G3,B0,F0];
@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G4,B0,F0];@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G5,B0,F0];
@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G6,B0,F0];@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G7,B0,F0];
@REPEAT ^D35 {[C,L] -> [X,X,X]; } [C,L] -> [G8,B0,F0];
@REPEAT ^ D35 {[C,L] ->  [X,X,X]; } [C,L] -> [G0,B0,F0]; "Check the final state rolls over to the first
                "This completes a run-through of ALL states, the following 2 vectors retest reset (START)
[C,L] -> [G0,B0,F1]; [C,H] -> [G0,B0,F0];
end Statexam
```

Figure 16. Triple State Machine (continued)

CYPRESS SEMICONDUCTOR

Using ABEL to Program the CY7C330

This application note describes how to access all the features of the Cypress CY7C330 using ABEL. Examples show how to put the features to work. ABEL is a versatile logic design tool that can program over 300 different devices.

The Cypress CY7C330 is a powerful PLD. Features such as input and buried registers allow the CY7C330 to fit into a wide variety of applications. Although, the same features can make programming the device a challenge, this application note should minimize the challenge.

ABEL 3.0 Bug

If you are still using ABEL 3.0 and trying to program the CY7C330 for the first time, note that the supplied device driver has a fatal flaw. Both Cypress and Data I/O offer updated device drivers.

ABEL 3.1 also supplies a correct device file, with a new name. P330 was used for revision 3.0, and P330A for 3.1, although 3.1 still compiles with the P330 device name. The only difference between these two device files is the syntax for specifying the shared feedback mux.

Input Registers

The CY7C330 contains 11 dedicated input registers. An input register is also associated with each one of the 12 output registers (more on this later).

Pin 3 can serve as an input register or a clock input. In fact, ten of the 11 input registers can be clocked from two different sources: pins 2 or 3. You can program the choice of the clock source individually, on a register-by-register basis. If an application requires only one input clock source, you can use pin 3 as a normal input. If an application requires both input clocks, however, you must use pin 3 as a clock input. A configuration bit must be changed to enable pin 3 as a clock input.

Like pin 3, pin 14 is a dual-function pin; it can be used as a registered input or a global, asynchronous, output-enable line. Control of the CY7C330's output enable can originate from the product term array or from pin 14. You can program the choice on a register-by-register basis. (The I/O macrocell section of this application note gives more information on controlling the output enable.)

You can control the input-register clock mux in two ways. The most descriptive way is to use the ".C" suffix, as shown in the DEMO330.ABL example file supplied with ABEL. This method works for the dedicated input registers (pins 4 - 7 and 9 - 14) but does not work in ABEL 3.1 for the input registers in the I/O macrocells. The reason for this problem is that for the 12 I/O macrocells, ABEL thinks the clock mux is for the output or state register and not the input register.

Thus, the recommended method for controlling the input-register clock mux is to use macro commands. The macro file supplied with ABEL 3.0 does not include the complete macro list needed to program all the clock muxes, but you can get the complete file from Cypress. This file, P330.INC, contains the macros needed to program all the clock muxes, including the input registers. A listing of the macro file appears in *Appendix A*. ABEL versions 3.1 and higher come with the complete macro file.

After you reference the macro file in the ABEL source file, the command CLK2 must enable the pin-3 clock. Then you set specific clock muxes by entering CLK2_n, where n is the input register's pin number. For example:

```
LIBRARY      'p330';
        "allows use of p330.inc macro file
CLK2;
        "enables pin 3 as a clock input
CLK2_5 ;
        "pin 5 input reg uses the pin 3 clock
CLK2_15;
        "pin 15 input reg uses the pin 3 clock
```

You do not need a macro statement to specify the use of clock 1 (pin 2) for input registers, because clock 1 is the clock mux default setting for both the dedicated input registers and the I/O macrocell input registers.

ABEL handles the accessing of data from one of the dedicated input registers (pins 3 - 14) the same as for a straight buffered input. The only difference is that for the dedicated input registers, input data is not available in the product term array until after the appropriate input clock pulse is received.

Controlling the Output Enable

You specify an output enable by appending the suffix ".OE" to the appropriate pin name. You must define whether control of the output enable mux comes from pin 14 or the product term array. Configuration bit C0 controls this choice, and you make the selection using the ISTYPE statement:

```
OUT1,OUT2,OUT3,OUT4   pin 15,16,17,18 ;
        "I/O pins
OUT1.OE,OUT2.OE ISTYPE 'EQN';
        "OE is product-term controlled
OUT3.OE,OUT4.OE ISTYPE 'PIN';
        "OE is controlled by pin 14
```

When controlling the output enable with a product term, you have the option of setting it always on, always off, or making it a combination of some number of inputs or outputs. All three choices are illustrated in this code:

```
[OUT1.OE,OUT2.OE] = [1,1];
        "permanently enable outputs
OUT3.OE = 0;
        "permanently disable output
OUT4.OE = IN1 & IN2 & OUT1 ;
        "OE controlled by IN1, IN2, OUT1
```

Using Set and Reset

The CY7C330 has global synchronous set and reset capability. When used, it sets or resets all 12 state registers and the four buried registers. Watch out for two conditions when using set or reset: First, when you reset the registers, all the outputs go High if they are enabled because of the inverter between the state register and the output (*Figure 1*). Second, be aware that the reset does not occur for two clock pulses if an input is designated as the set/reset pin. This occurs because the reset data must be clocked into the product term array using one of the two input clocks first. The output registers must then be clocked to cause the reset or set to occur.

You can access the CY7C330's set and reset capability in two ways: First, you can append the suffix ".PR" for preset or ".RE" for reset to any output-pin or buried-register node name. The syntax is:

```
OUT1, INP1, INP2   PIN   16, 5, 6;
OUT1.PR  =  INP1 ;
        "preset all output nodes on INP1=1
OUT.RE = INP2;
        "reset all output nodes on INP2=1
```

The second way to utilize set and reset is to employ the node notation shown in the following code, in which the set and reset product terms are designated node 30 and 29, respectively.

```
SET, RESET    NODE 30, 29 ;
SET = INP1 ;
        "preset all output nodes on INP1=1

RESET = INP2;
        "reset all output nodes on INP2=1
```

Even though the reset and preset functions are synchronous, an error occurs while parsing the equations if you use the ":=" notation, which signifies a registered operation.

Using the Macrocell as an Output Only

When using the I/O macrocell as an output, you need to consider two parameters. The first is the setting of the macrocell feedback mux, as controlled by configuration bit C1. The second parameter is the control of the output enable, as described in the previous section. As with the output-enable control, you set the configuration bit for the feedback mux using the ISTYPE statement. When the input register is not used, data from the output register is typically fed back to the product-term array through the macrocell feedback mux. When this feedback arrangement is used, ISTYPE is followed by the FEED_REG attribute:

```
OUT1            PIN 15 ;
        "located in initial pin definitions

OUT1 ISTYPE 'FEED_REG';
        "sets C1=0, allowing feedback mux
        "to pass data from state register

OUT1 :=  INP1 $ ((INP1 & INP2 )# INP3);
        "sample eq from 'equations' section
```

The ABEL default for the feedback mux configuration bit (C1) is to take data from the state register. Thus the "ISTYPE 'FEED_REG';" statement is not required, but it is recommended that the defaults be documented.

Using the Macrocell as an Input Only

When you use the I/O macrocell as an input register, the syntax differs from that of the previous example. Specifically, the output buffer must be three-stated, and the macrocell feedback mux must be set to accept data from the input register (C1 must be set to 1). The following example assumes that the output register is not used at

Table 1. Node Numbers for Shared Input Multiplexers

Node Number	Mux Between Pins
35	15, 16
36	17, 18
37	19, 20
38	23, 24
39	25, 26

all. Keep in mind that the input register clock defaults to clock 1 (pin 2) unless specifically changed.

```
INP1, INP2, OUT2        PIN 5, 15, 16 ;

INP2  ISTYPE 'FEED_PIN';
        "set C1=1, allowing feedback mux to
        "take data from the input register

INP2.OE  ISTYPE  'EQU';
"set C0=0 for product term OE

EQUATIONS

INP2.OE = 0 ;
        "three-state output buffer permanently
OUT2 :=  INP1 & INP2;
```

Shared Input Mux

Each pair of I/O macrocells has a shared input mux. This mux feeds data from the input pin into the product-term array if both registers are fed back in an I/O macrocell. A configuration bit (C3) controls whether the mux's input is from an even- or odd-pin-number macrocell. The ABEL default is that the data is supplied from the even-pin-number macrocell. Changing to an odd pin requires that you invoke macros located in the P330.INC file. (The example in the next section shows how to make this change.)

The purpose of the shared input mux is to provide another input path to the product-term array, when registered feedback is used, without losing input capability.

Using the Input and Output Registers

When using both the input and output registers in the I/O macrocell, the most difficult task is to get the data into the product-term array.

You can use two muxes to feed data from the registers into the product-term array. The state-register information must be fed back through the feedback mux controlled by configuration bit C1. You can route input-register data through the feedback mux or through the shared input mux (*Figure 1*).

The state-register output is referred to by the pin name associated with the macrocell. The data clocked into the input register is referred to by using the node name assigned to the shared input mux. *Table 1* lists the node numbers of the shared input muxes.

In ABEL, the configuration bit controlling the shared input mux (C3) defaults to an even I/O pin. When the input data is on an odd pin, you can use a macro in the P330.INC macro file to change the C3 configuration bit.

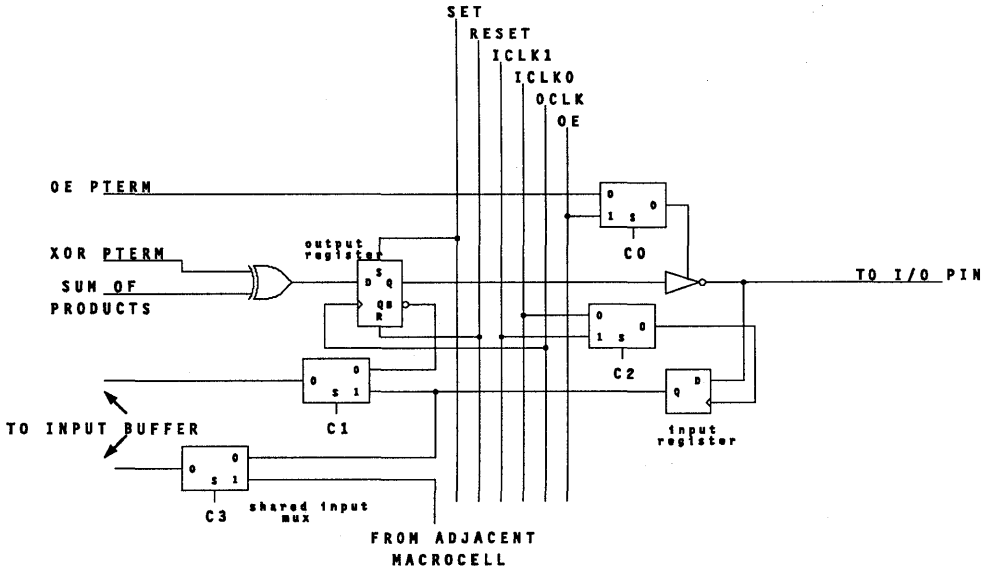


Figure 1. The CY7C330 Macrocell

The following example also uses clock 2 (pin 3) to clock the input register:

```
BREG            PIN 15 ;
        "BREG is output register for pin 15

INP1 NODE 35 ;
        "INP1 is the input register for pin 15
BREG ISTYPE 'FEED_REG';
        "C1 is set to 0, mux routes Q of BREG
BREG ISTYPE 'EQN';
        "OE is product term controlled
LIBRARY 'P330' ;
        "enables use of the P330.INC file
CLK2;
        "enables pin-3 clock
CLK2_15;
        "enables CLK2 on pin-15 input reg
FEEDPIN_15;
        "shared input mux control bit (C3) set
        "This gives pin 15 an input path
EQUATIONS

BREG.OE  =  0 ;
        "disable output

BREG  :=  BREG $ (INP1 & INP2);
        "BREG is fed back and INP1 is an input
```

The Exclusive-OR Gate

The CY7C330 provides an exclusive-OR (XOR) gate on the D input of the 12 I/O-macrocell output registers and the four buried registers. You can use this gate for two purposes. First, you can invert the polarity of a signal going into the output register. This inversion is accomplished by setting one of the XOR inputs to a logic 1, using the ABEL "$" symbol for XOR. In ABEL, you can use the following format:

```
OUT1 := 1 $ (INP1 & INP2 & INP3);
```

In ABEL versions before 3.1, however, the reduction algorithms do not recognize a 1 mixed with variables in an equation. The equivalent expression for earlier versions is:

```
OUT1 := (INP1 # !INP) $ (INP1&INP2&INP3);
```

The second use for the XOR gate is to emulate JK or T flip-flops in software. T flip-flops are more efficient than D flip-flops for implementing counters and state machines. You can emulate T-type flip-flops by feeding back the output register's Q output and tying it to the XOR product term. The sum-of-products input to the XOR becomes the T input (*Figure 2*). You can configure this emulation with Boolean equations:

```
TFLOP := TFLOP $ (T input expression);
```

where "T input expression" is a legal sum-of-products expression. A JK flip-flop is emulated using the same configuration, and the relationship:

```
T = J!Q # KQ
```

The second way to configure an output flip-flop as a T-type flop is to use an ISTYPE statement such as the one in the next example. The following syntax describes a simple 2-bit counter:

```
CLK, INSTB, !OE         PIN 1, 2, 3, 14;
Q0, Q1                  PIN 28, 27;
Q0, Q1          ISTYPE          'REG_T' ;
Q0.OE, Q1.OE    ISTYPE          'PIN';
CNT = [Q1,Q0];
EQUATIONS
Q0.OE = OE;
Q1.OE = OE;
CNT = (CNT + 1);
```

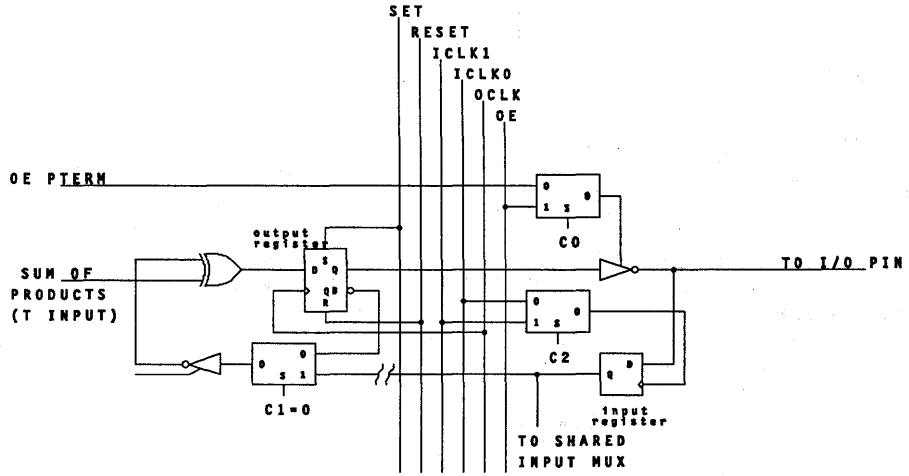


Figure 2. The CY7C330 Macrocell as a T-Type Flip-Flop

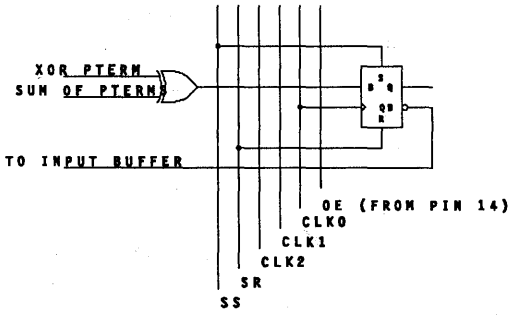


Figure 3. A Buried Register

Buried Registers

As mentioned before, the CY7C330 contains four buried registers. You access these registers by assigning a name to the buried register node number. *Table 2* lists the node numbers, and *Figure 3* shows a diagram of a buried register.

To use a buried register, assign a name to the node and use it as if it were a normal output. The only difference is that the I/O macrocell has an inverter between the state register and the output pin, which causes ABEL to handle the polarity differently (more on this in the next section).

Polarity Conventions

As shown in later examples, you typically do not have to worry about signal polarity except when sending data to an output pin. This is because all data enters the product-term array in both the non-inverted and inverted states. ABEL chooses the right polarity to obtain the output as specified by the equations.

When you export data from the device via an output pin, polarity is more critical—especially when using the set or reset. As shown by the block diagrams, the macrocell includes an inverter between the output register and output pin. Therefore, if you use the reset capability, the registers' Q output goes Low, and the output pins go High. If your application requires all the outputs to start out Low, use preset instead of reset.

In the following example, the output is defined as positive, and a 1 and a 0 are passed through the device.

Table 2. Node Numbers of Buried Registers

Buried Register	Node Number	Product Terms
1	31	13
2	32	17
3	33	11
4	34	19

ABEL compensates for the lack of inversion in the output by inverting the data coming out of the input register.

```
"inputs
CKS, CK1, CK2, INP   PIN  1, 2, 3, 4;
"output
OUT  PIN  15 ;

EQUATIONS
OUT  :=  INP ;

TEST_VECTORS
([CKS,CK1,CK2,INP]  ->  [OUT])
 [0,   C,   0,    0]  ->  [X];
 [C,   0,   0,    X]  ->  [0];
 [C,   0,   0,    X]  ->  [0];
 [0,   C,   0,    1]  ->  [0];
 [C,   0,   0,    X]  ->  [1];
 [C,   0,   0,    X]  ->  [1];
END
```

When using state machine syntax, ABEL does not handle the polarity of the buried registers correctly. Not only do the equations not work, but the simulation also fails. You can easily fix the problem, however, by negating the names in the node declaration:

```
CLK1, CLK2, CLK3  PIN  1,2,3 ;
INP, OUT          PIN  4,15 ;
        "hidden register declaration (negated)
!C1, !C2, !C3 NODE  31,32,33 ;
```

As with the state machine syntax, when using the "COUNT = COUNT +1" syntax, you also must invert the polarity of any buried registers. The easiest place to accomplish the inversion is at the node definitions statement, as shown in the previous example. Additionally, refer to the counter example at the end of this application note.

State Machine Syntax

ABEL supports state machine syntax on the CY7C330. The only drawback is that you can only use the toggle flip-flop emulation mode for very simple state machines. Up to revision 3.1, the results of using state machine syntax with T flip-flop emulation are unpredictable.

The T flip-flop is efficient for state machines because it holds its state unless told otherwise and thus needs a product term only for a state change. In contrast, a state machine using D flip-flops needs a product term both to change states and to hold states. Even with this limitation, the CY7C330 contains from nine to 19 product terms per output and usually handles a medium-size state machine with ease.

Simulation Caveat

Be aware of a limitation to what ABEL can simulate. Specifically, when writing simulation test vectors, you can use only one of the three clock lines on a single test-vec-

tor line. The following example does not simulate correctly:

```
TEST_VECTORS

([CKS,CK1,CK2,INP] -> [OUT])
[C,C,0,0] -> [0];
```

The following modified version does simulate correctly:

```
TEST_VECTORS
([CKS,CK1,CK2,INP]      -> [OUT])
[0,C,0,0]               -> [X];
[C,0,0,X]               -> [0];
```

ABEL supports the preload function. Refer to the 15-bit counter example for more information on how to use it.

16-Bit Up/Down Counter

This application, COUNTER6, is an example of a 15-bit up counter with a terminal-count output. The application shows how to use ABEL's "COUNT = COUNT + 1" syntax and corrects the polarity problem that crops up when combining normal I/O macrocell output registers and buried registers. This example also illustrates how to use the preload function. The ABEL source code for this example appears in *Appendix B*.

State-Machine-Based Modulo-11 Counter

This example is a state machine application implementing a modulo-11 counter using state machine syntax. This example again shows how to handle polarity using both normal registers and buried registers. *Appendix C* lists the ABEL source code for this example.

Appendix A. P330.INC -- Macro Listing

```
" P330.INC
"The following select Clock 2 (pin 3) for the Output Macrocell Input register.
        CLK2_28 macro () {FUSES[17030] = 1;}
        CLK2_27 macro () {FUSES[17034] = 1;}
        CLK2_26 macro () {FUSES[17037] = 1;}
        CLK2_25 macro () {FUSES[17041] = 1;}
        CLK2_24 macro () {FUSES[17044] = 1;}
        CLK2_23 macro () {FUSES[17048] = 1;}
        CLK2_20 macro () {FUSES[17051] = 1;}
        CLK2_19 macro () {FUSES[17055] = 1;}
        CLK2_18 macro () {FUSES[17058] = 1;}
        CLK2_17 macro () {FUSES[17062] = 1;}
        CLK2_16 macro () {FUSES[17065] = 1;}
        CLK2_15 macro () {FUSES[17069] = 1;}

"The following enables clock 2 (pin 3)
        CLK2   macro () {FUSES[17070] = 1;}
        CLK2_4 macro () {FUSES[17072] = 1;}
        CLK2_5 macro () {FUSES[17073] = 1;}
        CLK2_6 macro () {FUSES[17074] = 1;}
        CLK2_7 macro () {FUSES[17075] = 1;}
        CLK2_9  macro () {FUSES[17076] = 1;}
        CLK2_10 macro () {FUSES[17077] = 1;}
        CLK2_11 macro () {FUSES[17078] = 1;}
        CLK2_12 macro () {FUSES[17079] = 1;}
        CLK2_13 macro () {FUSES[17080] = 1;}
        CLK2_14 macro () {FUSES[17081] = 1;}

"The following program the C3 bit in the Output Macrocell and selects feedback from the lower pin.
        FEEDPIN_27 macro () {FUSES[17031] = 1;}
        FEEDPIN_25 macro () {FUSES[17038] = 1;}
        FEEDPIN_23 macro () {FUSES[17045] = 1;}
        FEEDPIN_19 macro () {FUSES[17052] = 1;}
        FEEDPIN_17 macro () {FUSES[17059] = 1;}
        FEEDPIN_15 macro () {FUSES[17066] = 1;}
```

Appendix B. ABEL Source Code for the 16-Bit Counter Example

```
module _counter6
title 'Counter application for CY7C330 application note - Cypress Semiconductor  June 19,1989'

counter6device   'p330' ;
                        " This is example of a 15 bit counter showing:
                        "1. How to handle the polarity when combining normal output registers and buried regs.
                        "2. How to use the 'count = count + 1' syntax.
                        "3. How to use preload for simulation vectors and handle the polarity inversion for the
                        "       buried registers.
" inputs pins
        clk,clk1,clk2,preset        pin       1,2,3,4 ;
" output pins
        c0,c1,c2,c3,c4,c5,c6 pin 15,28,26,17,24,19,20 ;
        c11,c12,c13,c14      pin  25,18,16,27 ;
        tci                  pin  23 ;
        spreset              node 30 ;
        !c7,!c8,!c9,!c10     node 31,32,33,34 ;
" macros
        c_cntr = [c14, c13, c12, c11, c10, c9, c8, c7, c6, c5, c4, c3, c2, c1, c0] ;
                        " this is used to handle the preload  inversion of the  buried registers. See test vectors below.
        c_cntrs = [c14, c13, c12, c11, !c10, !c9, !c8, !c7, c6, c5, c4, c3, c2, c1, c0] ;
        c,x,p           =       .c., .x., .p. ;
equations
        spreset         =       preset ;
        c_cntr   :=     (c_cntr + 1) ;
         tci            :=      (c_cntr == 2346) ;

                        " Example of using preset with simulation

test_vectors
([clk,clk1,preset,c_cntrs] -> [c_cntr,tci])
 [ 0 , 0 ,  x  ,  x     ] -> [  x     , x ];
 [ 0 , c ,  1  ,  x     ] -> [  x     , x ];
 [ c , 0 ,  x  ,  x     ] -> [  0     , 0 ];
 [ 0 , c ,  0  ,  x     ] -> [  0     , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  1     , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  2     , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  3     , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  4     , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  5     , 0 ];
 [ p , 0 ,  x  ,  62    ] -> [  x     , 0 ];
 [ 0 , 0 ,  x  ,  x     ] -> [  62    , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  63    , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  64    , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  65    , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  66    , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  67    , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [  68    , 0 ];
 [ p , 0 ,  x  , 2345 ] -> [  x     , 0 ];
 [ 0 , 0 ,  x  ,  x     ] -> [ 2345 , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [ 2346 , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [ 2347 , 1 ];
 [ c , 0 ,  x  ,  x     ] -> [ 2348 , 0 ];
 [ c , 0 ,  x  ,  x     ] -> [ 2349 , 0 ];

end
```

Appendix C. ABEL State Machine Source Code for Modulo 11 Counter

```
module _statem
title 'Application Note State Machine Example -  Cypress Semiconductor  5-12-89'

        statem   device   'P330';

        clk1,clk2,clk3    pin       1,2,3 ;
        c1,c2             pin       15,16 ;
        res               pin 4 ;
        reset             node 30 ;
        !c3,!c4           node 31,32 ;
        count             =         [c4,c3,c2,c1] ;
        c4,c3,c2,c1                 istype    'feed_reg' ;
        c,x,z,h,l         =         .c.,.x.,.z.,1,0 ;

                          " This is an example of implementing a modulo counter using state  machine syntax.
                          " This example also shows how to use the hidden registers.

                          " counter states
 s0 = ^b0000 ;  s3 = ^b0011 ;  s6 = ^b0110 ;  s9 =   ^b1001;
 s1 = ^b0001 ;  s4 = ^b0100 ;  s7 = ^b0111 ;  s10 = ^b1010 ;
 s2 = ^b0010 ;  s5 = ^b0101 ;  s8 = ^b1000 ;

equations
        c4.pr             =         res ;

state_diagram  [ c4,c3,c2,c1 ]
        state s0: goto s1 ;
        state s1: goto s2 ;
        state s2: goto s3 ;
        state s3: goto s4 ;
        state s4: goto s5 ;
        state s5: goto s6 ;
        state s6: goto s7 ;
        state s7: goto s8 ;
        state s8: goto s9 ;
        state s9: goto s10 ;
        state s10: goto s0 ;

test_vectors
([clk1,clk2,res] -> [count])
 [ 0 , c , 1 ] -> [ 15  ];
 [ c , 0 , 0 ] -> [  0  ];
 [ 0 , c , 0 ] -> [  0  ];
 [ c , 0 , 0 ] -> [  1  ];
 [ c , 0 , 0 ] -> [  2  ];
 [ c , 0 , 0 ] -> [  3  ];
 [ c , 0 , 0 ] -> [  4  ];
 [ c , 0 , 0 ] -> [  5  ];
 [ c , 0 , 0 ] -> [  6  ];
 [ c , 0 , 0 ] -> [  7  ];
 [ c , 0 , 0 ] -> [  8  ];
 [ c , 0 , 0 ] -> [  9  ];
 [ c , 0 , 0 ] -> [  10 ];
 [ c , 0 , 0 ] -> [  0  ];

 end
```

CYPRESS SEMICONDUCTOR

Using ABEL to Program the Cypress CY7C331

This application note describes how to program the CY7C331 using Data I/O's ABEL. Each section of the application note describes a configuration and presents the relevant ABEL source code. (You can obtain all the examples presented in this application note from the Cypress Bulletin Board at (408) 943-2954. Retrieve the file 331APNT.EXE; it unarchives itself automatically.)

The information presented here can simplify the jobs of circuit designers, who are under a lot of pressure to shorten design cycles and fit numerous functions into a small footprint. The latest programmable logic devices (PLDs) give you the ability to increase circuit density with a reduced design cycle. When you combine multiple types of PLDs from multiple vendors on the same board, using a general programmable logic compiler such as ABEL makes a lot of sense.

Unfortunately, as PLDs get more complex, the concept and implementation of a universal compiler becomes non-trivial. A compiler vendor such as Data I/O must define a syntax that is both easy to use and powerful enough to accommodate hundreds of different PLDs. The ABEL PLD compiler succeeds with a vast array of features. It does an admirable job of supporting over 300 different types of PLD source equations with a multitude of different architectures.

The architecture covered in this application note is that of the Cypress CY7C331. This device belongs to a family of high-speed, high-density, 28-pin PLDs. Features such as individual set, reset, and clock product terms for each of the 24 registers make the device one of the most versatile PLDs on the market today.

Controlling the Output Enable

The CY7C331 has two different methods of controlling the output enable on each of the twelve outputs (see the CY7C331 diagram in *Figure 1* of "Using the CY7C331 as a Waveform Generator"). Either pin 14 or a product term can control each output enable. Controlling the output enable by a product term means using any combination of inputs and outputs ANDed together. Because only one term is available, OR terms are not allowed in the equation.

The advantage to using pin 14 rather than a product term is that the pin enables or disables the output buffers 5 ns faster. This is because the output enable signal does not travel through the array.

Any I/O pin (pins 15 - 28) used on the left side of an equation, by default, has its output enable programmed as asserted. For example:

```
   I1, OUT15       PIN 1, 15;
EQUATIONS
   OUT15           = I1 ;
    is the same as
   I1, OUT15       PIN 1, 15;
EQUATIONS
   OUT15           = I1;
   OUT15.OE        = 1;
```

If you use the direct connection to pin 14, the signal must be configured as active Low. The way ABEL configures the output enable mux depends on the equations. If the right hand side of an ".OE" equation has just an inverted pin 14 on it, ABEL assumes you want to use the direct connection to pin 14. For example, the following equations use the direct connection to pin 14 (C0 = 1):

```
   I14, I15              PIN 14,15;
   OUT15, OUT16          PIN 15,16;
EQUATIONS
   [OUT15,OUT16].OE  =  !I14;
```

The same example uses the product term array if you change the equation to:

```
 [OUT15,OUT16].OE =  I15; OR
 [OUT15,OUT16].OE = I14 & I15;
    or even:
 [OUT15,OUT16].OE = I14;
```

In some cases, you might want to use pin 14 to control the output enable, but for timing reasons use the product term array instead of the direct connection. ABEL allows you to do this by using an ISTYPE statement. In the following example, the output enable for

pin 16 goes through the product term array, and pin 17 uses a direct connection:

```
   I2, I14                PIN 2,14;
   OUT16, OUT17           PIN 16,17;
   OUT16.OE               ISTYPE 'EQN' ;
EQUATIONS
   [OUT16,OUT17].OE   = !I14;
   OUT16              = I2 ;
   OUT17              = I2 ;
    TEST_VECTORS
   ([I2,I14]      ->      [OUT16,OUT17])
   [ X , 0 ]      ->      [ Z , Z ];
   [ 0 , 1 ]      ->      [ 0 , 0 ];
   [ 1 , 1 ]      ->      [ 1 , 1 ];
```

Note that in most cases when an output register is buried and the I/O pin serves as an input, ABEL does not automatically disable the output enable. In fact, you cannot disable the output enable unless you define it with an ISTYPE 'EQN' statement.

In the following example, the OUT15.OE = 0 statement does not disable the output enable unless the statement is preceded with OUT15.OE ISTYPE 'EQN':

```
" The following code is for testing
" polarity on the CY7C331.
"input pins
   I1,CLK              PIN 1,2;
   RES,PRE,OE          PIN 4,5,6;
    "output pins
   OUT15,OUT16         PIN 15,16;
   OUT17,OUT18         PIN 17,18;
    "constants
   C,X,Z               = .C., .X., .Z.;
   OUT15.OE            ISTYPE 'EQN';
EQUATIONS
" the example below shows using the
" feedback from register 15 to
" control the preset and set of
" register 16.
   OUT15      = I1;
   OUT15.C    = CLK;
   OUT15.RE       = RES;
   OUT15.PR       = PRE;
" The following statement is ignored
" without previous istype'eqn'.
   OUT15.OE   = 0;
   OUT16.RE   = OUT15;
   OUT16.PR   = !OUT15;
   OUT16.Oe   = 1;

TEST_VECTORS
   ([I1,CLK,RES,PRE]    ->    [OUT15,OUT16])
   [ 0, 0, 0, 0 ]           ->    [ Z , 1 ];
   [ 0, C, 0, 0 ]     ->    [ Z , 0 ];
   [ 1, C, 0, 0 ]     -> [ Z , 1 ];
" This tests what happens to the
" polarity of the register feedback
" when you go from register to
" transparent.
   [ 0, 0, 1, 1 ]     ->    [ Z , 0 ];
   [ 1, 0, 1, 1 ]     ->    [ Z , 1 ];
```

In general, it is advisable to use the ISTYPE 'EQN' for all I/O pins that use a product term to control the output enable, especially when trying to disable an output buffer.

Registered Output Only

You can use the CY7C331 macrocell as a registered output, without using the input register, as illustrated in the following example:

```
"input pins
   D_INP, CLK              PIN 1, 2 ;
"output pins
   OUT15              PIN 15 ;
"constants
   C, X, Z            = .C., .X., .Z. ;
EQUATIONS
   OUT15              := D_INP ;
   OUT15.C        = CLK ;
TEST_VECTORS
   ([D_INP,CLK]       ->    OUT15)
   [ X, 0 ]           ->    1;
   [ 0, C ]           ->    0;
   [ 1, C ]           ->    1;
```

As shown in this example, the minimum requirement to configure an output into a register is the OUTPUT := INPUT equation and an equation describing where the clock is coming from. The latter is necessary because the CY7C331 has no dedicated clock pin.

Because the following equations are ABEL defaults, you do not need to explicitly define them:

```
   OUT15.RE = 0 ;          "disable reset
   OUT15.PR = 0 ;          "disable preset
" permanently enable output buffer
   OUT15.OE = 1 ;
```

The next example uses all the output register's features. For example, you can dynamically switch from registered mode to combinatorial and back to registered. Although the ABEL simulation always shows the register returning to the same state when switching from combinatorial to registered mode, the actual state varies from device to device.

Also note that this example adds OUT17 to show that even when the pin 15 output buffer is disabled, the register's state still feeds back to the product term array via the feedback mux. The ABEL default for the feedback mux in the registered mode is to take information from the register (C1 = 0).

```
"input pins
   D_INP, CLK              PIN 1, 2 ;
   RES, PRE, OE            PIN 3, 4, 5 ;
"output pins
   OUT15, OUT16            PIN 15, 16 ;
   OUT17, OUT18            PIN 17, 18 ;
"constants
   C, X, Z                 = .C., .X., .Z. ;
```

```
EQUATIONS
" OUT15 is using the output register in both registered
"and combinatorial mode by manipulating the
" set and reset terms.
   OUT15                  := D_INP ;
   OUT15.C                = CLK ;
   OUT15.RE               = RES ;
   OUT15.PR               = PRE ;
   OUT15.OE               = OE ;
   OUT17                  = OUT15 ;
TEST_VECTORS
   ([D_INP,CLK,RES,PRE,OE]      -> [OUT15,OUT17])
    [0 , 0  , 0 , 0 , 0]        -> [ Z , 1 ];
"with no external help, the registers initialize to the
"reset state, which means the outputs are high,
" because of the non-bypassable inverter in
"the output path.
    [ 0 , 0 , 0 , 0 , 1]        -> [ 1 , 1 ];
    [ 0 , 0 , 0 , 1 , 1]        -> [ 0 , 0 ];
    [ 0 , 0 , 1 , 0 , 1]        -> [ 1 , 1 ];
    [ 0 , C , 0 , 0 , 1]        -> [ 0 , 0 ];
    [ 1 , C , 0 , 0 , 1]        -> [ 1 , 1 ];
" The register becomes combinatorial
" when the reset and preset are both asserted
    [ 0 , 0 , 1 , 1 , 1]        -> [ 0 , 0 ];
    [ 1 , 0 , 1 , 1 , 1]        -> [ 1 , 1 ];
" this is the state the register returned to
" when going from combinatorial to registered mode.
    [ 0 , 0 , 0 , 0 , 1]        -> [ 0 , 0 ];
```

Remember that the ABEL default for the feedback mux in the registered mode is to take information from the register (C1 = 0). This is not the case when you configure the output register as transparent, however, as shown in the next example.

Combinatorial Output Only

ABEL allows you to configure the output register as transparent by using the "=" symbol instead of ":=" in the equations, as this example shows:

```
"input pins
   I1                     PIN 1;
"output pins
   OUT15                  PIN 15;
"constants
   C, X, Z                = .C., .X., .Z. ;
EQUATIONS
   OUT15                  = I1 ;
TEST_VECTORS
   ( I1          ->       OUT15 )
    0            ->       0 ;
    1            ->       1 ;
```

In this example, the following equations are ABEL defaults, and you do not have to write them. Including these equations does not cause an error.

```
   OUT15.PR               = 1;  "set and reset
   OUT15.RE               = 1;  "high = transparent.
   OUT15.OE               = 1;  " enable on.
```

When you configure the output register as transparent, the input register path data is automatically fed to the product term array (C1 = 1). Because ABEL also defaults to transparent input registers, the data fed to the product term array is not the same as the registered output data.

You can feed data back to the product term array from before the output buffer—even when the output register is configured as transparent—by using an IS-TYPE 'FEED_REG' statement:

```
 "input pins
   I6, I7                 PIN      6,7 ;
"output pins
   OUT16, OUT18           PIN      16, 18;
"constants
   C, X, Z                = .C., .X., .Z. ;
   OUT16                  ISTYPE   'FEED_REG';
EQUATIONS
   OUT16                  = I6 ;
   OUT16.OE               = I7 ;
   OUT18                  = OUT16 ;
TEST_VECTORS
   ([I6,I7]               ->       [OUT16,OUT18])
   [ 0, 0]                ->       [ Z , 0 ];
   [ 1, 0]                ->    [ Z , 1 ];
   [ 0, 1]                ->    [ 0 , 0 ];
   [ 1, 1]                ->    [ 1 , 1 ];
```

If you omit the FEED_REG statement, an error occurs in the simulation. The FEED_REG statement changes the feedback-mux configuration bit from One to Zero (C1 = 0).

Transparent Input Only

The ABEL 3.2 default is to make the input register transparent. Thus, to specify an I/O macrocell as a combinatorial input, place the specification on the right side of an equation:

```
"INPUTS
   INP16, OUT18           PIN 16, 18;
EQUATIONS
   OUT18                  = INP16 ;
TEST_VECTORS
   (INP16                 ->    OUT18)
    0                     ->    0 ;
    1                     ->    1 ;
```

In this example, only one operator (=) serves to configure both registers as transparent. This method works because the equals sign controls only the output register configuration (OUT18), which is possible because the default configuration for an input register is transparent. Changing the "=" to ":=" changes the pin-18 output register from transparent to registered, but does not affect the pin-16 input register.

The Macrocell as a Registered Input Only

To change an input register from transparent to registered, you configure the register using its node

number. *Table 1* lists the node assignment for each register.

To use an input register as a register, place the signal on the right side of the equation and add the rest of the terms needed. In the following example, INP17 is a registered input pin. The register itself is called INP17REG. OUT19 is a transparent or combinatorial output.

```
"pin definitions
   INP17, RESET                PIN 17, 3;
   SET, CLK                    PIN 4, 5;
   OUT19                       PIN 19;
   INP17REG                    NODE 145;
EQUATIONS
   OUT19          = INP17 ;
   INP17REG.C     = CLK ;
   INP17REG.PR    = SET ;
   INP17REG.RE    = RESET ;
TEST_VECTORS
   ([INP17,CLK,SET,reset]  ->   out19)
   [ X , X , 0 , 1]        ->    0 ;
   [ X , X , 1 , 0]        ->   1 ;
   [ 0 , C , 0 , 0]        ->   0 ;
   [ 1 , C , 0 , 0]        ->   1 ;
   [ 0 , X , 1 , 1]        ->   0 ;
   [ 1 , X , 1 , 1]        ->   1 ;
```

To access the data stored in an input register, use the pin name. Access the set, reset, and clock using the input register node name.

Burying the Output Register/Registered Input

The CY7C331 allows you to bury an output register and still use the pin as a registered input by using the shared-input mux. The CY7C331 provides a shared-input mux between pins 15 and 16, 17 and 18, 19 and 20, etc. Thus, there are three paths into the product term array for every pair of macrocells. You therefore cannot bury both of a pair's output registers and still use the pin as an input. If you bury the output register at pin 15 and use the pin for an input, for example, you cannot bury the output register at pin 16 and also use the pin for an input.

Use the pin name to access the information fed back to the product term array from the output register. Use the node number of the shared-input mux to access the input data coming from the pin and passing through the input register. The shared-input mux node number assignments appear in *Table 2*.

The shared-input mux can take information from one of the two macrocells. ABEL defaults to selecting the macrocell of the even pin number. However, macros are available that select the odd pin's macrocell. You can access these macros by using the following syntax:

```
LIBRARY      'P331';
FEEDPIN_27;
```

Table 1. CY7C331 Input Register Node Assignments

Pin Number	Register Node
15	143
16	144
17	145
18	146
19	147
20	148
23	149
24	150
25	151
26	152

The LIBRARY statement inserts a copy of all the possible CY7C331 macros into the source during compilation. You can observe the result by looking at the listing file (.LST). The FEEDPIN_27 statement selects pin 27 to pass through the shared-input mux, overriding the default, which is pin 28.

The following code is the complete listing of a test program that shows how to bury a register and employ the pin as an input, using macros to change the shared-input mux:

```
module _test3
title 'CY7C331 test programs for applications note
Cypress Semiconductor Inc. 3/16/90'
   TEST3   DEVICE 'P331';
"This is an example of burying the output register
"of a CY7C331 and using the I/O pin as an input.
"input pins
   I1, CLK1, CLK2          PIN 1, 2, 3;
   RES, PRE, OE            PIN 4, 5, 6;
   CLK3                    PIN 7;
"output pins
   OUT15, OUT16            PIN 15, 16;
   OUT17, OUT18            PIN 17, 18;
"constants
   C, X, Z                 = .C., .X., .Z.;
"LIBRARY statement is used to access the macros
"needed to change the shared-input mux selection.
LIBRARY 'P331';
"Data from pin 1 gets clocked through the buried
"register on pin 15, and output on pin 16.
"Output register 15 is configured as a register and the
"pin 16 output register is transparent.
"Data also gets input on pin 15 and output on pin 17.
"Both are configured as registers.
```

```
   INP15REG            NODE 143 ;
   INP15MUX            NODE 29 ;
   OUT15.OE            ISTYPE 'EQN';
   OUTPUTS        =  [OUT16,OUT17];
   FEEDPIN_15;
EQUATIONS
   OUT17          := INP15MUX ;
   OUT17.C        = CLK3 ;
   INP15REG.C     = CLK2 ;
   OUT15          := I1 ;
   OUT15.C        = CLK1 ;
   OUT15.RE       = 0 ; " disable reset,
   OUT15.PR       = 0 ; " preset, and oe
   OUT15.OE       = 0 ;
   OUT16          = OUT15 ;
TEST_VECTORS
([I1,CLK1,OUT15,CLK2,CLK3]  -> [OUTPUTS])
 [X, 0, 0 , 0,0 ]                -> [1,1];
 [0, C, X , 0,0 ]                -> [0,1];
 [1, C, X , 0,0 ]                -> [1,1];
 [X, 0, 0 , C,0 ]                -> [1,1];
 [X, 0, X , 0,C ]                -> [1,0];
 [X, 0, 1 , C,0 ]                -> [1,0];
 [X, 0, X , 0,C ]                -> [1,1];
END
```

The ABEL 3.2 compiler contains a bug that relates to this example. If you remove the line OUT15.OE ISTYPE 'EQN';, the code compiles and simulates correctly. However, if you look at the resulting JEDEC map for the equations, the output buffer for pin 15 is enabled, which should cause the simulation to fail. Contact Data I/O for more information.

When you use macros, be cautious about several aspects of ABEL. In equations, for instance, the ABEL parser allows spaces between the end of the equation and the semicolon. However, you must place a semicolon immediately after a library statement and a macro. The parser does not allow a space between a semicolon and a library statement or a macro.

Additionally, because the key words of the macros that are accessed using the library statement are in upper case, you must put all references to the macros (e.g., FEEDPIN_27) in upper case. This is the only place where ABEL is case sensitive.

Finally, although you can put the library statement anywhere in the source code's declaration section, you must put macros last in the declaration section, before the equations section.

Table 2. CY7C331 Shared Input Mux Node Assignment

Pin Numbers	Shared Input Mux Node
15/16	143
17/18	144
19/20	145
23/24	146
25/26	147

Transparent Output with Registered Input

This example shows how to configure a buried transparent output register with a registered input. As described in the earlier section on transparent output registers, when you configure the output as transparent, the feedback to the product term array passes through the input register, unless programmed otherwise. The following code shows how to override the default using the ISTYPE 'FEED_REG' statement.

(Note that in the input section of the simulation, OUT15 represents the data being input on pin 15. This representation is somewhat confusing because in the equations OUT15 refers to the information coming from the pin-15 output register. See the simulation section of this application note for an explanation of this apparent discrepancy.)

```
"input pins
   I1, CLK2                PIN  1, 2;
   CLK3                    PIN  3;
"output pins
   OUT15, OUT16            PIN  15, 16;
   OUT17, OUT18            PIN  17, 18;
"constants
   C, X, Z                 =  .C., .X., .Z.;
LIBRARY 'P331';
"Input data from pin 1 goes through the buried
"register on pin 15, and is output on pin 16.
"Output registers 15, 16 are configured as transparent.
"Data is also input on pin 15 and output on pin 17.
"Pin 15 input, pin 17 output are registered.
   INP15REG                NODE 143 ;
   INP15MUX                NODE 29 ;
   OUT15                   ISTYPE 'FEED_REG';
   FEEDPIN_15;
EQUATIONS
   OUT17          := INP15MUX ;
   OUT17.C        = CLK3 ;
   INP15REG.C     = CLK2 ;
   OUT15          = I1 ;
   OUT15.OE       = 0 ;
   OUT16          = OUT15 ;
TEST_VECTORS
   ([I1,OUT15,CLK2,CLK3]      ->  [OUT16,OUT17])
   [0,X ,0 ,0 ]                ->  [ 0,  1];
   [1,X ,0 ,0 ]                ->  [ 1,  1];
   [1,0 ,C ,0 ]                ->  [ 1,  1];
   [1,X ,0 ,C ]                ->  [ 1,  0];
   [1 ,1 ,C ,0 ]               ->  [ 1,  0];
   [1 ,X ,0 ,C ]               ->  [ 1,  1];
    "end
```

Using the CY7C331 for Counting

You can use the CY7C331 to create a synchronous counter. The only limitation to using the device in a synchronous mode is that all feedback must be internal to the part, because the input-data hold time is not compatible with the output-data hold time.

ABEL provides many ways to implement a counter, including describing it explicitly in D or T flip-flop form.

The following example shows how to use the "count = count + 1" capability with the CY7C331 to implement a basic counter. The ABEL compiler uses the CY7C331's XOR gate to implement T flip-flops without any external instructions such as ISTYPE 'REG_T'.

```
"input pins
   I1, CLK2, CLK3          PIN  1, 2, 3;
   RES, PRE, OE            PIN  4, 5, 6;
"output pins
   OUT15, OUT16            PIN  15, 16;
   OUT17, OUT18            PIN  17, 18;
"constants
LIBRARY 'CONSTANT';
   COUNT  =[OUT18, OUT17, OUT16, OUT15];
EQUATIONS
" Example of 4-bit counter
" that starts and wraps around at 15.
   COUNT.C      =  CLK2;
   COUNT        :=  COUNT + 1;
" Example of how to use set and reset with this form
   COUNT.RE     =  RES;
   COUNT.PR     =  PRE;
TEST_VECTORS
   ( CLK2          ->  COUNT)
     0             ->  15;
     C             ->  0;
     C             ->  1;
     C             ->  2;
     C             ->  3;
TEST_VECTORS
   ([CLK2, RES, PRE]       ->  COUNT)
    [ 0, 0,0]              ->   3;
    [ 0, 0,1]              ->   0;
    [ 0, 1,0]              ->   15;
    [ 0, 0,1]              ->   0;
    [ C, 0,0]              ->   1;
    [ C, 0,0]              ->   2;
    [ C, 0,0]              ->   3;
    "end
```

Polarity Issues

The CY7C331's outputs do not have programmable polarity control in the same sense as the 22V10. The CY7C331 has a hard-wired inverter between the output register and the output pin that results in an active low output. You generally control the device's polarity using the XOR gate located in front of the output register.

ABEL makes polarity control transparent by allowing you to write equations with both positive- and negative-polarity outputs. Most of the examples in the previous sections, for instance, had active-High outputs. But hard-wired polarity becomes an issue when using set and reset. Keep in mind that a reset causes the output to go High.

ABEL takes care of the necessary inversions in the device to get the correct output polarity. This operation can be tricky when the internal feedback from a register controls another register's set or reset. Because both polarities are available in the product term array, it is not obvious which polarity should be used. Refer to the last example in the "Controlling the Output Enable" section of this application note for an example of indirect set and reset control.

Although the CY7C331 has active-Low outputs, defining the outputs active High (using OUT15 ISTYPE 'POS') sometimes causes ABEL's Reduce module to create equations that suit the CY7C331 better. This effect is especially true when you use the XOR gate. Refer to pages 3 - 4 in the *ABEL 3.2 User Notes* for more information.

Simulation

Simulation is very important with a part as versatile as the CY7C331. All the examples in this application note have been simulated to verify their function.

The ABEL simulator is powerful enough to simulate most of the configurations possible with the CY7C331. For example, the simulator supports multiple clock inputs controlling different registers. An application that illustrates this capability is a ripple counter. This counter has the clock input driven from the previous stage's output, with the least-significant bit driven by an external clock.

The following is an example of a 4-bit decrementing ripple counter implemented in the CY7C331.

```
"input pins
   CLK2, RESET             PIN  2, 3;
"output pins
   OUT15, OUT16            PIN  15, 16;
   OUT17, OUT18            PIN  17, 18;
"constants
   LIBRARY 'CONSTANT';
   COUNT  =[OUT18, OUT17, OUT16, OUT15];
EQUATIONS
"example of a 4-bit ripple counter that starts at 15
"and wraps around at 0.
   COUNT.RE                =  RESET;
   OUT15.C                 =  CLK2;
   OUT15                   :=  !OUT15;
   OUT16.C                 =  OUT15;
   OUT16                   :=  !OUT16;
   OUT17.C                 =  OUT16;
   OUT17                   :=  !OUT17;
   OUT18.C                 =  OUT17;
   OUT18                   :=  !OUT18;
```

```
TEST_VECTORS
   ([CLK2,RESET ]          ->  COUNT)
    [ 0 , 0 ]              ->   X;
    [ 0 , 1 ]              ->  15;
    [ C , 0 ]              ->  14;
    [ C , 0 ]              ->  13;
    [ C , 0 ]              ->  12;
    [ C , 0 ]              ->  11;
    [ C , 0 ]              ->  10;
    [ C , 0 ]              ->   9;
```

The CY7C331 powers-up with all registers in the reset state. The simulator, in most cases, mimics the device power-up characteristics. However, in certain applications, including the previous one, the simulation consistently initializes to a non-reset state.

Another interesting problem with simulating the CY7C331 is naming the input data when you bury the output register and use an I/O pin as an input. Although the input-register data is accessed in the equations using a node name, the ABEL simulator only works with pin names. In this application note's "Transparent Output with Registered Input" section, the example's equations section uses the node name (INP15MUX) to access the data being input on pin 15; the pin name (OUT15) is used to represent the data from the output register, which is fed back to the product-term array and then to pin 16. In the simulation section, however, OUT15 now represents the data being input on pin 15. The ABEL simulator is smart enough to know which data you are referring to.

Remember that simulation preload does not work with registered asynchronous parts such as the CY7C331 or 20RA10. However, if your design has an extra input, you can preset to a specific value by using the set and preset product terms individually. For example:

```
"input pins
   CLK, PRE                PIN  2, 3;
"output pins
   OUT15, OUT16            PIN  15, 16;
   OUT17, OUT18            PIN  17, 18;
   OUT19, OUT20            PIN  19, 20;
"constants
   LIBRARY 'CONSTANT';
   COUNT  = [OUT20, OUT19, OUT18,
      OUT17, OUT16, OUT15];
   PRESET = [OUT20, OUT19, OUT18,
      OUT17, OUT16, OUT15].RE;
   RESET  = [OUT20, OUT19, OUT18,
      OUT17, OUT16, OUT15].PR;
EQUATIONS
   COUNT.C                 = CLK;
   COUNT                   := COUNT + 1 ;
   WHEN (PRE == 1)
      THEN PRESET          = [1,0,1,0,1,1];
   WHEN (PRE == 1)
      THEN RESET           = [0,1,0,1,0,0];
TEST_VECTORS
   (CLK          ->   COUNT)
    0            ->   63;
    C            ->   0;
    C            ->   1;
    C            ->   2;
    C            ->   3;
TEST_VECTORS          "preload simulation test
   ([clk,pre]    ->  count)
    [ 0 , 0 ]    ->   3 ; "remembers from previous sim.
    [ C , 0 ]    ->   4 ;
    [ 0 , 1 ]    ->  43 ;
    [ C , 0 ]    ->  44 ;
    [ C , 0 ]    ->  45 ;
    "end
```

CYPRESS SEMICONDUCTOR

Using LOG/iC to Program the CY7C330

This application note provides you with a running start towards using the LOG/iC design synthesis tool for designs using the Cypress CY7C330 programmable logic device.

Of the steps required for implementing designs using PLDs, generating JEDEC files from high-level descriptions is probably the most time consuming. Unfortunately, the documentation that comes with many high-level synthesis packages does not provide enough detailed information to use advanced PLDs without a significant learning curve. Although the LOG/iC documentation is quite good, this application note should help flatten the LOG/iC learning curve further.

Isdata's LOG/iC is an advanced universal logic synthesis program that generates designs targeted for PROMs, PLDs, and gate arrays. The LOG/iC package's basic algorithms were developed in the Electrical Engineering Department of the University of Karlsruhe, West Germany. Although a relative newcomer to the PLD software market in the U.S., LOG/iC has become very popular in Europe.

LOG/iC is available for a variety of operating environments including PC DOS and SPARC-based SUNoS platforms. The software is available as four different packages with two options. The first (PLC) package supports PAL designs. It offers input in either equations or tables with syntax constructs that include address ranges and functional blocks. Also available are hexadecimal, decimal, octal, and binary representations.

The second (PLUS) package extends the first and supports the design of sequential controllers via inclusion of their FSM (finite state machine) syntax. This package includes an automatic test vector generation feature.

Package three (PERFECT) extends support to include designs partitioning across multiple devices. Package four (GATES) supports the design of multi-level-structure gate arrays by producing netlists from the various input formats. The two option packages offered are the Functional Verifier and PLD Database.

This application note deals with the functions available in package two (PLUS).

LOG/iC Language Overview

LOG/iC offers three different entry methods: Boolean equations, truth tables, and FSM. Declarations partition an input file into sections. Additionally, designs can be logically partitioned into functional blocks within a LOG/iC design file. These options are described briefly before proceeding to CY7C330-specific information.

Declarations

Declarations are directives to the LOG/iC compiler that identify the design, indicate the inputs and outputs, specify compiler options, assign pin numbers to variables, and specify the type of input format. These declarations separate the input file into discrete sections that describe the design's various aspects. LOG/iC declarations consist of a key word preceded by an asterisk (*).

The first section is the *Identification section, where you enter comments regarding the function of the design, etc. Variable declarations follow this information. LOG/iC supports input, output, local, and state variables for both Mealy and Moore machines. You can specify variables in ranges for compactness of expression, such as Address[0..31]. Variables can also have special function extensions that control the function of the device, such as RAS.OE.

Following the variable declarations is the design description. It is denoted by the declaration *Boolean Equation s, *Function-Table, or *Flow-Table for Boolean, truth table, and FSM entry methods, respectively. In the design description, you specify the circuit's function. Drawing an analogy to programming a computer in a high-level language, you could say that most of the other declarations describe the circuit's variables. The design description implements the algorithm to perform the function you wish to create.

Next are the *PLD, *PINS, *Run-Control, and *END sections. The *PLD declaration describes the device type targeted for use in this design. *PINS controls the assignment of the external variables to device pins. Finally, *Run-Control provides compiler directives, and *END signifies the end of the design file.

Table 1. LOG/iC Operators

Operation	Symbol	Example
Unregistered Output	=	Z = X;
Registered Output	:=	Z := X;
Negation	/	Z = /X;
AND	&	Z = X & Y;
OR	+	Z = X + Y;
XOR	#	Z = X # Y;
Constant 1	VCC	Z = VCC;
Constant 0	GND	Z = GND;

Boolean Design Entry

The simplest design entry method is by Boolean equations. *Table 1* shows the operators supported by LOG/iC in order of precedence. The labels X, Y, and Z can represent either a single variable or a range of variables.

Logic polarity often creates an amazing amount of confusion for a methodology that has only two values. LOG/iC removes the burden of considering whether a given signal is active Low or High, because Boolean equations always have a positive polarity. Thus, if a given input variable is specified without a '/', that variable is deemed to be true independently of the active level of the signal on the pin.

LOG/iC deals with signals that are active Low via the *Level declaration. You therefore write equations for an active-Low signal exactly the same as those for an active-High signal. The *Level section identifies the polarity of given input signals and manages negative/positive polarity issues for you.

Another useful aspect of Boolean entry is the use of ranges, which provide a compact method of referring to many variables in a succinct fashion. Typical examples include references to address or data buses. *Figure 1* shows an example of Boolean entry that utilizes variable ranges. This example features an 8-bit data bus whose values are captured in a register when a load command is issued.

Truth Table Entry

Truth table entry represents one of the most compact entry methods to describe a combinatorial system. With this entry format, you map the outputs as a function of the input variables. The basic format of truth tables appears in *Figure2*.

This example contains several noteworthy characteristics. The first is the ordering of the inputs and outputs. Note that the labels after the key word "*Function-Table" are comments, indicated by the leading semicolon (;). Thus, the ordering of the X and Y variables in the *X-Names and *Y-Names declarations specifies their ordering in the function table.

If you want some other ordering, you can specify it with a header. A header is a logical line preceded by the dollar sign symbol ($). When using a header, you separate the variables into fields delimited by commas.

```
*Identification
    Parallel Load Register with acknowledge
    MMA - Cypress Semiconductor
*X-Names
    Load, Data[0..7];
*Y-Names
    Qout[0..7],ACK;
*Boolean-Equations
    Qout[0..7] := Load & Data[0..7];
    ACK = Load;
```

Figure 1. Boolean Entry Example

Figure3 shows an example in which a header changes the variable ordering. This example uses two important constructs that can assist in reducing the logic design to the minimum number of product terms. The first construct is the Don't Care entries designated by a hyphen (-), which appear on both the input and output sides of the table.

The use of the Don't Care input is unique to the function table entry method and can significantly improve the compiler's ability to produce minimized logic. Note that Don't Cares are only available when using bit fields and that the table ends with word "REST" on the input side. The use of the rest statement stems from the fact that, to uniquely identify all possible input patterns with N input variables, you would require 2^N table entries. A single Don't Care in any given line represents two entry lines rather that one. The rest statement provides a brief way to specify all remaining possible input values and the output the values should produce.

The header line has an additional benefit beyond merely changing the order of bit data. You can also use the header line to indicate logical groupings of data as fields. Data that is not entered in groups must be entered as binary data. Grouped variables, however, can represent input data that is in binary, octal, decimal, or hexadecimal representations. Suffixes that indicate the

```
*Identification
    Truth Table Example
    MMA - Cypress Semiconductor
*X-Names
    X[6..1];
*Y-Names
    Y[1..4];
*Function-Table
;       Input Side                  :   Output Side
;   X   X   X   X   X   X   :   Y   Y   Y   Y
;   6   5   4   3   2   1   :   1   2   3   4
    0   -   -   -   1   0   :   1   -   0   -;
    1   1   1   0   0   0   :   0   0   0   1;
    0   1   -   -   0   1   :   1   -   1   0;
    1   -   -   1   1   1   :   0   1   -   1;
    0   -   1   -   0   0   :   1   0   0   1;
    0   -   -   1   0   0   :   1   0   0   1;
    REST                    :   1   -   -   0;
```

Figure 2. Truth Table Example

```
*Identification
    Truth Table Example with header
    MMA - Cypress Semiconductor
*X-Names
    X[6..1];
*Y-Names
    Y[1..4];
*Function-Table
    ; Input Side                  :  Output Side
$   X6, X5, X4, X3, X1, X2       :  Y4, Y3, Y2, Y1
    0,  -,  -,  -,  0,  1        :  -,  0,  -,  1;
    1,  1,  1,  0,  0,  0        :  1,  0,  0,  0;
    0,  1,  -,  -,  1,  0        :  0,  1,  -,  1;
    1,  -,  -,  1,  1,  1        :  1,  -,  1,  0;
    0,  -,  1,  -,  0,  0        :  1,  0,  0,  1;
    0,  -,  -,  1,  0,  0        :  1,  0,  0,  1;
    REST                         :  0,  -,  -,  1;
```

Figure 3. Truth Table with Header

data format appear in *Table 2*. It is important to note that a field is always totally occupied by a number; if necessary, leading zeros are added to completely fill the field.

In addition to fields, function tables allow the use of ranges. This feature permits efficient implementation of address decoders (*Figure 4*). The function table for this decoder specifies the address as ordered from 15..0. This order is significant because it is the same order as that of the hexadecimal numbers entered in the ranges below, when you view the hexadecimal numbers as individual bits. Also note the double parenthesis surrounding the outputs in the header line, which label this field as a bit field, eliminating the need for separating commas.

Finite State Machine Entry

FSM entry is probably the design methodology that correlates best with the CY7C330's target application as a high-speed state machine. LOG/iC's documentation defines an FSM as a circuit that has combinatorial logic and state registers of arbitrary type that feed back to a combinatorial array. Add to this definition multi-clocked input registers that minimize set-up and hold time requirements and you have a high-level description of the CY7C330.

More generally described, state machines have memory elements that describe the present condition and inputs that influence both the transition to the next state and the outputs. FSMs are typically classified in two general categories: Moore and Mealy machines. LOG/iC differentiates between these types by stating that machines whose outputs might change arbitrarily within a state, even without a clock pulse, exhibit "Mealy behavior." Moore machines, on the other hand, have outputs that change only with the state clock and are free of glitches. This output is typified as "Moore behavior" and is characteristic of the CY7C330. These outputs are tied to the state clock and are referred to in LOG/iC as Z-variables.

Table 2. Numeric Base Indicator Suffixes

B	Binary (default - can be omitted)
O	Octal
Q	Octal (alternate - to eliminate confusion between 0 and O)
D	Decimal
H	Hexadecimal

Four variables describe an FSM's behavior: the input variables' values, the present state, the output variables' values, and the next state. An FSM's variable declarations section has options for all these parameters. As in the previous entry methods, *X-Names describe the circuit's inputs. *Y-Names are values that exhibit Mealy behavior. *Z-Names are outputs that change relative to the state clock, as do the CY7C330's.

State information can assume one of two forms. The most common (and easiest) way to store the machine's state is to determine the total number of states required and dedicate N register bits (where 2^N = the number of states) to maintain state information. This method is reliable and produces discrete non-overlapping state assignments. The disadvantage is that you must dedicate register resources (i.e., macrocells) that might have served better in another capacity.

The second method available for state assignment is assignment of states based purely on the output values. This method requires more thought, as it is critical that all output patterns be unique. A design that might meet this criteria on first pass, might not be realizable if you add features — or remove them, in the case of undesirable "features."

```
*Identification
  Address Decoder Example
  MMA - Cypress Semiconductor
*X-Names
  Enable, Adr[0..15];
*Y-Names
  ROM[1..3], Port[1,2];
*Function-Table
  ; Input Side          : Output Side
$ Enable, (Adr[15..0]) : ((ROM[1..3], Port[1..2]));
  1,          -        : 111 --  ; Disabled
  0, 00000H..007ffH    : 011 11  ; ROM1 Selected
  0, 00800H..00fffH    : 101 11  ; ROM2 Selected
  0, 01000H..017ffH    : 110 11  ; ROM3 Selected
  0, 08000H..08007H    : 111 01  ; I/O Port 1 Selected
  0, 08008H..0800FH    : 111 10  ; I/O Port 2 Selected
  0, 0f800H..0ffffH    : 011 11  ; ROM1 (Shadow)
  REST                 : 111 11  ; Disabled
```

Figure 4. Address Decoder Function Table

```
*Identification
    Counter with 247 states and overflow signal
    MMA - Cypress Semiconductor
*X-Names
    Reset;
*Y-Names
    Overflow;
*Z-Names
    Q[1..8]
*Flow-Table
    S[1..247],   X 1, Y 0, F1          ; Reset condition
    S[1..246],   X 0, Y 0, F[2..247]   ; Count
    S[247],      X 0, Y 1, F1          ; Overflow
```

Figure 5. FSM Counter

LOG/iC can implement designs using either type of state assignment. The *State-Assignment directive provides the options of binary, number, gray, 1-out-of-N, and Z-variables. The binary option dedicates registers to state values and encodes the state values in binary. LOG/iC can do this encoding automatically, or you can specify the encoding explicitly.

Using the number option ensures that the binary code for each state is the same as the state numbers used in the high-level description, i.e., state 1 = 001, etc. The gray option assigns the states using gray coding to minimize transitions. 1-out-of-N assignment again uses registers but does not binary-encode states; instead, each discrete register represents a single state. This approach is especially demanding on macrocell resources but minimizes the number of state bits switching at a single clock edge. Finally, the Z-variable option allows the output values themselves to represent the states.

You enter the FSM design as a table after the directive *Flow-Table. Each line in the flow table has as many as four fields separated by commas. These fields represent the present state, inputs, outputs, and next state. Not all designs require all four states. Counters are good examples of applications that require only three fields to describe the machine, because the count value is the same as the state value.

The order in which the fields appear is not significant, because a letter indicating the field type precedes each field. The letters S, X, Y, and F indicate the state-number, input, output, and next-state fields, respectively.

A line in an FSM that describes part of a machine might look like this:

```
*Flow-Table
 S1, X 0 1, Y - 1, F2;
```

When in state 1, with inputs at 0 and 1, this machine causes the second output to go True and transitions to state 2. In this case, the first output is not relevant to the design. In a large machine, many of the inputs and outputs might not be relevant to a subset of the machine's sequence of operations. Rather than force you to specify the status of all variables, LOG/iC has a directive that lets you specify what variables are significant. This statement is called Relevant and stays in effect until the next Relevant statement or until the end of the design. As an example, you can describe the simple machine as:

```
*Flow-Table
 Relevant =  X1, X2 : Y2;
 S1, X 0 1, Y 1, F2;
```

Omitting Y1 from the Relevant statement indicates that Y1 is a Don't Care. If, instead, you want Y1 always to be off for the subsequent lines, you can state Y1 = 0.

Another powerful statement is Xrest. Similar to the REST statement in function tables, Xrest provides a brief way to assign all remaining non-specified input patterns and these conditions' desired output and next state.

You can also use ranges in flow tables for compact machine descriptions. In only three lines, the counter definition in *Figure 5* completely specifies a state machine with 247 states through the use of ranges. The only limitation is the number of states that LOG/iC allows in a machine.

The table-driven LOG/iC optimizer allows a maximum of 1024 states. For most true state machine applications, you would be hard pressed to fit 1024 states into a single PLD. But this syntax's attractiveness for use in counters as large as 16 bits (64K states) in the CY7C330 can lead you to run up against the 1024-state limitation in short order.

Fortunately, LOG/iC can partition designs into blocks. This capability allows you to partition the design into smaller chunks that are optimized individually and merged after compilation. Blocks also tend to mimic optimal approaches to finding solutions by segmenting designs into smaller functional units (more on this later).

LOG/iC also includes a simple statement that determines the type of flip-flop for implementing the state registers via the *Flip-Flop directive. The default is D-FlipFlops, but the T-FlipFlops statement can also be used. The LOG/iC reduction algorithm automatically generates optimized equations for the flip-flop type specified. This capability is especially significant for the CY7C330, because LOG/iC understands how to use the XOR product term for both polarity control and T flip-flop creation. The CY7C330 can implement large counters extremely efficiently using T flip-flops automatically generated by LOG/iC.

Optimization Levels

You control LOG/iC's optimizer via the Compute and Nocompute statements, which you can place in the design file's *Run-Control section. Optimization levels are essentially binary. Nocompute allows you to indicate

outputs for which you desire no reduction. Compute is complementary and allows you to explicitly specify the outputs you want reduced. Another directive, CPU-Time = nn, allows you to specify the maximum amount of time the compiler can take to attempt an exact solution. After this time, the compiler computes approximated solutions.

CY7C330 Characteristics

Cypress's CY7C330 is a high-performance PLD optimized for state machine applications. It features a pipelined architecture that achieves a 66-Mhz state transition speed. The device's 11 dedicated registered inputs offer small set-up and hold times. These versatile input registers can be clocked with either of two input clocks. You select the input clock by programming a configuration fuse unique to each input register. The CY7C330 has a total of three clock pins — two for the input registers and one for the output/state registers. This feature allows you to synchronize input data without using an external register. You can tie the clock pins together if you need only a single clock source.

The CY7C330 provides 12 I/O macrocells and four buried macrocells. The 12 I/O macrocells have an input register structure identical to that of the dedicated inputs.

The outputs from the CY7C330 logic array feature variable product-term distribution with nine to 19 product terms per output. These product terms are XORed with an additional product term, which you can use for equations that require an XOR, polarity control, or T flip-flop implementation.

A fuse-configurable feedback mux allows you to program the CY7C330 macrocell for feedback from the input register or the output register (buried). The device's output enable is configurable for control via a product term or pin 14. This pin allows you to enable the output buffers asynchronously. Product term OE (output enable) is synchronous to the input register values that comprise the OE equation. You can also program this equation to permanently enable or disable the output buffer.

When the feedback is programmed for state-register (rather than input register) buried feedback, you have an additional feedback connection between pairs of I/O macrocells. This connection provides an input path for the pin that would otherwise be lost. You thus have the flexibility of burying six of the 12 I/O macrocells and using the associated pins as dedicated inputs. The four hidden macrocells have the same product-term structure as the I/O macrocells, with fixed state-register feedback to the logic array. The CY7C330 also furnishes two product terms that permit you to set or reset all the state registers synchronously.

Selecting the CY7C330's Input Clock

The CY7C330's input registers are clocked with either pin 2 or 3. LOG/iC refers to these pins as CLK1 and CLK2, respectively. The default clock used is CLK1. To specify CLK2 instead, use the *Special Functions directive along with the .IC2 pin name suffix. Thus, to select CLK2 for input Fred, use the following syntax:

```
*Special Functions
 Fred.IC2 = YES;
```

Controlling Output Enable

The default for OE in LOG/iC is asynchronous, pin-14 control of the output buffer. If you use the macrocell for input only (pure input), the OE-select fuse is left intact, which selects OE from the product term. Because none of the product term fuses are blown, selecting OE from the product term results in the output driver being turned off. Finally, if you use the macrocell for both input and output, the OE again defaults to asynchronous, pin-14 control.

You have several options for changing this default behavior. First, you can use the OE special function. If the macrocell is called A0, then:

```
A0.OE = 0 ;
; Sets OE to synchronous product term control and permanently turns OFF the driver
A0.OE = 1 ;
; Sets OE to synchronous product term control and permanently turns ON the driver
 A0.OE = EQN;
; Sets OE to synchronous product term control, output driver is controlled by the specified equation (EQN).
```

These constructs should allow you to create any desired OE configuration, while maintaining readability. You can also use the FUSES statement to control the OE mux, as follows:

```
; BLOWN Selects synchronous product term output buffer control
; INTACT Selects asynchronous pin 14 output buffer control
*Fuses;                Pin #
 $17067 = INTACT;      15
 $17063 = INTACT;      16
 $17060 = INTACT;      17
 $17056 = INTACT;      18
 $17053 = INTACT;      19
 $17049 = INTACT;      20
 $17046 = INTACT;      23
 $17042 = INTACT;      24
 $17039 = INTACT;      25
 $17035 = INTACT;      26
 $17032 = INTACT;      27
 $17028 = INTACT;      28
```

Use of the XOR Product Term

LOG/iC supports use of the XOR product term to implement polarity control and T flip-flops. Polarity control is automatic for all entry formats and is controlled via the *Level directive. LOG/iC uses the XOR to create T flip-flops by using the *Flip-Flops directive and specifying T-FlipFlops. The LOG/iC optimizer then automatically produces reduced equations targeted at T flip-flops.

Macrocell Feedback

LOG/iC defaults to selecting feedback from the state register. If you use the macrocell as a pure input, feedback is automatically routed from the input pin register. Designs that use the macrocell state register and the input pin register can specify feedback via the .FBK function or FUSES statements.

As an example, say you use the state register as an adder, and the associated macrocell input-pin register holds a base value. In this case, you want to drive the result onto the output pins during normal operation, while the macrocell input register uses the feedback path to provide the base value to the adder equations. During base-value updates, you three-state the output buffers and clock a new value into the macrocell input registers. LOG/iC defaults to selecting feedback from the state register. The following statements configure the desired feedback:

```
SUM3.FBK = PIN;
    or
; BLOWN Selects feedback from macrocell input
register
; INTACT Selects feedback from macrocell output
register
*Fuses;                    Pin #
 $17068 = BLOWN;           15
 $17064 = BLOWN;           16
 $17061 = BLOWN;           17
 $17057 = BLOWN;           18
 $17054 = BLOWN;           19
 $17050 = BLOWN;           20
 $17047 = BLOWN;           23
 $17043 = BLOWN;           24
 $17040 = BLOWN;           25
 $17036 = BLOWN;           26
 $17033 = BLOWN;           27
 $17029 = BLOWN;           28
```

Controlling Synchronous Reset and Preset

The CY7C330 has a single product term that controls the synchronous resets of all of the state/output registers. Similarly, a single product term controls all the state/output registers' synchronous presets. These two product terms are controlled via the $PS and $RS statements in the *Boolean-Equations section. Avoid a potential pitfall by remembering that resetting the register to Zero causes a value of One to appear on the output pin because of the inverting output buffer.

The following code shows the usage of the preset and reset statements, where variable Paul presets the register, and variable Ray resets the register.

```
*Boolean-Equations
$PS = Paul;
$RS = Ray;
```

Using the Shared-Input Feedback Mux

As mentioned previously, the CY7C330 has a shared-input feedback mux, which allows you to use a given macrocell for both input and output. This feature is useful for several configurations, such as when the state register is buried as an internal state bit that is fed back to the array, and the pin serves as a dedicated input. In this case, the OE product term is typically configured to disable the output buffer.

Another good application for the shared-input feedback mux occurs when you use the input register to hold a seldom-changed value used by the machine. For example, a counter might have an upper limit that is loadable. During normal operation, the output buffer OE is enabled and the count appears on the output pins. When a new limit is desired, the output is three-stated, and the limit value is clocked into the input register. The machine can then access this value via the shared-input feedback mux.

LOG/iC deals with these situations by referring to the state register as a buried node. LOG/iC provides a list of the node numbers and the pins they correspond to. The input to the macrocell is assigned to the pin number. Using this notation, LOG/iC automatically uses the shared-input feedback mux for the input. The following statements correctly configure and use the shared-input feedback mux for a buried macrocell that has a variable assigned to the state register named S1 and an input named X29:

```
*X-names
    X29;
*Y-names
    S1;
;
; Design entry here
;
*Pins
    X29 = 27;
*Nodes
    S1 = 15;
```

Remember that the shared-input feedback mux is available for only one of every pair of macrocells. Node numbers, the corresponding pin numbers, and their

available product terms are as follows (h1 - 4 are the hidden macrocells):

Node:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Pin :	h1	h2	h3	h4	15	16	17	18	19	20	23	24	25	26	27	28
PTs :	19	11	17	13	09	19	11	17	13	15	15	13	17	11	19	09

Design Examples

An old adage about designing asserts that good engineers borrow and great engineers steal! Most often used to describe the practice of re-using existing software in a new design, this proverb applies equally well to doing new PLD designs. Also, examples tend to be the best way to flatten the learning curve for a new language. The examples that follow highlight features of LOG/iC and the CY7C330.

Example 1: Modulo-11 Counter

The ability of the CY7C330's XOR product term to implement T flip-flops proves ideal for building complex counters. This first example is a small counter that counts to 11 and resets to 0. The design also features a clear, a hold, and count-up/down controls. *Appendix A* shows the LOG/iC source code for this design. This counter is an excellent example of the expression compactness with which LOG/iC describes designs.

The counter's four inputs are CLR, UP, HOLD, and OE. CLR resets the counter to zero when asserted. UP determines the direction in which the counter operates. HOLD causes the counter to stop at its current count value until HOLD is released. OE is tied to pin 14 and serves as an asynchronous output enable. The outputs are count bits Q0 - Q3.

Note that the *Level statement has been used to indicate that OE and CLR are active Low. As noted earlier, the design needs no polarity conversion; LOG/iC automatically creates the proper reduced equations for these active-Low inputs.

Because each output value is unique, this design uses Z-Values state assignment. Thus, the states are the counter values. Examining the flow table, you can see that whenever CLR is active, the counter goes to state 1, which has a value of zero. Entered this way, the flow table values cause LOG/iC to use the macrocell product terms to implement the CLR function. You could use the CY7C330's preset and reset product terms to achieve the same result. This design falls well within the limit of nine to 19 product terms per output, however, and the design is very readable in the current format.

The next line in the flow table shows the counting-down state: If in state 1, wrap around to state 11; if in any other state, move to the next lower state. The third line does not have to restate the state section of the flow table, because no change occurs from the second line. The third line specifies the design's up counter: If in state 1 - 10, go to the next higher state; if in state 11, wrap around to state 1.

The flow table's last line shows the hold state. Notice that the file contains statements for both T and D flip-flops. This practice allows you to comment one of the two out easily and see the number of product terms necessary for each type of design implementation. As expected, the T flip-flop design generates a more efficient counter implementation.

Example 2: 15-Bit Counter with Carry Out

The previous example generated a counter with a very compact design expression. If you want a larger counter, you might wish to borrow that example and edit the numbers to provide more count bits. Doing so quickly runs you into the wall of the 1024-states maximum, however. The solutions is to use LOG/iC's block structure to partition the task into multiple smaller counters that cascade to form a large counter. An example of this technique appears in *Appendix B*.

This design consists of three smaller design blocks named CTR1, CTR2, and CTR3 — all identical. The design has global inputs called RESET, HOLD, and UP that perform obvious functions. Global outputs include 15 bits of counter value and a carry out. Between the design blocks are two local variables, INT1 and INT2, which provide carry out internally between counter blocks. The *Link statement reconciles all the global variables and the local variables that each block declares.

Although this design looks rather large, bear in mind that when the optimization is complete, the internal variables completely disappear, and only two product terms are required per output. Finally, note the block titled HW_RESET. This block uses the CY7C330 preset product term to reset all the output pins to zero.

Example 3: T-Bird Tail Lights via Truth Table

The T-Bird tail lights example is a simple design that emulates the function of the early 1960s Ford Thunderbird tail lights. The original design used a motorized assembly that caused the left or right cluster of three lights to turn on sequentially from the inside to the outside when the driver activated the directional signal.

The design presented here has five inputs: left turn (LT), right turn (RT), ignition (IGN), brake, and flash. For this design, the six output lights are also listed as inputs, because the truth table uses them to determine present state — similar to Z-values in an FSM. The six output lights are designated the right and left inside, middle, and outside. The brakes and emergency flashers operate regardless of whether or not the ignition is active. The turn signals, however, operate only with the ignition on. The brake and turn inputs activate all the lights on the side that is not sequencing through a turn indication.

This design introduces the concept of a bus through the constructs LEFT = LO,LM,LI and RIGHT = RI,RM,RO. Also note that the design uses string substitution to describe the output states. *Appendix C* shows this example.

Example 4: T-Bird Tail Lights via Flow Table

FSM syntax can also implement the T-Bird tail lights. For this approach, state bits are assigned to guarantee that all states are unique and non-overlapping. The CY7C330's hidden macrocells are ideal for this use. Refer to *Appendix D* for this design.

Although this FSM implementation is safer than the truth-table version from the aspect of uniquely assigned states, the FSM approach is not without cost. Specifically, the truth table implementation was able to incorporate additional functions for invalid conditions such as LT and RT active simultaneously.

Example 5: 8-Bit Adder for Servo Control

This servo example is covered in detail in the application note, "Using the CY7C330 as a Closed-Loop Servo Controller." The basic idea is that you can use the CY7C330 to calculate the difference between the desired position and the actual position to provide feedback to the servo loop.

In the servo application, the target position is loaded into the I/O macrocell's input register during a special update cycle. During this cycle, a microprocessor provides data to the dedicated inputs as a delta from the current position. The CY7C330 adds the position value to the current position and makes the result available at the output pins in three clocks. Then the second input clock is toggled once to load the new desired position into the I/O-macrocell input registers. This operation is possible because the outputs are driving the macrocell input registers.

This design uses nearly all of the registers in the CY7C330. To provide a difference between desired and current position during normal operation, the input values are furnished in two's complement form and added to the target position stored in the I/O macrocell's input registers. *Appendix E* shows the source code for this example.

One difference between this example and the earlier ones is that both the X and Y input sections contain the variables A[0..7]. This arrangement is due to the fact that the same macrocells provide the desired position (input) and the difference value (output). The *Local attribute identifies intermediate values that are not needed for output but are used to generate the correct results via substitution into other equations. Because the basic equation for an adder uses an XOR to calculate the sum, this example specifies the .XRB attribute to use the CY7C330's XOR product term as an XOR — a technique that reduces the number of other product terms required.

The adder completes an 8-bit add in three clock cycles, producing two intermediate carry bits, which are generated and stored in two of the four internal hidden registers. The special functions attributes .IC2 and .FBK configure the output macrocell appropriately.

Summary

The examples presented here frequently optimized to levels exceeding results produced previously. The ability to specify Don't Cares for output cases, along with LOG/iC's table-driven optimizer, produced results much more quickly than has previously been typical. Documentation, an Achilles heel for many PLD tools, proved quite readable in this case and minimized the dreaded learning curve. LOG/iC's finite state machine syntax allows compact descriptions of complex designs that produced correct results — quite a contrast to previous experiences.

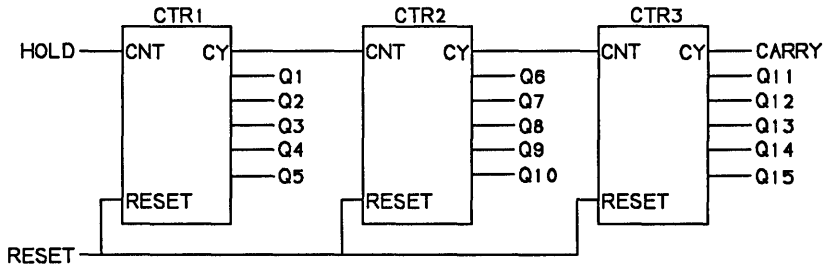
Clearly, LOG/iC can implement designs that use all the features available in the CY7C330. LOC/iC has quickly become an essential tool for Cypress PLD designs.

Appendix A. LOG/iC Source Code for Modulo-11 Counter

```
*IDENTIFICATION
 Bit - modulo 11 counter using LOG/iC FSM entry
 Z-Value state assignment used to specify absolute output value associated with each of the 11 states
 MMA - Cypress Semiconductor

*X-NAMES
    CLR, UP, HOLD, OE;

*Z-NAMES
    Q[3..0];

*LEVEL
    LOW =  CLR, OE;  Active low level for these pins

*Z-VALUES
    S[1..11] = [0..10];

*FLOW-TABLE
    RELEVANT =  CLR, UP, HOLD;
    S[1..11],X  1   -   -,   F1          ; Clear counter to zero
    S[1..11],X  0   0   0,   F[11,1..10] ; Count Down
          X     0   1   0,   F[2..11,1]  ; Count Up
    S[1..11],X  0   -   1,   F[1..11]    ; Hold Counter value

;Spacing between X variables above added only to improve clarity

*STATE-ASSIGNMENT
    Z-Values;

*FLIP-FLOPS
;   D-FlipFlops;
;   D-F/F uses total of 22 Product Terms

    T-FlipFlops;
;   T-F/F uses total of 16 Product Terms

*PLD
    TYPE = PLD7C330;

*PINS
    Q[3..0] =  [28..25],
    CLR     =  3,
    UP      =  4,
    HOLD    =  5,
    OE      =  14;

*RUN-CONTROL
    PROG = JEDEC;
    LIST = PLOT, EQUATIONS, PINOUT, FUSEPLOT;
*END
```

Appendix B. 15-Bit Counter with Carry Out

```
*Identification
    15 bit counter - Using 7C330 hardware Reset
    Using Block Syntax to implement large counter w/FSM input Syntax (bypasses problem with exceeding maximum
    number of states when building large counters - block structure adds NO extra product terms to compiled design.
    INT1 & 2 are completely eliminated)
    MMA
    Cypress Semiconductor
    ;
```




```
*X-Names
    RESET, HOLD, UP;

*Y-Names
    CARRY,Q[1..15];

*Local
    INT[1,2];

*Link
    RESET     =  CTR1:R,CTR2:R,CTR3:R;
    RESET     =  HW_RESET:R;
    HOLD      =  CTR1:CNT;
    UP        =  CTR1:UP,CTR2:UP,CTR3:UP;
    CARRY     =  CTR3:CY;
    INT1      =  CTR1:CY,CTR2:CNT;
    INT2      =  CTR2:CY,CTR3:CNT;
    Q[1..5]   =  CTR1:QQ[1..5];
    Q[6..10]  =  CTR2:QQ[1..5];
Q[11..15]     =  CTR3:QQ[1..5];

;*** First 5-bit counter stage here ************
@BLOCK  =  CTR1;

*X-Names
    CNT,R,UP;

*Y-Names
    CY;

*Q-Names
    QQ[5..1];
```

Appendix B. 15-Bit Counter with Carry Out (continued)

```
*Flow-Table
    ;Using '330s Internal Reset
    Relevant = CNT,UP:CY;
    S[1..32],X  0   -,   Y  0,  F[1..32] ;Hold Condition
    S[1..31],X  1   1,   Y  0,  F[2..32] ;Counting
    S[32],   X  1   1,   Y  1,  F1       ;Maximum Count Reached
    S[32..2],X  1   0,   Y  0,  F[31..1] ;Counting
    S[1],    X  1   0,   Y  1,  F32      ;Minimum Count Reached

*Flip-Flops
    T-FLIPFLOPS;

*State-Assignment
    binary;

@ENDBLOCK  =  CTR1;

;*** Second 5-bit counter stage here ************
@BLOCK  =  CTR2;

*X-Names
    CNT,R,UP;

*Y-Names
    CY;

*Q-Names
    QQ[5..1];

*Flow-Table
; Using '330s Internal Reset
    Relevant = CNT,UP:CY;
    S[1..32],X  0   -,   Y  0,  F[1..32];Hold Condition
    S[1..31],X  1   1,   Y  0,  F[2..32];Counting
    S[32],   X  1   1,   Y  1,  F1       ;Maximum Count Reached
    S[32..2],X  1   0,   Y  0,  F[31..1];Counting
    S[1],    X  1   0,   Y  1,  F32      ;Minimum Count Reached

*Flip-Flops
    T-FLIPFLOPS;

*State-Assignment
    Binary;

@ENDBLOCK  =  CTR2;

;*** Third 5-bit counter stage here ************
@BLOCK  =  CTR3;

*X-Names
    CNT,R,UP;

*Y-Names
    CY;
```

Appendix B. 15-Bit Counter with Carry Out (continued)

```
*Q-Names
    QQ[5..1];

*Flow-Table
; Using '330s Internal Reset
    Relevant = CNT,UP:CY;
    S[1..32],X  0   -,  Y  0,      F[1..32] ;Hold Condition
    S[1..31],X  1   1,  Y  0,      F[2..32] ;Counting
    S[32],   X  1   1,  Y  1,      F1       ;Maximum Count Reached
    S[32..2],X  1   0,  Y  0,      F[31..1] ;Counting
    S[1],    X  1   0,  Y  1,      F32      ;Minimum Count Reached

*Flip-Flops
    T-FLIPFLOPS;

*State-Assignment
    Binary;

@ENDBLOCK = CTR3;
;******* End of Counter Blocks **********

@BLOCK = HW_RESET;

 *X-Names
    R;

 *Boolean Equations
    $PS = R;

@ENDBLOCK = HW_RESET;

*PLD
    Type = PLD7C330;


*Pins
    REGCLK   =  1,
    INPCLK   =  2, ! needed for creating testvectors
    Q[5..10] =  [15..20],
    Q[11..15] = [23..27],
     CARRY   =  28,
     RESET   =  4,
     HOLD    =  5;
     UP      =  6;

*Nodes
    Q[1..4] =  [1..4];

*Run-control
     Listing    =  Pinout, Plot;
     Progformat =  Jedec;

*END
```

Appendix C. T-Bird Tail Lights Example

```
*IDENTIFICATION
    Thunderbird sequencing Tail lights example for 7C330 using ISDATA LOG/IC
    Truth Table Implementation
    MMA
    Cypress Semiconductor

*X-NAMES
    LT, RT, BRAKE, FLASH, IGN, RI, RM, RO, LI, LM, LO;


*Y-NAMES
    RI, RM, RO, LI, LM, LO;

*BUS
     LEFT  = LO,LM,LI;
     RIGHT = RI,RM,RO;

*LEVEL
    LOW = FLASH;

;Macros for All desired output combinations:
*STRING
    ON        =  1,  1,  1;
    OFF       =  0,  0,  0;
    LEFT1     =  0,  0,  1;
    LEFT2     =  0,  1,  1;
    RIGHT1    =  1,  0,  0;
    RIGHT2    =  1,  1,  0;
    ONE       =  1,  0,  0;
    TWO       =  0,  1,  0;
    THREE     =  0,  0,  1;
    TRI       =  -,  -,  -;

*FUNCTION-TABLE
    $ IGN,FLASH,LT,RT,BRAKE,LEFT ,RIGHT   : LEFT   ,RIGHT
    ;Quiescent
    1,  0,  0,  0,  0,  'TRI',     'TRI'     :  'OFF','OFF';
    0,  0,  -,  -,  0,  'TRI',     'TRI'     :  'OFF','OFF';

    ;Flash
    -,  1,  -,  -,  -,  ON',       'ON'      :  'OFF','OFF';
    -,  1,  -,  -,  -,  'OFF',     'OFF'     :  'ON','ON';

    ;Brake
    -,  0,  0,  0,  1,  'TRI',     'TRI'     :  'ON','ON';
    0,  0,  -,  -,  1,  'TRI',     'TRI'     :  'ON','ON';

;Left Turn
    1,  0,  1,  0,  0,  'OFF'      ,'TRI'    :  'LEFT1','OFF';
    1,  0,  1,  0,  0,  'LEFT1',   'TRI'     :  'LEFT2','OFF';
    1,  0,  1,  0,  0,  'LEFT2',   'TRI'     :  'ON','OFF';
    1,  0,  1,  0,  0,  'ON',      'TRI'     :  'OFF','OFF';
```

Appendix C. T-Bird Tail Lights Example (continued)

```
;Right Turn
    1,  0,  0,  1,  0,  'TRI',     'OFF'     :  'OFF','RIGHT1';
    1,  0,  0,  1,  0,  'TRI',     'RIGHT1'  :  'OFF','RIGHT2';
    1,  0,  0,  1,  0,  'TRI',     'RIGHT2'  :  'OFF','ON';
    1,  0,  0,  1,  0,  'TRI',     'ON'      :  'OFF','OFF';

;Left Turn + Brake
    1,  0,  1,  0,  1,  'OFF',     'TRI'     :  'LEFT1','ON';
    1,  0,  1,  0,  1,  'LEFT1',   'TRI'     :  'LEFT2','ON';
    1,  0,  1,  0,  1,  'LEFT2',   'TRI'     :  'ON','ON';
    1,  0,  1,  0,  1,  'ON',      'TRI'     :  'OFF','ON';

;Right Turn + Brake
    1,  0,  0,  1,  1,  'TRI',     'OFF'     :  'ON','RIGHT1';
    1,  0,  0,  1,  1,  'TRI',     'RIGHT1'  :  'ON','RIGHT2';
    1,  0,  0,  1,  1,  'TRI',     'RIGHT2'  :  'ON','ON';
    1,  0,  0,  1,  1,  'TRI',     'ON'      :  'ON','OFF';

;Both Turn - lights flash in reverse sequence
    1,  0,  1,  1,  0,  'OFF',     'OFF'     :  'ON','ON';
    1,  0,  1,  1,  0,  'ON',      'ON'      :  'LEFT2','RIGHT2';
    1,  0,  1,  1,  0,  'LEFT2',   'RIGHT2'  :  'LEFT1','RIGHT1';
    1,  0,  1,  1,  0,  'LEFT1',   'RIGHT1'  :  'OFF','OFF';

;Illegal condition, All ON
    1,  0,  1,  1,  1,  'OFF',     'OFF'     :  'ONE','THREE';
    1,  0,  1,  1,  1,  'ONE',     'THREE'   :  'TWO','TWO';
    1,  0,  1,  1,  1,  'TWO',     'TWO'     :  'THREE','ONE';
    1,  0,  1,  1,  1,  'THREE',   'ONE'     :  'OFF','OFF';

*FLIP-FLOPS
    D-FLIPFLOPS;
;   T-FLIPFLOPS;

*PLD
    TYPE = PLD7C330;

*PINS
    LT      =  4,
    RT      =  5,
    BRAKE   =  6,
    FLASH   =  7,
    IGN     =  9,

    RI      =  23,
    RM      =  24,
    RO      =  25,
    LI      =  20,
    LM      =  19,
    LO      =  18;

*RUN-CONTROL
    PROG = JEDEC;
    LIST = PLOT, EQUATIONS, PINOUT, FUSEPLOT;

*END
```

Appendix D. T-Bird Tail Lights via Flow Table

```
*IDENTIFICATION
    Thunderbird sequencing Tail lights example for 7C330 using ISDATA LOG/IC
    State Machine Implementation
    MMA
    Cypress Semiconductor

*X-NAMES
    LT,RT,BRAKE,FLASH,IGN;

*Z-NAMES
    LO,LM,LI,RI,RM,RO;

*LEVEL
    LOW = FLASH;

*Q-NAMES
    Q[1..4];

*Z-VALUES
    S1 =  000000;All lights off or Flash Off
    S2 =  001000; Left Turn 1
    S3 =  011000; Left Turn 2
    S4 =  111000; Left Turn 3
    S5 =  000100; Right Turn 1
    S6 =  000110; Right Turn 2
    S7 =  000111; Right Turn 3
    S8 =  001111; Brake + Left Turn 1
    S9 =  011111; Brake + Left Turn 2
    S10 = 111111; Brake + Left Turn 3
    S11 = 000111; Brake + Left Turn 4
    S12 = 111100; Brake + Right Turn 1
    S13 = 111110; Brake + Right Turn 2
    S14 = 111111; Brake + Right Turn 3
    S15 = 111000; Brake + Right Turn 4
    S16 = 111111; Brake or Flash On

*FLOW-TABLE
;   Sn,  LT RT Brake Flash IGN, Fn
    S1, X  0  0   0     0   -,  F1;     All Lights Off
        X  -  -   -     1   -,  F16;
        X  0  0   1     0   1,  F16;
        X  -  -   1     0   0,  F16;
        X  1  0   0     0   1,  F2;
        X  0  1   0     0   1,  F5;
        X  1  0   1     0   1,  F8;
        X  0  1   1     0   1,  F12;
        XREST,                  F1;

    S2, X  1  -   -     -   1,  F3;     Left Turn Sequence
        XREST,                  F1;

    S3, X  1  -   -     -   1,  F4;
        XREST,                  F1;

    S4, XREST,                  F1;
```

Appendix D. T-Bird Tail Lights via Flow Table (continued)

```
    S5, X - 1 - - 1, F6;      Right Turn Sequence
       XREST,          F1;

    S6, X - 1 - - 1, F7;
       XREST,          F1;

    S7, XREST,          F1;


    S8, X 1 - 1 - 1, F9;      Left Turn + Brake Sequence
       XREST,          F1;

    S9, X 1 - 1 - 1, F10;
       XREST,          F1;

    S10,X 1 - 1 - 1, F11;
       XREST,          F1;

    S11,X 1 - 1 - 1 ,F8;
    XREST,             F1;

    S12,X - 1 1 - 1, F13;       Right Turn + Brake Sequence
       XREST,          F1;

    S13,X - 1 1 - 1, F14;
       XREST,          F1;

    S14,X - 1 1 - 1, F15;
    XREST,             F1;

    S15,X - 1 1 - 1, F12;
    XREST,             F1;

    S16,X 0 0 1 0 1, F16;       Brake/Flash Tail lights
       X - - 1 0 0, F16;
       XREST,          F1;

*STATE-ASSIGNMENT
    Binary

*FLIP-FLOPS
    D-FLIPFLOPS;
;   T-FLIPFLOPS;

*PLD
;   TYPE = HYPERPLD;
    TYPE = PLD7C330;

*PINS
    LT      = 4,
    RT      = 5,
    BRAKE   = 6,
    FLASH   = 7,
    IGN     = 9,
```

Appendix D. T-Bird Tail Lights via Flow Table (continued)

```
    RI      =  23,
    RM      =  24,
    RO      =  25,
    LI      =  20,
    LM      =  19,
    LO      =  18;

*NODES
    Q[1..4] =  [1..4];

*RUN-CONTROL
    PROG = JEDEC;
    LIST = PLOT, EQUATIONS, PINOUT, FUSEPLOT;

*END
```

Appendix E. 8-Bit Adder Example

```
*Identification
    8-Bit multi-stage adder - as detailed in 7C330 Servo control Application Note
    Mark Aaldering
    Cypress Semiconductor

*X-Names
    CIN,C2,C5,A[0..7],B[0..7];

*Y-Names
    A[0..7],C2,C5,CARRY;

*Local
    C[0..1,3..4,6..7];

*Boolean-Equations

    A[0..7].XRB = A[0..7];

    A0 = B0 # CIN;
    C0 = (A0 & B0) + (A0 & CIN) + (B0 & CIN);

    A[1..7] = B[1..7] # C[0..6];
    C[1..6] = (A[1..6] & B[1..6]) + (A[1..6] & C[0..5]) + (B[1..6] & C[0..5]);

    CARRY = (A7&B7) + (A7&C6) + (B7&C6);

*Flip-Flops
    D-FLIPFLOPS;

*PLD
    Type = PLD7C330;

*Nodes
    C2 =  1;
    C5 =  3;

*Pins
    OUTCLK    =  1,
    INCLK     =  2,
    ACLK      =  3,
    CIN       =  4,
    B[0..7]   =  [5..7,9..13],
    A0        =  28,
    A1        =  15,
    A2        =  20,
    A3        =  17,
    A4        =  26,
    A5        =  23,
    A6        =  19,
    A7        =  24,
    CARRY     =  18,

*Special-Functions
    A0.IC2 =  Yes;
    A1.IC2 =  Yes;
    A2.IC2 =  Yes;
```

Appendix E. 8-Bit Adder Example (continued)

```
A3.IC2 =   Yes;
A4.IC2 =   Yes;
A5.IC2 =   Yes;
A6.IC2 =   Yes;
A7.IC2 =   Yes;

A0.FBK =   Pin;
A1.FBK =   Pin;
A2.FBK =   Pin;
A3.FBK =   Pin;
A4.FBK =   Pin;
A5.FBK =   Pin;
A6.FBK =   Pin;
A7.FBK =   Pin;

*RUN-CONTROL
    PROG  =  JEDEC;
    LIST  =  PLOT, EQUATIONS, PINOUT, FUSEPLOT;

*END
```

CYPRESS SEMICONDUCTOR

State Machine Design Considerations and Methodologies

The use of state machines provides a systematic way to design complex sequential logic circuits—an increasingly popular approach since the advent of PLD (Programmable Logic Device) circuitry. This application note describes the many options encountered during the state machine design cycle. By exhaustively walking through the PLD-based design example presented here, you can weigh the merits of several design approaches.

Definitions of Commonly Used Terms

1. *External input vector*—External signals (stimulus) applied to the state machine.
2. *System outputs*—Signals generated by the state machine that are explicitly designed for availability to the external system (hardware outside of the state machine). Registered system outputs can also be fed back into the state machine as part of the State Vector, which is then used in the decode of the state machine's next state.
3. *State registers*—Registers used exclusively for determining the next state of the machine (feedback).
4. *State outputs*—Outputs of the state registers that are available to the external system. (They are typically available to the external machine for debug or due to the lack of buried registers.)
5. *State vector or machine state*—The registered feedback information defining the present state of the machine and required to determine the next state of the machine.
6. *State path*—The transitional condition that must be met for the state machine to progress from one state to another. The state path typically consists of one or more product terms generated from external inputs, although other state paths are possible.
7. *Total input vector*—The combination of the external input vector and the state vector. The total input vector is decoded to generate the next state of the machine.

State Machine Entry Methods

There are many ways of describing a state machine, each with distinct advantages and disadvantages. Three popular description methods are state diagrams, state tables, and high-level languages (HLLs). The state diagram provides an easily observable flow description of the state machine. Because the ability to view the flow of states provides distinct documentation advantages, state diagrams will be used throughout this application note to describe the example state machine.

Upon completing a state diagram, you can easily convert the diagram's visual information into the other types of state machine description or directly into Boolean equations. Several available software programs accept their own forms of state table, HLL, and/or Boolean entry. You can enter all these formats easily via your favorite text editor. The software then translates the inputs into suitable forms (usually a JEDEC map) for hardware implementation.

Another method of describing a state machine, the state table, offers perhaps the most concise description. Its major advantage over the other entry methods is the availability of state table reduction methods (see *Reference 1*). When applied to your state table definition, a reduction program generates a minimal model for the function. The software used for state machine synthesis throughout this application note uses the state table method of entry. The program is called LOG/iC from ISDATA Corporation.

Finally, high level language (HLL) state machine entry is probably the most popular form of state

machine design. HLLs typically offer C-language-like instructions (e.g., case,if-then-else, etc.) to describe the machine.

An Example State Machine

The example state machine is a clock generator for a pipelined (three system execution stages), bit-slice-based, central processing unit (CPU). Each of the three system execution stages contains two clocks for a total of six system clocks for every instruction execution. With pipelining enabled, each instruction takes an average of two clock periods. Further, external hardware unaffected by CPU wait and stop states (e.g., cache memory) needs both polarities of an additional free-running clock.

To minimize clock edge skew, the state machine provides both versions of the clock. To put the timing of this application into perspective, executing each pipeline stage in an 80-ns period (or 12.5 MHz) requires the state machine to run at 25 MHz. This speed is well within the range of the available PALs , EPLDs and PROMs that can be used to implement the state machine.

Each of the pipeline's three execution stages has a specific function. Briefly, the first stage of the pipeline accesses the Writable Control Store (WCS) RAM. The Arithmetic Logic Unit (ALU) execution occurs during the second stage of the pipeline. Finally, the third pipeline stage clocks status and memory address registers. The function(s) performed during each of the three stages are described in greater detail in the "State Machine Output Definition" section of this application note.

If this design only generates a simple set of pipelined clocks, why not use shift registers and miscellaneous glue logic instead of a state machine? There are two reasons to consider a state machine. First, it is usually desirable to minimize the number of chips required; the state machine in PLD form might need external glue logic, but significantly less than the shift register solution.

The second reason for considering a state machine is that this application requires more then just a simple set of pipeline clocks. The function of the clock signals is to provide control of the CPU in multiple modes of operation. The desired modes of operation are as follows:

PIPELINED RUN Mode

In this mode, the CPU simultaneously performs the instructions in all three stages of the pipeline. For example, while instruction n does an ALU operation, instruction n+1 accesses WCS, and instruction n-1 clocks ALU status.

NONPIPELINED RUN Mode

NONPIPELINED RUN mode performs all three stages of instruction execution without overlap. The time to complete one nonpipelined instruction equals the average of three pipelined instructions.

CPU STOP

The system must have a way to perform an orderly stop of CPU execution from both of the above run modes. This stop might be the result of several possible conditions, including a utility stop from a system control unit, a single step, a breakpoint, or a response to external hardware (e.g., a logic analyzer). The free-running clocks continue to run during the CPU STOP mode and remain running at all times, except during a reset condition.

CPU WAIT

In CPU WAIT mode, an external condition causes a delay in an instruction's execution. The instruction pauses until the external condition is removed. One application for the CPU WAIT mode is to handle a cache miss. When a cache miss occurs, the CPU remains in the CPU WAIT mode until the cache completes its memory transfer.

SINGLE STEP

The ability to execute one instruction at a time is needed to debug the CPU. You can easily implement SINGLE STEP external to the clock state machine by pulsing the RUN signal. SINGLE STEP mode is described further in the State Machine Input Definition section of this application note.

INTERRUPT

A variety of system conditions can interrupt the CPU out of its normal execution sequence and immediately start the execution of the interrupt handler. The influence of the INTERRUPT mode on the system clocks will be discussed in greater detail later in this application note.

REPEAT INSTRUCTION

The REPEAT INSTRUCTION mode is a CPU debug feature. It is a good idea to implement this mode external to the clock state machine. By dubbing the clock to the instruction register and the interrupt line to the clock state machine, the CPU continually executes the instruction in the instruction register.

Synchronous vs. Asynchronous Machine

At this point in the state machine design, an appropriate type of state machine must be chosen to match the application. Two major types are the asynchronous and the synchronous implementations. The asynchronous machine changes state when one or more of its inputs changes from a previously stable input state. After a state change, the outputs of the state machine settle, while the machine stabilizes once again. A basic example of an asynchronous state machine would be a simple SR latch built from two NAND gates (*Figure 1*). For the clocking application considered in this application note, the asynchronous state machine

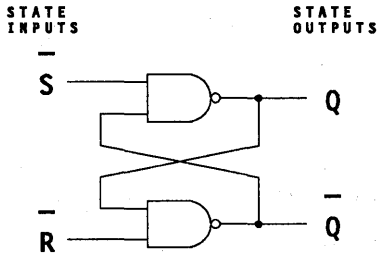


Figure 1. SR Latch, Asynchronous State Machine Example

implementation would be a poor choice, due to the instability of the system outputs.

The synchronous state machine offers a better choice. A synchronous state machine block diagram appears in *Figure 2*. Generally, a synchronous state machine samples the total input vector at specific periods to determine the machine's next state. When designing synchronous state machines, it is important to avoid state register metastability. External inputs to the machine must be synchronized to guarantee stable state register inputs, and the feedback time plus data setup time to the state register clock must be less then or equal to the state clock period.

The modern theory of synchronous state machines was pioneered by Mealy and Moore (see *Reference 1*). Mealy and Moore machines differ slightly from each other in they way they control the system outputs. During a specific machine state, a Mealy machine allows the input conditions to alter the system outputs (the outputs depend on the "total" input state). In contrast, a Moore machine system outputs depend only on the present machine state. Thus, the system outputs remain stable until the next time period, when the Moore machine samples the total input vector to determine the next state. If all design conditions are met (external inputs are stable prior to the next state clock), the Moore machine provides glitch-free system outputs—a desirable characteristic for the CPU system clock. The design described here is therefore implemented as a Moore machine.

Clock Generator Output Definition

As explained earlier, each of the three system execution stages contains two clocks for a total of six system clocks for every instruction execution. The naming convention for these clocks is

CLK_xy
where x = 1, 2, or 3, representing the first, second, or third stage of the instruction execution
and y = A or B, representing the first or second half of the execution stage.

Following this convention, the state machine's two free-running clocks are named CLK_A and CLK_B. These clocks run at half the state clock frequency and 180 degrees out of phase. The free-running clocks occur at the same time as their respective CLK_xA and CLK_xB clocks.

The major clock functions for this application are:
CLK_1B: The leading edge of this clock updates the instruction register.
CLK_2A: This clock's leading edge marks the start of ALU execution. The information on the ALU input bus clocks into the appropriate input registers at this time. The instruction cycle is considered recoverable up through and including CLK_2A (i.e., the status of the machine from the previous instruction has not been altered).

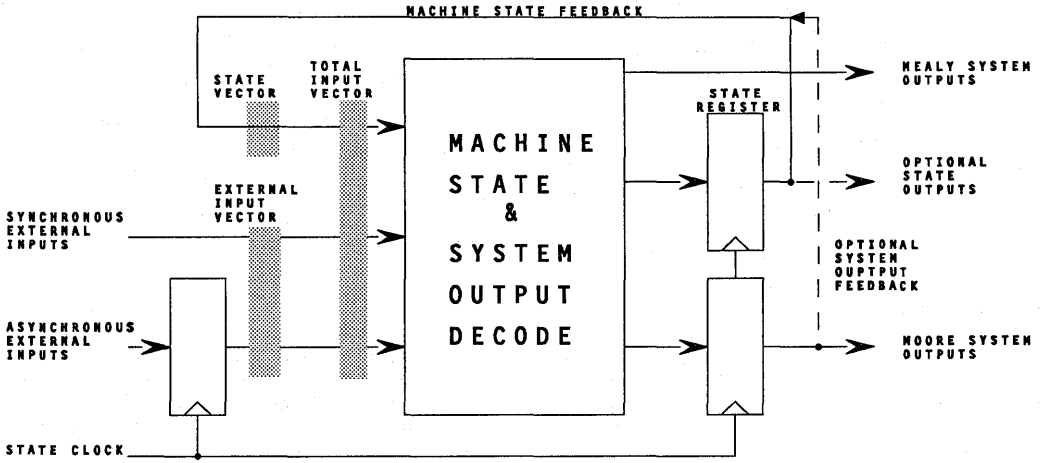


Figure 2. Synchronous State Machine Block Diagram

CLK_2B: Used to control the second half of the ALU execution stage, this clock initiates a write to RAM, triggers counters, gates ALU output into its latch, and clocks the ALU output information into any of the distributed destination registers.
CLK_3A: On this clock the memory address register can be updated. The ALU output bus status and ALU status is also clocked into the CPU status register.

Clock Generator Inputs

A set of inputs (external stimulus to the state machine) controls the state machine. The clock state machine described here has eight external inputs, including the state machine clock. These inputs are:

STATECLK: The state machine clock.

RESET: An asynchronous or synchronous reset input that can be connected directly to the state registers' preset or clear or to all clocked register inputs (D or T input). If connected to the preset or clear, RESET need not be synchronized. In this case, RESET forces the state machine into the machine's initial state, regardless of the present state. RESET can result from any combination of the following sources:

1. Power up circuit (system reset)
2. System controller software decodes system reset
3. System controller software decodes module reset
4. CPU software decodes module reset

RUN: This signal controls the start and stop sequence of the CPU clocks. In PIPELINE RUN mode, the start sequence generates the proper clock progression to fill up the pipeline registers, and the stop sequence empties the pipeline. RUN is externally manipulated to implement the single step and breakpoint functions.

NPL: Used to select NONPIPELINED RUN vs. PIPELINED RUN modes, this signal must be set to the selected mode prior to activating the RUN signal. Setting NPL = 1 selects NONPIPELINED RUN mode, and NPL = 0 selects PIPELINED RUN mode. The single step function operates properly in NONPIPELINED RUN mode only.

INTR: This signal indicates an external interrupt. When INTR is received, and IEN (interrupt enable, described below) is active, the CPU executes its interrupt handler. An interrupt inhibits the instruction register update clock (CLK_1B) and the ALU update clock (CLK_2B). CLK_1A for the interrupt instruction executes on the next cycle. The interrupt condition has priority over a wait condition and therefore starts generating clocks to permit execution of the interrupt instructions.

IEN: This interrupt enable signal qualifies INTR. IEN is likely to be a bit in the instruction word, allowing the user to define sections of un-interruptable code.

WAIT: The wait condition is initiated when both WAIT and WEN (wait enable, described below) are active. The CPU remains in the wait condition until WAIT goes inactive.

WEN: This wait enable signal qualifies WAIT for entrance into the wait condition. Like IEN, WEN is usually a bit in the instruction word, allowing the user to define sections of wait-sensitive code.

State Machine Partitioning

When architecting a state machine, it is generally a good practice to break up large machines into workable blocks, with each of the smaller machines containing states that require common inputs and generate common outputs. The example clock state machine is small enough to be designed as a single state machine, although it would be trivial to design logic to generate the free-running clocks as a separate machine from the rest of the clock state machine. Equations for the free-running clocks are:

CLK_A := /RESET * /CLK_A

CLK_B := /RESET * CLK_A

where ":=" indicates a registered output.

By examining these output equations, you can see that the free-running clocks have only two dependencies in common with the remaining portion of the clock state machine, i.e., RESET and STATECLK. The free-running clocks are required as inputs to the other state machine to synchronize the additional system outputs, however.

The example presented here implements the free-running clocks and the other system outputs within the same state definition. The resulting output equations can be verified against the equations for the free-running clocks alone.

The Initial Machine State

Regardless of the preferred state machine entry method, attacking the problem starts with defining the initial state of the machine. This initial state (INIT in the example) must be consistent with the power-on condition and/or an external input used to initialize the machine (RESET).

The state of the machine can be decoded from the present values of the system outputs, state registers, or a combination of the two. (The advantages and disadvantages of the state definition options will be discussed in greater detail later in this application note.) The initial machine state is generally, but not always, a decode of all 0s or all 1s. In the example design, INIT is the decode of all 0s.

Naming the States

With the exception of INIT, each state in the example design is named to indicate the active system clocks occurring during that state. For example, during state A, only CLK_A is active. Similarly, state 123B has only CLK_1B, CLK_2B, CLK_3B, and CLK_B active. Additionally, an "N" suffix designates a nonpipelined state and a "W" suffix designates a wait condition state; this convention differentiates between states with identical active system outputs.

CPU Inactive States

The RESET input causes the state machine to enter the INIT state from any state in the machine. From the INIT state, the machine unconditionally starts to generate the free-running clocks. As shown in *Figure 3*, a line pointing from the INIT state to the A state, with a path equation equal to 1, indicates an unconditional branch. The state machine progression continues from the A state unconditionally into the B state. In the B state a multi-branch condition exists. If the RUN input remains inactive, then the A and B states continue to toggle, generating only the free-running clocks. Hence the INIT, A, and B states are referred to as "CPU inactive states".

Nonpipelined States

If the NPL input is active while the RUN input becomes active, the state machine operates in NON-PIPELINED RUN mode and follows the model portrayed in *Figure4*.

Pipelined States

If the NPL input is inactive when the RUN input goes active, thus indicating PIPELINED RUN mode, the state machine operates as depicted in *Figure5*.

Unique States

When the RUN input goes active, the next state executed is either the 1A or the 1AN state, depending upon the value of the NPL input (refer to *Figures4* and *5*). Notice that the active system outputs in these two states are identical. Why generate two identical states—when an additional state register might be required to differentiate between the states? (This assumes you use the system outputs to decode the machine's states.) The redundant states are not a problem because the additional state register needed to differentiate between the states is not an issue. There are two reasons for this. First, if you eliminate the redundant states, the state machine would require at least one additional state register anyway to differentiate between the B and the BW or BWN states, which would be needed without 1A and 1AN. (Separation of states BW and BWN from state B is required for correct functionality.) Second, adding another state only increases the number of state registers if the new total number of states exceeds an additional binary boundary (2, 4, 8, 16, ...). This is not a problem here.

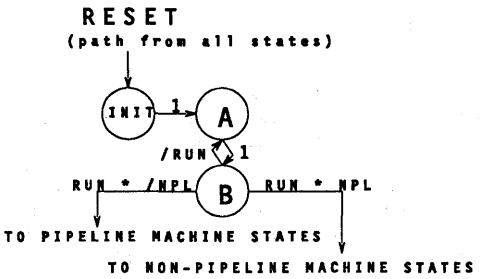


Figure 3. CPU Inactive States

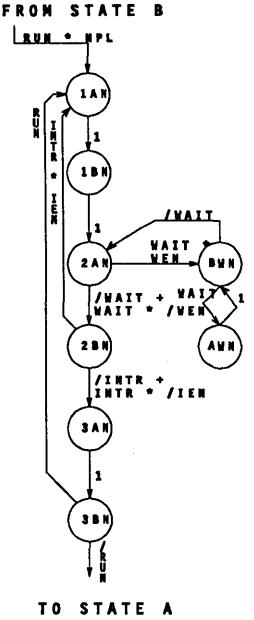


Figure 4. Non-Pipelined States

You might also choose to widen your state machine (increase the number of state registers) to reduce the number of product terms to the state or system output registers. This decision should take into account the desired circuit implementation (PLDs, PROMS, discrete hardware, etc.) and is often an iterative process. In general, you can initially architect the state machine in the manner that is the easiest for you to understand, then make additional changes or small adjustments later if they become necessary.

State Description Verification

Now that all the pieces of the state machine are functionally defined (refer to *Figure6* for the completed state diagram), consider methods for verifying the validity of the design. Some software you can use to describe and implement state machines would already offer verification at this point in a design. For other methods, read on!

One way to verify a state machine design is to recognize a rule of thumb: Out of every state, there should be a state path to another state for every possible combination of relevant external inputs. For example, there are two paths out of state 123B, with INTR and IEN as the relevant external inputs:

Path 1 = INTR * IEN

Path 2 = /INTR + INTR * /IEN

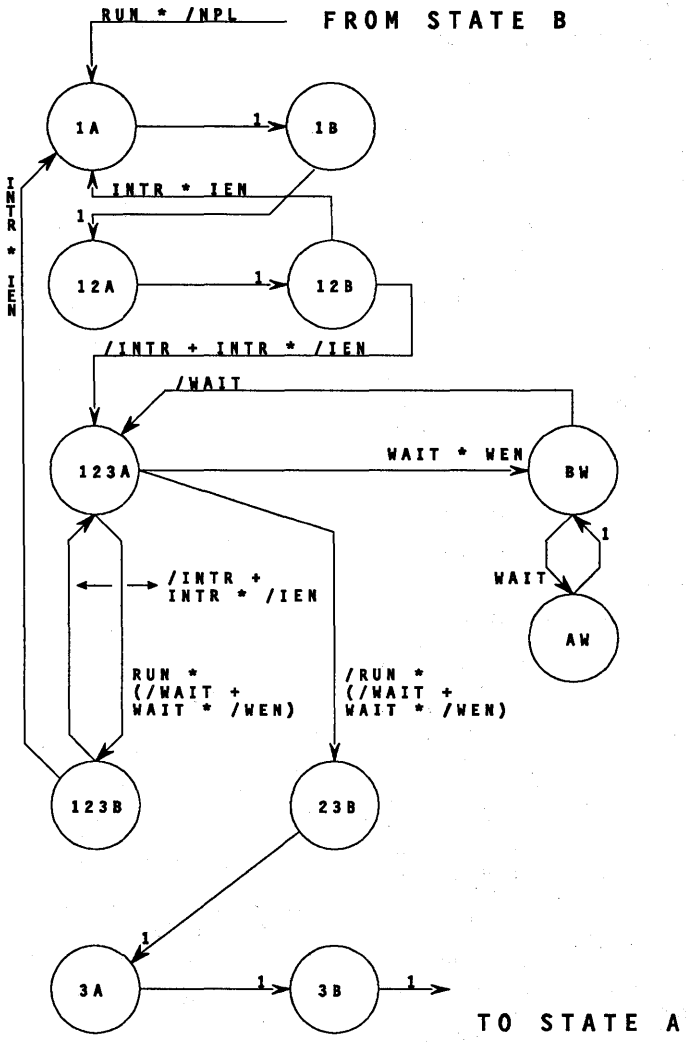


Figure 5. Pipelined States

If there are no known restrictions on the external inputs, a simple method of verifying the above rule of thumb is to generate an equation where all of the paths out of a state are ORed together as follows:

```
OUT_STATE_123B    = Path 1 + Path 2;
OUT_STATE_123B    = (INTR * IEN)
                  + /INTR
                  + (INTR * /IEN);
OUT_STATE_123B    = 1
```

If the equation's terms equal 1 after Boolean reduction, then every state path out of the state is accounted for. The main advantage to this verification method is that you can easily do it using readily available Boolean reduction software.

If there are known restrictions to the external inputs, you can use this information to reduce the complexity of the machine. If it is impossible for the INTR * /IEN condition to occur externally, for example, then you can leave this condition out of the Path 2 equation.

In that case, the reduction of the OUT_STATE_123B equation yields a non-1 result.

Because the method of verification just described does not detect redundant path equations, it is useful to revise the original rule of thumb to: Out of every state, there should be one and only one state path to another state for every possible combination of relevant external inputs.

This revised condition is not as easily verified as the original statement. The easiest way to verify the more restrictive case is to simulate the state machine. To do this, you must generate a test vector for every possible external input that is relevant to each state simulated. Automatic test vector generation programs are available that produce every possible combination. After running the vectors against the design, you must visually inspect the output to verify that the machine never enters an illegal state.

System and State Register Output Generation

The model defining the clock state machine is complete, but there are still quite a few important

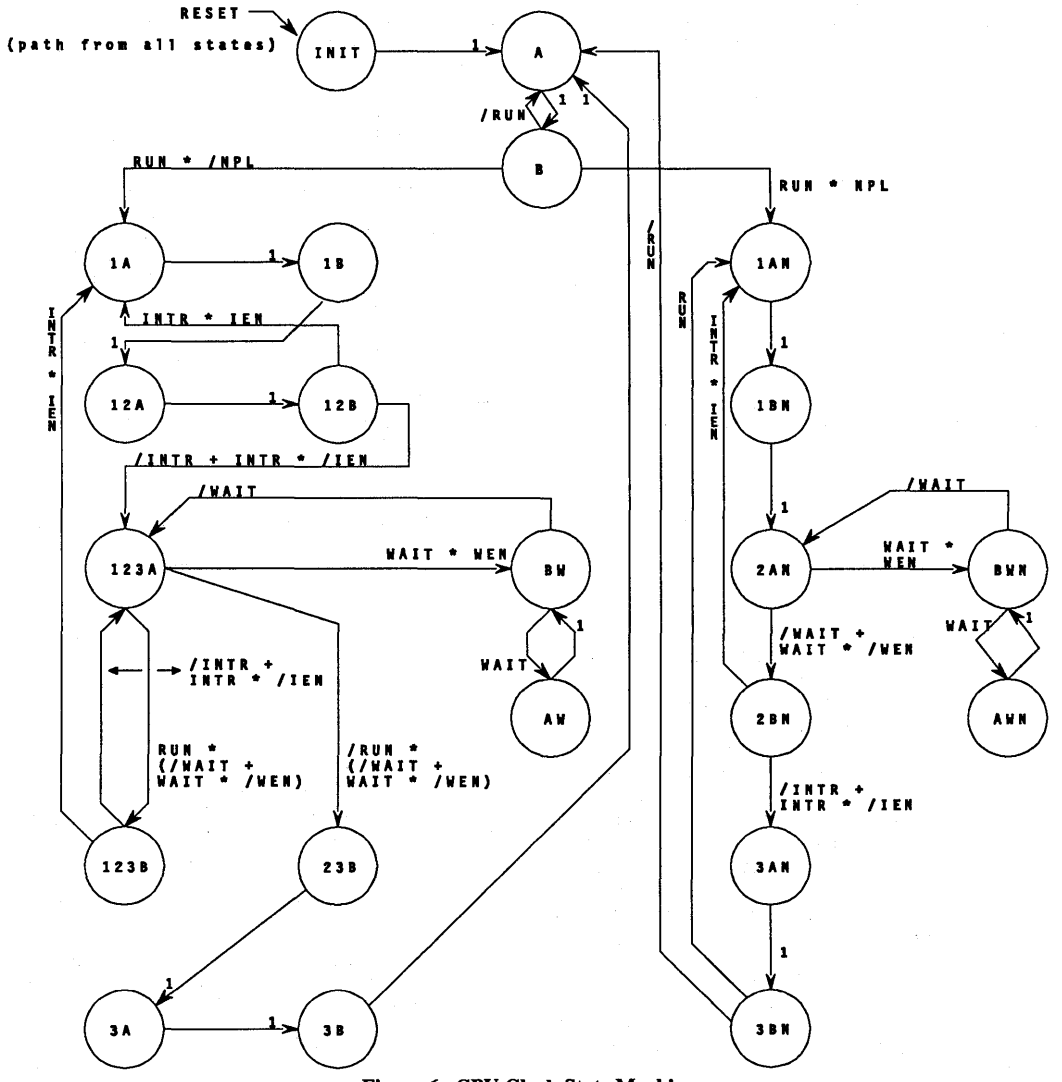


Figure 6. CPU Clock State Machine

decisions to be made regarding the final circuit implementation. Some of the major alternatives for final implementation are:

- System output vs. exclusive state register state decode
- D flip-flop vs. T flip-flop implementation
- PLD vs. PROM implementation

To gain some insight into these choices, consider how the output or feedback equations are assembled. Take, for example, the generation of CLK_3A using a D flip-flop (FF) implementation. By referring to *Figure 6*, you can find all the states in which CLK_3A is active. These are 123A, 3A, and 3AN. The CLK_3A output is generated by ORing the state decodes that, when ANDed with their respective state paths, advance the state machine into the three states listed above. Specifically:

```
CLK_3A :=
        (Decode of 12B)*(/INTR+INTR*/IEN) ;-123A
        +(Decode of BW)*(/WAIT)           ;-123A
        +(Decode of 23B)*(1)              ;-3A
        +(Decode of 2BN)*(/INTR+INTR*/IEN);-3AN
```

When you define the state decodes, the CLK_3A equations are completely specified in terms of the state machine inputs (state path), state registers, and/or system outputs (state decode). Typically, you then multiply the equation out to form a sum of products. This format provides for easy implementation in a PLD, which has a sum of products architecture, and also provides a useful foundation for further equation reduction.

State Decode

As discussed earlier, the next state of the machine can be decoded from the present values of the system outputs, the state registers, or a combination of the two. The choice typically comes down to weighing the maximum number of product terms verses the maximum number of flip-flops available in an implementation. For a Moore machine, with registered system outputs, using the system outputs to uniquely define the states uses the smallest number of flip-flops to define the state machine. However, it is often necessary to add one or more state registers to uniquely define the states.

State assignment for this state decoding method is quite simple, but also rigidly defined, allowing limited flexibility when assigning the additional state registers. After reduction, the feedback and output equations of this "narrow" state machine might contain too many product terms to be implemented in a specific PLD, although product term complexity is never a problem with a PROM implementation.

Exclusive State Registers

Another consideration in state machine design is that you might be able to distribute the number of product terms more evenly among the equations implementing the state machine by using state registers exclusively to decode the states. Because the state decodes in the state registers can be selected to assist in Boolean reduction, proper state assignment enables the more complex equations to fit into a specific implementation.

This type of decode is useful in a PLD implementation, where there is a shortage of product terms for a specific state flip-flop, but extra flip-flops are available. Adding an extra state register can simplify the decode logic enough to fit the design in a single PLD.

The total number of exclusive state registers required to implement a state machine varies from a minimum of LOG(2)X (rounded up to the nearest integer) to a maximum of X, where X is the total number of states in the machine. You can iteratively change this number, along with the state assignment, to obtain a suitable solution.

The state assignment itself is a non-trivial issue, with almost limitless possibilities and no known method of obtaining the optimal solution. There are, however, some guidelines that can be used to obtain workable solutions:

1. Two or more states that potentially enter the same state with identical path equations should be adjacent (their binary codes differ in exactly one position). As an example, refer to *Figure 5*. States 12B and 123B both proceed into state 1A if the path condition INTR * IEN is true. When generating the CLK_1A equation, two of the terms of the equation look like this:

```
CLK_1A :=
        (Decode of 12B) * (INTR * IEN)      ;-1A
        + (Decode of 123B) * (INTR * IEN)   ;-1A
        ...
```

If the decode of 12B and 123B differ in exactly one position, then Boolean reduction (which uses the A*B + /A*B = B relationship) converts the two product terms into one smaller product term.

2. Two or more states that might proceed into different states with identical path equations, and an identical active output, should be adjacent. This situation occurs in the previous CLK_3A equation, shown again here:

```
CLK_3A :=
        (Decode of 12B)*(/INTR+INTR*/IEN) ;-123A
        +(Decode of BW)*(/WAIT)           ;-123A
        +(Decode of 23B)*(1)              ;-3A
        +(Decode of 2BN)*(/INTR+INTR*/IEN);-3AN
```

Note that if states 12B and 2BN are adjacent, then you can reduce the CLK_3A equation to three product terms.

Clock Generator Implementation

As mentioned earlier, there are many ways to implement state machines. The following sections discuss some of the pros and cons associated with some of the more common state machine implementations.

D Flip-Flop Implementation

There are more products available that support a D flip-flop solution than any other implementation.

Table 1. Optimized Results for Clock Generator: T Flip-Flop Implementation

LOG/IC OPTIMIZATION SUMMARY (FACT)				
CPU TIME QUOTA PER FUNCTION: 100 SEC				
FUNCTION	INV	P-TERMS	CPU-TIME	FLAGS
CLK_1A.T	NO	6	<1	
	YES	7	1	
CLK_1B.T	NO	4	1	
	YES	3	1	
CLK_2A.T	NO	5	1	
	YES	4	<1	
CLK_2B.T	NO	4	1	
	YES	3	<1	
CLK_3A.T	NO	5	<1	
	YES	6	2	
CLK_3B.T	NO	4	<1	
	YES	2	<1	
CLK_A.T	NO			C
	YES			C
CLK_B.T	NO	2	1	
	YES	1	<1	
QQ1.T	NO	3	<1	
	YES	5	1	
QQ2.T	NO	6	<1	
	YES	11	2	

C: Constant Function
FACT MINIMIZATION: 11 SEC

Table 2. Non-optimized Results for Clock Generator: D Flip-Flop Implementation

LOG/IC OPTIMIZATION SUMMARY (FACT)				
CPU TIME QUOTA PER FUNCTION: 100 SEC				
FUNCTION	INV	P-TERMS	CPU-TIME	FLAGS
CLK_1A.D	NO	12	<1	N
	YES	27	<1	N
CLK_1B.D	NO	5	<1	N
	YES	34	1	N
CLK_2A.D	NO	8	<1	N
	YES	31	<1	N
CLK_2B.D	NO	7	<1	N
	YES	32	<1	N
CLK_3A.D	NO	8	<1	N
	YES	31	<1	N
CLK_3B.D	NO	6	<1	N
	YES	33	<1	N
CLK_A.D	NO			NT
	YES			NT
QQ1.D	NO	6	<1	N
	YES	5	<1	N
QQ2.D	NO	10	<1	N
	YES	9	<1	N

N: No Optimization
T: Trivial Function
FACT MINIMIZATION: 2 SEC

Therefore, it is usually the most cost-effective solution for a state machine.

Table 1 lists the number of product terms per output obtained by compiling the clock generator state machine definition with the LOG/iC software, using D flip-flops. The compiler input file appears in *Appendix A*. Optimizing the design (*Table 2*) significantly reduces the number of product terms needed.

T Flip-Flop Implementation

Even though D flip-flop solutions are more widely available, there are times when the logic needed for this implementation is prohibitively complex. Under these circumstances, a T flip-flop implementation might be more cost effective, because using T flip-flops reduces the logic significantly.

The best example of this situation is a simple synchronous binary counter. While the most significant bit (MSB) of an N-bit counter in a D flip-flop implementation requires N product terms, the T flip-flop solution requires only one product term. Note that the Cypress family of CY7C33x devices offers you a configurable T or D type implementation if you place an XOR gate prior to the D flip-flop; route the AND/OR array to one of the XOR's inputs and the flip-flop's Q output (via an additional product term) to the other XOR input.

It isn't clear from simple observation, however, whether the T flip-flop implementation is beneficial for the clock generator state machine. One way to clarify this question is to change three command lines in the state machine description shown in *Appendix A* and recompile to produce a T flip-flop implementation. *Table 3* contains the product term results using T flip-

flops. A quick study of the results reveals that the optimized version using D flip-flops (*Table 2*) requires fewer product terms than the T flip-flop version.

PLD Implementation

With the LOG/iC PLD Database option, the software assists in selecting a PLD, and it shows that the non-optimized version of the clock state machine fits in a PALC22V10 without further reduction. If the equations are reduced using Boolean reduction, however, a lower-cost solution is available. The results shown in *Table 3* indicate that the less expensive PALC20G10 would work. *Appendix A* shows the listing for the 20G10 LOG/iC implementation. Waveforms for the completed design appear in *Appendix B*. You can verify the CLK_A and CLK_B equation results against the equations generated in the State Machine Partitioning section of this application note.

PROM Implementation

You can obtain very high speed solutions by implementing state machines using PROMs. A PROM uses a look-up table to decode the machine's next state, as opposed to the AND/OR array in a PLD. The main advantage of using a look-up table to decode the next state is that every combination of the inputs can be decoded. Thus, you can create an extremely complex machine, without equation reductions.

The look-up table's drawback is that the PROM's depth grows exponentially (2^N, where N = # of inputs to the look-up table) with every additional input to the look-up table. To determine the depth required, notice that the present total input vector provides the inputs to the look-up table. The clock generator state machine has seven external inputs, six system outputs, and two state outputs, which indicates a feasible implementation using the CY7C277 (32K X 8) registered PROM.

Using a registered PROM such as the CY7C277 to implement the machine also helps to reduce the parts count, because the PROM implements both the state and system output registers. LOG/iC offers support for implementing state machines in PROMs, and only a few minor changes to the state machine description shown in *Appendix A* are required. *PROM replaces the *PAL command, some simple statements indicating the CY7C277 architecture (INPUTS = 15 AND OUTPUTS = 8) replaces the TYPE = statement, and PROGFORMAT = INTEL-HEX.

CY7C361 Implementation

A new way to obtain high-speed operation of state machines became available with Cypress Semiconductor's development of a revolutionary architecture that enables a CMOS PLD state machine part to operate at speeds in the 125-MHz range. The first part in this family is the CY7C361. The architectural innovations used to obtain 125-MHz operation require that you approach state machine design slightly differently than you would when designing with traditional PLD architectures.

Table 3. Optimized Results for Clock Generator: D Flip-Flop Implementation

LOG/IC OPTIMIZATION SUMMARY (FACT)				
CPU TIME QUOTA PER FUNCTION: 100 SEC				
FUNCTION	INV	P-TERMS	CPU-TIME	FLAGS
CLK_1A.D	NO	6	1	
	YES	11	2	
CLK_1B.D	NO	3	1	
	YES	4	<1	
CLK_2A.D	NO	4	1	
	YES	7	<1	
CLK_2B.D	NO	3	1	
	YES	4	<1	
CLK_3A.D	NO	4	1	
	YES	9	1	
CLK_3B.D	NO	3	<1	
	YES	3	1	
CLK_A.D	NO	1	<1	
	YES	2	<1	
CLK_B.D	NO	1	1	
	YES	2	<1	
QQ1.D	NO	3	<1	
	YES	3	1	
QQ2.D	NO	6	16	
	YES	6	2	
FACT MINIMIZATION: 29 SEC				

To fully understand the information in this section, consult the Cypress Semiconductor application note, "Understanding the CY7C361."

Using the clock generator state machine example, this section shows how you can generate a state diagram for the CY7C361 by following some simple rules. This diagram allows you to determine whether the design can fit in a CY7C361. The rule of thumb is that a state diagram with 32 or fewer state nodes will probably fit. (The likelihood of the implementation at that point depends totally upon split-input-array fitting issues.) You can convert the state diagram directly into Boolean equations (with no Boolean reduction required) and compile the equations into JEDEC code for the final implementation.

The CY7C361's condition-decode array has been optimized for use in state machine applications. As shown in *Figure 7*, the CY7C361 condition decoder contains the necessary logic to generate two kinds of state machine operations. The Entering a State operation should look familiar. The process used to generate the system output and state register equations (in the System and State Register Output Generation section of this application note) utilizes a similar equation form. There are two small differences, though.

First, the Entering a State equation shown in *Figure 7* assumes the present state conditions are available as single entities on the input array. That is, one state register uniquely defines each state, and therefore the present state is not encoded using multiple flip-flops, as is typical in traditional state machines. There is a special case, however, that allows you to encode the states into 32 or fewer state registers (providing a maximum total of 2^{32} states) so long as only one product term per state register is required in the decode of the next state. An example of this is a simple synchronous 32-bit binary counter using the TOGGLE (or T flip-flop) configuration.

The second design difference with the CY7C361 is that the Entering a State equation also shows all states (*SA*, *SB*, *SC*) that have an identical state path to *S0*. This is not necessarily the case when designing with traditional PLDs (as shown in *Figure 6*). The CY7C361, however, requires a machine definition in which all state paths into any given state are identical. You can easily convert an existing state diagram and satisfy the new condition by simply adding additional states for those states that do not meet the above condition.

To remain consistent with the naming conventions already defined for the clock generator example, two additional suffixes, "X" and "Y", indicate the additional states. For example, state BW in *Figure 6* has two state paths entering into it: WAIT * WEN from state 123A and a path from state AW. To meet the design conditions for the CY7C361, you add an additional state, BWX, such that state AW enters BWX with a state path of 1, and state 123A enters state BW with a state path of WAIT * WEN.

The CY7C361 implementation of the clock generator state machine appears in *Figure 8*. Note that both of the new states (BW and BWX) have exits with the same state path equation. Thus, the number of states in the state machine does not grow geometrically due to this new methodology.

In addition to the normal Entering a State equation, the CY7C361 supports operations in which multiple state paths go from one state to another, and each state path term contains only one input. *Figure 9* shows a diagram of this condition.

Another operation, called Leaving a State, proves especially useful in conjunction with the (Wait Until) TERMINATE state macrocell configuration in the

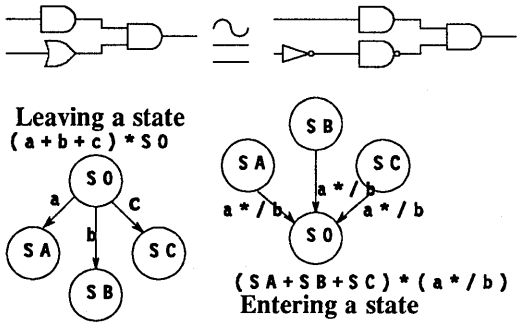


Figure 7. The Condition Decoder - Optimized for Two State Machine Operations

CY7C361. The flip-flop in this macrocell configuration is unconditionally set by the active previous state macrocell. The flip-flop remains set until the condition decoder equation (a Leaving a State equation) for the TERMINATE macrocell goes active. *Figure 10* shows how the TERMINATE configuration looks within a state diagram.

When implementing the clock generator state machine in the CY7C361 using the conversion techniques discussed above, the number of states slightly exceeds 32. But by allowing the machine's pipelined and nonpipelined portions to share common states, (1A, 1B, and 3B) the total number of states reduces to less than 32.

Note that you can use this same kind of state reduction for the original implementation (refer to the Unique States section). *Figure 8* shows the resulting state diagram.

It is a simple matter to convert information from the state diagram to PLD ToolKit Equations (refer to *Appendix C* for the PLD ToolKit source file). You must generate an Entering a State equation for every state node in the diagram. (The TERMINATE configuration was not used in this example, but it can be useful for implementing wait states.)

You generate the equations in *Appendix C* using the <PROD> and <INV_PROD> connectives for the AND and NAND terms, respectively. Then generate the system outputs by ORing the appropriate states in the OR-based output array. For example, the CLK_1B output is active during the 1B, 12B, 123B, or 123BX states. The PLD ToolKit connective for the OR array is <INV_SUM>. The CY7C361 implementation of the clock generator state machine was simulated using the PLD ToolKit (see *Appendix D*).

Reference

1. Donald D. Givone, *Introduction to Switching Circuit Theory* (New York: McGraw-Hill, Inc., 1970)

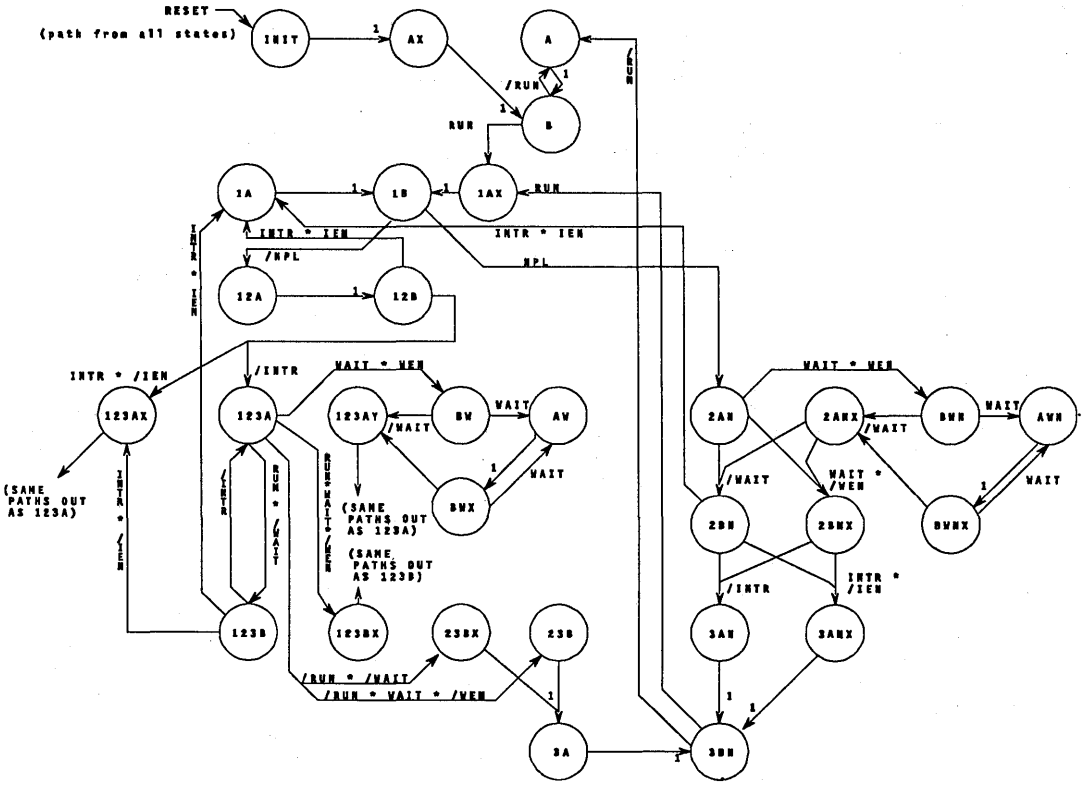


Figure 8. CY7C361 Implementation, CPU Clock State Machine

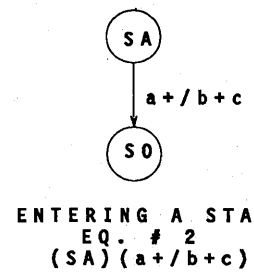


Figure 9. Entering a State Along Multiple Paths

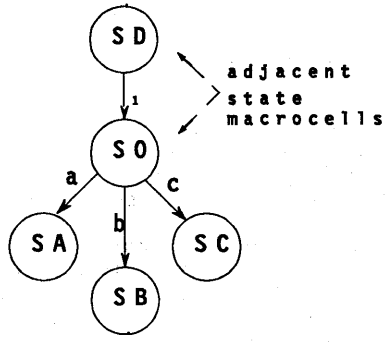


Figure 10. Leaving a State (TERMINATE Configuration)

Appendix A. LOG/iC PLD Source Code: Clock State Machine

```
LOG/iC-PAL      Rel 3.2/2-2328-1721/00034 # 32-5955 90/03/15  23:49:45
-------------------------------------------------------------------------------
LOG/iC - COPYRIGHT (C) 1985,1988 BY ISDATA GMBH, 7500 KARLSRUHE WEST-GERMANY

Cypress Semiconductor                    LICENCE FOR IBM-PC/XT/AT


Data Set: OD20G10.DCB

  1    1: *IDENTIFICATION
  2    2: PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
  3    3: ERIC B. ROSS
  4    4: CYPRESS SEMICONDUCTOR
  5    5: NAMING CONVENTION
  6    6: OD     = SYSTEM OUTPUTS ARE DFLOPS AND ARE USED FOR STATE DEF
  7    7: 20G10  = PALC20G10 IMPLEMENTATION
  8    8: *PAL
  9    9: TYPE= PALC20G10
 10 I  10:
 11    11: *X-NAMES
 12 I  12: ;-------------------------------------------------------------------
 13 I  13: ;INPUT DEFINITIONS :
 14 I  14: ; RUN    = START & STOP EXECUTION OF OUTPUT CLOCKS (NORMAL, SINGLE
 15 I  15: ;         STEP, & BREAK PT. EXECUTION
 16 I  16: ; NPL    = PIPELINED VS NON-PIPELINED MODE OF EXECUTION
 17 I  17: ; INTR   = EXTERNAL INTERRUPT CONDITION (TLB MISS, PARITY ERROR,...)
 18 I  18: ; IEN    = INTERRUPT ENABLE
 19 I  19: ; WAIT   = WAIT ENABLE (CACHE MISS)
 20 I  20: ; WEN    = WAIT ENABLE
 21 I  21: ;-------------------------------------------------------------------
 22 I  22: ;
 23    23: RUN, NPL, INTR, IEN, WAIT, WEN, RESET;
 24 I  24:
 25    25: *Z-NAMES
 26 I  26: ;-------------------------------------------------------------------
 27 I  27: ;OUTPUT DEFINITIONS :
 28 I  28: ;
 29 I  29: ; 3 CLOCK STAGES 1, 2, 3
 30 I  30: ; 2 CLOCKS PER STATE A, B
 31 I  31: ;   CLK_XX WHERE XX =  1A,1B,2A,2B,3A,3B
 32 I  32: ;
 33 I  33: ; 2 FREE RUNNING CLOCKS
 34 I  34: ;   CLK_A, CLK_B
 35 I  35: ;
 36 I  36: ; ADDITIONAL REGISTERS FOR STATE DEFINITION
 37 I  37: ;   QQ1, QQ2
 38 I  38: ;-------------------------------------------------------------------
 39 I  39: ;
 40    40: CLK_1A, CLK_1B, CLK_2A, CLK_2B, CLK_3A, CLK_3B, CLK_A, CLK_B, QQ1, QQ2;
 41 I  41:
 42    42: *Z-VALUES
 43 I  43:
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
44 I 44: ;                ADDITIONAL OUTPUTS
45 I 45: ;    SYSTEM OUTPUTS   FOR STATE DEFINITION
46 I 46: ;    ---------------  ------------------
47 I 47: ;
48 I 48: ;    C C C C C C C C  Q Q
49 I 49: ;    L L L L L L L L  Q Q
50 I 50: ;    K K K K K K K K  1 2
51 I 51: ;    1 1 2 2 3 3 A B
52 I 52: ;    A B A B A B
53 I 53:
54   54: S1 = 0 0 0 0 0 0 0 0  - -      ;INIT  COMMON STATES
55   55: S2 = 0 0 0 0 0 0 1 0  0 -      ;SA     - INACTIVE
56   56: S3 = 0 0 0 0 0 0 0 1  0 -      ;SB       MODE STATES
57 I 57:
58   58: S4 = 1 0 0 0 0 0 1 0  - 0      ;S1A   PIPELINE STATES
59   59: S5 = 0 1 0 0 0 0 0 1  - 0      ;S1B
60   60: S6 = 1 0 1 0 0 0 1 0  - -      ;S12A
61   61: S7 = 0 1 0 1 0 0 0 1  - -      ;S12B
62   62: S8 = 1 0 1 0 1 0 1 0  - -      ;S123A
63   63: S9 = 0 1 0 1 0 1 0 1  - -      ;S123B
64   64: S10 = 0 0 0 1 0 1 0 1  - -     ;S23B
65   65: S11 = 0 0 0 0 1 0 1 0  - 0     ;S3A
66   66: S12 = 0 0 0 0 0 1 0 1  - 0     ;S3B
67   67: S13 = 0 0 0 0 0 0 1 0  1 0     ;SAW
68   68: S14 = 0 0 0 0 0 0 0 1  1 0     ;SBW
69 I 69:
70   70: S15 = 1 0 0 0 0 0 1 0  - 1     ;S1AN  NON-PIPLINE
71   71: S16 = 0 1 0 0 0 0 0 1  - 1     ;S1BN
72   72: S17 = 0 0 1 0 0 0 1 0  - -     ;S2AN
73   73: S18 = 0 0 0 1 0 0 0 1  - -     ;S2BN
74   74: S19 = 0 0 0 0 1 0 1 0  - 1     ;S3AN
75   75: S20 = 0 0 0 0 0 1 0 1  - 1     ;S3BN
76   76: S21 = 0 0 0 0 0 0 1 0  1 1     ;SAWN
77   77: S22 = 0 0 0 0 0 0 0 1  1 1     ;SBWN
78 I 78:
79   79: *STRING
80   80: INIT  = 1              ; COMMON STATES
81   81: SA    = 2              ; -INACTIVE MODE
82   82: SB    = 3              ;  STATES
83 I 83:
84   84: S1A   = 4              ; PIPELINE STATES
85   85: S1B   = 5              ;
86   86: S12A  = 6              ;
87   87: S12B  = 7              ;
88   88: S123A = 8              ;
89   89: S123B = 9              ;
90   90: S23B  = 10             ;
91   91: S3A   = 11             ;
92   92: S3B   = 12             ;
93   93: SAW   = 13             ;
94   94: SBW   = 14             ;
95 I 95:
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
 96   96: S1AN   = 15                  ; NON-PIPLINE
 97   97: S1BN   = 16                  ;
 98   98: S2AN   = 17                  ;
 99   99: S2BN   = 18                  ;
100  100: S3AN   = 19                   ;
101  101: S3BN   = 20                   ;
102  102: SAWN   = 21                    ;
103  103: SBWN   = 22                    ;
104  104: LASTSTATE =  22;
105 I 105:
106  106: *FLOW-TABLE
107 I 107: ;
108 I 108: ;-------------------------------------------------------------------------
109 I 109: ;RESET STATE
110 I 110: ;ALL STATES MUST RESET TO THE INITIAL STATE (ALL OUTPUTS REGISTERS 0) UPON
111 I 111: ;AN ACTIVE RESET INPUT. SINCE THE 20G10 HAS NO GLOBAL OR INDIVIDUAL
112 I 112: ;RESETS TO THE OUTPUT REGISTERS, RESET TO INITIAL STATE MUST BE EMBEDDED
113 I 113: ;INTO THE STATE MACHINE
114 I 114: ;
115  115: RELEVENT =  RESET                ;
116  116: S['1'..'LASTSTATE'], X 1    , F 'INIT' ;ALL STATE  INIT UPON RESET
117  138: RELEVENT =  RESET =  0             ;
118 I 139: ;
119 I 140: ;-------------------------------------------------------------------------
120 I 141: ;INACTIVE MODE STATES
121  142: RELEVANT =  RUN, NPL             ;
122  143: S 'INIT'  , X - -       , F 'SA'   ;INITIAL STATE AFTER RESET
123 I 144:
124  145: S 'SA'    , X - -       , F 'SB'   ;INACTIVE MODE STATE, ONLY
125 I 146:
126  147: S 'SB'    , X 0 -       , F 'SA'   ;FREE RUN CLKS A & B ARE ACTIVE
127  148:           X 1 0       , F 'S1A'  ; PIPELINE VS.
128  149:           X 1 1       , F 'S1AN' ; NON-PIPELINE DECISION
129 I 150:
130 I 151: ;-------------------------------------------------------------------------
131 I 152: ;PIPELINE MODE STATES
132 I 153:
133  154: RELEVANT =  INTR, IEN              ;*PRIMING THE PIPELINE*
134  155: S 'S1A'   , X - -       , F 'S1B'  ;
135 I 156:
136  157: S 'S1B'   , X - -       , F 'S12A' ;
137 I 158:
138  159: S 'S12A'  , X - -       , F 'S12B' ;
139 I 160:
140  161: S 'S12B'  , X 1 1       , F 'S1A'  ;  INTERRUPT CONDITION ? YES
141  162:           X 1 0       , F 'S123A';                  NO
142  163:           X 0 -       , F 'S123A';                  NO
143 I 164:
144  165: RELEVANT =  RUN, INTR, IEN, WAIT, WEN; *FULL PIPELINE*
145  166: S 'S123A' , X - - - 1 1 , F 'SBW'    ;  WAIT CONDITION
146  167:           X 0 - - 0 - , F 'S23B'   ;  /RUN COND., EMPTY PIPELINE
147  168:           X 0 - - 1 0 , F 'S23B'   ;  /RUN COND., EMPTY PIPELINE
148  169:           X 1 - - 0 - , F 'S123B'  ;  RUN CONDITION
149  170:           X 1 - - 1 0 , F 'S123B'  ;  RUN CONDITION
150 I 171:
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
151  172: S 'S123B' , X - 1 1 - - , F 'S1A'    ;  INTERUPT CONDITION
152  173:          X - 0 - - - , F 'S123A'  ;  RUN CONDITION
153  174:          X - 1 0 - - , F 'S123A'  ;  RUN CONDITION
154 I 175:
155  176: RELEVANT =  RUN                    ; *EMPTY PIPELINE*
156  177: S 'S23B'  , X -         , F 'S3A'  ;
157 I 178:
158  179: S 'S3A'   , X -         , F 'S3B'  ;
159 I 180:
160  181: S 'S3B'   , X -         , F 'SA'   ;  BACK TO INACTIVE STATE
161 I 182:
162  183: RELEVANT =  WAIT                   ; *PIPELINE WAIT STATES*
163  184: S 'SBW'   , X 1         , F 'SAW'  ;  WAIT
164  185:          X 0         , F 'S123A' ;  /WAIT
165 I 186:
166  187: S 'SAW'   , X -         , F 'SBW'  ;
167 I 188:
168 I 189: ;-------------------------------------------------------------------------
169 I 190: ;NON-PIPELINE MODE STATES
170 I 191:
171  192: S 'S1AN'  , X -         , F 'S1BN' ;
172 I 193:
173  194: S 'S1BN'  , X -         , F 'S2AN' ;
174 I 195:
175  196: RELEVANT =  WAIT, WEN              ;
176  197: S 'S2AN'  , X 1 1       , F 'SBWN'   ; WAIT CONDITION
177  198:          X 0 -       , F 'S2BN'   ; /WAIT CONDITION
178  199:          X 1 0       , F 'S2BN'   ; /WAIT CONDITION
179 I 200:
180  201: RELEVANT =  INTR, IEN              ;
181  202: S 'S2BN'  , X 1 1       , F 'S1AN'   ; INTERRUPT CONDITION
182  203:          X 0 -       , F 'S3AN'   ; /INTERRUPT CONDITION
183  204:          X 1 0       , F 'S3AN'   ; /INTERRUPT CONDITION
184 I 205:
185  206: RELEVANT =  RUN                    ;
186  207: S 'S3AN'  , X -         , F 'S3BN' ;
187 I 208:
188  209: S 'S3BN'  , X 1         , F 'S1AN' ;
189  210:          X 0         , F 'SA'   ; BACK TO INACTIVE STATE
190 I 211:
191  212: RELEVANT =  WAIT                   ;*NON-PIPELINED WAIT STATES*
192  213: S 'SBWN'  , X 1         , F 'SAWN' ;  REMAIN IN WAIT
193  214:          X 0         , F 'S2AN' ;  END OF WAIT CONDITION
194 I 215:
195  216: S 'SAWN'  , X -         , F 'SBWN' ;  REMAIN IN WAIT
196 I 217:
197  218: *STATE-ASSIGNMENT
198  219: Z-VALUES
199 I 220:
200 I 221:
201  222: *PIN
202  223:  STATECLK =  1, RUN =  2, NPL =  3, INTR =  4, IEN =  5, WAIT =  6, WEN =  7,
203  223:  RESET =  8, CLK_1A =  14, CLK_1B =  15, CLK_2A =  16, CLK_2B =  17,
204  223:  CLK_3A =  18, CLK_3B =  19, CLK_A =  20, CLK_B =  21, QQ1 =  22, QQ2 =  23;
205 I 224:
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
206  225: *RUN-CONTROL
207  226: LISTING= LONG,SYMBOL-TABLE,EQUATIONS,PINOUT;
208  227: PROGFORMAT= L-EQUATIONS
209  228: OPTIMIAZATION= P-TERMS;
210  229: *END


 LOG/IC SYMBOL TABLE


 SYMBOL   TYPE        REG LEVEL PIN/NODE

 GND      LOCAL       - HIGH
 VCC      LOCAL       - HIGH
 RUN      X-VARIABLE   - HIGH      2
 NPL      X-VARIABLE   - HIGH      3
 INTR     X-VARIABLE   - HIGH      4
 IEN      X-VARIABLE   - HIGH      5
 WAIT     X-VARIABLE   - HIGH      6
 WEN      X-VARIABLE   - HIGH      7
 RESET    X-VARIABLE   - HIGH      8
 CLK_1A   X-VARIABLE   - HIGH     14
 CLK_1B   X-VARIABLE   - HIGH     15
 CLK_2A   X-VARIABLE   - HIGH     16
 CLK_2B   X-VARIABLE   - HIGH     17
 CLK_3A   X-VARIABLE   - HIGH     18
 CLK_3B   X-VARIABLE   - HIGH     19
 CLK_A    X-VARIABLE   - HIGH     20
 CLK_B    X-VARIABLE   - HIGH     21
 QQ1      X-VARIABLE   - HIGH     22
 QQ2      X-VARIABLE   - HIGH     23
 CLK_1A.D Z-VARIABLE   DFF HIGH     14
 CLK_1B.D Z-VARIABLE   DFF HIGH     15
 CLK_2A.D Z-VARIABLE   DFF HIGH     16
 CLK_2B.D Z-VARIABLE   DFF HIGH     17
 CLK_3A.D Z-VARIABLE   DFF HIGH     18
 CLK_3B.D Z-VARIABLE   DFF HIGH     19
 CLK_A.D  Z-VARIABLE   DFF HIGH     20
 CLK_B.D  Z-VARIABLE   DFF HIGH     21
 QQ1.D    Z-VARIABLE   DFF HIGH     22
 QQ2.D    Z-VARIABLE   DFF HIGH     23


 EXPANDED FUNCTION TABLE (INCLUDING LOCAL VARIABLES):
 ----------------------------------------------------
                    : CCCC CC
                    : LLLL LLCC
           CCC CCC  : KKKK KKLL
          RLLL LLLC C : ____ __KK QQ
      I W EKKK KKKL L : 1122 33__ QQ
   GVRN NIAW S___ ___K KQQ : ABAB ABAB 12
   NCUP TEIE E112 233_ _QQ : .... .... ..
   DCNL RNTN TABA BABA B12 : DDDD DDDD DD
   ---------------------------------------
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
---- ---- 1000 0000 0-- : 0000 0000 --;   1/ 116
---- ---- 0000 0000 0-- : 0000 0010 0-;   2/ 143
---- ---- 1000 0001 00- : 0000 0000 --;   3/ 117
---- ---- 0000 0001 00- : 0000 0001 0-;   4/ 145
---- ---- 1000 0000 10- : 0000 0000 --;   5/ 118
--0- ---- 0000 0000 10- : 0000 0010 0-;   6/ 147
--10 ---- 0000 0000 10- : 1000 0010 -0;   7/ 148
--11 ---- 0000 0000 10- : 1000 0010 -1;   8/ 149
---- ---- 1100 0001 0-0 : 0000 0000 --;   9/ 119
---- ---- 0100 0001 0-0 : 0100 0001 -0;  10/ 155
---- ---- 1010 0000 1-0 : 0000 0000 --;  11/ 120
---- ---- 0010 0000 1-0 : 1010 0010 --;  12/ 157
---- ---- 1101 0001 0-- : 0000 0000 --;  13/ 121
---- ---- 0101 0001 0-- : 0101 0001 --;  14/ 159
---- ---- 1010 1000 1-- : 0000 0000 --;  15/ 122
---- 11-- 0010 1000 1-- : 1000 0010 -0;  16/ 161
---- 10-- 0010 1000 1-- : 1010 1010 --;  17/ 162
---- 0--- 0010 1000 1-- : 1010 1010 --;  18/ 163
---- ---- 1101 0101 0-- : 0000 0000 --;  19/ 123
---- --11 0101 0101 0-- : 0000 0001 10;  20/ 166
--0- --0- 0101 0101 0-- : 0001 0101 --;  21/ 167
--0- --10 0101 0101 0-- : 0001 0101 --;  22/ 168
--1- --0- 0101 0101 0-- : 0101 0101 --;  23/ 169
--1- --10 0101 0101 0-- : 0101 0101 --;  24/ 170
---- ---- 1010 1010 1-- : 0000 0000 --;  25/ 124
---- 11-- 0010 1010 1-- : 1000 0010 -0;  26/ 172
---- 0--- 0010 1010 1-- : 1010 1010 --;  27/ 173
---- 10-- 0010 1010 1-- : 1010 1010 --;  28/ 174
---- ---- 1000 1010 1-- : 0000 0000 --;  29/ 125
---- ---- 0000 1010 1-- : 0000 1010 -0;  30/ 177
---- ---- 1000 0101 0-0 : 0000 0000 --;  31/ 126
---- ---- 0000 0101 0-0 : 0000 0101 -0;  32/ 179
---- ---- 1000 0010 1-0 : 0000 0000 --;  33/ 127
---- ---- 0000 0010 1-0 : 0000 0010 0-;  34/ 181
---- ---- 1000 0001 010 : 0000 0000 --;  35/ 128
---- ---- 0000 0001 010 : 0000 0001 10;  36/ 187
---- ---- 1000 0000 110 : 0000 0000 --;  37/ 129
---- --1- 0000 0000 110 : 0000 0010 10;  38/ 184
---- --0- 0000 0000 110 : 1010 1010 --;  39/ 185
---- ---- 1100 0001 0-1 : 0000 0000 --;  40/ 130
---- ---- 0100 0001 0-1 : 0100 0001 -1;  41/ 192
---- ---- 1010 0000 1-1 : 0000 0000 --;  42/ 131
---- ---- 0010 0000 1-1 : 0010 0010 --;  43/ 194
---- ---- 1001 0001 0-- : 0000 0000 --;  44/ 132
---- --11 0001 0001 0-- : 0000 0001 11;  45/ 197
---- --0- 0001 0001 0-- : 0001 0001 --;  46/ 198
---- --10 0001 0001 0-- : 0001 0001 --;  47/ 199
---- ---- 1000 1000 1-- : 0000 0000 --;  48/ 133
---- 11-- 0000 1000 1-- : 1000 0010 -1;  49/ 202
---- 0--- 0000 1000 1-- : 0000 1010 -1;  50/ 203
---- 10-- 0000 1000 1-- : 0000 1010 -1;  51/ 204
---- ---- 1000 0101 0-1 : 0000 0000 --;  52/ 134
---- ---- 0000 0101 0-1 : 0000 0101 -1;  53/ 207
---- ---- 1000 0010 1-1 : 0000 0000 --;  54/ 135
--1- ---- 0000 0010 1-1 : 1000 0010 -1;  55/ 209
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
   --0- ---- 0000 0010 1-1  :  0000 0010 0-;   56/  210
   ---- ---- 1000 0001 011  :  0000 0000 --;   57/  136
   ---- ---- 0000 0001 011  :  0000 0001 11;   58/  216
   ---- ---- 1000 0000 111  :  0000 0000 --;   59/  137
   ---- --1- 0000 0000 111  :  0000 0010 11;   60/  213
   ---- --0- 0000 0000 111  :  0010 0010 --;   61/  214
   REST                     :  ---- ---- --;   62
   ------------------------------------------
   1234 5678 9012 3456 789     1234 5678 90


STATE ASSIGNMENT:
-----------------
   CCCC CC
   LLLL LLCC
   KKKK KKLL
   _____KK QQ
   1122 33__ QQ
   ABAB ABAB 12
   ------------
   0000 0000 --; 1
   0000 0010 0-; 2
   0000 0001 0-; 3
   1000 0010 -0; 4
   0100 0001 -0; 5
   1010 0010 --; 6
   0101 0001 --; 7
   1010 1010 --; 8
   0101 0101 --; 9
   0001 0101 --; 10
   0000 1010 -0; 11
   0000 0101 -0; 12
   0000 0010 10; 13
   0000 0001 10; 14
   1000 0010 -1; 15
   0100 0001 -1; 16
   0010 0010 --; 17
   0001 0001 --; 18
   0000 1010 -1; 19
   0000 0101 -1; 20
   0000 0010 11; 21
   0000 0001 11; 22
EXPANDED FUNCTION TABLE (LOCAL VARIABLES REMOVED):
--------------------------------------------------
                    :  CCCC CC
                    :  LLLL LLCC
         C CCCC C   :  KKKK KKLL
        RL LLLL LCC :  _____KK QQ
    I  W EK KKKK KLL:  1122 33__ QQ
   RNNI AWS_____KKQ Q  :  ABAB ABAB 12
   UPTE IEE1 1223 3__Q Q  :  .... .... ..
   NLRN TNTA BABA BAB1 2  :  DDDD DDDD DD
   ---------------------------------------
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
---- --10 0000 000- - : 0000 0000 --;   1/ 116
---- --00 0000 000- - : 0000 0010 0-;   2/ 143
---- --10 0000 0100 - : 0000 0000 --;   3/ 117
---- --00 0000 0100 - : 0000 0001 0-;   4/ 145
---- --10 0000 0010 - : 0000 0000 --;   5/ 118
0--- --00 0000 0010 - : 0000 0010 0-;   6/ 147
10-- --00 0000 0010 - : 1000 0010 -0;   7/ 148
11-- --00 0000 0010 - : 1000 0010 -1;   8/ 149
---- --11 0000 010- 0 : 0000 0000 --;   9/ 119
---- --01 0000 010- 0 : 0100 0001 -0;  10/ 155
---- --10 1000 001- 0 : 0000 0000 --;  11/ 120
---- --00 1000 001- 0 : 1010 0010 --;  12/ 157
---- --11 0100 010- - : 0000 0000 --;  13/ 121
---- --01 0100 010- - : 0101 0001 --;  14/ 159
---- --10 1010 001- - : 0000 0000 --;  15/ 122
--11 --00 1010 001- - : 1000 0010 -0;  16/ 161
--10 --00 1010 001- - : 1010 1010 --;  17/ 162
--0- --00 1010 001- - : 1010 1010 --;  18/ 163
---- --11 0101 010- - : 0000 0000 --;  19/ 123
---- 1101 0101 010- - : 0000 0001 10;  20/ 166
0--- 0-01 0101 010- - : 0001 0101 --;  21/ 167
0--- 1001 0101 010- - : 0001 0101 --;  22/ 168
1--- 0-01 0101 010- - : 0101 0101 --;  23/ 169
1--- 1001 0101 010- - : 0101 0101 --;  24/ 170
---- --10 1010 101- - : 0000 0000 --;  25/ 124
--11 --00 1010 101- - : 1000 0010 -0;  26/ 172
--0- --00 1010 101- - : 1010 1010 --;  27/ 173
--10 --00 1010 101- - : 1010 1010 --;  28/ 174
---- --10 0010 101- - : 0000 0000 --;  29/ 125
---- --00 0010 101- - : 0000 1010 -0;  30/ 177
---- --10 0001 010- 0 : 0000 0000 --;  31/ 126
---- --00 0001 010- 0 : 0000 0101 -0;  32/ 179
---- --10 0000 101- 0 : 0000 0000 --;  33/ 127
---- --00 0000 101- 0 : 0000 0010 0-;  34/ 181
---- --10 0000 0101 0 : 0000 0000 --;  35/ 128
---- --00 0000 0101 0 : 0000 0001 10;  36/ 187
---- --10 0000 0011 0 : 0000 0000 --;  37/ 129
---- 1-00 0000 0011 0 : 0000 0010 10;  38/ 184
---- 0-00 0000 0011 0 : 1010 1010 --;  39/ 185
---- --11 0000 010- 1 : 0000 0000 --;  40/ 130
EXPANDED FUNCTION TABLE (LOCAL VARIABLES REMOVED)- continued :

---- --01 0000 010- 1 : 0100 0001 -1;  41/ 192
---- --10 1000 001- 1 : 0000 0000 --;  42/ 131
---- --00 1000 001- 1 : 0010 0010 --;  43/ 194
---- --10 0100 010- - : 0000 0000 --;  44/ 132
---- 1100 0100 010- - : 0000 0001 11;  45/ 197
---- 0-00 0100 010- - : 0001 0001 --;  46/ 198
---- 1000 0100 010- - : 0001 0001 --;  47/ 199
---- --10 0010 001- - : 0000 0000 --;  48/ 133
--11 --00 0010 001- - : 1000 0010 -1;  49/ 202
--0- --00 0010 001- - : 0000 1010 -1;  50/ 203
--10 --00 0010 001- - : 0000 1010 -1;  51/ 204
---- --10 0001 010- 1 : 0000 0000 --;  52/ 134
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
---- --00 0001 010- 1  :  0000 0101 -1;   53/  207
---- --10 0000 101- 1  :  0000 0000 --;   54/  135
1--- --00 0000 101- 1  :  1000 0010 -1;   55/  209
0--- --00 0000 101- 1  :  0000 0010 0-;   56/  210
---- --10 0000 0101 1  :  0000 0000 --;   57/  136
---- --00 0000 0101 1  :  0000 0001 11;   58/  216
---- --10 0000 0011 1  :  0000 0000 --;   59/  137
---- 1-00 0000 0011 1  :  0000 0010 11;   60/  213
---- 0-00 0000 0011 1  :  0010 0010 --;   61/  214
REST                   :  ---- ---- --;   62
---------------------------------------
1234 5678 9012 3456 7     1234 5678 90


PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
ERIC B. ROSS
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45

**************************************************
*** NET DESCRIPTION TABLE FOR AND/OR STRUCTURE  ***
**************************************************

                 :  CCCC CC
                 :  LLLL LLCC
     C CCCC C    :  KKKK KKLL
    RL LLLL LCC  :  ____ __KK QQ
   I  W EK KKKK KLL   :  1122 33__ QQ
RNNI AWS_ ____ _KKQ Q  :  ABAB ABAB 12
UPTE IEE1 1223 3_ Q Q  :  .... .... ..
NLRN TNTA BABA BAB1 2  :  DDDD DDDD DD
---------------------------------------
              INV  .... .... ..
              REG  DDDD DDDD DD
---------------------------------------
---- 0-0- --0- 0-11 0  :  A... .... .. ;  1
1--- --0- --0- 1--- 1  :  A... .... .. ;  2
---- --0- 1--- ---- 0  :  A... .... .. ;  3
---- --0- 1-1- ---- -  :  A... .... .. ;  4
--11 --0- --1- 0--- -  :  A... .... .. ;  5
1--- --0- 0-0- 0-10 -  :  A... .... .. ;  6
---- --01 ---0 ---- -  :  .A.. .... .. ;  7
1--- -001 ---- ---- -  :  .A.. .... .. ;  8
1--- 0-01 ---- ---- -  :  .A.. .... .. ;  9
---0 --0- 1--- ---- -  :  ..A. .... .. ; 10
--0- --0- 1--- ---- -  :  ..A. .... .. ; 11
---- 0-0- --0- 0-11 -  :  ..A. .... .. ; 12
---- --0- 1-0- ---- -  :  ..A. .... .. ; 13
---- 0-0- -1-- ---- -  :  ...A .... .. ; 14
---- -00- -1-- ---- -  :  ...A .... .. ; 15
---- --01 -1-0 ---- -  :  ...A .... .. ; 16
---0 --0- --1- ---- -  :  .... A... .. ; 17
--0- --0- --1- ---- -  :  .... A... .. ; 18
---- --0- 0-1- 1--- -  :  .... A... .. ; 19
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
---- 0-0- 0-0- 0-11 0 : .... A... .. ; 20
---- 0-0- ---1 ---- - : .... .A.. .. ; 21
---- -00- ---1 ---- - : .... .A.. .. ; 22
---- --0- -0-1 ---- - : .... .A.. .. ; 23
---- --0- ---- -0-- - : .... ..A. .. ; 24
---- --0- ---- -1-- - : .... ...A .. ; 25
---- ---- ---- -1-1 - : .... .... A. ; 26
---- ---- ---- 0-11 - : .... .... A. ; 27
---- ---- -1-- ---- - : .... .... A. ; 28
---- ---- --0- 1--- - : .... .... .A ; 29
---- ---- 0-1- 0--- - : .... .... .A ; 30
---- ---0 -1-- ---- - : .... .... .A ; 31
---- ---- -0-- -1-- 1 : .... .... .A ; 32
-1-- ---0 0--0 0--0 - : .... .... .A ; 33
---- ---- 00-- 0--1 1 : .... .... .A ; 34
-------------------------------------
1234 5678 9012 3456 7 : 1234 5678 90
 PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
 ERIC B. ROSS
 CYPRESS SEMICONDUCTOR
 90/03/15  23:49:45

 ****************************************************
 ***          BOOLEAN EQUATIONS            ***
 ****************************************************

CLK_1A.D   :=
     /WAIT  & /RESET  & /CLK_2B     & /CLK_3B     & CLK_B
          & QQ1   & /QQ2
    +  RUN   & /RESET  & /CLK_2B     & CLK_3B     & QQ2
    +  /RESET  & CLK_1B     & /QQ2
    +  /RESET  & CLK_1B     & CLK_2B
    +  INTR  & IEN   & /RESET  & CLK_2B     & /CLK_3B
    +  RUN   & /RESET  & /CLK_1B     & /CLK_2B     & /CLK_3B
          & CLK_B  & /QQ1   ;

CLK_1B.D   :=
     /RESET  & CLK_1A     & /CLK_3A
    +  RUN   & /WEN   & /RESET  & CLK_1A
    +  RUN   & /WAIT  & /RESET  & CLK_1A     ;

CLK_2A.D   :=
     /IEN   & /RESET  & CLK_1B
    +  /INTR  & /RESET  & CLK_1B
    +  /WAIT  & /RESET  & /CLK_2B     & /CLK_3B     & CLK_B
          & QQ1
    +  /RESET  & CLK_1B     & /CLK_2B     ;

CLK_2B.D   :=
     /WAIT  & /RESET  & CLK_2A
    +  /WEN   & /RESET  & CLK_2A
    +  /RESET  & CLK_1A     & CLK_2A     & /CLK_3A     ;
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
CLK_3A.D   :=
     /IEN   & /RESET & CLK_2B
   + /INTR  & /RESET & CLK_2B
   + /RESET & /CLK_1B    & CLK_2B    & CLK_3B
   + /WAIT  & /RESET & /CLK_1B    & /CLK_2B    & /CLK_3B
        & CLK_B & QQ1   & /QQ2   ;

CLK_3B.D   :=
     /WAIT  & /RESET & CLK_3A
   + /WEN   & /RESET & CLK_3A
   + /RESET & /CLK_2A    & CLK_3A    ;

CLK_A.D    :=
     /RESET & /CLK_A ;

CLK_B.D    :=
     /RESET & CLK_A ;

QQ1.D :=  CLK_A & QQ1
   + /CLK_3B    & CLK_B & QQ1
   +  CLK_2A    ;

QQ2.D :=  /CLK_2B    & CLK_3B
   + /CLK_1B    & CLK_2B    & /CLK_3B
   + /CLK_1A    & CLK_2A
   + /CLK_2A    & CLK_A & QQ2
   +  NPL   & /CLK_1A    & /CLK_1B    & /CLK_3A
        & /CLK_3B    & /QQ1
   + /CLK_1B    & /CLK_2A    & /CLK_3B    & QQ1   & QQ2   ;

PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
ERIC B. ROSS
 CYPRESS  SEMICONDUCTOR
90/03/15  23:49:45

           PALC20G10


    STATECLK  1      24  @VCC

       RUN  2       23  QQ2

       NPL  3       22  QQ1

      INTR  4       21  CLK_B

       IEN  5       20  CLK_A

      WAIT  6       19  CLK_3B

       WEN  7       18  CLK_3A

     RESET  8       17  CLK_2B
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
@09  9        16 CLK_2A

@10 10        15 CLK_1B

@11 11        14 CLK_1A

@GND 12       13 @OE
```

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
ERIC B. ROSS
 CYPRESS  SEMICONDUCTOR
90/03/15  23:49:45

                S
                T
                A
                T
          I     E  @
          N N R C V Q  Q
          T P U L C Q  Q
          R L N K C 2  1

          4 3 2 1 28 27 26

        5             25 CLK_B

  IEN   6              24 CLK_A

 WAIT   7              23 CLK_3B
         PALC20G10
        8             22 CLK_3A
            LCC
  WEN   9              21 CLK_2B

RESET  10              20 CLK_2A

       11             19

         12 13 14 15 16 17 18

          @  @  @  @  @  C  C
          0  1  1  G  O  L  L
          9  0  1  N  E  K  K
                D
                    _  _
                    1  1
                    A  B

PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
ERIC B. ROSS
 CYPRESS  SEMICONDUCTOR
90/03/15  23:49:45
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (Continued)

```
                S
                T
                A
                T
                E   @
           N R C   V Q
           P U L   C Q
           L N K   C 2

        4 3 2 1 28 27 26

   INTR  5            25 QQ1

    IEN  6            24 CLK_B

   WAIT  7            23 CLK_A
         PALC20G10
    WEN  8            22 CLK_3B
           PLCC
  RESET  9            21 CLK_3A

   @09  10            20 CLK_2B

      11              19 CLK_2A

       12 13 14 15 16 17 18

        @ @ @ @ C C
        1 1 G O L L
        0 1 N E K K
            D
              1 1
              A B
```

LOG/iC - PAL CPU TIME USED: 45 SEC

Appendix B. LOG/iC Simulation: Clock State Machine

```
PIPELINED CLOCKING SYSTEM OD20G10 3/7/90
                              C C C C C C
 E S                        R  L L L L  L L C C
 v t        I    W         E  K K K K  K K L L
 e a   R N N I  A W S       _ _ _ _ _  _ K K
 n t   U P T E  I E E       1 1 2 2 3  3 _ _
 t e   N L R N  T N T       A B A B A  B A B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

Top of trace buffer
 1  1 IU : :
 1  1 IC :          :
 1  1 IU :          :
 2  1 IU :          :
 2  1 IC :          :
 2  1 IU :          :
 3  1 IU :         :
 3  1 IC :          :
 3  2 IU :          :
 4  2 IU :          :
 4  2 IC :          :
 4  3 IU :          :
 5  3 IU :          :
 5  3 IC :          :
 5  2 IU :          :
 6  2 IU :          :
 6  2 IC :          :
 6  3 IU :          :
 7  3 IU :         :
 7  3 IC :          :
 7  4 IU :          :
 8  4 IU :          :
 8  4 IC :          :
 8  5 IU :          :
 9  5 IU :          :
 9  5 IC :          :
 9  6 IU :          :
10  6 IU :           :
10  6 IC :          :
10  7 IU :          :
11  7 IU :          :
11  7 IC :          :
11  8 IU :          :
12  8 IU :          :
12  8 IC :          :
12  9 IU :          :
13  9 IU :          :
13  9 IC :          :
13  8 IU :          :
14  8 IU :          :
14  8 IC :          :
14  9 IU :          :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
                                C C C C  C C
 E  S                        R  L L L L  L L  C C
 v  t          I    W       E  K K K K  K K  L L
 e  a    R N N I  A W S                       K K
 n  t    U P T E  I E E  1_ 1_ 2_ 2_ 3_ 3_
 t  e    N L R N  T N T  A B A B  A B A B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

15  9 IU :          :
15  9 IC :           :
15  8 IU :           :
16  8 IU :           :
16  8 IC :           :
16 10 IU :            :
17 10 IU :            :
17 10 IC :            :
17 11 IU :            :
18 11 IU :            :
18 11 IC :            :
18 12 IU :            :
19 12 IU :            :
19 12 IC :            :
19  2 IU :            :
20  2 IU :            :
20  2 IC :            :
20  3 IU :            :
21  3 IU :            :
21  3 IC :            :
21  2 IU :            :
22  2 IU :          :
22  2 IC :           :
22  3 IU :           :
23  3 IU :          :
23  3 IC :           :
23 15 IU :            :
24 15 IU :            :
24 15 IC :            :
24 16 IU :            :
25 16 IU :            :
25 16 IC :            :
25 17 IU :            :
26 17 IU :            :
26 17 IC :            :
26 18 IU :            :
27 18 IU :            :
27 18 IC :            :
27 19 IU :            :
28 19 IU :            :
28 19 IC :            :
28 20 IU :            :
29 20 IU :            :
29 20 IC :            :
29 15 IU :            :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10 3/7/90
                                C C C C C C
 E S                          R  L L L L  L L C C
 v t       I    W    E  K K K K  K K L L
 e a   R N N I  A W S  _ _ _ _  _ _ K K
 n t   U P T E  I E E  1 1 2 2  3 3 _ _
 t e   N L R N  T N T  A B A B  A B A B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

30 15 IU :          :
30 15 IC :          :
30 16 IU :          :
31 16 IU :          :
31 16 IC :          :
31 17 IU :          :
32 17 IU :          :
32 17 IC :          :
32 18 IU :          :
33 18 IU :          :
33 18 IC :          :
33 19 IU :          :
34 19 IU :          :
34 19 IC :          :
34 20 IU :          :
35 20 IU :          :
35 20 IC :          :
35 15 IU :          :
36 15 IU :         :
36 15 IC :          :
36 16 IU :          :
37 16 IU :          :
37 16 IC :          :
37 17 IU :          :
38 17 IU :          :
38 17 IC :          :
38 18 IU :          :
39 18 IU :          :
39 18 IC :          :
39 19 IU :          :
40 19 IU :          :
40 19 IC :          :
40 20 IU :          :
41 20 IU :          :
41 20 IC :          :
41  2 IU :          :
42  2 IU :          :
42  2 IC :          :
42  3 IU :          :
43  3 IU :          :
43  3 IC :          :
43  2 IU :          :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10 3/7/90
                                C C C C  C C
 E  S                        R  L L L L  L L C C
 v  t        I   W           E  K K K K  K K L L
 e  a   R N N I  A W S       _ _ _ _ _ _ K K
 n  t   U P T E  I E E       1 1 2 2 3 3 _ _
 t  e   N L R N  T N T       A B A B A B A B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

44  2 IU :       :
44  2 IC :         :
44  3 IU :         :
45  3 IU :        :
45  3 IC :         :
45  4 IU :         :
46  4 IU :         :
46  4 IC :         :
46  5 IU :         :
47  5 IU :         :
47  5 IC :         :
47  6 IU :         :
48  6 IU :         :
48  6 IC :         :
48  7 IU :         :
49  7 IU :         :
49  7 IC :         :
49  8 IU :         :
50  8 IU :         :
50  8 IC :         :
50  9 IU :         :
51  9 IU :         :
51  9 IC :         :
51  8 IU :         :
52  8 IU :        :
52  8 IC :         :
52  9 IU :         :
53  9 IU :         :
53  9 IC :         :
53  4 IU :         :
54  4 IU :         :
54  4 IC :         :
54  5 IU :         :
55  5 IU :        :
55  5 IC :         :
55  6 IU :         :
56  6 IU :         :
56  6 IC :         :
56  7 IU :         :
57  7 IU :         :
57  7 IC :         :
57  8 IU :         :
58  8 IU :         :
58  8 IC :         :
58  9 IU :         :
59  9 IU :         :
59  9 IC :         :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
                                C C C C  C C
 E  S                        R  L L L L  L L  C C
 v  t        I    W    E  K K K K  K K  L L
 e  a   R N N I  A W  S                    K K
 n  t   U P T E  I E  E  1 1 2 2  3 3
 t  e   N L R N  T N  T  A B A B  A B  A B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

59  8 IU :          :
60  8 IU :         :
60  8 IC :          :
60  9 IU :          :
61  9 IU :          :
61  9 IC :          :
61  8 IU :          :
62  8 IU :         :
62  8 IC :          :
62 14 I  :          :
63 14 I  :          :
63 14 IC :           :
63 13 I  :          :
64 13 I  :          :
64 13 IC :           :
64 14 I  :          :
65 14 I  :         :
65 14 IC :           :
65 13 I  :          :
66 13 I  :          :
66 13 IC :           :
66 14 I  :          :
67 14 I  :         :
67 14 IC :           :
67  8 IU :          :
68  8 IU :         :
68  8 IC :          :
68  9 IU :          :
69  9 IU :          :
69  9 IC :          :
69  8 IU :          :
70  8 IU :          :
70  8 IC :          :
70  9 IU :          :
71  9 IU :         :
71  9 IC :          :
71  1 IU :          :
72  1 IU :          :
72  1 IC :          :
72  1 IU :          :
73  1 IU :          :
73  1 IC :          :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10 3/7/90
                                C C C C C C
 E S                          R L L L L L C C
 v t       I    W    E  K K K K K K L L
 e a   R N N I  A W S  _ _ _ _ _ _    K K
 n t   U P T E  I E E  1 1 2 2 3 3  _ _
 t e   N L R N  T N T  A B A B A B  A B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

 73  1 IU :          :
 74  1 IU :         :
 74  1 IC :          :
 74  2 IU :          :
 75  2 IU :          :
 75  2 IC :          :
 75  3 IU :          :
 76  3 IU :          :
 76  3 IC :          :
 76  4 IU :          :
 77  4 IU :          :
 77  4 IC :          :
 77  5 IU :          :
 78  5 IU :         :
 78  5 IC :          :
 78  6 IU :          :
 79  6 IU :          :
 79  6 IC :          :
 79  7 IU :          :
 80  7 IU :          :
 80  7 IC :          :
 80  4 IU :          :
 81  4 IU :          :
 81  4 IC :          :
 81  5 IU :          :
 82  5 IU :         :
 82  5 IC :          :
 82  6 IU :          :
 83  6 IU :          :
 83  6 IC :          :
 83  7 IU :          :
 84  7 IU :          :
 84  7 IC :          :
 84  8 IU :          :
 85  8 IU :          :
 85  8 IC :          :
 85  9 IU :          :
 86  9 IU :          :
 86  9 IC :          :
 86  8 IU :          :
 87  8 IU :         :
 87  8 IC :          :
 87  9 IU :          :
 88  9 IU :          :
 88  9 IC :          :
 88  8 IU :          :
 89  8 IU :         :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
                                C  C  C  C   C  C
 E  S                     R   L  L  L  L   L  L  C  C
 v  t         I    W      E   K  K  K  K   K  K  L  L
 e  a    R  N  N  I  A  W  S   _  _  _  _   _  _  K  K
 n  t    U  P  T  E  I  E  E   1  1  2  2   3  3  _  _
 t  e    N  L  R  N  T  N  T   A  B  A  B   A  B  A  B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

 89   8 IC :              :
 89   1 IU :              :
 90   1 IU :             :
 90   1 IC :              :
 90   2 IU :              :
 91   2 IU :              :
 91   2 IC :              :
 91   3 IU :              :
 92   3 IU :              :
 92   3 IC :              :
 92  15 IU :               :
 93  15 IU :               :
 93  15 IC :               :
 93  16 IU :               :
 94  16 IU :             :
 94  16 IC :               :
 94  17 IU :               :
 95  17 IU :               :
 95  17 IC :               :
 95  18 IU :               :
 96  18 IU :               :
 96  18 IC :               :
 96  15 IU :               :
 97  15 IU :               :
 97  15 IC :               :
 97  16 IU :               :
 98  16 IU :             :
 98  16 IC :               :
 98  17 IU :               :
 99  17 IU :             :
 99  17 IC :               :
 99  22 I  :              :
100  22 I  :              :
100  22 IC :               :
100  21 I  :              :
101  21 I  :              :
101  21 IC :               :
101  22 I  :              :
102  22 I  :              :
102  22 IC :               :
102  21 I  :              :
```

Appendix B. LOG/iC Simulation: Clock State Machine (Continued)

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90
                                C C C C  C C
 E  S                        R  L L L L  L L C  C
 v  t        I    W         E  K K K K  K K L  L
 e  a   R N N I  A W S       _  _ _ _  _ _ K  K
 n  t   U P T E  I E E       1  1 2 2  3 3 _  _
 t  e   N L R N  T N T       A  B A B  A B A  B

 #  #    0-1 0-1 0-1 0-1  0-1 0-1 0-1 : 0-1 0-1 0-1 0-1  0-1 0-1 0-1 0-1  0

103 21 I :          :
103 21 IC :           :
103 22 I :           :
104 22 I :           :
104 22 IC :           :
104 17 IU :           :
105 17 IU :           :
105 17 IC :           :
105 18 IU :           :
106 18 IU :           :
106 18 IC :           :
106 19 IU :           :
107 19 IU :           :
107 19 IC :           :
107 20 IU :           :
108 20 IU :           :
108 20 IC :           :
108 15 IU :           :
109 15 IU :           :
109 15 IC :           :
109 16 IU :           :
110 16 IU :           :
110 16 IC :           :
110 17 IU :           :
111 17 IU :           :
111 17 IC :           :
111 18 IU :           :
112 18 IU :           :
112 18 IC :           :
112 19 IU :           :
113 19 IU :           :
113 19 IC :           :
113 20 IU :           :
114 20 IU :          :
114 20 IC :           :
114  2 IU :           :
115  2 IU :           :
115  2 IC :           :
115  3 IU :           :
116  3 IU :           :
116  3 IC :           :
116  2 IU :           :
```

Appendix C. Cypress PLD ToolKit: CY7C361 Implementation

```
CY7C361;

{PIPELINED CLOCKING SYSTEM AN1_361 4/27/90
 ERIC B. ROSS
 CYPRESS SEMICONDUCTOR}

CONFIGURE;

{-------------------------------------------------------------------------------------------- ----------
; INPUT DEFINITIONS :
; RUN    = START & STOP EXECUTION OF OUTPUT CLOCKS (NORMAL, SINGLE STEP,
;            & BREAK PT. EXECUTION
; NPL    = PIPELINED VS NON-PIPELINED MODE OF EXECUTION
; INTR   = EXTERNAL INTERUPT CONDITION (TLB MISS, PARITY ERROR,...)
; IEN    = INTERRUPT ENABLE
; WAIT   = WAIT ENABLE (CACHE MISS)
; WEN    = WAIT ENABLE
; RPT_EO = USED TO DUB CLK_1B, CLK USED TO UPDATE THE EO REG

;-------------------------------------------------------------------------------------------- ----------
;OUTPUT DEFINITIONS :
;
; 3 CLOCK STAGES 1, 2, 3
; 2 CLOCKS PER STATE A, B
;    CLK_XX WHERE XX =  1A,1B,2A,2B,3A,3B
;
; 2 FREE RUNNING CLOCKS
;    CLK_A, CLK_B
;
;-------------------------------------------------------------------------------------------- ----------
}
RUN(node= 3), STATECLK, NPL, INTR,                    {*INPUTS*}
IEN(node= 9), WAIT, WEN, RESET,
/RPT_EO,

/CLK_A(node= 16), /CLK_B, /CLK_1A,                    {*OUTPUTS*}
/CLK_2B(node= 24), /CLK_1B(and), /CLK_2A, /CLK_3A,
/CLK_3B,

{ LOCAL 8      LOCAL 8      HALF 16      GLOBAL 32    *STATE MACROCELLS*
  FEEDBACK     FEEDBACK     FEEDBACK     FEEDBACK
  ___________  ___________  ___________  ___________  START = DEFAULT}

AX(node= 32),  A,           B,           1B,          {LOCAL 8 =  1, HALF =  1}
1A,            1AX,         12A,         123A,

BW,            AW,          12B,         123AX,       {LOCAL 8 =  2, HALF =  1}
BWX,                                     123AY(node= 47),

AWN,           BWN,         2AN,         123B,        {LOCAL 8 =  1, HALF =  2}
               BWNX(node= 53),2ANX,      123BX,

23B,           23BX,        2BNX,        2BN,         {LOCAL 8 =  2, HALF =  2}
3A,            3AN,         3ANX,        3BN,
```

Appendix C. Cypress PLD ToolKit: CY7C361 Implementation (Continued)

```
IENA(node= 29),IENB,                                    {*MISC*}
GLBRST(node= 64),
GND(NODE= 73),
CLKDB(NODE= 74)


EQUATIONS;

  GLBRST = < prod> RESET;                               {*MISC*}

  IENA = < INV_SUM> /GND;

  IENB = < INV_SUM> /GND;

  AX  = < prod> /RESET;                 {*STATE MACROCELLS}

  A     = < prod> /RUN
          < inv_prod> /B * /3BN;

  B     = < prod>
          <inv_prod> /AX * /1AX

  1B    =  < prod>
           < inv_prod> /1A * /1AX;

  1A    =  < prod> INTR * IEN
           < inv_prod> /123B * /123BX * /12B * /2BN;

  1AX   =  < prod> RUN
           < inv_prod> /B * /3BN;

  12A   =  < prod> /NPL
           < inv_prod> /1B;

  123A  =  < prod> /INTR
           < inv_prod> /12B * /123B * /123BX;

  BW    =  < prod> WAIT * WEN
           < inv_prod> /123A * /123AX * /123AY;

  AW    =  < prod> WAIT
           < inv_prod>  /BW * /BWX;

  12B   =  < prod> 12A;

  123AX=   < prod> INTR * /IEN
           < inv_prod> /12B * /123B * /123BX;

  BWX   =  < prod> AW;

  123AY=   < prod> /WAIT
           < inv_prod> /BW * /BWX;

  AWN   =  < prod> WAIT
           < inv_prod> /BWN * /BWNX;
```

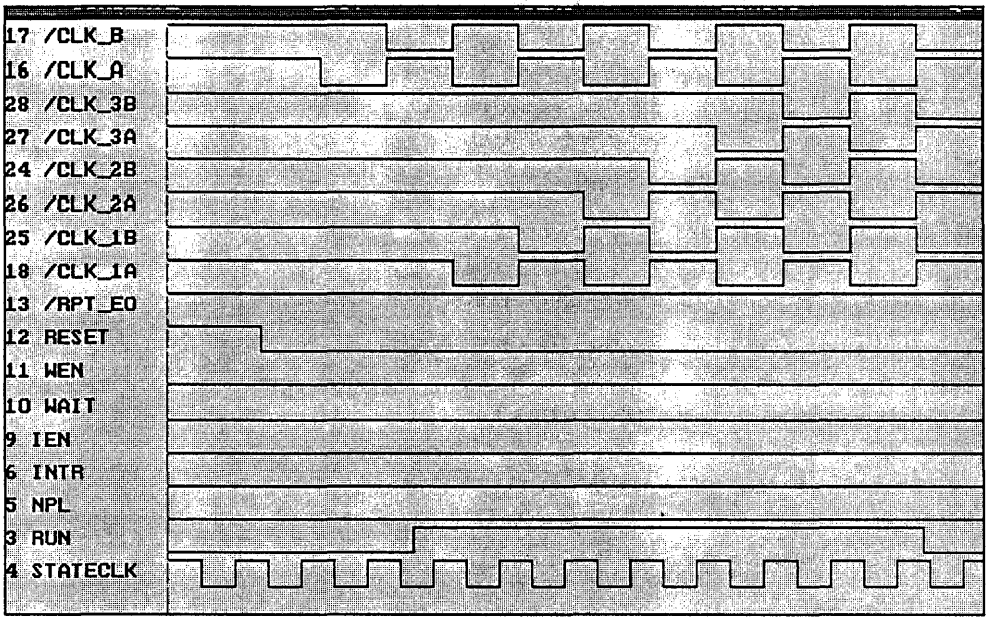
Appendix C. Cypress PLD ToolKit: CY7C361 Implementation (Continued)

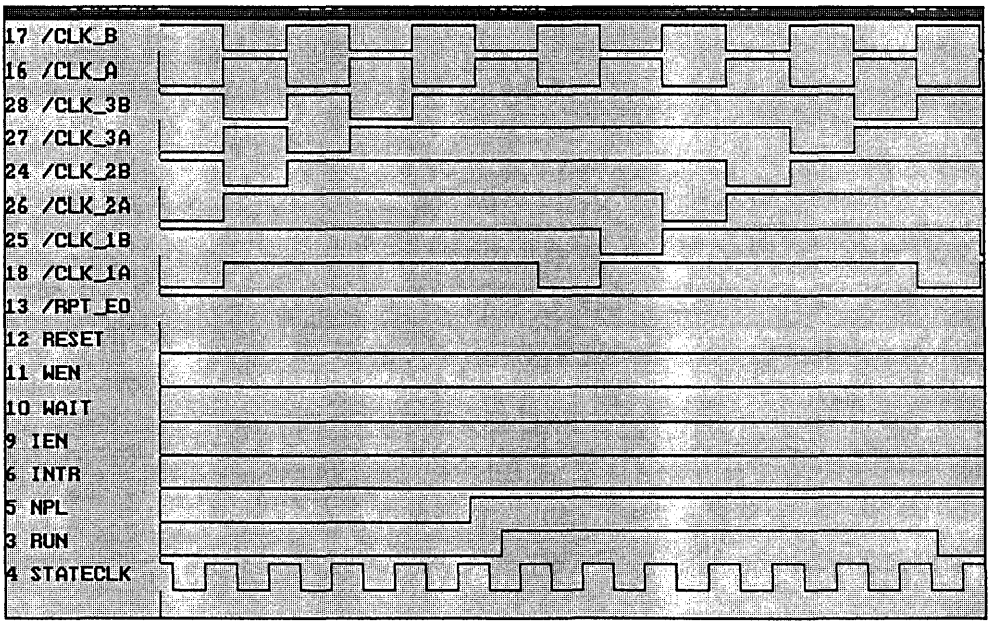
```
BWN  = < prod>  WAIT * WEN
       < inv_prod>  /2AN;

2AN  = < prod>  NPL
       < inv_prod>  /1B;

123B = < prod>  RUN * /WAIT
       < inv_prod>  /123A * /123AX * /123AY;

BWNX = < prod>  AWN;

2ANX = < prod>  /WAIT
       < inv_prod>  /BWN * /BWNX;

123BX = < prod>  RUN * WAIT * /WEN
        < inv_prod>  /123A * /123AX * /123AY;

23B  = < prod> /RUN * WAIT * /WEN
       < inv_prod>  /123A * /123AX * /123AY;

23BX = < prod>  /RUN * /WAIT
       < inv_prod>  /123A * /123AX * /123AY;

2BNX = < prod>  WAIT * /WEN
       < inv_prod>  /2AN * /2ANX;

2BN  = < prod>  /WAIT
       < inv_prod>  /2AN * /2ANX;

3A   = < prod>
       < inv_prod>  /23B * /23BX;

3AN  = < prod> /INTR
       < inv_prod>  /2BN * /2BNX;

3ANX = < prod>  INTR * /IEN
       < inv_prod>  /2BN * /2BNX;


3BN  = < prod>
       < inv_prod>  /3A * /3AN * /3ANX;
```

Appendix C. Cypress PLD ToolKit: CY7C361 Implementation (Continued)

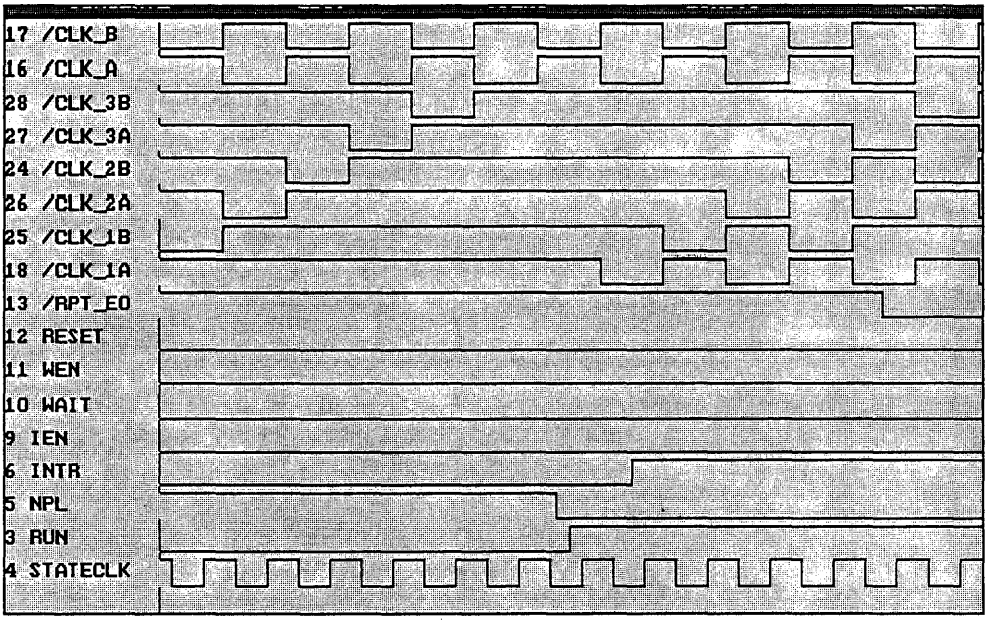
```
CLK_A  = < inv_sum>  /A * /AX * /1A * /1AX *  {*OUTPUTS*}
                     /12A * /123A * /123AX *
                     /123AY * /AW * /3A * /2AN *
                     /2ANX * /AWN * /3AN * /3ANX;

CLK_B  = < inv_sum>  /B * /1B * /12B * /123B *
                     /123BX * /BW * /BWX * /23B *
                     /23BX * /2BN * /2BNX * /BWN *
                     /BWNX * /3BN;

CLK_1A = < inv_sum>  /1A * /1AX * /12A * /123A *
                     /123AX * /123AY;

CLK_1B = < inv_sum>  /1B * /12B * /123B * /123BX;

CLK_2A = < inv_sum>  /12A * /123A * /123AX *
                     /123AY * /2AN * /2ANX;

CLK_2B = < inv_sum>  /12B * /123B * /123BX *
                     /23B * /23BX * /2BN * /2BNX;

CLK_3A = < inv_sum>  /123A * /123AX * /123AY *
                     /3A * /3AN * /3ANX;

CLK_3B = < inv_sum>  /123B * /123BX * /23B *
                     /23BX * /3BN;
```

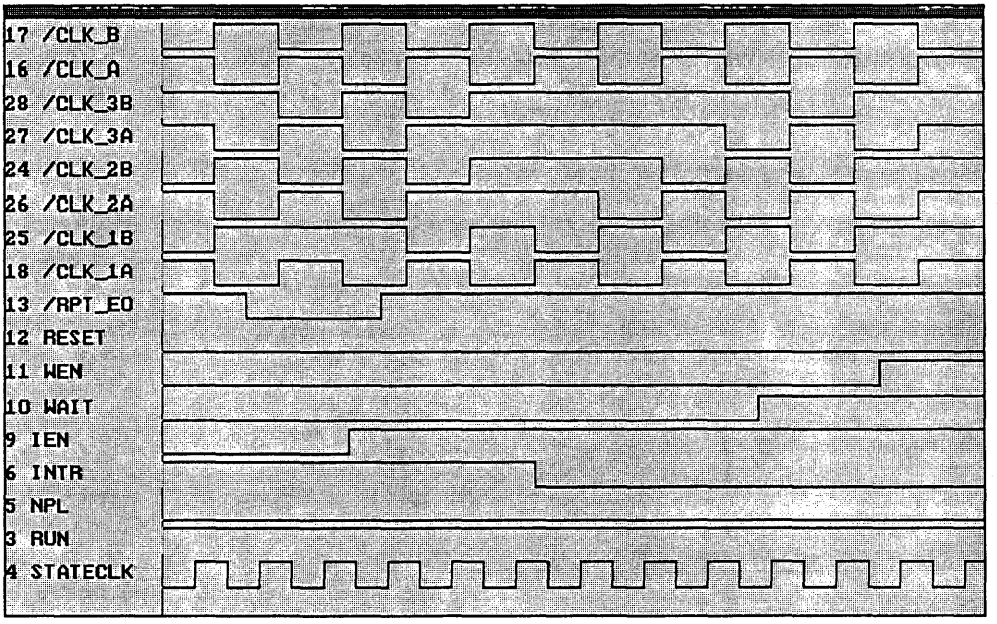
Appendix D. Cypress PLD ToolKit: CY7C361 Simulation



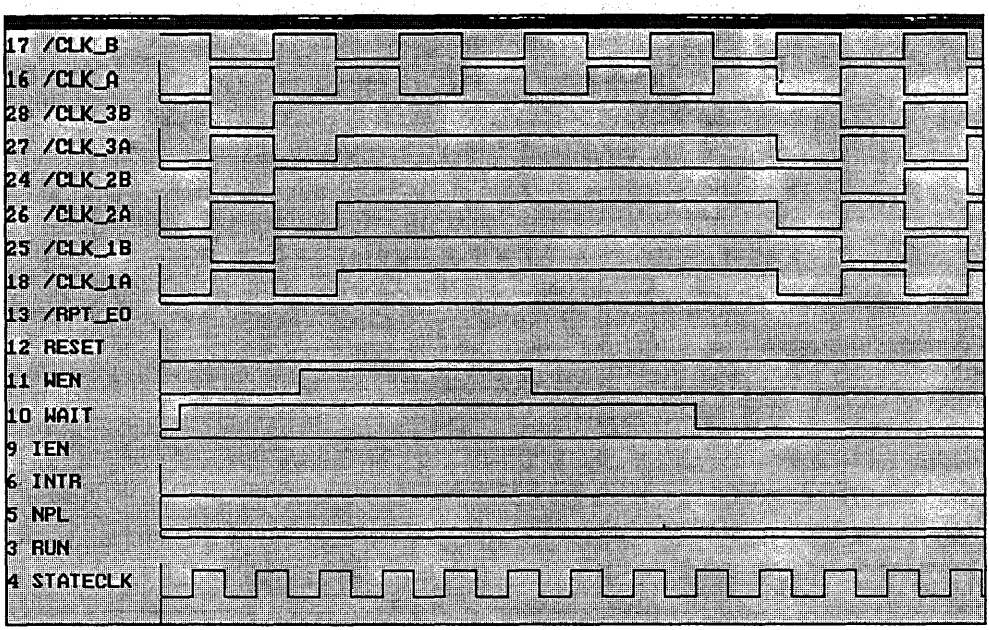


Appendix D. Cypress PLD ToolKit: CY7C361 Simulation (Cont.)





Appendix D. Cypress PLD ToolKit: CY7C361 Simulation (Cont.)



CYPRESS SEMICONDUCTOR

Understanding the CY7C330 Synchronous EPLD

This application note provides basic information on the CY7C330 and presents four design examples: a high-speed up/down counter with limits, a 16x16 crossbar switch, a pipelined buffer, and a simple toggle counter. Also included is an internal product term numbering chart. All example source code is in Cypress PLD ToolKit syntax.

The Cypress CY7C330 is the first in a family of high-speed, application-optimized CMOS EPLDs. This fully synchronous part is designed to implement state machines and other clocked systems. The CY7C330 offers new solutions for systems designers, with a truly usable high clock rate, 39 total registers, and 17,000 programmable bits providing up to 1200-gate complexity.

Other devices in the family are the CY7C331 and the CY7C332. All family members are packaged in 28-pin, 300-mil dual in-line and LCC/PLCC packages. The technology is low-power CMOS and UV erasable. The application-specific family from Cypress provides the CY7C330 for sequential state machine applications, the CY7C331 for general-purpose asynchronous designs, and the CY7C332 for decoders and combinational logic applications.

This family of high-speed devices provides the optimal solution for each system design using Cypress's 0.8-micron, dual-level-metal, CMOS technology. Systems using other types of programmable logic devices for synchronous state machine applications can use the CY7C330 as a higher-density, lower-power solution at speeds up to 66 MHz.

The Cypress PALC22V10, PLDC20G10 and PAL20 devices proved the popularity of high-speed, low-power, erasable CMOS logic. The CY7C330 builds on that base. One CY7C330 can easily replace four PALC22V10s because the CY7C330 extends the number of state registers to 16, extends the number of product terms per output to 19 maximum, adds an XOR logic function, and provides the ability to use pins as bidirectional I/O.

The CY7C330 increases the speed of synchronous systems to 66 MHz. This is the actual usable speed, as determined by the total 15-ns feedback time from the Q output of a flip-flop to the D input of any flip-flop in the device. To ensure the 66-MHz operation, all 23 inputs to the device have registers. This structure permits pipelined operations, which allow external data to be synchronized or CPU bus-oriented data to be latched. Input registers can be clocked from either of two input clock sources on either pin 2 or 3.

The CY7C330 offers 258 variable product terms for 16 state registers. This allows you to design very complex sequential machines with virtually no limitation of product terms. These designs can easily exceed the size you want to manage with Karnaugh mapping. However, the new generation of advanced EPLD compilers can manage very complex state machine designs on workstations such as the IBM PC/XT .

Overview of the CY7C330

An easy way to picture the CY7C330 is with the block diagrams in *Figure 1*. On the input side of the CY7C330 (pins 1 - 7 and 9 - 14) are 11 input registers and three clocks. Pin 1 is the state clock. Each of the 11 input registers is edge triggered, and each can use either device pin 2 (clock 1) or pin 3 (clock 2) (shown in *Figure 2*) as a clock. An architecture bit for each input register controls the selection of the input clock. This approach allows input data to be synchronized to a clock edge or loaded into the device from a CPU data bus, with the clocks being decoded I/O-write signals. The registers' setup and hold times are very short, allowing high system throughput. Note that the outputs of the registers feed the device's AND-OR-XOR array.

Pin 14 has an additional function that affects the input register: You can use the pin as a fast, asynchronous output enable to the device, allowing a CPU to move data in the state machine registers onto a bus, for example.

On the I/O side of the device (pins 15 - 20 and 23 - 28) are 12 macrocells. Each I/O macrocell (see *Figure 1* in "Using ABEL to program the CY7C330") contains a D-type register, an input register with clock controls, and output-enable resources. Architecture bits for feed-

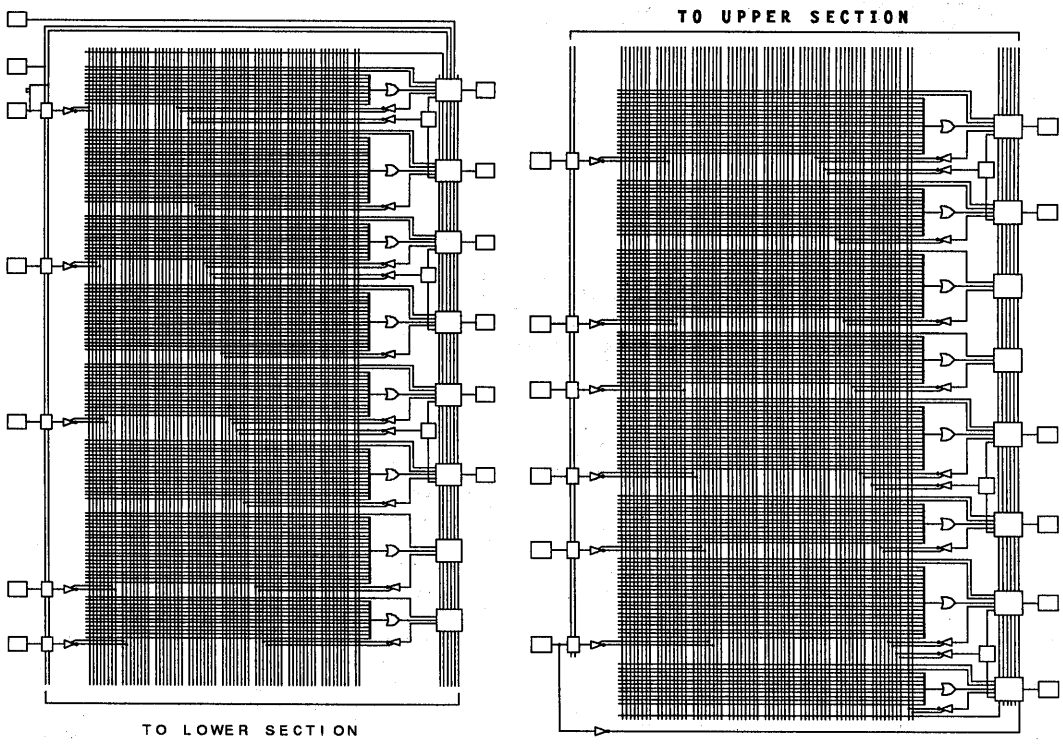


Figure 1. The CY7C330 Block Diagram

back selection, output-enable configuration, and input-register clock selection allow you to configure each macrocell independently.

Each adjacent I/O macrocell shares an input multiplexer (*Figure 3*). This allows either macrocell register to be hidden, while the I/O pin is used as an input. In addition, four hidden register macrocells (see *Figure 3* in "Using ABEL to Program the CY7C330") provide additional state registers without direct output connections.

The AND-OR-XOR array in *Figure1* has 66 inputs and 244 product terms driving 16 OR-XOR gates. The 16 OR gates have from nine to 19 inputs (variable product terms), which allow complex designs to fit into each stage. An XOR product term for each OR output permits equations to be solved either with D or T flip-flops in the output stage, or for active-High or active-Low equations. 12 product terms provide the output-enable function. A global reset and preset is also generated out of the array. Each product term forms an AND function with up to 66 inputs. The 66 inputs are the true and complement signals of 33 internal nodes in the CY7C330.

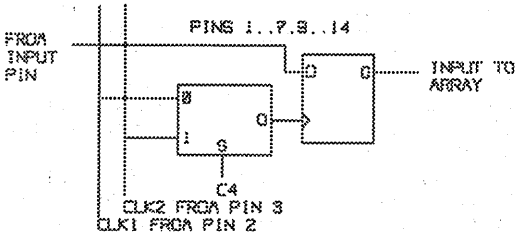


Figure 2. The CY7C330 Input Macrocell

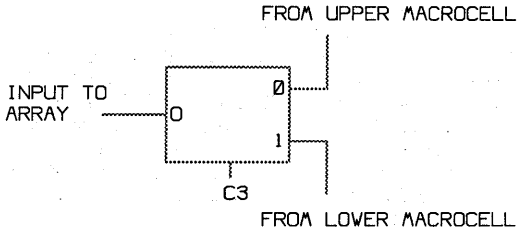


Figure 3. The CY7C330 Shared Input Multiplexer

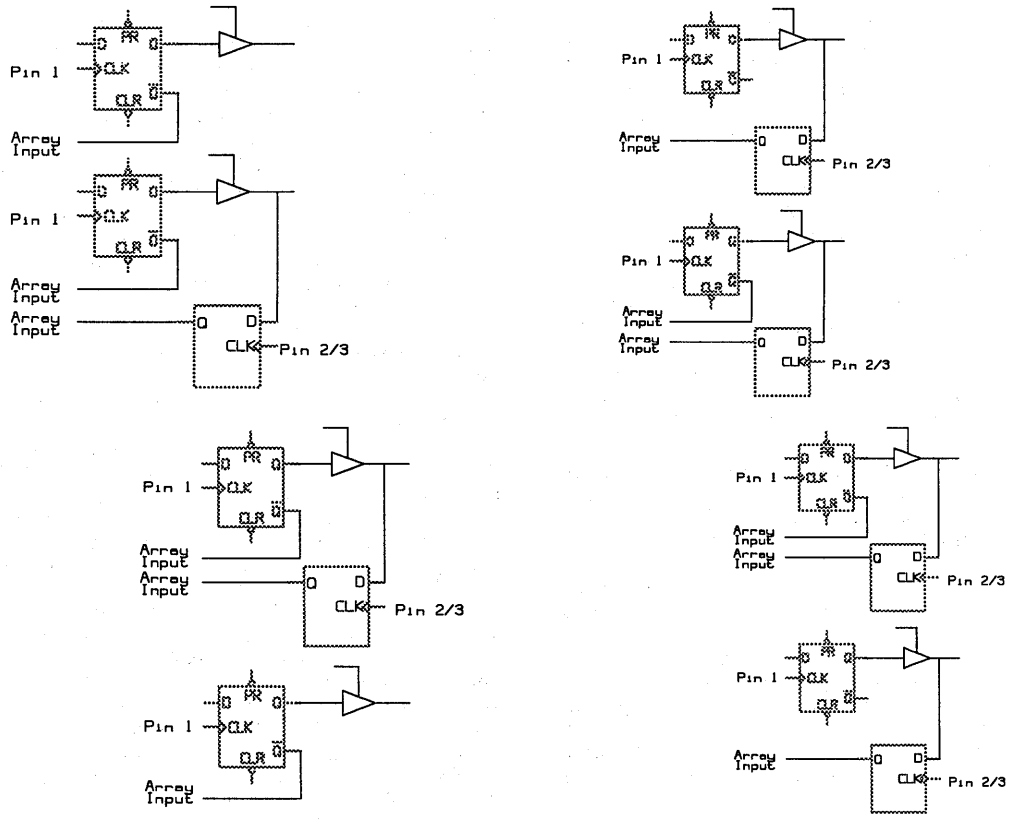


Figure 4. Four CY7C330 I/O Macrocell Configurations

Macrocell State Registers

The CY7C330's OR-XOR gates feed into 16 state registers. These registers are edge-triggered D flip-flops with pin 1 serving as clock. The outputs from these state registers feed back into the array, allowing you to construct high-speed state machines. The total feedback time period from Q to D and the array delay from input register to state register is 15 ns, allowing a full, usable clock rate of 66 MHz.

Four of the CY7C330's state registers are always hidden inside the device. A hidden register lets you build intermediate states or other functions without loading an I/O pin. Of the 12 remaining registers, up to six can be hidden. This gives a total of 10 maximum usable hidden registers, while allowing the 28-pin device to have 17 dedicated input pins, six I/O pins, and many other combinations. Valid I/O macrocell configurations appear in *Figure4*.

Each I/O macrocell (pins 15 - 20 and 22 - 28) also has an edge-triggered input register with either pin 2 or pin 3 serving as clock. The total register count is 39—16 state registers and 23 input registers. To keep the device speed as high as possible, the number of inputs to the array is limited to 33 (x2); six of the array inputs from the I/O macrocells are multiplexed (shared). Thus, three feedbacks are provided for the two output and two input registers for each set of two I/O pins. The easiest way to understand the net result is that the maximum number of hidden registers in the 12 I/O macrocells is six. Output registers that have no feedback to the array are useful for data outputs or single-clock-delayed Mealy outputs from the state machine.

The 12 macrocells have 24 registers total and 18 feedbacks. When you assign functions in your application to physical pins in the device, consider the number of feedbacks available and the number of product terms required.

Center Pinning

All Cypress CY7C330 family products use center pins for V_{CC} and V_{SS} connections. In addition, the V_{SS}

for the internal logic and the V_{SS} for the output drivers are on different pins. Center power pins eliminate noise generated by both TTL and CMOS devices. This noise is inductive noise proportional to the package lead inductance. Moving the power pins to the center lowers pin inductance and noise by a factor of 3 compared with corner-pin power connections.

Splitting ground lines—with the ground for input and logic on pin 8 and the ground for output drivers on pin 21—has additional noise benefits. Ground-bounce noise is caused when outputs switch from High to Low. The more pins switching at the same time, the more noise generated. Several hundred millivolts can be induced on the chip's internal ground from this effect. Although the level is low enough to meet output V_{ol} specs, the noise voltage must be considered when designing the input buffers on a chip because the noise influences the V_{il} spec of 0.8V. 400 mV of ground-bounce noise shifts the AC effective V_{il} to 1.2V.

By separating the input reference ground from the output ground where the noise is generated, ground noise compensation is lowered or eliminated. This permits Cypress offer a faster input buffer. Externally, the two grounds are connected together. Also, by placing the V_{CC} pin close to the GND pin, external 0.1 µF capacitors (as usual, one per chip) can be very close to the actual device power pins.

All Cypress EPLDs permit the registers to be preloaded into any configuration. This capability can vastly reduce the test time and allows all patterns programmed into an EPLD to be completely tested. Without preload, for example, testing a multibit counter that has no reset product term could be very slow or impossible.

CY7C33X Family Technology

The CY7C330 and most other new Cypress products are built in the Cypress 0.8 micron, N-well CMOS, high-speed technology. New Cypress EPLDs use a dual-metal-layer connection method to further increase speed. This technology allows Cypress to build static RAMs with 7-ns access times, 35-MHz FIFOs, a 33-MHz RISC processor, and many other high-performance products.

Cypress uses an EPROM technology (as distinct from fuse-link, or EEPROM technology) for all its EPLD s and (E)PROMS because of the tremendous increase in manufacturing yields and 100-percent testability offered by EPROM technology. This UV-erasable EPROM technology provides proven data retention, testability, and manufacturability.

In addition, the Cypress 2T (2 transistor) cell design allows very high speed circuits to be built. Cypress uses this 2T cell design for performance. One transistor is used only for programming and the other for reading, with each optimized for only one function. The program transistor can be larger and slower. It is designed to withstand 15V source to drain, which is the maximum program charge on the floating gate. The read transistor can be very small and fast. Because the read bit line is only switching between 0 and 5V, the sense amp is smaller and faster, and no high-current 15V driver MOSFETs are present. The result is very fast (sub 10 ns) array times.

All Cypress devices offer protection against static discharge (ESD). This means the devices are no more sensitive than bipolar devices. By using a unique -3V substrate bias generator (V_{bb}), Cypress devices are protected from latchup caused by transient voltages below ground, which are commonly seen in TTL systems. This internally generated V_{bb} also allows the device to maintain high speed over a wide temperature range by controlling switching thresholds. No current flows in an input even under extreme undershoot situations, and the input transistor requires no recovery time after an undershoot.

In addition to substrate bias for latchup elimination, Cypress uses a stacked TTL output driver. This feature removes the pin-to-P-channel-transistor connection, a major source of latchup. Reducing the energy in High-to-Low transitions also improves overshoot and noise generation. Virtually all high-performance systems using TTL or CMOS adhere to the TTL standard voltage specification—2.0V for a TTL High and 0.8V for a TTL Low. Thus, a P-channel output transistor that pulls the output to V_{cc} causes more problems than it solves because it overdrives the output. The lower voltage output from a stacked N-channel output drive of 3.5V vs. 5.0V causes less noise on the High-to-Low transition because less energy needs to be switched.

Cypress uses stacked N-channel transistors on the outputs of all devices, eliminating latchup and fast transition to an overly high output 1 level. The devices are more compatible with the TTL devices Cypress replaces.

Resource Planning

Planning the assignment of functions to pins in the CY7C330 is an important step in a CY7C330 design. The resource planning sheet presented in *Table 1* should be helpful for this procedure. Examples of its use are included with each application presented here.

The decision on which pin to use is based on:

1. Asynchronous output enable, set to pin 14 or synchronous enable with a product term
2. State clock is pin 1
3. Input clock is pin 2
4. Second input clock is pin 3, or use pin 3 as a normal input if pin 2 will be the only input clock
5. Input only on pins 4 - 7 and 9 - 13
6. Device outputs: Assign pins keeping in mind that they have different product term widths. The widths are: 9, 11, 13, 15, 17, 19 for pins 28/15, 26/17, 24/19, 23/20, 25/18, 27/16, respectively

7. Use of hidden registers:
   a. Four registers — H1 to H4 — are always hidden
   b. Up to six additional hidden registers can be defined; Cypress suggests this sequence: 25, 18, 27, 16, 23, 20
   c. Assign input names to these six registers that are defined. Cypress suggests this sequence: 25, 18, 27, 16, 23, 20
   d. Assign input names to these six registers that are different from the physical device pin names
   e. The optionally hidden registers can be viewed if their output enable is made active and the external logic driving the pin is in a high-impedance state; otherwise the OE (output enable) product term of the hidden register must be set to Zero (NAME.ENA = 0)

Table 1. A CY7C330 Resource Planning Sheet

CY7C330 Resources Planning Sheet
Project : Your project name

Pin	Input Register Function	Input Register Clock	Register Function	Output Enable	# of PTerms
1	State Clk				
2	Clk 1				
3	Input/Clk 2	1 if Input			
4	Input	1/2			
5	Input	1/2			
6	Input	1/2			
7	Input	1/2			
8	VSS				
9	Input	1/2			
10	Input	1/2			
11	Input	1/2			
12	Input	1/2			
13	Input	1/2			
14	Input/OE	1/2 if Input			
15	Input	1/2 if Input	Output	Pin 14/Pterm	9
16	Input	1/2 if input	Output	Pin 14/Pterm	19
17	Input	1/2 if input	Output	Pin 14/Pterm	11
18	Input	1/2 if input	Output	Pin 14/Pterm	17
19	Input	1/2 if input	Output	Pin 14/Pterm	13
20	Input	1/2 if Input	Output	Pin 14/Pterm	15
21	VSS				
22	VCC				
23	Input	1/2 if input	Output	Pin 14/Pterm	15
24	Input	1/2 if input	Output	Pin 14/Pterm	13
25	Input	1/2 if input	Output	Pin 14/Pterm	17
26	Input	1/2 if input	Output	Pin 14/Pterm	11
27	Input	1/2 if input	Output	Pin 14/Pterm	19
28	Input	1/2 if input	Output	Pin 14/Pterm	9
H1	None	-	-	None	19
H2	None	-	-	None	11
H3	None	-	-	None	17
H4	None	-	-	None	13
Notes :	Input Register Clock	#1 is pin 2 #2 is pin 3			

See the Application Note for the meaning of the pin names.
Output Enable = 14 means the asynchronous pin 14 direct enable.
Z means the pin is never active

8. The remaining visible registers can still be used in applications where both inputs of a macrocell pair are used. However, one of the output registers of each adjacent pair cannot have a feedback; it is used only as an output synchronized by the state clock on pin 1. If, after this assignment, the compiler or assembler complains that not enough product terms are available, some pins might have to be re-assigned

Software Design Tools

You can compile logic for the CY7C330 with a number of packages available from independent software vendors. These packages include ABEL V3.0 from DATA I/O® and LOG/iC V3.0 from ISDATA®. Cypress has developed the PLD ToolKit (CY7C3101), which you can use to design any PLD that Cypress makes. All these packages are logic compilers capable of converting state machine or binary logic descriptions into a JEDEC file that can program the device.

The JEDEC file is the standard interface from a software development tool to a logic programmer. See the examples section for more detail on the software tools.

Logic Programmers

The CY7C330 can be programmed today on the QuickPro plug-in board for IBM and compatible personal computers. Soon you will also be able to use the DATA I/O®, STAG®, and other programmers.

Some software tools require you to set fuses or bits in the device to enable certain functions, whereas others set the architecture bits automatically. Note that bit 17070 requires special attention: it must be set to 1 if any input register uses a clock from pin 3. This requirement will disappear in future releases of the software packages, and the bits will be set automatically.

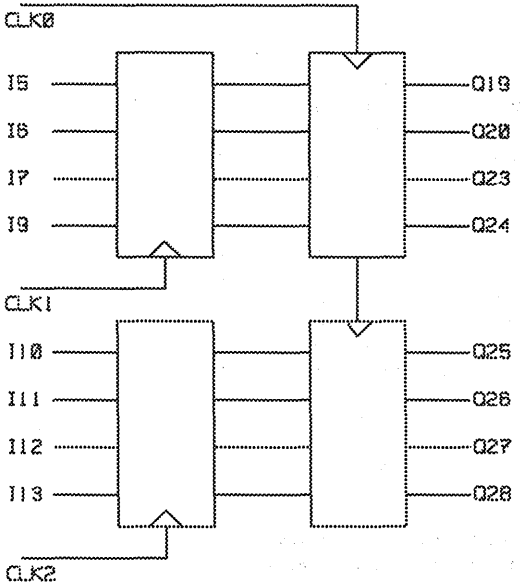


Figure 5. Pipelined Buffer Block Diagram

Pipelined Buffer

The Pipe330 example is a two-stage pipeline that shifts parallel data from the inputs to the outputs (*Figure 5*). This example demonstrates the overall Cypress PLD ToolKit source syntax and shows how macrocells are configured.

In the Pipe330 example, the output enable for specific macrocells is under control of either pin 14 or the associated product term. The latter case is the default. To control the output enable of a macrocell with pin 14, add NENBPT to the list of attributes following the node assignment in the configuration section.

If NENBPT does not appear in the attribute list for a node, the expression that follows the construct <OE> in the equations controls the output enable. If <OE> is not part of the equation, the output is permanently disabled. If <OE> is present, but no expression follows it, the output is permanently enabled.

The pin 1 signal always clocks the output registers in the CY7C330. Either the pin 2 or 3 signals can clock the input registers. Because pin 2 is the default clock, no special attributes are required for this configuration. If you wish to clock an input register with pin 3, the attribute list for that node must contain ICLK= 3.

The resource planning sheet for the pipelined buffer appears in *Table 2*, and the source code appears in *Appendix A*.

Test patterns for the Pipe330 example are relatively simple, but keep in mind a few guidelines. At first, for example, the state of the registers in the device is unknown, and all registers are put in a known state before any outputs are checked (non-X). Another aspect of CY7C330 simulation is the need to consider multiple clocks. The input and output clocks should be treated separately, because the simultaneity of clock assertion is not guaranteed in programmers—or in any real system, for that matter.

Up/Down Toggle Counter with Preloads

The Tog330 example shows how you can use the CY7C330's XOR product terms to emulate a T-type flip-flop. The statement:

Q = < XSUM> Q
< SUM> T;

programs the XOR product term with the feedback of the register output, making the register into a T type. The T-type register configuration is active Low because, by architecture, all the outputs are active Low. You can emulate a JK-type flip-flop by using the configuration above with the following relation:

T = J!Q + KQ

Table 3 presents the resource planning sheet for the toggle counter example, and the source code appears in *Appendix B. Figure 6* shows the block diagram for the design.

Up/Down Counter with Limits

The up/down counter example shows how you can assign the pins for maximum use in the CY7C330. This counter operates at 66 MHz, counting up until reaching the value stored in the 8-bit upper-limit register, then down until reaching the lower limit. Also included is a device reset and a method to preload the counter to either the upper or lower limit.

Consider an application in which the two 8-bit limit registers are loaded from a CPU. The lower limit is on pins 4 to 12, with a 9th bit for preload on pin 13. The clock for this lower limit is on pin 2. The upper limit is loaded via pins 15 - 27, with pin 27 providing 9th preload bit. These pins are also used for reading out the counter value, and pin 14 is the output enable for the up/down counter.

Table 2. Resource Planning Sheet for Pipelined Buffer

CY7C330 Resources Planning Sheet

Project : Pipelined Buffer

Pin	Input Register Function	Input Register Clock	Register Function	Output Enable	# of PTerms
1	State Clk				
2	Clk 1 (LHS)				
3	Clk 2 (RHS)				
4	I4	1			
5	I5	1			
6	I6	1			
7	I7	1			
8	VSS				
9	I9	1			
10	I10	2			
11	I11	2			
12	I12	2			
13	I13	2			
14	OE	-			
15	-	-	-	Z	9
16	-	-	-	Z	19
17	-	-	-	Z	11
18	-	-	-	Z	17
19	-	-	Q19	Pterm (Eqn)	13
20	-	-	Q20	Pterm (Eqn)	15
21	VSS				
22	VCC				
23	-	-	Q23	Pterm (Eqn)	15
24	-	-	Q24	Pterm (Eqn)	13
25	-	-	Q25	Pin 14	17
26	-	-	Q26	Pin 14	11
27	-	-	Q27	Pin 14	19
28	-	-	Q28	Pin 14	9
H1	None	-	-	None	19
H2	None	-	-	None	11
H3	None	-	-	None	17
H4	None	-	-	None	13

Notes : Input Register Clock

1 is pin 2
#2 is pin 3
See the Application Note for the meaning of the pin names.
Output Enable = 14 means the asynchronous pin 14 direct enable.
Z means the pin is never active

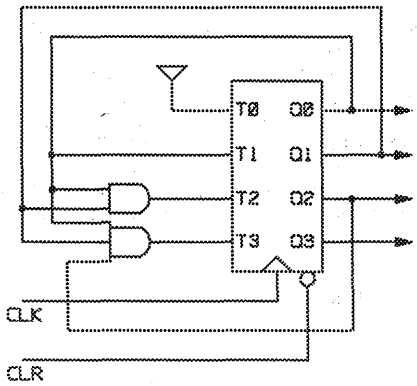


Figure 6. Toggle Counter Block Diagram

Four buried registers detect equality of the counter with the limits to maintain up/down direction and to detect the preload request as an edge-triggered signal. By using the XOR product terms, the counter needs only nine total products even on the most significant bit. Without XOR, the 8th bit would need 18 product terms because of the two preload sources. Due to the large number of product terms per output in the CY7C330, this counter can operate at 66 MHz.

The counter's contents can be read out when pin 14 (direct output enable) is Low. In a bus-oriented system, a microprocessor can read the register if a decoded I/O read signal is applied to pin 14. Note that the other method of output enable, via the array, requires a clock edge to load the enable input condition into the input registers. When pin 14 is High, the upper-limit register can be loaded—from a processor bus, for example. The lower-limit register can be loaded at any time.

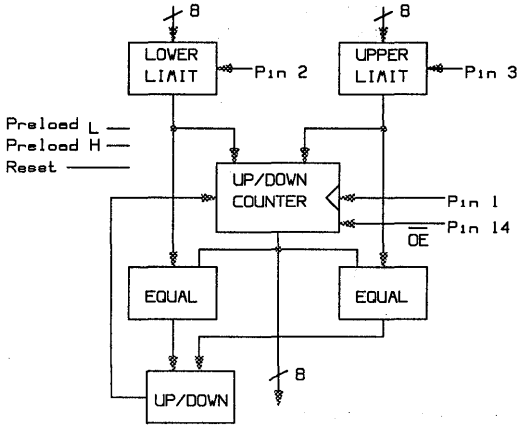


Figure 7. UP/DOWN Counter Block Diagram

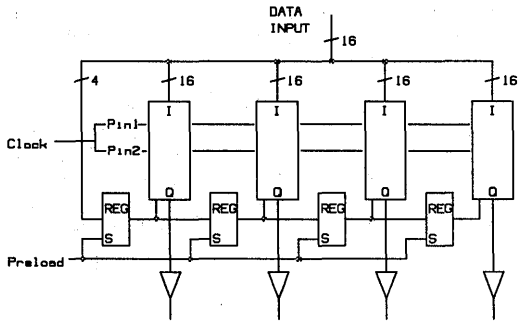


Figure 8. 16X16 Crossbar Switch Block Diagram

Figure 7 shows the block diagram for this design. The resource planning sheet appears in *Table 4*, and the code is in *Appendix C*.

In operation, the up/down counter counts between the limits stored in two registers. Lower-limit (LL) data is loaded on the positive edge of the pin 2 clock. There are 8 data bits plus 2 control bits, LPL and Reset. If LPL is Low, only the limit compare register is changed. If LPL is High, the LL data is loaded into the counter on the next clock edge, and the counter counts up. The LL data is one count higher than the actual lower limit. If RESET is active, all internal registers are reset to 0, so long as the reset bit is set in the LL register.

Upper-limit (UL) data is loaded on the positive edge of the pin 3 clock. This part of the counter uses 8 data bits plus a preload control bit, UPL. If UPL is Low, only the limit-compare register is changed. If UPL is High, the UL data is loaded into the counter on the next clock edge, and the counter counts down. UL data is multiplexed with the counter output data. The UL data is one count lower than the actual upper limit. Pin 16 is the RESET input. Pin 14 is the active-Low output enable for the counter; the counter can be read at any time. Pin 1 is the clock for the counter. Pins 18 and 20 are connected together for data bit 6. Pins 23 and 25 are connected together for data bit 7.

The buried (hidden) registers are used as follows: H1 is loaded with the result of the comparison between the counter and UL. H2 is UPL or LPL, delayed by one clock edge; H2 serves as an edge detect. H3 is loaded with the result of the comparison between the counter and LL. H4, when High, forces the counter to count up.

16 x 16 Crossbar Switch

A data switch capable of multiplexing 16 inputs into four outputs can be built with one CY7C330. The 66-MHz clock rate allows even asynchronous input signals of up to 33 MHz to be switched through the device. The compact 300-mil package saves PCB space, in contrast to the space such a multiplexer would otherwise

need. At least 40 pins would normally be required, partitioned as follows:

16 input pins,
4 output pins,
4 x 4 = 16 selection inputs
4 pins for power and clock connections

No other PLD today can perform this function using a single device, due to the logic requirement (the number of product terms required per output) as well as the timing requirement.

The crossbar switch uses 12 state registers plus four input registers to act as the 4 x 4-bit selection registers. Each output channel needs a 4-bit register to select one of 16 input channels. A 4-stage, 4-bit-wide shift register implemented in the device holds the select status. This allows the 4 x 4 selection bits to be loaded via only four pins, without needing any address pins.

When the PL (PRELOAD) signal on pin 3 is Low, input data bits 0 to 3 become the selector data lines; five clock pulses shift the select data through the device

Table 3. Resource Planning Sheet for Toggle Counter

CY7C330 Resources Planning Sheet
Project : 4 Bit Toggle Counter

Pin	Input Register Function	Input Register Clock	Register Function	Output Enable	# of PTerms
1	State Clk				
2	Clk 1				
3	Clear	1			
4	-				
5	-				
6	-				
7	-				
8	VSS				
9	-				
10	-				
11	-				
12	-				
13	-				
14	-				
15	-	-	!Q0	Pterm	9
16	-	-	!Q1	Pterm	19
17	-	-	!Q2	Pterm	11
18	-	-	!Q3	Pterm	17
19	-	-		Z	13
20	-	-		Z	15
21	VSS				
22	VCC				
23	-	-		Z	15
24	-	-		Z	13
25	-	-		Z	17
26	-	-		Z	11
27	-	-		Z	19
28	-	-		Z	9
H1	None	-	-	None	19
H2	None	-	-	None	11
H3	None	-	-	None	17
H4	None	-	-	None	13

Notes : Input Register Clock

1 is pin 2
#2 is pin 3
See the Application Note for the meaning of the pin names.
Output Enable = 14 means the asynchronous pin 14 direct enable.
Z means the pin is never active

into selectors 1, 2, and 3, as well as the output pins. Setting pin 3 High after the fifth pulse loads the signals on the output data pins into select register 0. This last load operation utilizes the function of pin 3 as a data pin as well as a clock. Setting the signal on pin 3 Low switches the internal logic from a selector into a shift register; the clock edge created by applying a High to pin 3 loads the data outputs into the input registers associated with output pins 16, 18, 25, and 27.

This design buries the output registers of several I/O macrocells and uses the pin as an input by utilizing a shared-input mux. The source file's configuration section specifies this arrangement by first assigning the name of the output register to the macrocell node number. Because the default configuration is for the output register's Q output to feed back into the array, no other configuration attributes are needed here. Next, the input's name is assigned to the node number of the shared input mux adjacent to the pin. The default for the shared input muxes is to pass the data on the even-numbered pin into the array. If the input should come from an odd-numbered pin, you must add the attribute

Table 4. Resource Planning Sheet for Up/Down Counter

CY7C330 Resources Planning Sheet

Project : Up/Down Counter with Limits

Pin	Input Register Function	Input Register Clock	Register Function	Output Enable	# of PTerms
1	State Clk				
2	Clk 1				
3	Clk 2				
4	LL01				
5	LL11				
6	LL21				
7	LL31				
8	VSS				
9	LL41				
10	LL51				
11	LL61				
12	LL71				
13	PRELOAD LOW	1			
14	COUNTER OE-				
15	UL12	CNT1	Pin 14	9	
16	Reset	1-Z19			
17	UL32	CNT3	Pin 14	11	
18	UL62-Z17				
19	UL42	CNT4	Pin 14	13	
20	--CNT6		Pin 14	15	
21	VSS				
22	VCC				
23	--CNT7		Pin 14	15	
24	UL52	CNT5	Pin 14	13	
25	UL72-Z17				
26	UL22	CNT2	Pin 14	11	
27	PRELOAD HIGH	2-Z19			
28	UL02	CNT0	Pin 14	9	
H1	None-Up EqualsNone		19		
H2	None-L/H Prel'DoneNone		11		
H3	None-Down EqualsNone		17		
H4	None-Up CountNone		13		

Notes :Input Register Clock #1 is pin 2
#2 is pin 3
See the Application Note for the meaning of the pin names.

SRC=N (where N is the pin number) to the list of attributes in parentheses following the node name. For an example of this syntax, refer to d10 and sa2 in the source file.

The space advantage of the CY7C330 in this crossbar switch application becomes especially important as the size of the matrix increases. A 32 x 32 matrix requires only 16 devices vs. 64 PALC22V10s or 96 TTL parts. You can easily load the internal data selection registers with a Cypress 24-pin EPLD, the PLDC20G10, and a FIFO. A CPU can load the 16 x 4-bit selector information into the FIFO, and the PLDC20G10 can move the data from the FIFO into the device. One PLDC20G10 and one 16 x 4 (or larger) FIFO is required. The Cypress CY7C403 is an ideal FIFO for this application

Table 5 shows the resource planning sheet for the 16 X 16 crossbar switch, and a block diagram of the design appears in *Figure 8*. The source code can be found in *Appendix D*.

Table 5. Resource Planning Sheet for Crossbar Switch

CY7C330 Resources Planning Sheet

Project :16 X 16 Crossbar Switch

Pin	Input Register Function	Input Register Clock	Register Function	Output Enable	# of PTerms
1	State Clk				
2	Clk 1				
3	Sel PRELOAD	1			
4	Data 0	1			
5	Data 1	1			
6	Data 2	1			
7	Data 3	1			
8	VSS				
9	Data 4	1			
10	Data 5	1			
11	Data 6	1			
12	Data 7	1			
13	Data 8	1			
14	Data 9	1			
15	Data 10	1	Select A2	Z	9
16	Select D0	2	Output 3	Pterm	19
17	Data 11	1	Select A1	Z	11
18	Select C0	2	Output 2	Pterm	17
19	Data 12	1	Select C1	Z	13
20	-	1	Select D1	Z	15
21	VSS				
22	VCC				
23	-	1	Select B2	Z	15
24	Data 13	1	Select A2	Z	13
25	Select B0	2	Output 1	Pterm	17
26	Data 14	1	Select C2	Z	11
27	Select A0	2	Output 0	Pterm	19
28	Data 15	1	Select D2	Pterm	9
H1	None	-	Select A3	None	19
H2	None	-	Select B3	None	11
H3	None	-	Select C3	None	17
H4	None	-	Select D3	None	13

Notes : Input Register Clock

1 is pin 2
#2 is pin 3
See the Application Note for the meaning of the pin names.
Output Enable = 14 means the asynchronous pin 14 direct enable.
Z means the pin is never active

Reading the CY7C330 JEDEC Map

Table 6 should help you read the JEDEC map of a CY7C330. The pin or node reference number is on the left. These numbers correspond to the pin and node numbers on the block diagram in *Figure1*.

The column labeled Input True gives the sequential number (left to right) of the column corresponding to the non-inverted input to the array. If the number is even, then the false input is the next-higher integer; if the number is odd, then the false input is the next lower integer.

The table lists the number of product terms in each output stage, along with the JEDEC offset (sequential fuse position) for each.

Table 6. The CY7C330 Internal Array Reference List

Pin or Node	Function	Input True	# of Pterms	OE	XOR	1st OR
1	State Clock					
2	Input Clock1					
3	Input Clock2	0				
4	Input Register	2				
5	Input Register	4				
6	Input Register	6				
7	Input Register	8				
8	VSS					
9	Input Register	10				
10	Input Register	12				
11	Input Register	14				
12	Input Register	16				
13	Input Register	18				
14	Input Register	20				
15	I/O Regs, mux	65	9	L16236	L16302	L16368
N-35	mux input(node)	62				
16	I/O Regs, mux	61	19	L14850	L14916	L14982
17	I/O Regs, mux	59	11	L13992	L14058	L14124
N-36	mux input(node)	56				
18	I/O Regs, mux	55	17	L12738	L12804	L12870
19	I/O Regs, mux	49	13	L9636	L9702	L9768
N-37	mux input(node)	46				
20	I/O Regs, mux	45	15	L8514	L8580	L8646
21	VSS					
22	VCC					
23	I/O Regs, mux	39	15	L5280	L5346	L5412
N-38	mux input(node)	36				
24	I/O Regs, mux	35	13	L4290	L4356	L4422
25	I/O Regs, mux	33	17	L3036	L3102	L3168
N-39	mux input(node)	30				
26	I/O Regs, mux	29	11	L2178	L2244	L2310
27	I/O Regs, mux	27	19	L792	L858	L914
N-40	mux input(node)	24				
28	I/O Regs, mux	23	9	L66	L132	L198
N-29	Sync. Reset			L0		
N-30	Sync. Preset			L16962		
N-31	Buried Register	40	13		L11814	L11870
N-32	Buried Register	42	17		L10626	L10692
N-33	Buried Register	50	11		L7722	L7788
N-34	Buried Register	52	19		L6402	L6468

Appendix A. PLD ToolKit Source Code for Pipelined Buffer

```
CY7C330;                                    {Pipe330}

CONFIGURE;

CkS (node=1),                               {Output register clock}
Ck1,                                        {Input register clock 1}
Ck2,                                        {Input register clock 2}
I0 (iclk=3),                                {Input 0, clocked by Ck2 (pin 3)}
I1 (iclk=3),                                {Input 1, clocked by Ck2 (pin 3)}
I2 (iclk=3),                                {Input 2, clocked by Ck2 (pin 3)}
I3 (iclk=3),                                {Input 3, clocked by Ck2 (pin 3)}
I4 (node=9),                                {Input 4, clocked by Ck1 (pin 2)}
I5,                                         {Input 5, clocked by Ck1 (pin 2)}
I6,                                         {Input 6, clocked by Ck1 (pin 2)}
I7,                                         {Input 7, clocked by Ck1 (pin 2)}
OE1,                                        {output enable for Q<7:4>}
!OE2(node=14),                              {direct output enable for Q<7:0>}
Q7,                                         {Output 7, clocked by CkS, enabled by OE1&!OE2}
Q6,                                         {Output 6, clocked by CkS, enabled by OE1&!OE2}
Q5,                                         {Output 5, clocked by CkS, enabled by OE1&!OE2}
Q4,                                         {Output 4, clocked by CkS, enabled by OE1&!OE2}
Q3(nenbpt),                                 {Output3, clocked by CkS, enabled: pin14}
Q2(nenbpt),                                 {Output2, clocked by CkS, enabled: pin14}
Q1(node=23,nenbpt),                         {Output1, clk: CkS, OE: pin14}
Q0(nenbpt),                                 {Output0, clocked by CkS, enabled: pin14}
!RST(iop),                                  {low asserted reset, I/O macrocell as input}
reset(node=29),                             {internal reset node}

EQUATIONS;

reset = RST;

!Q0 = <sum>  !I0;

!Q1 = <sum>  !I1;

!Q2 = <sum>  !I2;

!Q3 = <sum>  !I3;

!Q4 = <oe> OE1 & OE2
     < sum> !I4;

!Q5 = <oe> OE1 & OE2
     < sum> !I5;

!Q6 = <oe> OE1 & OE2
     < sum> !I6;

!Q7 = <oe> OE1 & OE2
     < sum> !I7;
                                            {end of file}
```

Appendix B. PLD ToolKit Source Code for a Toggle Counter

```
CY7C330;                                {Tog330}

CONFIGURE;

CkS,                                    {Count clock, This is pin1 since it is first in the list.}
Ck1,                                    {Input clock, This is pin2 since it is next.}
!Clr,                                   {Low true clear, Pin3 is next in sequential order.}
!OE(node = 14),                         {Low asserted output enable pin, pin 14}
!Q0(nenbpt),                            {Q0-Q3 are the counter outputs - pins 15-18.}
!Q1(nenbpt),
!Q2(nenbpt),
!Q3(nenbpt),
reset(node=29),                         {The reset product term is node 29.}

EQUATIONS;

reset = Clr;

Q0 = <xsum> Q0                          {Feeding the register output back into the XOR emulates a T flop.}
     < sum> ;                           {T input - No expression after the connective < sum> means always asserted}

Q1 = <xsum> Q1                          {Feeding the register output back into the XOR emulates a T flop.}
    < sum> Q0;                          {T input}

Q2 = <xsum> Q2                          {Feeding the register output back into the XOR emulates a T flop.}
     < sum> Q1 & Q0;                        {T input}

Q3 = <xsum> Q3                          {Feeding the register output back into the XOR emulates a T flop.}
     < sum>  Q2 & Q1 & Q0;                  {T input}

                                        {end of file}
```

Appendix C. PLD ToolKit Source Code for Up/Down Counter

```
CY7C330;                                {File: COUNTER.CYP Date: 11/9/1988 }
CONFIGURE;
CLK(node=1),  LLC(node=2),  ULC(node=3),    {Count clock, Lower Limit Clock, Upper Limit Clock}
LL0(node= 4, iclk= 2), LL1,  LL2, LL3,  {The Lower Limit register is clocked by pin 2-LLC- by default.}
LL4(node= 9), LL5,  LL6,  LL7,          {The register is located at pins 4-7, 9-12 - pin 8 is Vss.}
LPL(node=13),                           {Lower limit PreLoad}
/CNTOE  (node=14),                      {Counter output enable on pin 14}
CNT0 (node= 28, nenbpt, oclk= 1,iclk= 3),  {The counter itself is in the output register of various I/O macrocells}
CNT1  (node=15,  ,nenbpt,  iclk=3),     {as noted in the node numbers after the names. Pin 1 always clocks the}
CNT2  (node=26,  nenbpt,  iclk=3),      {output registers-oclk = 1 was included once for documentation.}
CNT3  (node=17,  nenbpt,  iclk=3),      {'nenbpt' specifies that the output enable is controlled by pin 14}
CNT4  (node=19,  nenbpt,  iclk=3),      {rather than the output enable product terms in each macrocell}
CNT5  (node= 24, nenbpt, iclk= 3),      {Most of these macrocells will be bidirectional, with the Upper Limit}
CNT6  (node=20,  nenbpt),               {register residing in the input registers. 'iclk =  3' specifies that pin 3}
CNT7  (node=23,  nenbpt),               {clocks the input registers. This overrides the default, pin2.}
                                        {The output register is fed back into array by default.}
UL0  (node=40,  src=28),                {UL0 is the input reg of pin28, routed thru shared input mux-node40}
UL1  (node=35,  src=15),                {UL1 is the input reg of pin15, routed thru shared input mux-node35}
UL2  (node=39,  src=26),                {UL2 is the input reg of pin26, routed thru shared input mux-node39}
UL3  (node=36,  src=17),                {UL3 is the input reg of pin17, routed thru shared input mux-node36}
UL4  (node=37,  src=19),                {UL4 is the input reg of pin19 routed thru shared input mux-node37}
UL5  (node=38,  src=24),                {UL5 is the input reg of pin24 routed thru shared input mux-node38}
UL6  (node=18,  iop,iclk=3),            {UL6 is the input reg of pin18, 'iop' selects array input from input reg}
UL7  (node=25,  iop,  iclk=3),          {UL7 is the input reg of pin25, 'iop' selects array input from input reg}
UPL  (node=27,  iop,  iclk=3),          {Upper limit PreLoad, array input from input reg, clocked by pin 3}
/reset  (node=16,  iop),                {Low asserted clear, array input from input reg, clocked by pin 2}
node29  (node=29),                      {The reset product term is node 29}
UP  (node=31),                          {buried node 31 selects the counter direction, clocked by pin 1}
LEQUAL  (node=32),                      {buried node 32 compares counter with lower limit, clocked by pin 1}
PLDONE  (node=33),                      {buried node 33 is the preload done flag, clocked by pin 1}
UEQUAL  (node=34),                      {buried node 34 compares counter with upper limit, clocked by pin 1}

EQUATIONS;

/CNT0= < XSUM>  /CNT0
       < SUM>  /LPL  & /UPL
       < SUM>  /PLDONE
       < SUM>  /LL0  & LPL  & CNT0
       < SUM>  /CNT0 & UL0 & UPL
       < SUM>  LL0  & LPL  & /CNT0
       <SUM>  CNT0  &  /UL0  &  UPL;

/CNT1=  < XSUM>  /CNT1
       < SUM>  /LPL & CNT0 & /UPL & /UP
       < SUM>  /LPL & /CNT0 & /UPL & UP
       < SUM>  /LL1 & LPL & PLDONE & CNT1
       < SUM>  LL1 & LPL & PLDONE & /CNT1
       < SUM>  UPL & PLDONE & /UL1 & CNT1
       < SUM>  UPL & PLDONE & UL1 & /CNT1
       < SUM>  CNT0 & /PLDONE & /UP
       <SUM>  /CNT0  &  /PLDONE  &  UP;
```

Appendix C. Source Code for Up/Down Counter (continued)

```
/CNT2= < XSUM>  /CNT2
       < SUM>  /LPL  & CNT0 & /UPL & /UP & CNT1
       < SUM>  /LPL  & /CNT0 & /UPL & UP & /CNT1
       < SUM>  /LL2  & LPL  & CNT2 & PLDONE
       < SUM>  LL2  & LPL  & /CNT2 & PLDONE
       < SUM>  UPL & CNT2 & /UL2 & PLDONE
       < SUM>  UPL & /CNT2 & UL2 & PLDONE
       < SUM>  CNT0 & /PLDONE & /UP & CNT1
       <SUM> /CNT0  &  /PLDONE  &  UP  &  /CNT1;


/CNT3= < XSUM>  /CNT3
       <SUM>/LPL&CNT0&/UPL&CNT2&/UP&CNT1
       <SUM>/LPL&/CNT0&/UPL&/CNT2&UP&/CNT1
       < SUM>  /LL3 & LPL & PLDONE & CNT3
       < SUM>  LL3 & LPL & PLDONE & /CNT3
       < SUM>  UPL & PLDONE & /UL3 & CNT3
       < SUM>  UPL & PLDONE & UL3 & /CNT3
       <SUM>CNT0&CNT2&/PLDONE&/UP&CNT1
       <SUM>/CNT0&/CNT2&/PLDONE&UP&/CNT1;


/CNT4=  <XSUM>  /CNT4
       < SUM>  /LL4 & LPL & PLDONE & CNT4
       < SUM>  LL4 & LPL & PLDONE & /CNT4
       < SUM>  UPL & PLDONE & /UL4 & CNT4
       < SUM>  UPL & PLDONE & UL4 & /CNT4
       <SUM> /LPL  & CNT0  &  /UPL  &  CNT2  &  /UP  &  CNT3  &  CNT1
       <SUM> /LPL  &  /CNT0  &  /UPL  &  /CNT2  &  UP  &  /CNT3  &  /CNT
       < SUM>  CNT0 & CNT2 & /PLDONE & /UP  & CNT3 & CNT1
       <SUM> /CNT0  &  /CNT2  &  /PLDONE  &  UP  &  /CNT3  &  /CNT1;


/CNT5=  <XSUM>  /CNT5
       < SUM>  /LL5  & LPL  & CNT5 & PLDONE
       < SUM>  LL5  & LPL  & /CNT5 & PLDONE
       <SUM> UPL  &  CNT5  &  /UL5  &  PLDONE
       <SUM> UPL  &  /CNT5  &  UL5  &  PLDONE
       < SUM>  /LPL  & CNT0 & /UPL & CNT2 & CNT4  & /UP & CNT3 & CNT1
       < SUM>  /LPL  & /CNT0 & /UPL & /CNT2 & /CNT4  & UP & /CNT3 & /CNT1
       < SUM>  CNT0 & CNT2 & /PLDONE & CNT4 & /UP  & CNT3 & CNT1
       < SUM>  /CNT0 & /CNT2 & /PLDONE & /CNT4  & UP  & /CNT3 & /CNT1;

/CNT6=  <XSUM>  /CNT6
       < SUM>  /LL6  & LPL  & PLDONE & CNT6
       < SUM>  LL6  & LPL  & PLDONE & /CNT6
       < SUM>  UPL & PLDONE & CNT6 & /UL6
       < SUM>  UPL & PLDONE & /CNT6 & UL6
       < SUM> /LPL&CNT0&/UPL&CNT2&CNT5&CNT4  & /UP & CNT3 & CNT1
       < SUM>  /LPL & /CNT0 & /UPL & /CNT2 & /CNT5 & /CNT4 & UP & /CNT3 & /CNT1
       < SUM> CNT0&CNT2&CNT5&/PLDONE&CNT4  & /UP & CNT3 & CNT1
       < SUM>  /CNT0 & /CNT2 & /CNT5 & /PLDONE  & /CNT4 & UP & /CNT3 & /CNT1;
```

Appendix C. Source Code for Up/Down Counter (continued)

```
/CNT7=  <XSUM>  /CNT7
        < SUM>  /LL7  & LPL  & CNT7 & PLDONE
        < SUM>  LL7  & LPL  & /CNT7 & PLDONE
        <SUM>  UPL  & /UL7  & CNT7  & PLDONE
        <SUM>  UPL  & UL7  & /CNT7  & PLDONE
        < SUM>  /LPL & CNT0 & /UPL & CNT2 & CNT5  & CNT6 & CNT4 & /UP & CNT3 & CNT1
        <SUM> /LPL  & /CNT0 & /UPL  & /CNT2 & /CNT5 & /CNT6 & /CNT4 & UP & /CNT3 & /CNT1
        < SUM>  CNT0 & CNT2 & CNT5 & /PLDONE  & CNT6 & CNT4 & /UP & CNT3 &CNT1
        < SUM>  /CNT0 & /CNT2 & /CNT5 & /PLDONE   & /CNT6 & /CNT4 & UP & /CNT3 & /CNT1;

node29  =  <SUM>  reset;

UP= < XSUM>  UP
        <SUM>  /UEQUAL  &  /UP
        <SUM>  /LEQUAL  &  UP
        < SUM>  UPL & PLDONE & /UP
        < SUM>  LPL  & PLDONE & UP;

PLDONE= < SUM>  /LPL  & /UPL;

LEQUAL= < SUM>  LL6  & /CNT6
        < SUM>  /LL7  & CNT7
        < SUM>  LL7  & /CNT7
        < SUM>  LL3  & /CNT3
        < SUM>  /LL5  & CNT5
        < SUM>  LL5  & /CNT5
        < SUM>  /LL1  & CNT1
        < SUM>  LL0  & /CNT0
        < SUM>  /LL2  & CNT2
        < SUM>  /LL4  & CNT4
        < SUM>  LL4  & /CNT4
        < SUM>  /LL0  & CNT0
        < SUM>  LL1  & /CNT1
        < SUM>  /LL6  & CNT6
        < SUM>  /LL3  & CNT3
        < SUM>  LL2  & /CNT2;

UEQUAL= < SUM>  /CNT6 & UL6
        < SUM>  /UL7 & CNT7
        < SUM>  UL7 & /CNT7
        < SUM>  UL3 & /CNT3
        < SUM>  CNT5 & /UL5
        < SUM>  /CNT5 & UL5
        < SUM>  /UL1 & CNT1
        < SUM>  /CNT0 & UL0
        < SUM>  CNT2 & /UL2
        < SUM>  /UL4 & CNT4
        < SUM>  UL4 & /CNT4
        < SUM>  CNT0 & /UL0
        < SUM>   UL1 & /CNT1
        < SUM>  CNT6 & /UL6
        < SUM>  /UL3 & CNT3
        < SUM>  /CNT2 & UL2;
```

Appendix D. Source Code for Crossbar Switch

```
CY7C330;

configure;
clk (node=1), iclk, pl,
d0, d1, d2, d3,
d4 (node =9), d5, d6, d7, d8, d9,
d10 (node=35,src=15), d11 (node=36, src=17),
d12 (node=37,src=19), d13 (node=38, src=24),
d14 (node=39, src=26), d15 (node=40, src=28),
sa1 (node=17), sa2 (node=15), sa3 (node=34),
sb1 (node=24), sb2 (node=23), sb3 (node=33),
sc1 (node=19), sc2 (node=26), sc3 (node=32),
sd1 (node=20), sd2 (node=28), sd3 (node=31),
y0(node=27,iop,iclk=3),                    {Input reg is sa0}
y1(node=25,iop,iclk=3),                    {Input reg is sb0}
y2(node=18,iop,iclk=3),                    {Input reg is sc0}
y3(node=16,iop,iclk=3),                    {Input reg is sd0}

EQUATIONS;

/sa1 = <SUM> /pl & /sa2
       <SUM> pl & /sa1;

 /sa2 =<SUM> /pl & sa3
       <SUM> pl & /sa2;

sa3 =   <SUM> /pl & d0
        <SUM> pl & sa3;

/sb1=   <SUM> /pl & /sb2
       <SUM> pl & /sb1;

/sb2=   < SUM> /pl & sb3
       <SUM> pl & /sb2;

sb3 =  <SUM> /pl & d1
       <SUM> pl & sb3;

/sc1=   < SUM> /pl & /sc2
       <SUM> pl & /sc1;

/sc2 =  < SUM> /pl & sc3
       <SUM> pl & /sc2;

sc3 =  < SUM> /pl & d2
       <SUM> pl & sc3;

sd1 =  < SUM> /pl & /sd2
       <SUM> pl & /sd1;

/sd2 =  < SUM> /pl & sd3
       <SUM> pl & /sd2;

sd3 =  < SUM> /pl & d3
       < SUM> pl & sd3;
```

Appendix D. Source Code for Crossbar Switch (continued)

```
/y3 = < OE>  /pl
      <SUM> pl & /d0 & /sa3 & /sb3 & /sc3 & /sd3
      <SUM> pl & /d1 & sa3 & /sb3 & /sc3 & /sd3
      <SUM> pl & /d2 & /sa3 & sb3 & /sc3 & /sd3
      <SUM> pl & /d3 & sa3 & sb3 & /sc3 & /sd3
      <SUM> pl & /d4 & /sa3 & /sb3 & sc3 & /sd3
      <SUM> pl & /d5 & sa3 & /sb3 & sc3 & /sd3
      <SUM> pl & /d6 & /sa3 & sb3 & sc3 & /sd3
      <SUM> pl & /d7 & sa3 & sb3 & sc3 & /sd3
      <SUM> pl & /d8 & /sa3 & /sb3 & /sc3 & sd3
      <SUM> pl & /d9 & sa3 & /sb3 & /sc3 & sd3
      <SUM> pl & /sa3 & sb3 & /sc3 & sd3 & /d10
      <SUM> pl & sa3 & sb3 & /sc3 & sd3 & /d11
      < SUM>  pl & /sa3 & /sb3 & /d12 & sc3 & sd3
      <SUM> pl & /d13 & sa3 & /sb3 & sc3 & sd3
      <SUM> pl & /d14 & /sa3 & sb3 & sc3 & sd3
      <SUM> pl & /d15 & sa3 & sb3 & sc3 & sd3
      <SUM> /pl  &  sd1;

/y2 = < OE>  /pl
      < SUM>  pl & /d0 & sd2 & sc2 & sb2 & sa2
      < SUM>  pl & /d1 & sd2 & sc2 & sb2 & /sa2
      < SUM>  pl & /d2 & sd2 & sc2 & /sb2 & sa2
      < SUM>  pl & /d3 & sd2 & sc2 & /sb2 & /sa2
      < SUM>  pl & /d4 & sd2 & /sc2 & sb2 & sa2
      < SUM>  pl & /d5 & sd2 & /sc2 & sb2 & /sa2
      < SUM>  pl & /d6 & sd2 & /sc2 & /sb2 & sa2
      < SUM>  pl & /d7 & sd2 & /sc2 & /sb2 & /sa2
      < SUM>  pl & /d8 & /sd2 & sc2 & sb2 & sa2
      < SUM>  pl & /d9 & /sd2 & sc2 & sb2 & /sa2
      < SUM>  pl & /sd2 & sc2 & /sb2 & /d10 & sa2
      < SUM>  pl & /sd2 & sc2 & /sb2 & /d11 & /sa2
      < SUM>  pl & /sd2 & /sc2 & sb2 & /d12 & sa2
      < SUM>  pl & /sd2 & /sc2 & /d13 & sb2 & /sa2
      < SUM>  pl & /sd2 & /sc2 & /d14 & /sb2 & sa2
      < SUM>  pl & /sd2 & /d15 & /sc2 & /sb2 & /sa2
      <SUM> /pl  &  sc1;

/y1 = < OE>  /pl
      < SUM>  pl & /d0 & sb1 & sd1 & sc1 & sa1
      < SUM>  pl & /d1 & sb1 & sd1 & sc1 & /sa1
      < SUM>  pl & /d2 & /sb1 & sd1 & sc1 & sa1
      < SUM>  pl & /d3 & /sb1 & sd1 & sc1 & /sa1
      < SUM>  pl & /d4 & sb1 & sd1 & /sc1 & sa1
      < SUM>  pl & /d5 & sb1 & sd1 & /sc1 & /sa1
      < SUM>  pl & /d6 & /sb1 & sd1 & /sc1 & sa1
      < SUM>  pl & /d7 & /sb1 & sd1 & /sc1 & /sa1
      < SUM>  pl & /d8 & sb1 & /sd1 & sc1 & sa1
      < SUM>  pl & /d9 & sb1 & /sd1 & sc1 & /sa1
      < SUM>  pl & /sb1 & /sd1 & sc1 & sa1 & /d10
      < SUM>  pl & /sb1 & /sd1 & sc1 & /d11 & /sa1
      < SUM>  pl & sb1 & /sd1 & /d12 & /sc1 & sa1
      < SUM>  pl & sb1 & /d13 & /sd1 & /sc1 & /sa1
      < SUM>  pl & /d14 & /sb1 & /sd1 & /sc1 & sa1
      < SUM>  pl & /d15 & /sb1 & /sd1 & /sc1 & /sa1
      <SUM> /pl  &  sb1;
```

Appendix D. Source Code for Crossbar Switch (continued)

```
/y0 = < OE>  /pl
      < SUM>  pl & /d0 & /y0 & /y1 & /y2 & /y3
      < SUM>  pl & /d1 & y0 & /y1 & /y2 & /y3
      < SUM>  pl & /d2 & /y0 & y1 & /y2 & /y3
      < SUM>  pl & /d3 & y0 & y1 & /y2 & /y3
      < SUM>  pl & /d4 & /y0 & /y1 & y2 & /y3
      < SUM>  pl & /d5 & y0 & /y1 & y2 & /y3
      < SUM>  pl & /d6 & /y0 & y1 & y2 & /y3
      < SUM>  pl & /d7 & y0 & y1 & y2 & /y3
      < SUM>  pl & /d8 & /y0 & /y1 & /y2 & y3
      < SUM>  pl & /d9 & y0 & /y1 & /y2 & y3
      < SUM>  pl & /y0 & y1 & /y2 & y3 & /d10
      < SUM>  pl & y0 & y1 & /y2 & /d11 & y3
      < SUM>  pl & /y0 & /y1 & /d12 & y2 & y3
      < SUM>  pl & y0 & /y1 & /d13 & y2 & y3
      < SUM>  pl & /y0 & /d14 & y1 & y2 & y3
      < SUM>  pl & /d15 & y0 & y1 & y2 & y3
      <SUM>  /pl  &  sa1;
```

CYPRESS SEMICONDUCTOR

Using the Cypress CY7C330 in Closed-Loop Servo Control

This application note examines a common facet of engineering design — control systems — and offers an alternative to common implementations. Along with an overview of the subject, this application note explores the tradeoffs among several implementation strategies.

Also included here is a description of a PLD-based method that offloads the processing bandwidth requirements of a controlling CPU. Implemented in a Cypress CY7C330 PLD, this method has been successfully employed in a high-speed customer application — a laser mirror-positioning servo.

Control System Concepts

Control system theory is applied to areas as diverse as pneumatic controls and economic models. Analyzing control system behavior mathematically relies heavily on an understanding of Laplace and Z-transforms (see the *References*). However, this application note deals with the subject on a more practical level.

Control systems fall into two major categories: open loop and closed loop. An open-loop system generates outputs based on input conditions, but has no feedback from the output to verify or correct the output condition. Examples of open-loop systems include light switches (although you could reasonably argue that the human is the feedback loop) and self-timed, free-running traffic-control signals.

Closed-loop systems, on the other hand, provide information on system status to the controller. Examples of closed-loop systems include the eye-brain system you are using to read this line of text, the engine thermostat in most automobiles, and the print head of a dot-matrix printer. The closed-loop application described later in this application note consists of a motor-driven mirror that can rotate 360 degrees in either direction.

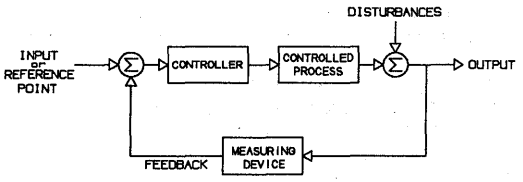


Figure 1. Closed-Loop Servo System

Closed-loop systems use information from the environment under control to influence the output. Block diagrams such as the one in *Figure 1* typically represent such a control system.

Control System Influences

In a closed-loop design, numerous factors influence the system behavior. Among them are:

Input, I(t): The system input is the signal from an external source that references the desired steady-state behavior. In the mirror servo system, the steady-state output is the absolute position at a given location within a given accuracy. The input is also known as the reference or set point.

Summing function: This is the section of the control system that determines the amount of error, E(t), currently in the system. It is the difference between the reference point and the controlled environment's present state. In a motor servo system, E(t) is the difference between the target reference position and the motor's present position. In an analog circuit, an operational amplifier usually implements the summing function.

Controller: Most control systems incorporate a controller that receives the error signal as an input and generates an output that attempts to reduce this error to within a specific tolerance (ideally 0). The controller has a control mode that determines how the controller should manipulate the error signal to produce a control signal. Common control modes include proportional, integral, differential, and the combination of these three — PID. Approximately 80 - 90 percent of industrial control implementations use variations of the PID method.

Controlled device: The object of the control system is to have a controlled device perform satisfactorily. This is the motor, in the case of the mirror servo.

Output, O(t): This is the physical characteristic to be controlled. In an automobile thermostat system, O(t) is the engine's temperature. In the mirror servo, O(t) is the mirror's position.

Disturbance, D(t): Any influence on the system that negatively affects the desired output is called a disturbance. In an automobile, operation in bumper-to-bumper traffic that reduces airflow through the radiator is a disturbance to the thermostat.

This is only a partial list of the influences in a closed-loop control system, but the factors mentioned are the most significant for the mirror servo. (For more complete information, consult any of the *References.*)

Control System Parameters

Some of the parameters used to quantize control system behavior are:

Accuracy: the difference between ideal and actual steady-state system behavior.

Settling time: the time required to reach steady state after the reference point is changed or set.

Percentage overshoot: the difference between the reference point and the maximum excursion after passing through the reference point.

Jitter: a condition that occurs when the controlling element improperly overcompensates for an overshoot of the reference point. The overcompensation results in an undershoot that is again overcompensated for and produces overshoot. Jitter can increase the system's settling time or result in unstable oscillations that never attain the reference point.

Rise time: the time required for the system's output to increase from 10 to 90 percent of the final value.

Control System Implementations

Control system implementations vary from purely analog to completely digital. Many popular implementations use a hybrid of digital and analog techniques. The approach described here uses a digital element to perform the summing, the control, and part of the feedback function. This approach and the pure-analog method are possibly the most often used.

Each approach has its own tradeoffs. Because analog systems continuously perform the summing function (usually with an op-amp), they are immune to the problems associated with data quantization. Thus analog systems usually offer excellent stability.

Digital hybrids offer good sensitivity, immunity to noise, resolution, and flexibility, along with minimized drift. These systems are usually easier to design at a lower cost, compared to alternatives. Microprocessors make it relatively easy to implement the system's controller and summing function on one chip.

When you use a microprocessor, you can take advantage of several algorithms for generating the control signal. The simplest is proportional control, in which the correction made is proportional to the error signal. The value by which the error is scaled is the system's proportionality constant or gain. Proportional control offers an intuitively reasonable solution: the larger the error, the larger the corrective signal.

Another control algorithm is integral control, where the corrective signal is based on the error's time integral multiplied by a weighting factor. You typically calculate this value using a numeric approximation. Integral control is usually combined with proportional control to increase accuracy or reduce steady-state error.

One other control algorithm, derivative control, employs a corrective signal comprised of the error signal's derivative over time multiplied by a weighting factor. Again, a numeric approximation is used to calculate the derivative. Combining this method with proportional control contributes a stabilizing influence to the system. However, noisy systems often omit the derivative function because it amplifies high-frequency disturbances.

When all three control algorithms are combined, they constitute proportional + integral + derivative, or PID control. You can verify the influences of the integral and derivative methods on PID with analysis based on Laplace transforms. A PID tradeoff is that it reduces the processor bandwidth available to perform other tasks. PID systems also require a finite amount of time to calculate the output value.

Another factor to consider in a hybrid control system is the system's sampling/processing rate. Several reference books indicate that the sampling rate for a closed-loop control system should be significantly above the minimum dictated by Shannon's sampling theorem. Thus, rather than operating at the Nyquist frequency (twice the highest frequency sampled), the sampling rate would be eight to ten times the highest sampled frequency. The reasons for this practice include an uncertainty associated with determining the sampled signal's highest frequency component, the possibility of aliasing, and the decrease in system stability that can result from a too-low sampling rate. Unfortunately, increasing the sampling rate quickly consumes the available bandwidth of a microprocessor-based implementation.

Using the CY7C330 in Servo Control

The Cypress CY7C330 can help offload the microprocessor in a high-speed servo control system. The application described here positions a mirror to form images with a laser beam. A previous implementation of this system used a 68000 microprocessor in the servo loop. But as the number of tasks on the 68000 increased, the processor's ability to maintain a stable servo system became marginal. The CY7C330-based version maintains servo loop stability as well as freeing processor system throughput with a minimum of additional cost and complexity.

Several features of the CY7C330 are fundamental to understanding this design (see the CY7C330 block

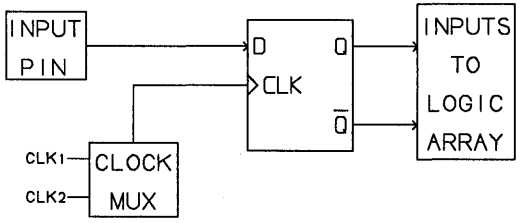


Figure 2. CY7C330 Dedicated Input Register

diagram in *Figure 1* of "Understanding the CY7C330 Synchronous EPLD"). The dedicated input registers (*Figure2*), for example, allow data to be loaded into the chip with either of two data input clocks — CLK1 (pin 2) or CLK2 (pin 3). You choose the input clock at program time via an EPROM configuration fuse.

The macrocells (*Figure 3*) also feature input registers, again with two clocks for data entry. The ability to three-state the macrocell output drivers and load data into the macrocell input register allows you to use these macrocell input registers to hold reference values. This is handy in applications such as up/down counters, where the input registers can hold the counter upper/lower limit.

In the mirror-servo design, the macrocell input registers store the mirror's calculated target position and are clocked by CLK2. While actively controlling the servo, this design uses the dedicated input registers for loading the present mirror position from the servo loop. In command mode, though, the dedicated input registers hold data from the microprocessor that is used to calculate a new target position. In either case, CLK1 loads the dedicated input registers.

Mirror Servo Fundamentals

As *Figure 1* shows, the basic mechanism of control loops is proportional feedback of the error signal. If this loop acts as a self-contained coprocessor to the main CPU, the CPU is only required to input the reference point to which the mirror should be moved. Now the CPU no longer needs to perform the control algorithm at a pace equal to the sampling rate. Essentially, the processor can "set and forget" the servo coprocessor.

One way to implement this servo coprocessor is to add another microprocessor. This would add software and hardware (CPU, RAM, ROM, clock, I/O, interrupt control, etc.), and possibly require an in-circuit emulator for development if a low-cost microcontroller is used. Another possibility is to use an analog servo controller, but the accuracy requirements preclude this when drift is considered.

Another approach is to use several simple PLDs in a hybrid control-loop implementation. The system block

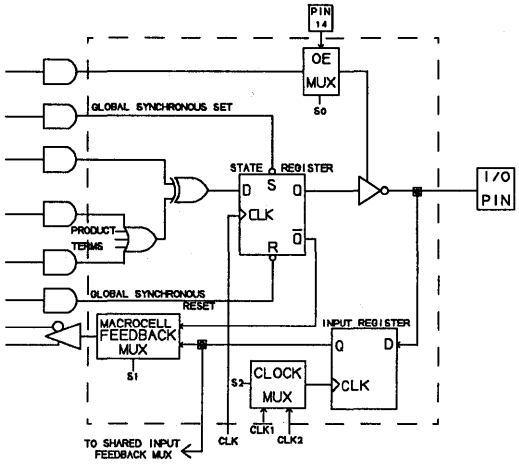


Figure 3. CY7C330 I/O Macrocell

diagram in *Figure 4* shows the general approach used. The design employs three CY7C330s that each generate an 8-bit accumulate for 24-bit precision. The microprocessor provides the CY7C330s with a 24-bit position reference target for the mirror. The CY7C330s latch this 24-bit value into their on-board registers.

The CY7C330s perform the control loop's summing and proportional feedback functions. The PLDs compare the 24-bit desired position to the present position, which is maintained in an external 24-bit present-position counter. The result is the error multiplied by a fixed unity gain. This proportional control signal is then converted to an analog signal, which is converted to a current level to control the positioning mirror's motor.

The motor's shaft has an optical encoder that creates a sin-cos analog signal. When converted to digital form, this signal indicates the direction of rotation and provides a pulse that increments or decrements the external 24-bit present-position counter. This allows the

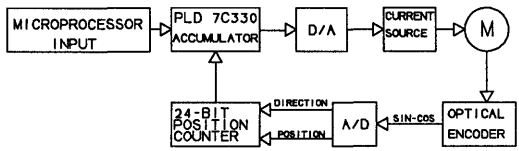


Figure 4. CY7C330 Servo Control Loop

loop to operate as fast as the slowest of the following elements: the CY7C330s configured as a multistage accumulator/subtractor, the D/A converter, or the A/D converter. The host microprocessor is completely decoupled from the servo loop. Should the microprocessor halt, the servo circuitry continues to maintain the desired reference position without intervention.

Details of the Mirror Servo

Getting into the inner workings of the mirror servo loop, the CY7C330 macrocell output registers act essentially as an accumulator. Depending on the mode of operation, the accumulator generates a value that is either a new servo-motor target position or the proportional error feedback value to the servo.

When the system starts, the macrocell input registers wake up with an initial value of 0. These registers are dedicated to holding the motor's present target position. At the same time, the external position counter is set to zero. Then the microprocessor steps the target position until the laser targets an alignment sensor.

The following steps accomplish this sequence: First, the outputs of the external 24-bit position counter are placed in a three-state condition. These outputs and the microprocessor's outputs act as inputs to the CY7C330's dedicated input registers. The processor drives a step value onto the inputs, and CLK1 clocks the value into the CY7C330's dedicated input registers. On CLK1's rising edge, this value is added to the present value in the macrocell input registers. The result of this addition moves to the macrocell output registers, and CLK2 clocks it into the same macrocell input registers that were a source value for the add.

Thus, in this mode, the CY7C330s use the present value on the dedicated input pins to adjust the target position in the macrocell input registers with an accumulate cycle. This target-position update cycle is pictured in *Figure 5*. The microprocessor always provides data as a delta or step from the present position. The accumulate can be either an add or subtract. Subtracts are accomplished by providing the step data from the microprocessor in 2's-complement form. After alignment, the position and accumulator values are reset to zero, and the system is ready for operation.

In operation, the outputs from the microprocessor are three-stated, and the value from the 24-bit position counter is loaded into the the dedicated input registers. This value is always provided in a 2's-complement form by inverting the position counter's outputs (1's complement) and setting the carry in (C_{in}) input to one. The position-counter value is thus subtracted from the present-target-position value stored in the macrocell input registers; this forms the proportional error feedback value used to control the servo motor. *Figure 6* illustrates this servo control mode.

Note that the D/A converter does not need a 24-bit digital value for control. In practice, the circuit uses an 8-bit D/A value biased such that the eighth bit provides direction control (clockwise vs. counterclockwise). In the actual design, the upper 16 bits from the two most-significant CY7C330s are tested for rail High and Low conditions and generate generate two offscale bits each

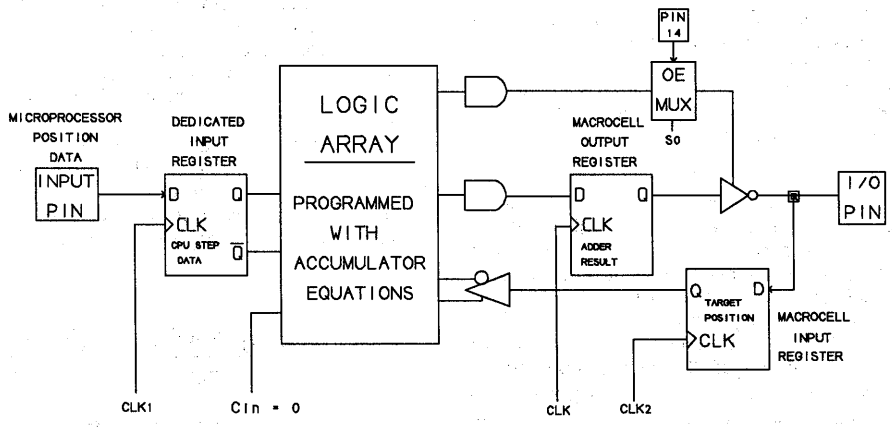


Figure 5. Target Update Mode Operation Sequence

(1) With external position counter's output three-state, host microprocessor drives position step data.
(2) Step data (provided in 2's complement form if a subtract is desired) is loaded into the 330 with CLK1.
(3) Step data is added or subtracted from present target position with logic equations to create new target position.
(4) New target position is clocked into macrocell output registers with CLK.
(5) On CLK2, the new target position is clocked into the macrocell input register.

for these conditions. The seven low-order bits, along with the four offscale bits, are passed to a second PLD (22V10), which drives the output to the D/A in the correct direction (eighth bit) and with the correct magnitude. If the four offscale bits indicate that the upper bits are all close to 0, the seven bits to the D/A are masked to 0. Likewise, if the upper bits are mostly 1, the D/A bits are set to 1.

The offscale bits are generated to minimize the number of inputs required for the subsequent PLD that feeds the D/A converter. The determination of how to use the offscale bits for compensation in the second PLD is specific to a given application.

The Accumulator Design

The backbone of the logic in this design is the CY7C330-based accumulator. The logic that implements this synchronous full adder is described by an equation for the sum and an equation for the carry of a given bit. The equation for the sum (S) at bit position n, with inputs A, B, and carry in (C_{in}) is:

$$Sn = (An \text{ XOR } Bn \text{ XOR } C_{in}).$$

The equation for the carry out is:

$$COUTn = (An * Bn) + (An * C_{in}) + (Bn * C_{in})$$

Figure 7 shows the equations for a 4-bit synchronous adder, whose sequence completes in four clocks. Because the objective is to calculate a complete 24-bit sum as quickly as possible, the equation for carry out (C0) from the adder's first bit can be substituted into the equation for the adder's second bit. This arrangement allows the first two bits to be added in a single clock cycle. Similarly, the equation for the carry out from the second bit can be substituted into the equation for the third sum, and so on. The resulting equations for three bits of substitution appear in *Figure8*.

The CY7C330's XOR product term is useful for reducing the number of product terms required for a given sum bit. However, even after Boolean reduction and utilization of the XOR product term, the fourth bit of the adder requires 30 product terms for the sum bit and 31 product terms for the carry out bit to generate a 4-bit result in a single clock cycle. Because a given CY7C330 macrocell provides a maximum of 19 product terms, the device must run the accumulate process over multiple 3-bit stages. The addition of the first three bits finishes after one clock cycle, the second three bits after two cycles, and so on. Implemented in three CY7C330s, the complete 24-bit accumulate therefore requires nine clock cycles. With 66-MHz devices, nine clock cycles translates to a complete calculation cycle of 120 ns.

Appendix A lists the minimized equations for one of the three 8-bit adder stages. The syntax used in this example is that of the Cypress PLD ToolKit. Variables B0 - B7 are the eight dedicated inputs sourced from either the microprocessor or the 24-bit position counter. INCLK is the CLK1 pin on the CY7C330 used to clock in the B0 - B7 variables. C_{in} is the carry in from external logic (set to one for subtraction when in control mode

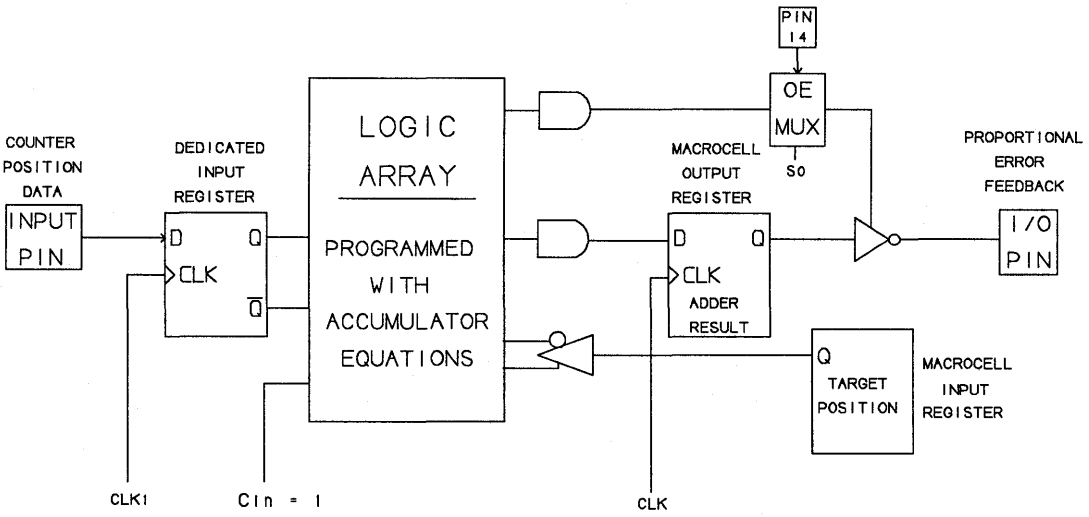


Figure 6. Control Mode Operation Sequence

(1) CLK1 loads external 24-bit position data (in 1's complement form) into CY7C330's dedicated input register.
(2) With carry in set to 1, logic equations subtract current position from target position to form error amount.
(3) Error result is clocked into macrocell output register with CLK and is available to servo motor interface.

```
/* Four Bit Adder - General Case */

Inputs: An, Bn ; Inputs to be added at Bit n
        CIN    ; Carry in to Adder

Outputs: Sn    ; Sum out for Bit n
         Cn    ; Carry out from adder stage n

/* Equations to be reduced */

S0 = A0 XOR B0 XOR CIN
C0= (A0 * B0) + (A0 * CIN) + (B0 * CIN)

S1 = A1 XOR B1 XOR C0
C1 = (A1 * B1) + (A1 * C0) + (B1 * C0)

S2 = A2 XOR B2 XOR C1
C2 = (A2 * B2) + (A2 * C1) + (B2 * C1)

S3 = A3 XOR B3 XOR C2
C3 = (A3 * B3) + (A3 * C2) + (B3 * C2)

/* C3 == Carry Out of Four Bit Adder */
```

Figure 7. Equations for Four-Bit Adder

on the first 8-bit adder stage) or from the previous stage of the adder.

A0 - A7 are the sum outputs for either target update or control mode. If the processor is updating the target position by a step increment, A0 - A7 are loaded into the macrocell input registers with CLK2 (named ACLK). When this new position update is being loaded, the output drivers of the macrocells are not three-stated with the OE pin or a product term equation. This allows ACLK to load the macrocell output registers (which have the newly calculated target position) into the macrocell input registers (which are used to hold the target position).

C2 and C5 are internal carry-out bits generated from the first and second 3-bit adder stages, respectively. Finally, COUT is the carry out generated as either the final carry out or as the input to the next 8-bit adder stage's carry in.

Appendix B shows the implementation of the two upper CY7C330 stages. The equations for the accumulator function are the same as in the previous equations. The additions here are the equations for detecting rail conditions and generating the offscale bits.

Note that the intent here has been to focus on a different approach to implementing a closed-loop servo

```
/* Synchronous 3 bit adder - derivative of General Case */
/* Uses substitution of Carry Out in first 3 bits to generate 3 bit
result in one clock cycle */

S0 = A0 XOR B0 XOR CIN
/ * C0= (A0 * B0) + (A0 * CIN) + (B0 * CIN) */

S1 = A1 XOR B1 XOR [(A0 * B0) + (A0 * CIN) + (B0 * CIN)]

/* C1 = (A1 * B1)
+ (A1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])
+ (B1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)]) */

S2 =  A2 XOR B2 XOR
{(A1 * B1)
+ (A1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])
+ (B1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])}

C2 = (A2 * B2)
+  (A2 *
{(A1 * B1)
+ (A1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])
+ (B1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])})
+  (B2 *
{(A1 * B1)
+ (A1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])
+ (B1 * [(A0 * B0) + (A0 * CIN) + (B0 * CIN)])}
```

Figure 8. Equations for a Synchronous 3-Bit Adder

controller, with the CY7C330 as the central element, and to disclose the details unique to the CY7C330. Many hardware implementation details are left to the designer, including the D/A design, feedback design, and the lead/lag compensation.

References

Houpis & Lamont, *Digital Control Systems - Theory, Hardware, Software (New York: McGraw-Hill, 1985)*

Ball & Pratt, *Engineering Applications of Microcomputers* (Prentice Hall Int'l (UK) Ltd, 1986)

Kuo, *Digital Control Systems* (New York: Holt, Rinehart, & Winston, Inc., 1980)

Gayakwad & Sokoloff, *Analog and Digital Control Systems* (Prentice Hall, 1988)

Bollinger & Duffie, *Computer Control of Machines & Processes* (New York: Addison - Wesley, 1988)

For more information on implementing the CY7C330-based, 24-bit up/down position counter mentioned in this application note, consult the application note, "66-MHz CY7C330 Synchronous State Machine."

Appendix A. PLD ToolKit Code for an 8-Bit Accumulator

```
{Mark Aaldering - Cypress Semiconductor - 8-bit accumulator - June 14, 1989}

CY7C330;

CONFIGURE;                          { Dedicated input registers. Default configuration is use of pin 2 for clock }

Outclk(node=1),
Inclk(node=2),
Aclk(node=3),
CIN(node=4),
B0(node=5),
B1(node=6),
B2(node=7),
B3(node=9),
B4(node=10),
B5(node=11),
B6(node=12),
B7(node=13),
oe(node=14),

{Output nodes assigned to maximize available product term utilization. In the following declarations, the 7C330's
macrocell outputs are configured as follows:
        ireg--This sets the macrocell feedback MUX for feedback from the macrocell input register instead of the
              (default) macrocell output register (rgd)

        iclk=3--This selects the clock on pin 3 instead of the default (used for the inputs above) of clock on pin 2 for the
                macrocell input register

        IOP--Same as ireg.

        nenbpt--Selects OE control from pin 14 instead of a product term }

A0(node=28,iop,iclk=3,ireg,nenbpt),     { Sum 0 / Accum. Feedback Register 0 }
A1(node=15,iop,iclk=3,ireg,nenbpt),     { Sum 1 / Accum. Feedback Register 1 }
A2(node=20,iop,iclk=3,ireg,nenbpt),     { Sum 2 / Accum. Feedback Register 2 }
A3(node=17,iop,iclk=3,ireg,nenbpt),     { Sum 3 / Accum. Feedback Register 3 }
A4(node=26,IOP,iclk=3,ireg,nenbpt),     { Sum 4 / Accum. Feedback Register 4 }
A5(node=23,IOP,iclk=3,ireg,nenbpt),     { Sum 5 / Accum. Feedback Register 5 }
A6(node=19,IOP,iclk=3,ireg,nenbpt),     { Sum 6 / Accum. Feedback Register 6 }
A7(node=24,IOP,iclk=3,ireg,nenbpt),     { Sum 7 / Accum. Feedback Register 7 }
COUT(node=18,nenbpt),                       { Carry out }
C2(node= 32),                               { Carry 2 - Hidden }
C5(node= 34),                               { Carry 5 - Hidden }

        { Available nodes     #  P.T.'s }
        { I/O macrocell      - 16 - 19   }
        { I/O macrocell      - 25 - 17   }
        { I/O macrocell      - 27 - 19   }
        { hidden macrocell - 31 - 13     }
        { hidden macrocell - 33 - 11     }

        {End of configuration section}
```

Appendix A. PLD ToolKit Code for an 8-Bit Accumulator (continued)

```
        {Logic equation section}

EQUATIONS;

        {A0: 2 product terms, pin 28: 9 P.T. Available}

/A0  =  < XSUM>   CIN
        < SUM>  /A0 * /B0
            +     A0 *  B0;

        {A1: 6 product terms, pin 15: 9 P.T. Available}

/A1  =  < XSUM>   /A1
        < SUM>   B1 * /B0 * /CIN
            +     /B1 *  B0 *  CIN
            +     /B1 *  A0 *  CIN
            +     /B1 *  A0 *  B0
            +      B1 * /A0 * /CIN
            +      B1 * /A0 *  B0;

        {A2: 14 product terms, pin 20: 15 P.T. Available}

/A2  =  < XSUM>  /A2
        < SUM>    B2 * /A1 * /B1
            +     /B2 *  B1 *  B0 *  CIN
            +     /B2 *  A1 *  B0 *  CIN
            +     /B2 *  B1 *  A0 *  CIN
            +     /B2 *  A1 *  A0 *  CIN
            +     /B2 *  B1 *  A0 *  B0
            +     /B2 *  A1 *  A0 *  B0
            +      B2 * /B1 * /B0 * /CIN
            +      B2 * /A1 * /B0 * /CIN
            +     /B2 *  A1 *  B1
            +      B2 * /B1 * /A0 * /CIN
            +      B2 * /A1 * /A0 * /CIN
            +      B2 * /B1 * /A0 * /B0
            +      B2 * /A1 * /A0 * /B0;

        {C2: 15 product terms, virtual pin 32: 17 P.T. Available}

C2   =  < SUM>    B2 * B1 * B0 * CIN
            +     A2 * B1 * B0 * CIN
            +     B2 * A1 * B0 * CIN
            +     A2 * A1 * B0 * CIN
            +     B2 * B1 * A0 * CIN
            +     A2 * B1 * A0 * CIN
            +     B2 * A1 * A0 * CIN
            +     A2 * A1 * A0 * CIN
            +     B2 * B1 * A0 * B0
            +     A2 * B1 * A0 * B0
            +     B2 * A1 * A0 * B0
            +     A2 * A1 * A0 * B0
            +     B2 * A1 * B1
            +     A2 * A1 * B1
            +     A2 * B2;
```

Appendix A. PLD ToolKit Code for an 8-Bit Accumulator (continued)

```
        {A3: 2 product terms, pin 17: 11 P.T. Available}

/A3 =  < XSUM>  C2
       < SUM>  /A3 * /B3
         +       A3 * B3;

        {A4: 6 product terms, pin 26: 11 P.T. Available}

/A4 =  < XSUM>  /A4
       < SUM>   B4 * /B3 * /C2
         +      /B4 * B3 * C2
         +      /B4 * A3 * C2
         +      /B4 * A3 * B3
         +       B4 * /A3 * /C2
         +       B4 * /A3 * B3;

        {A5: 14 product terms, pin 23: 15 P.T. Available}

/A5 =  < XSUM>  /A5
       < SUM>   B5 * /A4 * /B4
         +      /B5 * B4 * B3 * C2
         +      /B5 * A4 * B3 * C2
         +      /B5 * B4 * A3 * C2
         +      /B5 * A4 * A3 * C2
         +      /B5 * B4 * A3 * B3
         +      /B5 * A4 * A3 * B3
         +       B5 * /B4 * /B3 * /C2
         +       B5 * /A4 * /B3 * /C2
         +      /B5 * A4 * B4
         +       B5 * /B4 * /A3 * /C2
         +       B5 * /A4 * /A3 * /C2
         +       B5 * /B4 * /A3 * /B3
         +       B5 * /A4 * /A3 * /B3;

        {C5: 15 product terms, virtual pin 34: 19 P.T. Available}

C5 =   < SUM>  B5 * B4 * B3 * C2
         +      A5 * B4 * B3 * C2
         +      B5 * A4 * B3 * C2
         +      A5 * A4 * B3 * C2
         +      B5 * B4 * A3 * C2
         +      A5 * B4 * A3 * C2
         +      B5 * A4 * A3 * C2
         +      A5 * A4 * A3 * C2
         +      B5 * B4 * A3 * B3
         +      A5 * B4 * A3 * B3
         +      B5 * A4 * A3 * B3
         +      A5 * A4 * A3 * B3
         +      B5 * A4 * B4
         +      A5 * A4 * B4
         +      A5 * B5;
```

Appendix A. PLD ToolKit Code for an 8-Bit Accumulator (continued)

```
{A6:  2 product terms, pin 19: 13 P.T. Available}


/A6 =   < XSUM>   C5
        < SUM>   /A6 * /B6
          +        A6 *  B6;

        {A7:  6 product terms, pin 24: 13 P.T. Available}

/A7 =   < XSUM>  /A7
        < SUM>    B7 * /B6 * /C5
          +      /B7 *  B6 *  C5
          +      /B7 *  A6 *  C5
          +      /B7 *  A6 *  B6
          +       B7 * /A6 * /C5
          +       B7 * /A6 *  B6;

        {COUT:  7 product terms, pin 18: 17 P.T. Available}

/COUT = < SUM>   /B7 * /B6 * /C5
          +      /A7 * /B6 * /C5
          +      /B7 * /A6 * /C5
          +      /A7 * /A6 * /C5
          +      /B7 * /A6 * /B6
          +      /A7 * /A6 * /B6
          +      /A7 * /B7;

        {End of file.}
```

Appendix B. PLD ToolKit Code for an Accumulator with Rail Condition

```
{Mark Aaldering - Cypress Semiconductor - 8-bit accumulator with rail condition outputs - June 14, 1989}

CY7C330;

CONFIGURE;                          { Dedicated input registers. Default configuration is use of pin 2 for clock }

Outclk(node=1),
Inclk(node=2),
Aclk(node=3),
Cin(node=4),
B0(node=5),
B1(node=6),
B2(node=7),
B3(node=9),
B4(node=10),
B5(node=11),
B6(node=12),
B7(node=13),
oe(node=14),

{Output nodes assigned to maximize available product term utilization. In the following declarations, the 330's
macrocell outputs are configured as follows:

    ireg--This sets the macrocell feedback MUX for feedback from the macrocell input register instead of the
        (default) macrocell output register (rgd)

    iclk=3--This selects the clock on pin 3 instead of the default (used for the inputs above) of clock on pin 2 for the
            macrocell input register

    IOP--Same as ireg.

    nenbpt--Selects OE control from pin 14 instead of a product term  }

A0(node=28,iop,iclk=3,ireg,nenbpt),     { Sum 0 / Accum. Feedback Register 0 }
A1(node=15,iop,iclk=3,ireg,nenbpt),     { Sum 1 / Accum. Feedback Register 1 }
A2(node=20,iop,iclk=3,ireg,nenbpt),     { Sum 2 / Accum. Feedback Register 2 }
A3(node=17,iop,iclk=3,ireg,nenbpt),     { Sum 3 / Accum. Feedback Register 3 }
A4(node=26,iop,iclk=3,ireg,nenbpt),     { Sum 4 / Accum. Feedback Register 4 }
A5(node=23,iop,iclk=3,ireg,nenbpt),     { Sum 5 / Accum. Feedback Register 5 }
A6(node=19,iop,iclk=3,ireg,nenbpt),     { Sum 6 / Accum. Feedback Register 6 }
A7(node=24,iop,iclk=3,ireg,nenbpt),     { Sum 7 / Accum. Feedback Register 7 }
COUT(node= 18,nenbpt),                  { Carry Out }
C2(node= 32),                                 { Carry 2 - Hidden }
C5(node= 34),                                 { Carry 5 - Hidden }
R0(node= 16,nenbpt),                    { Rail Bit 0 }
R1(node= 25,nenbpt),                    { Rail bit 1 }

    { Available nodes     #  P.T.'s }
    { I/O macrocell     - 27 - 19    }
    { Hidden macrocell - 31 - 13     }
    { Hidden macrocell - 33 - 11     }

    {End of configuration section}
```

Appendix B. PLD ToolKit Code for an Accumulator with Rail Condition (continued)

```
{Logic equation section}
EQUATIONS;

        {A0: 2 product terms, pin 28:  9 P.T. Available}

/A0 =   < XSUM>  CIN
        < SUM>  /A0 * /B0
           +      A0 *  B0;

        {A1: 6 product terms, pin 15:  9 P.T. Available}

/A1 =   < XSUM> /A1
        < SUM>   B1 * /B0 * /CIN
           +    /B1 *  B0 *  CIN
           +    /B1 *  A0 *  CIN
           +    /B1 *  A0 *  B0
           +     B1 * /A0 * /CIN
           +     B1 * /A0 *  B0;

        {A2: 14 product terms, pin 20:  15 P.T. Available}

/A2 =  < XSUM>  /A2
      < SUM>    B2 * /A1 * /B1
         +     /B2 *  B1 *  B0 *  CIN
         +     /B2 *  A1 *  B0 *  CIN
         +     /B2 *  B1 *  A0 *  CIN
         +     /B2 *  A1 *  A0 *  CIN
         +     /B2 *  B1 *  A0 *  B0
         +     /B2 *  A1 *  A0 *  B0
         +      B2 * /B1 * /B0 * /CIN
         +      B2 * /A1 * /B0 * /CIN
         +     /B2 *  A1 *  B1
         +      B2 * /B1 * /A0 * /CIN
         +      B2 * /A1 * /A0 * /CIN
         +      B2 * /B1 * /A0 * /B0
         +      B2 * /A1 * /A0 * /B0;

        {C2: 15 product terms, virtual pin 32:  17 P.T. Available}

C2 =  < SUM>    B2 * B1 * B0 * CIN
         +      A2 * B1 * B0 * CIN
         +      B2 * A1 * B0 * CIN
         +      A2 * A1 * B0 * CIN
         +      B2 * B1 * A0 * CIN
         +      A2 * B1 * A0 * CIN
         +      B2 * A1 * A0 * CIN
         +      A2 * A1 * A0 * CIN
         +      B2 * B1 * A0 * B0
         +      A2 * B1 * A0 * B0
         +      B2 * A1 * A0 * B0
         +      A2 * A1 * A0 * B0
         +      B2 * A1 * B1
         +      A2 * A1 * B1
         +      A2 * B2;
```

Appendix B. PLD ToolKit Code for an Accumulator with Rail Condition (continued)

```
        {A3: 2 product terms, pin 17: 11 P.T. Available}

/A3 =  < XSUM>  C2
       < SUM>  /A3 * /B3
         +      A3 * B3;

        {A4: 6 product terms, pin 26: 11 P.T. Available}

/A4  = <XSUM>  /A4
       < SUM>   B4 * /B3 * /C2
         +     /B4 * B3 * C2
         +     /B4 * A3 * C2
         +     /B4 * A3 * B3
         +      B4 * /A3 * /C2
         +      B4 * /A3 * B3;

        {A5: 14 product terms, pin 23: 15 P.T. Available}

/A5 =  < XSUM>  /A5
       < SUM>   B5 * /A4 * /B4
         +     /B5 * B4 * B3 * C2
         +     /B5 * A4 * B3 * C2
         +     /B5 * B4 * A3 * C2
         +     /B5 * A4 * A3 * C2
         +     /B5 * B4 * A3 * B3
         +     /B5 * A4 * A3 * B3
         +      B5 * /B4 * /B3 * /C2
         +      B5 * /A4 * /B3 * /C2
         +     /B5 * A4 * B4
         +      B5 * /B4 * /A3 * /C2
         +      B5 * /A4 * /A3 * /C2
         +      B5 * /B4 * /A3 * /B3
         +      B5 * /A4 * /A3 * /B3;

        {C5: 15 product terms, virtual pin 34: 19 P.T. Available}

C5 =  < SUM>  B5 * B4 * B3 * C2
         +     A5 * B4 * B3 * C2
         +     B5 * A4 * B3 * C2
         +     A5 * A4 * B3 * C2
         +     B5 * B4 * A3 * C2
         +     A5 * B4 * A3 * C2
         +     B5 * A4 * A3 * C2
         +     A5 * A4 * A3 * C2
         +     B5 * B4 * A3 * B3
         +     A5 * B4 * A3 * B3
         +     B5 * A4 * A3 * B3
         +     A5 * A4 * A3 * B3
         +     B5 * A4 * B4
         +     A5 * A4 * B4
         +     A5 * B5;
```

Appendix B. PLD ToolKit Code for an Accumulator with Rail Condition (continued)

```
        {A6: 2 product terms, pin 19: 13 P.T. Available}

/A6 =   < XSUM>  C5
      < SUM>  /A6 * /B6
        +       A6 *  B6;

        {A7: 6 product terms, pin 24: 13 P.T. Available}

/A7 =   < XSUM>  /A7
      < SUM>    B7 * /B6 * /C5
        +      /B7 *  B6 *  C5
        +      /B7 *  A6 *  C5
        +      /B7 *  A6 *  B6
        +       B7 * /A6 * /C5
        +       B7 * /A6 *  B6;

        {COUT: 7 product terms, pin 18: 17 P.T. Available}

/COUT = < SUM>  /B7 * /B6 * /C5
              +      /A7 * /B6 * /C5
              +      /B7 * /A6 * /C5
              +      /A7 * /A6 * /C5
              +      /B7 * /A6 * /B6
              +      /A7 * /A6 * /B6
              +      /A7 * /B7;

        {R0: rail bit 0; Arbitrarily equation chosen to detect when upper 5 bits are all 1 - this decision is a matter of
preference        output active low}

/R0  =  < SUM>  A7 * A6 * A5 * A4 * A3;

        { R1: rail bit 1; Again, arbitrarily chosen to reflect value of carry out, therefore this is a redundant output - active lo
        output}

/R1  =  < SUM>  COUT;

{End of file}
```

CYPRESS SEMICONDUCTOR

FDDI Physical Connection Management Using the CY7C330

This application note shows how you can use the Cypress CY7C330 programmable logic device (PLD) to implement the Physical Connection Management (PCM) state machine specified in the Station Management (SMT) of the Fiber Distributed Data Interface (FDDI) standard. Along with a brief overview of the FDDI standard, this application note explains the CY7C330's features, the design methodology used in this design, and an example of how you can synthesize a complex function into this device. Note, however, that this is not meant to be an in-depth tutorial of the FDDI standard and its various layers.

FDDI Overview

FDDI is a 100-Mbits/s dual token ring network that can connect as many as 500 nodes with a maximum link-to-link distance of 2 km and a total network circumference of about 100 km. The network employs a primary and a secondary ring. The primary ring handles data transmission, and the secondary ring mainly provides fault tolerance, but can be used for data transmission as well.

FDDI is a token ring network, in which rotating a token grants network access. The node with the token can transmit data. This arrangement ensures a deterministic, collision-free network, independent of the number of stations in the network.

Because of the dual-ring topology, FDDI defines a fault-recovery mechanism. If a fault is detected, such as a broken fiber-optic cable, the network can be restored by routing around the break with the second ring. This function is largely controlled by the state machine shown later, which is implemented with the CY7C330.

The ANSI X3T9.5 standards committee controls the FDDI standard, which was developed using the Open Systems Interconnection (OSI) model. FDDI implements the model's physical and data-link layers. The four FDDI layers are Physical Media Dependent (PMD), Physical (PHY), Media Access Control (MAC), and Station Management (SMT).

The state machine example described later in this application note was developed with the December 2, 1988 update of the SMT specification. The final FDDI specification might differ slightly, but the design methodology remains the same.

The PMD layer is the lowest and specifies the network's connectors, transceivers, and bypass switches. The PHY layer specifies the type of encoding used on the data (4B/5B) and specifies a set of line states. These line states implement a handshake mechanism between PHYs of adjacent nodes. The MAC layer performs higher-level, peer-to-peer communications. It also provides for system timer support, packet framing, and responses to various types of errors in the network. The SMT layer controls the activities of the MAC, PHY, and PMD. SMT includes functions such as connection management (CMT), fault detection, and ring reconfiguration.

The CMT is the portion of Station Management that controls the insertion, removal and logical connection of the PHY entities. Within the CMT is an area known as the Physical Connection Management (PCM). A chart showing a hierarchical view of the location of

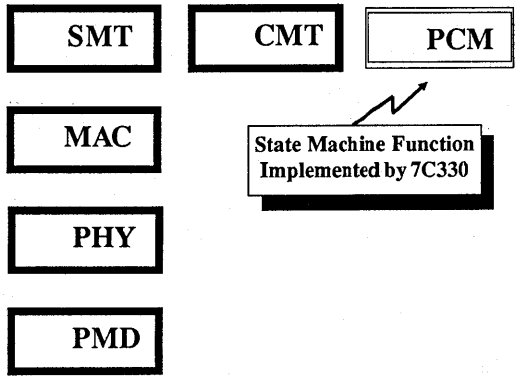


Figure 1. FDDI Hierarchy

the PCM appears in *Figure 1*. The PCM provides the signals to perform the following functions:

- Initialize a connection
- Reject a marginal connection
- Support maintenance

Figure 2 shows the synthesized state machine that performs these activities. This state machine is based on version 9.1 of the PCM state machine described in the SMT specification.

To keep within the CY7C330's 25 I/O constraint, a small amount of logic is implemented outside the CY7C330. For instance, the PCM uses two timers. The CY7C330 does not include these timers, but two decoded signals (timer1 and timer2) indicate that the timer has reached specific values. The timer1 and timer2 signals are inputs to the CY7C330. The chart in *Figure 3* shows all the macrofunctions, how they are decoded, and their functions.

Introduction to the CY7C330

The CY7C330 is a synchronous, 28-pin PLD. It is packaged in a 300-mil DIP as well as several types of surface mount packages, including a leadless ceramic chip carrier (LCC) and a plastic leaded chip carrier (PLCC). The device is fabricated with the Cypress 0.8-micron CMOS process and is available in speeds of 33, 50, and 66 MHz. The CY7C330 is also available as a military device in speeds of 33, 40, and 50 MHz. The device is optimized to implement high-speed state machine designs.

The CY7C330's features can be generalized into four groups:

1. Dedicated input cell
2. Product term array
3. I/O macrocell
4. Hidden state-register macrocell

The CY7C330 contains 11 of the dedicated input macrocells. This cell (*Figure 4*) contains a D flip-flop and a programmable multiplexer (mux) that allows a choice of two input clocks. The two input clocks are CK1 and CK2, which come directly from pins 2 and 3 of the device, respectively. Note that you cannot bypass any of the CY7C330's registers. The device is purely synchronous in nature.

As with any PLD, the CY7C330's product term array (see the CY7C330 block diagram in *Figure 1* of "Understanding the CY7C330") synthesizes the logical connections of the design. The product terms control a

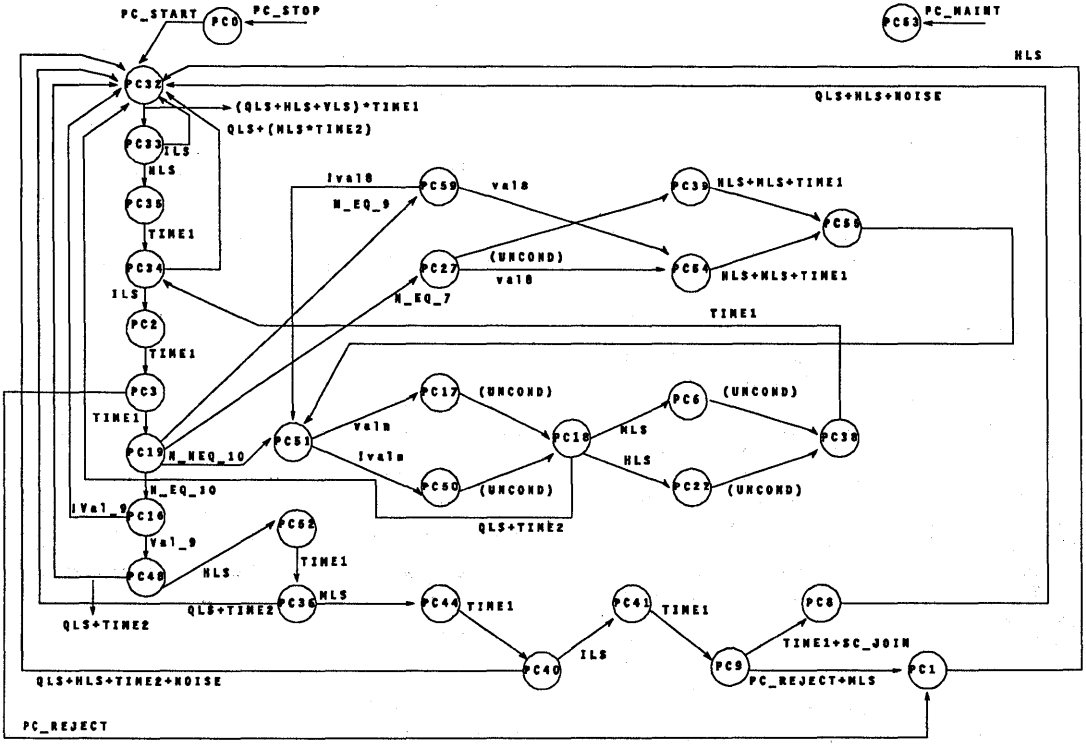


Figure 2. PCM State Machine

MACRO NAME	SYNTHESIZED SIGNAL	FUNCTION
MLS	!MLS	Master Line State
ILS	!ILS	Idle Line State
HLS	!HLS	Halt Line State
QLS	!QLS	Quiet Line State
pc_start	!pc0 & !pc1	State PCM State Machine
pc_reject	!pc0 & pc1	Enter Reject State
sc_join	pc0 & !pc1	Incorporate connection into token path
pcstop	!pc_stop	PCM state machine to enter OFF state
pcmaint	!pc_maint	Enter maintenance state
time1	!timer1	See timer explanations below.
time2	!timer2	See timer explanations below.
n_neq_10	!n0 & !n1	Counter indicating 10 bits of data have not been received or transmitted
n_eq_7	!n0 & n1	Counter indicating 7 bits have been transmitted or received
n_eq_9	n0 & !n1	Counter indicating 9 bits have been transmitted or received
n_eq_10	n0 & n1	Counter indicating 10 bits have been transmitted or received
noise	!noise_count	Noise counter threshold
valn	Val_n	Transmitted value n
val8	!Val_8	Transmitted or Received value = 8
val9	!Val_9	Transmitted or Received value = 9
	TIMER VALUES	
Timer 1		
0 ms	TB_Min	Minimum break time for link.
0.2 ms	A_Max	Maximum time required to achieve signal aquisition.
480 ns	LS_Min	Length of time reception of ILS
15 us	LS_Max	Max time required for line state recognition
25 ms	I_Max	Max optical bypass insertion/deinsertion time
200 ms	T_next(9)	Default time for MAC loopback
Timer 2:		
100 ms	T_Out	Signalling Timeout

Figure 3. Macro Definitions

global reset, a global preset, an Exclusive-OR gate, the output enables, and the product terms that go to the D input of the flip-flops in the output macrocells. (Most of these features are covered later in the explanation of the macrocell.) The device offers product term distribution that varies between nine and 19, depending on which output macrocell is being addressed. The 19 product terms become the limiting factor in the complexity of the design.

The I/O macrocell (see *Figure1* in "Using ABEL to program the CY7C330") contains two D flip-flops. One of the D flip-flops clocks data from the array to either the output pin or back to the array and is intended to be a state register. The I/O macrocell has a different clock than the input registers, called CLK, which comes directly from pin 1. The other D flip-flop is an input register, which can clock data from the I/O pin into the array. This flip-flop can be clocked from CK1 or CK2, as with the dedicated input cell.

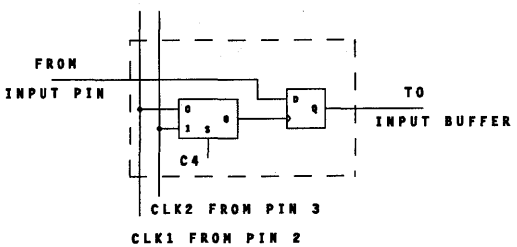


Figure 4. Input Macrocell

As mentioned earlier, the product term array feeds an XOR gate, which in turn feeds the D input of the state register. This gives you quite a bit of design flexibility. For example, you can use the XOR as an inverter by setting the XOR product term to a One. You can use the XOR to make the flip-flop a D, T, or JK type. Wrapping the Q output back to the XOR input changes the flip-flop from D to T, for instance. The design example described later uses this feature.

The output macrocell also allows you to choose the output-enable control for the pin. The output enable can come from a product term or directly from pin 14. The CY7C330 provides 12 I/O macrocells.

The hidden-state macrocell (*Figure 5*) contains a state register with no output pin associated with it. The CY7C330 contains four hidden-state macrocells. You can use these macrocells to synthesize a small 4-bit internal state machine or perform any function that is required only internally to the device itself.

The timing required for this design is 12.5 MHz, which allows use of the slowest CY7C330 version (33 MHz). The design requires one clock, although two pins are dedicated for clocks in the CY7C330. In this design, pins 1 and 2 are tied together externally, connecting the input-register and state-register clocks together. In the ABEL source code described below, the labels for the two clocks are CKS and CK1.

Design Methodology

The PCM design is implemented using the state machine syntax in ABEL version 3.0. The first-pass ABEL source code appears in *Appendix A*. Note that the state machine requires 31 states. This means that the state machine is implemented with 5 bits, which gives 32 total states and leaves one illegal state. When the design is run at reduction level 4 — the maximum reduction in ABEL — the software responds that the design requires more than 30 product terms per output. This is far more than the 19 product terms that are possible on any one output.

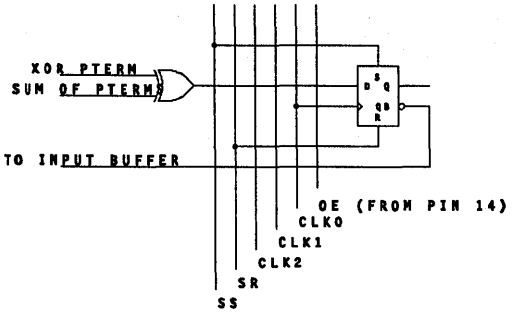


Figure 5. The CY7C330 Buried Register

Case 1.

	Decimal	Binary
	6	000110
	9	001001
		(4 bits toggle)

Case 2.

	6	000110
	7	000111
		(1 bit toggles)

Figure 6. State Change Comparison

At first glance, you might assume that the design is far too complex for the CY7C330. But further procedures make this implementation possible. To understand these procedures, it is necessary to understand some facts about ABEL.

ABEL reduces a design to a sum of products and does not make use of the XOR gate in the macrocell. To use the XOR gate, you must specify it in Boolean equation form and run the reduction at level 0. Specifying T flip-flops in version 3.0 also causes ABEL to reduce to a sum of products and not create T flip-flops using the XOR gate. ABEL 3.1 accepts T flip-flops, however, and corrects this situation.

Product Term Squeezing

The first method for reducing the number of product terms is to increase the number of bits in the state machine from 5 to 6 bits. Although the state machine only requires 31 states, a much broader range of choice results from having 64 possibilities for placing the states.

The next procedure involves changing from D flip-flops to T flip-flops. T flip-flops are more efficient because when the T input is High, the flip-flop toggles. Otherwise, the flip-flop retains its previous state.

Because a T flip-flop only needs one product term for a transition to occur, the state machine can be optimized by choosing state transitions that use a minimum number of bits. For example, a transition between states 6 and 9 requires more bits to change than a transition between states 6 and 7 (*Figure6*).

The 6-to-9 transition requires four product terms, while the 6-to-7 transition requires only one product term. Because the number of total states has been increased from 32 to 64 by adding one more bit to the state machine, you gain much more flexibility in choosing states. Carefully choosing the states in a state machine is the easiest way to reduce the number of product terms required.

Another way to make the design implementation more efficient is to use the CY7C330's synchronous global reset and preset to deal with illegal states. (Initially, the state machine is in state 0 because the CY7C330 has a power-on reset.) It is good design practice to make provisions for illegal states. Although an

illegal state should never occur, the state machine should be able to recover from such a state. Many times the recovery mechanism is built into the state machine itself, which requires more product terms.

If an illegal state is detected in this design, the state machine re-initializes itself and goes to state 0. Instead of building this requirement into the design, you can use a hidden register to detect the occurrence of illegal states. The signal from that register controls the CY7C330's synchronous reset, which returns the state machine to state 0. The CY7C330's synchronous nature causes the state machine to go to state 0 two clocks after the illegal state is encountered. One clock is required to detect the illegal state, and one clock is required to reset the device. This requirement is acceptable for this application.

In this design, it was noticed that the condition pcmaint was encountered in every state; the state machine was unconditionally required to go to this state. To reduce the state machine further, the state assigned to this condition is 63 (111111 binary). The synchronous preset is used to detect this signal. The assertion of pcmaint forces the state machine to state 63, thus avoiding the use of any product terms in the main body of the design.

This design requires several synchronous resets: an external pin (RST), the illegal state detect, and the signal pc_stop. Because only one product term is allowed for the device's synchronous reset, the other two resets must be developed by ANDing the reset signal with every product term associated with the outputs that are to be reset. This performs the same function as having multiple p terms for the synchronous reset but does not utilize any additional resources in the CY7C330.

Keep in mind that the CY7C330 has varied product term distribution. The state registers associated with pins 16 and 27 have 19 product terms. Put the state outputs that require the most product terms to these pins. In this example, Q0 requires 18 product terms, and Q5 requires 17. These outputs are assigned to pins 27 and 16. The remaining outputs are placed in the same manner.

Converting the state machine to Boolean equations is a straightforward procedure. By examining the state transitions, you can extract the Boolean equations. The reduced design is shown in *Figure7*.

```
State !S48:     if (HLS) then !S52
                else if (QLS # time2) then !S32
                else !S48;

                48 = 110000 (binary)
                52 = 110100
```

Q2 is the only bit that transitions

Therefore, a product term of:

```
Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & HLS
\_____________________________/
          state 48
```

would be added to the equation for Q2.

To continue the example:

```
48 = 110000
32 = 100000
```

Q4 is the only bit that transitions

Therefore, the product terms of:

```
      Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & QLS
# Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & time2
\_____________________________/
          state 48
```

would be added to the equation for Q4.

Figure 7. Boolean Equation Extraction Example

The Cypress PLD ToolKit is used as the development platform for the reduction process. The PLD ToolKit is a low-cost software development system for all Cypress PLDs. Although the reduced equations could have been obtained using ABEL, in many ways the PLD ToolKit is easier to use and more tailored to the Cypress devices. The PLD ToolKit source file appears in *Appendix B*. The PLD ToolKit also features a mouse-driven, interactive, simulator/waveform editor that makes design verification easy.

Appendix A. Orignal Abel Source Code

```
module pcm flag '-r3'
title 'Physical Connection Management (PCM) state Machine version 9.1
Steve Traum Cypress Semiconductor March 27, 1989'

                    U1      device 'P330';
"Inputs
                    CKS,Ck1,rst_            pin 1,2,3;
                    pc0,pc1                 pin 4,5;
                    timer1                  pin 6;
                    timer2                  pin 7;
                    mls,ils,hls,qls         pin 9,10,11,12;
                    Val_n                   pin 13;
                    n0,n1                   pin 14,15;
                    Val_8                   pin 16;
                    Val_9                   pin 17;
                    noise_count             pin 18;
                    pc_stop                 pin 19;
                    pc_maint                pin 20;
                    n1              istype  'feed_pin';
                    Val_8           istype  'feed_pin';
                    Val_9           istype  'feed_pin';
                    noise_count     istype  'feed_pin';

"Outputs

                    Reset                   node 29;
                    Q5,Q4,Q3,Q2,Q1,Q0               pin28,27,26,25,24,23;
                    Q5,Q4,Q3,Q2,Q1,Q0       istype  'pos,reg';
                    Qstate  =  [Q5,Q4,Q3,Q2,Q1,Q0];

"declarations
                    High,Low                = 1,0;
                    H,L,C,X,Z               = 1,0, .C.,.X.,.Z.;

"Qstate
S0   =  ^b000000;  S1  =  ^b000001;  S2  =  ^b000010;  S3 =  ^ b000011;  S4 =  ^ b000100;
S5   =  ^b000101;  S6  =  ^b000110;  S7  =  ^b000111;  S8 =  ^b001000;  S9 =  ^b001001;
S10  =  ^b001010;  S11 =  ^b001011;  S12 =  ^b001100;  S13 =  ^b001101;S14 =  ^b001110;
S15 =  ^b001111;  S16 =  ^b010000;  S17 =  ^b010001;  S18 =  ^b010010;S19 =  ^b010011;
S20 =  ^b010100;  S21 =  ^b010101;  S22 =  ^b010110;  S23 =  ^b010111;S24 =  ^b011000;
S25 =  ^b011001;  S26 =  ^b011010;  S27 =  ^b011011;  S28 =  ^b011100;S29 =  ^b011101;
S30 =  ^b011110;  S31 =  ^b011111;  S32 =  ^b100000;  S33 =  ^b100001;S34 =  ^b100010;
S35 =  ^b100011;  S36 =  ^b100100;  S37 =  ^b100101;  S38 =  ^b100110;S39 =  ^b100111;
S40 =  ^b101000;  S41 =  ^b101001;  S42 =  ^b101010;  S43 =  ^b101011;S44 =  ^b101100;
S45 =  ^b101101;  S46 =  ^b101110;  S47 =  ^b101111;  S48 =  ^b110000;S49 =  ^b110001;
S50 =  ^b110010;  S51 =  ^b110011;  S52 =  ^b110100;  S53 =  ^b110101;S54 =  ^b110110;
S55 =  ^b110111;  S56 =  ^b111000;  S57 =  ^b111001;  S58 =  ^b111010;S59 =  ^b111011;
S60 =  ^b111100;  S61 =  ^b111101;  S62 =  ^b111110;  S63 =  ^b111111;

MLS MACRO {(!mls)};
ILS MACRO {(!ils)};
HLS MACRO {(!hls)};
QLS MACRO {(!qls)};
```

Appendix A. Original Abel Source Code (continued)

```
pc_start MACRO {(!pc0 & !pc1)};
pc_reject MACRO {(!pc0 & pc1)};
sc_join MACRO {(pc0 & !pc1)};
pcstop MACRO {(!pc_stop)};
pcmaint MACRO {(!pc_maint)};
time1 MACRO {(!timer1)};
time2 MACRO {(!timer2)};
n_neq_10 MACRO {(!n0 & !n1)};
n_eq_7 MACRO {(!n0 & n1)};
n_eq_9 MACRO {(n0 & !n1)};
n_eq_10 MACRO {(n0 & n1)};
noise MACRO {(!noise_count)};
valn MACRO {(Val_n)};
val8 MACRO {(!Val_8 )};
val9 MACRO {(!Val_9)};

state_diagram  Qstate
        state !S0:
                if ( pc_start) then !S32
                else if ( pcmaint ) then !S31
                else !S0;
        state !S1:
                if (HLS) then !S32
                else if ( pcstop ) then !S0
                else if ( pcmaint ) then !S63
                else !S1;
        state !S2:
                if (time1) then !S3
                else !S2;
        state !S3:
                if (time1) then !S19
                else if ( pc_reject ) then !S1
                else !S3;
        state !S63:
                if ( pc_stop ) then !S0
                else !S63;
        state !S6:
                goto !S38;
        state !S8:
                if ( QLS # HLS # noise ) then !S32
                else if ( pc_stop ) then !S0
                else if (pc_maint) then !S63
                else if (pc_start) then !S32
                else !S8;
        state !S9:
                if ( sc_join&time1 ) then !S8
                else if ( pc_reject # MLS ) then !S1
                else !S9;
        state !S16:
                if ( Val_9 ) then !S48
                else !S32;
        state !S17:
                goto !S18;
```

Appendix A. Original Abel Source Code (continued)

```
state !S18:
        if ( QLS # time2 ) then !S32
        else if ( MLS ) then !S6
        else if (HLS) then !S22
        else !S18;
state !S19:
        if (n_neq_10 ) then !S51
        else if (n_eq_7) then !S27
        else if (n_eq_9) then !S59
        else if (n_eq_10) then !S16
        else !S19;
state !S22:
        goto !S38;
state !S27:
        if (val8 == High ) then !S54
        else !S39;
state !S39:     if ( HLS # MLS # time1 ) then !S55
        else !S39;
state !S32:     if (( QLS # HLS # MLS) & time1 ) then !S33
        else if ( pc_stop ) then !S0
        else if ( pc_maint) then !S63
        else !S32;
state !S33:
        if ( HLS ) then !S35
        else if (ILS) then !S32
        else !S33;
state !S34:
        if ( ILS ) then !S2
        else if ( QLS # (MLS & time2)) then !S32
        else !S34;
 state !S35:
        if ( time1 ) then !S34
        else !S35;
state !S36:
        if ( MLS ) then !S44
        else if ( QLS # time2 ) then !S32
        else if (pc_stop) then !S0
        else if (pc_maint) then !S63
        else !S36;
state !S38:
        if (time1) then !S34
        else !S38;
state !S40:
        if ( ILS ) then !S41
        else if ( QLS # HLS # time2 # noise ) then !S32
        else !S40;
state !S41:
        if ( time1 ) then !S9
        else !S41;
state !S44:
        if ( time1 ) then !S40
        else !S44;
```

Appendix A. Original Abel Source Code (continued)

```
        state !S48:
                if (HLS) then !S52
                else if (QLS # time2) then !S32
                else !S48;
        state !S50:
                goto !S18;
        state !S51:
                if ( valn == High ) then !S17
                else !S50;
        state !S52:
                if ( time1 ) then !S36
                else !S52;
        state !S55:
                goto !S51;
        state !S59:
                if (val8) then !S54
                else !S51;
        state !S54:
                if ( HLS # MLS # time1 ) then !S55;
                else !S54;
        state !4:       goto !S0;
        state !5:       goto !S0;
        state !7:       goto !S0;
        state !10:      goto !S0;
        state !11:      goto !S0;
        state !12:      goto !S0;
        state !13:      goto !S0;
        state !14:      goto !S0;
        state !15:      goto !S0;
        state !20:      goto !S0;
        state !21:      goto !S0;
        state !23:      goto !S0;
        state !24:      goto !S0;
        state !25:      goto !S0;
        state !26:      goto !S0;
        state !28:      goto !S0;
        state !29:      goto !S0;
        state !30:      goto !S0;
        state !31:      goto !S0;
        state !37:      goto !S0;
        state !42:      goto !S0;
        state !43:      goto !S0;
        state !45:      goto !S0;
        state !46:      goto !S0;
        state !47:      goto !S0;
        state !49:      goto !S0;
        state !53:      goto !S0;
        state !56:      goto !S0;
        state !57:      goto !S0;
        state !58:      goto !S0;
        state !60:      goto !S0;
        state !61:      goto !S0;
        state !62:      goto !S0;
equations
        Reset = !rst_;
end pcm
```

Appendix B. Cypress PLD ToolKit Source File

```
CY7C330;
 {This file is the Cypress ToolKit Source Code for FDDI Design }

CONFIGURE;

CKS,Ck1,RST_,
pc0, pc1, timer1, timer2, MLS (node= 9), ILS, HLS, QLS,
Val_n, n0, n1(iop,ireg), !Q0, Val_8(iop,ireg), !Q1, Val_9(iop,ireg), !Q2,
!Q3 (node=23), noise_count(iop,ireg), !Q4, pc_stop(iop,ireg), !Q5,
pc_maint(iop,ireg), RST, SET, ILSTATE (node= 34),

{*************************************************************************}

EQUATIONS;

RST = RST_;

SET = !pc_maint;

ILSTATE =    # Q2 & !Q1 & !Q4 & !Q5 & pc_stop
             # Q2 & Q0 & !Q4 & !Q5 & pc_stop
             # Q1 & Q3 & !Q4 & !Q5 & pc_stop
             # !Q1 & Q2 & Q4 & !Q5 & pc_stop
             # Q0 & Q2 & Q4 & !Q5 & pc_stop
             # Q3 & !Q1 & Q4 & !Q5 & pc_stop
             # Q3 & !Q0 & Q4 & !Q5 & pc_stop
             # Q3 & Q1 & !Q4 & Q5 & pc_stop
             # Q0 & !Q1 & Q2 & !Q4 & Q5 & pc_stop
             # !Q1 & Q3 & Q4 & Q5 & pc_stop
             # !Q1 & Q0 & Q4 & Q5 & pc_stop
             # Q3 & !Q0 & Q4 & Q5 & pc_stop;


Q0 :=        < oe>
             <xsum> Q0 & !ILSTATE & pc_stop
             # !Q5 & !Q4 & !Q3 & !Q2 & !Q1 & Q0 & !HLS & !ILSTATE & pc_stop
             # !Q5 & !Q4 & !Q3 & !Q2 & Q1 & !Q0 & !timer1 & !ILSTATE & pc_stop
             # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & Q0 & pc0 & !pc1 & !timer1 & !ILSTATE & pc_stop
             # !Q5 & Q4 & !Q3 & !Q2 & !Q1 & Q0 & !ILSTATE & pc_stop
             # !Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & n0 & n1 & !ILSTATE & pc_stop
             # Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & !Val_8 & !ILSTATE & pc_stop
             # Q5 & Q4 & !Q3 & Q2 & Q1 & !Q0 & !HLS & !ILSTATE & pc_stop
             # Q5 & Q4 & !Q3 & Q2 & Q1 & !Q0 & !MLS & !ILSTATE & pc_stop
             # Q5 & Q4 & !Q3 & Q2 & Q1 & !Q0 & !timer1 & !ILSTATE & pc_stop
             # !Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & Val_n & !ILSTATE & pc_stop
             # Q5 & !Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !QLS & !timer1 & !ILSTATE & pc_stop
             # Q5 & !Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !HLS & !timer1 & !ILSTATE & pc_stop
             # Q5 & !Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !MLS & !timer1 & !ILSTATE & pc_stop
             # Q5 & !Q4 & !Q3 & !Q2 & !Q1 & Q0 & !ILS & !ILSTATE & pc_stop
             # Q5 & !Q4 & !Q3 & !Q2 & Q1 & Q0 & !timer1 & !ILSTATE & pc_stop
             # Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !ILS & !ILSTATE & pc_stop
             # Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & !Val_n & !ILSTATE & pc_stop
             # Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & !pc0 & !pc1 & !ILSTATE & pc_stop;
```

Appendix B. Cypress PLD ToolKit Source File (continued)

```
Q1  :=      < oe>
            <xsum> Q1 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & !Q3 & !Q2 & Q1 & Q0 & !pc0 & pc1 & !ILSTATE & pc_stop
            # Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & !pc0 & !pc1 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & !Q1 & Q0 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !timer2 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & n0 & n1 & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & !Q2 & !Q1 & Q0 & !HLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & !Q2 & Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & !Q2 & Q1 & !Q0 & !timer2 & !MLS & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & Val_n & !ILSTATE & pc_stop;

Q2  :=      < oe>
            <xsum> Q2 & !ILSTATE & pc_stop
            # Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & !pc0 & !pc1 & !ILSTATE & pc_stop
            #!Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !HLS & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !MLS & !ILSTATE & pc_stop
            # Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & !Val_8 & !ILSTATE & pc_stop
            # !Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & !Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & !Q1 & !Q0 & !timer2 & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & Q1 & !Q0 & !timer1 & !ILSTATE & pc_stop
            # Q5 & !Q4 & Q3 & Q2 & !Q1 & !Q0 & !timer1 & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !HLS & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & Q2 & Q1 & Q0 & !ILSTATE & pc_stop;

Q3  :=      < oe>
            <xsum> Q3 & !ILSTATE & pc_stop
            # Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & !pc0 & !pc1 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !HLS & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !noise_count & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & Q0 & !pc0 & pc1 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & Q0 & !MLS & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & !n0 & n1 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & n0 & !n1 & !ILSTATE & pc_stop
            # Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & !Q1 & !Q0 & !MLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !HLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !timer2 & !ILSTATE & pc_stop
            # Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !noise_count & !ILSTATE & pc_stop;
```

Appendix B. Cypress PLD ToolKit Source File (continued)

```
Q4  :=      < oe>
            <xsum> Q4 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & !Q3 & !Q2 & Q1 & Q0 & !timer1 & !ILSTATE & pc_stop
            # Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & !pc0 & !pc1 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & Val_9 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !timer2 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !MLS & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & Q2 & Q1 & !Q0 & !ILSTATE & pc_stop
            # !Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & !Val_n & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & Q1 & Q0 & !HLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & Q1 & Q0 & !MLS & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & Q2 & Q1 & Q0 & !timer1 & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !timer2 & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & Q2 & !Q1 & !Q0 & !timer1 & !ILSTATE & pc_stop;

Q5  :=      < oe>
            <xsum> Q5 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !pc0 & !pc1 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & !Q3 & !Q2 & !Q1 & Q0 & !HLS & !ILSTATE & pc_stop
            # !Q5 & !Q4 & !Q3 & Q2 & Q1 & !Q0 & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !HLS & !ILSTATE & pc_stop
            # !Q5 & !Q4 & Q3 & !Q2 & !Q1 & !Q0 & !noise_count & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & !Q1 & !Q0 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !QLS & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !timer2 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & !n0 & !n1 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & n0 & !n1 & !ILSTATE & pc_stop
            # !Q5 & Q4 & !Q3 & Q2 & Q1 & !Q0 & !ILSTATE & pc_stop
            # !Q5 & Q4 & Q3 & !Q2 & Q1 & Q0 & !ILSTATE & pc_stop
            # Q5 & !Q4 & !Q3 & !Q2 & Q1 & !Q0 & !ILS & !ILSTATE & pc_stop
            # Q5 & !Q4 & Q3 & !Q2 & !Q1 & Q0 & !timer1 & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & !Q2 & Q1 & !Q0 & !ILSTATE & pc_stop
            # Q5 & Q4 & !Q3 & !Q2 & Q1 & Q0 & Val_n & !ILSTATE & pc_stop;


{end of file}
```

CYPRESS SEMICONDUCTOR

Bus-Oriented Maskable Interrupt Controller

This application note illustrates the design flexibility of Cypress's CY7C331 PLD by describing a single-chip interrupt controller based on the PLD.

Virtually all microprocessor designs require some type of interrupt support. Complex applications can take advantage of a dedicated interrupt controller chip from the microprocessor family. But for simple applications or where special requirements exist, a standard interrupt controller can prove inadequate or represent overkill for the design.

In such cases, you generally implement a custom-designed controller using some combination of MSI logic and PLDs. The single-chip design described here is implemented in two stages: The first stage comprises a simple 4-channel controller, which includes the major functional blocks. In the second stage, another controller is cascaded from the stage-1 design to provide support for up to eight interrupt channels.

The interrupt controller's design features include:

1. Programable-polarity, level-sensitive inputs
2. Interlocked REQ/ACK handshake
3. Simple MPU bus attachment for read and write
4. Masking of individual channels
5. Prioritized interrupt vector
6. Fully asynchronous operation

Design Description

The interrupt controller attaches to the MPU data bus and is controlled by the system processor through read and write ports on the data bus. The read port provides interrupt status and a prioritized vector for the processor, and the write port allows the processor to selectively mask individual interrupt channels. The controller provides a separate interrupt request line to the processor to signal a pending interrupt. Figure 1 shows the bit assignments for the read and write ports. In Figure 2. you can see the interrupt controller's major functional blocks.

CY7C331 Description

The device used to implement the interrupt controller is the CY7C331, an asynchronous PLD packaged in a 28-pin, 300-mil DIP. The device features 12 I/O macrocells and 13 dedicated inputs. The I/O macrocell has a separate input and output flip-flop, which is highly

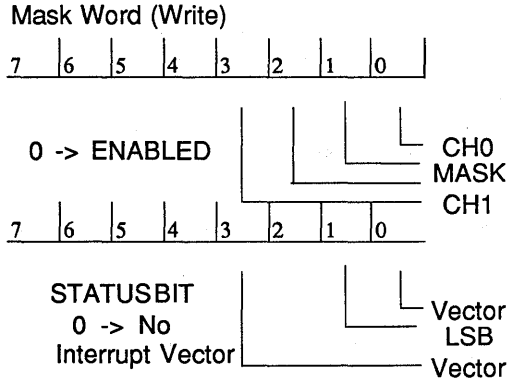


Figure 1. Data Bus Bit Assignments

useful in bus-oriented applications.

Each flip-flop has a separate product term for the clock, set, and reset. The output flip-flop's D input incorporates an XOR with the sum-of-products array. This allows you to select polarity or implement a toggle or JK flip-flop.

The macrocell flip-flops also offer a unique transparency feature: When the set and reset inputs are both asserted, the flip-flop's Q output follows its D input. Thus, you can use the flip-flop as a clocked register with independent clock, set and reset inputs or as a combinational path.

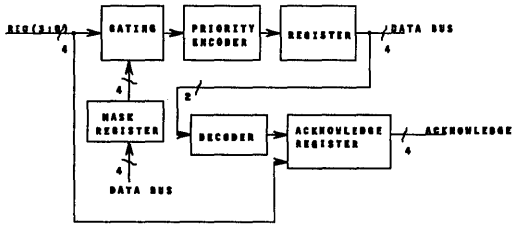


Figure 2. Interrupt Controller Block Diagram

Additionally, the CY7C331 includes six shared input multiplexers, which allow you to bury up to six output flip-flops without giving up any pins. Figure 3 shows a block diagram of the CY7C331. A diagram of the device's I/O macrocell appears in Figure 1 of the application note "Using the CY7C331 as a Waveform Generator."

Four-Channel Interrupt Controller

The interrupt controller's operation is quite simple. On reset, all interrupt channels are masked off, and no interrupts are permitted. The processor then loads the mask register with the desired interrupt-channel mask bits cleared. If the channel is not masked when an interrupt request occurs, the request is prioritized, and the controller asserts the Interrupt Request (IRQ) to the processor.

The processor responds to the IRQ by reading the interrupt vector port. When the interrupt controller detects the read, the controller latches the current interrupt priority and places the priority vector on the data bus. Latching the current priority while the vector is being read prevents the vector from being altered during the read cycle. In addition, the controller decodes the vector and asserts the corresponding channel's acknowledge line.

The acknowledge remains asserted until detected by the interrupting element, which responds by deasserting its interrupt request. This interlocking handshake ensures that a pending interrupt is not lost or responded to more than once. The controller also uses the acknowledge internally to disable the interrupt request into the priority encoder; this is done in the time between the interrupt acknowledge and the interrupt request being deasserted. A simple example of the timing sequence for a single interrupting channel appears in Figure 4.

Figure 5 defines the pin assignments for the first-stage interrupt controller.

Data Bus Interface

The data bus interface requires bidirectional operation. When CS and WE are asserted Low, the controller writes data into the mask register. When CS is asserted

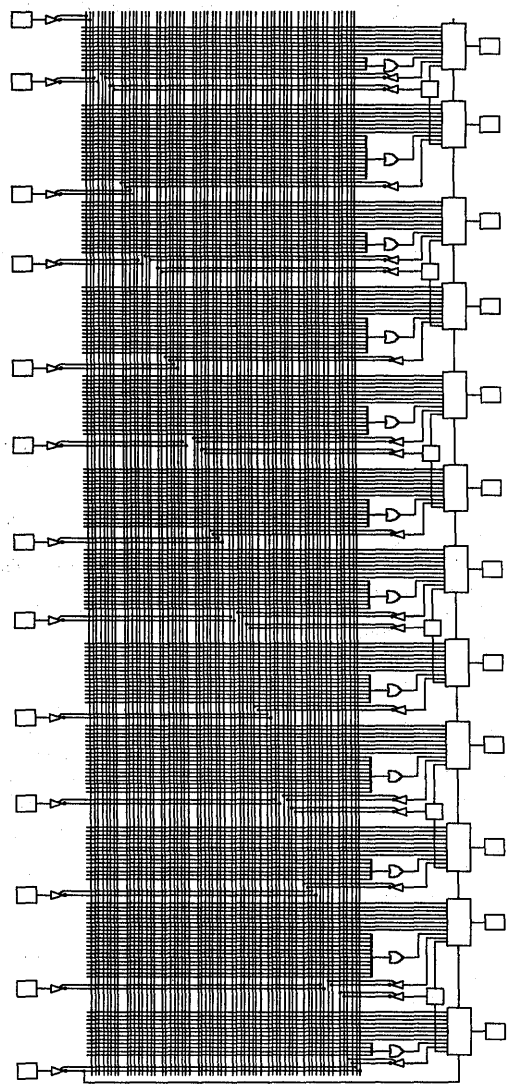


Figure 3. CY7C331 Block Diagram

Low and WE remains High, the controller holds the current priority vector and interrupt status and places them on the data bus. The CY7C331's I/O macrocell is readily adapted to this requirement, as illustrated in Figure 6.

The INTERRUPT status generation requires a different implementation. If any interrupt requests are pending when the controller detects a read cycle (CS

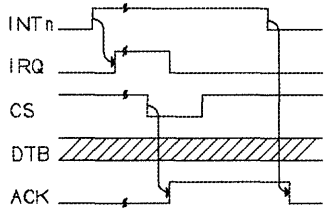


Figure 4. Timing Sequence for Single Interrupt Channel

Low, WE High), the interrupt status bit must be asserted High. Moreover, new interrupt requests are held off until the end of the read cycle. This requires a clocked implementation of the interrupt status bit on the data bus, as shown in Figure 7.

Acknowledge Generation

Acknowledge generation requires that the controller decode the priority vector placed on the data bus and assert the corresponding acknowledge line until the interrupt request line is deasserted. The controller must handle the timing carefully for correct operation. Specifically, because a valid priority vector is not available until after CS is asserted Low, the controller cannot decode the correct channel until the priority vector register has settled. Thus, a delay is required before the controller can generate an acknowledge.

The controller can generate a delay by taking advantage of the following sequence: If there is a pending interrupt request, the interrupt status bit is always asserted one propagation delay after CS is asserted on a read cycle. The interrupt status signal is then passed through an internal strobe stage, which causes an additional propagation delay. The internal strobe then initiates the acknowledge-generation sequence.

The delayed strobe assures that the priority vector value has settled and the setup requirements for decoding have been met. An SR flip-flop implements the acknowledge-generation function for each channel. The flip-flop is set when a read cycle occurs, the priority vector corresponds to the channel, and the delayed internal strobe occurs. The flip-flop is reset when the interrupt request for the channel is deasserted.

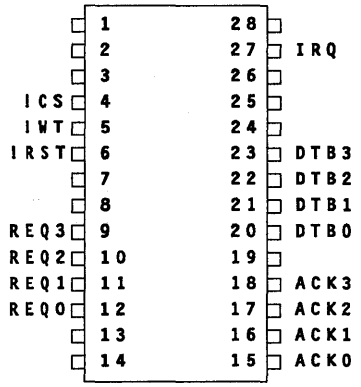


Figure 5. Interrupt Controller Pin Assignments

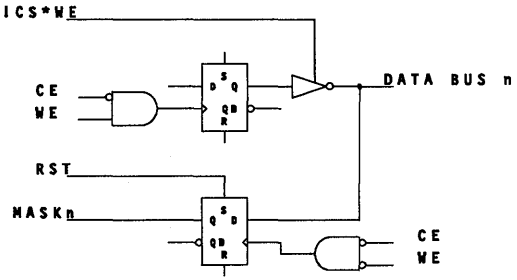


Figure 6. Mask/Priority Vector Function

A logic diagram for the internal strobe generation and a single acknowledge-generation block appears in Figure 8. The timing diagram in Figure 9 illustrates a typical operation.

Logic Equations

The Cypress PLD ToolKit assembles the Boolean equations for the interrupt controller (Appendix A). The equations are heavily commented for clarity. Because the PLD ToolKit does not currently support "DeMorganization," and because the CY7C331 contains inverting output buffers, the Boolean equations for output flip-flops are written for negative logic (i.e., solving for zero). In addition, the inversion requires swapping of the SET and RESET functions on the output flip-flops. Thus, the logical Boolean equation required to set the flip-flop must be implemented on the flip-flop's reset input. Similarly, the equation required to reset the flip-flop must be implemented on the flip-flop's set input.

Adding Cascade Capability

You can readily extend the interrupt controller design to accommodate four additional channels by in-

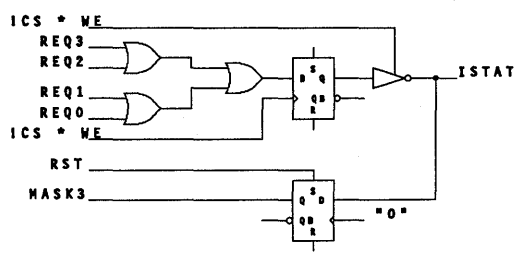


Figure 7. Interrupt Status Generation

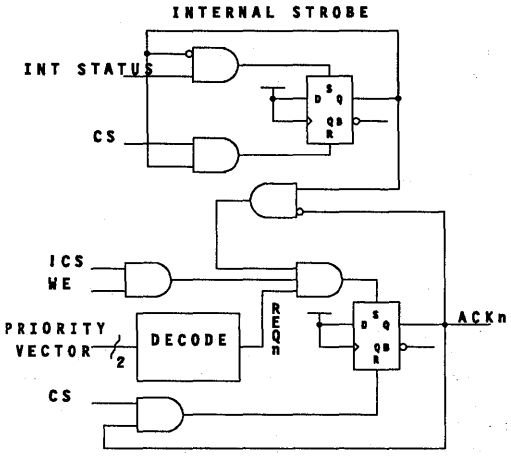


Figure 8. Internal Strobe/Acknowledge Generation

corporating a cascade mechanism. You can then attach a second interrupt controller to to the first (Figure 10). The additional channels require an extension to the formats of the mask register and the interrupt vector (Figure 11).

The lower interrupt controller supports the lower-priority interrupt channels, generates the IRQ to the processor, and places the interrupt status and priority vector on the data bus during a read cycle. The upper interrupt controller supports the higher-priority channels and passes its current status and priority vector down to the lower interrupt controller.

The interrupt status line is asserted High when the upper interrupt controller has a non-masked interrupt request pending. To permit the host processor to write into the upper interrupt controller's mask register, the controller monitors the data bus's upper four bits. Because the upper interrupt controller passes its priority vector directly to the lower interrupt controller, however, the upper interrupt controller does not need to output any data on the bus during a read cycle.

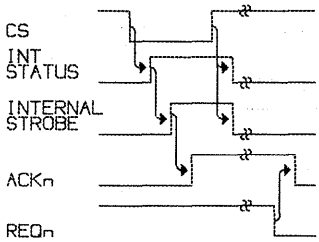


Figure 9. Timing Diagram

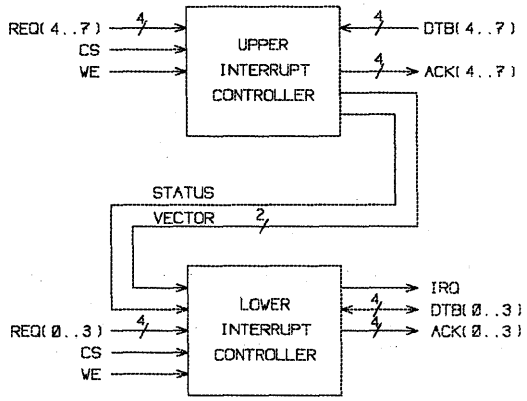


Figure 10. Cascading Interrupt Controllers

In operation, the lower interrupt controller must monitor the status interrupt line from the upper controller. The lower controller incorporates the interrupt into the IRQ to the host processor and into the interrupt vector placed on the data bus during a read cycle.

Modifying the interrupt vector is straightforward. Because the upper interrupt channels have higher priority, when the interrupt status from the upper controller is asserted, the interrupt vector's lower two bits are the two vector bits from the upper controller. When the status is not asserted, the interrupt vector's lower

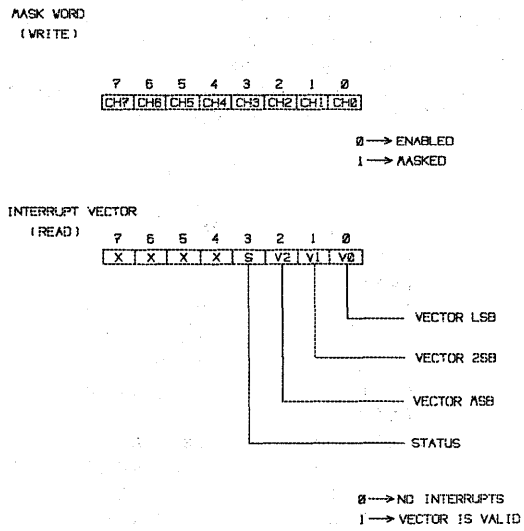


Figure 11. Extended Interrupt Vector

two bits are the lower priority interrupt vector encoded from the lower interrupt controller. The interrupt vector's third bit is simply the state of the interrupt status signal from the upper controller. The modified interrupt controller equations for the lower element appear in Appendix B. and the upper element equations in Appendix C.

Summary

The interrupt controller described in the application note can serve as the basis for flexible low-to-moderate-complexity interrupt controllers. You can extend the design as required for different request polarity levels, edge-sensitive inputs, or additional channels.

Simulations of the interrupt controller show that the design works as expected. You can obtain the PLD source files for the design from your local Cypress sales office.

Appendix A. PLD ToolKit Source Code
Stand Alone Interrupt Controller

```
{Stand Alone Interrupt Controller}
CY7C331;  {declare device type}

CONFIGURE;

CS(node =  4),              {pin 4, chip select}
WE(node =  5),              {pin 5, write enable}
RST(node =  6),             {pin 6, reset}
REQ3(node =  9),            {pin 9, interrupt request channel 3}
REQ2(node =  10),           {pin 10, interrupt request channel 2}
REQ1(node =  11),           {pin 11, interrupt request channel 1}
REQ0(node =  12),           {pin 12, interrupt request channel 0}

!IRQ(node =  27),           {pin 27, interrupt to processor}
ISTAT(node =  28),          {pin 28, data bus 3 - interrupt status}
PVEC2(node =  26),          {pin 26, data bus 2 - priority vector bit 2}
PVEC1(node =  24),          {pin 24, data bus 1 - priority vector bit 1}
PVEC0(node =  20),          {pin 20, data bus 0 - priority vector bit 0}
ACK3(node =  18),           {pin 18, acknowledge channel 3}
ACK2(node =  17),           {pin 17, acknowledge channel 2}
ACK1(node =  16),           {pin 16, acknowledge channel 1}
ACK0(node =  15),           {pin 15, acknowledge channel 0}
MSK3(node =  34,SRC =  28),   {shared input mux for pin 28}
MSK2(node =  33,SRC =  26),   {shared input mux for pin 26}
MSK1(node =  32,SRC =  24),   {shared input mux for pin 24}
MSK0(node =  31,SRC =  20),   {shared input mux for pin 20}

ISTB(node =  25),           {pin 25, internal strobe}

EQUATIONS;
 IRQ =  < oe>
     < set_out>  {make FF transparent}
     < clr_out>  {make FF transparent}
     < xsum>   {force invert}
     < sum>  REQ3 & !ACK3 & !MSK3
       # REQ2 & !ACK2 & !MSK2
       # REQ1 & !ACK1 & !MSK1
       # REQ0 & !ACK0 & !MSK0;

 !ISTAT =  < oe> !CS & WE
       < xsum>   {force invert}
       < set_out> CS & ISTAT {FF output is reset }
       < ck_out> !CS & WE
       < set_in> !RST  {interrupt is masked on reset}
       < ck_in> !WE & !CS
     < sum>  REQ3 & !ACK3 & !MSK3
       # REQ2 & !ACK2 & !MSK2
       # REQ1 & !ACK1 & !MSK1
       # REQ0 & !ACK0 & !MSK0;

 !PVEC2 =  < oe> !CS & WE
       < set_out>  {always zero}
       < set_in> !RST  {interrupt is masked on reset}
       < ck_in> !WE & !CS;
```

Appendix A. PLD ToolKit Source Code
Stand Alone Interrupt Controller (continued)

```
!PVEC1 =  < oe> !CS & WE
      < xsum>  {force invert}
      < ck_out> !CS & WE
      < sum>  !ACK3 & REQ3 & !MSK3
        #  !ACK2 & REQ2 & !MSK2
      < set_in> !RST  {interrupt is masked on reset}
      < ck_in> !WE & !CS;

!PVEC0 =  < oe> !CS & WE
      < xsum>  {force invert}
      < ck_out> !CS & WE
      < sum>  !ACK3 & REQ3 & !MSK3
        # !ACK1 & REQ1 & !MSK1 & MSK2 #  !MSK1 & !ACK1 & REQ1 & !REQ2
      < set_in> !RST  {interrupt is masked on reset}
      < ck_in> !WE & !CS;

!ACK3 =  < oe>
     < clr_out> !CS & WE & PVEC1 & PVEC0 & ISTB & !ACK3 {FF output is set }
     < set_out> CS & ACK3 & !REQ3; {FF output is reset }

!ACK2 =  < oe>
     < clr_out> !CS & WE & PVEC1 & !PVEC0 & ISTB & !ACK2 {FF output is set}
     < set_out> CS & ACK2 & !REQ2; {FF output is reset }

!ACK1 =  < oe>
     < clr_out> !CS & WE & !PVEC1 & PVEC0 & ISTB & !ACK1 {FF output is set}
     < set_out> CS & ACK1 & !REQ1; {FF output is reset }

!ACK0 =  < oe>
     < clr_out> !CS & WE & !PVEC1 & !PVEC0 & ISTB & !ACK0 {FF output is set}
     < set_out> CS & ACK0 & !REQ0; {FF output is reset }

!ISTB =  < oe>
     < clr_out> ISTAT & !ISTB {FF output is set }
     < set_out> CS & ISTB; {FF output is reset }
```

Appendix B. PLD ToolKit Source Code
Cascadable Interrupt Controller-Lower Element

```
{Cascaded Interrupt Controller - Lower Element}
CY7C331;  {declare device type}

CONFIGURE;
USTAT(node =  1),          {pin 1, upper element interrupt status}
RVEC1(node =  2),          {pin 2, ripple vector bit 1 from upper element}
RVEC0(node =  3),          {pin 3, ripple vector bit 0 from upper element}
CS(node =  4),             {pin 4, chip select}
WE(node =  5),             {pin 5, write enable}
RST(node =  6),            {pin 6, reset}
REQ3(node =  9),           {pin 9, interrupt request channel 3}
REQ2(node =  10),          {pin 10, interrupt request channel 2}
REQ1(node =  11),          {pin 11, interrupt request channel 1}
REQ0(node =  12),          {pin 12, interrupt request channel 0}

!IRQ(node =  27),          {pin 27, interrupt to processor}
ISTAT(node =  28),         {pin 28, data bus 3 - interrupt status}
PVEC2(node =  26),         {pin 26, data bus 2 - priority vector bit 2}
PVEC1(node =  24),         {pin 24, data bus 1 - priority vector bit 1}
PVEC0(node =  20),         {pin 20, data bus 0 - priority vector bit 0}
ACK3(node =  18),          {pin 18, acknowledge channel 3}
ACK2(node =  17),          {pin 17, acknowledge channel 2}
ACK1(node =  16),          {pin 16, acknowledge channel 1}
ACK0(node =  15),          {pin 15, acknowledge channel 0}
MSK3(node =  34,SRC =  28),   {shared input mux for pin 28}
MSK2(node =  33,SRC =  26),   {shared input mux for pin 26}
MSK1(node =  32,SRC =  24),   {shared input mux for pin 24}
MSK0(node =  31,SRC =  20),   {shared input mux for pin 20}

ISTB(node =  25),          {pin 25, internal strobe}

EQUATIONS;
 IRQ =  < oe>
     < set_out>  {make FF transparent}
     < clr_out>  {make FF transparent}
     < xsum>   {force invert}
     < sum>  REQ3 & !ACK3 & !MSK3
        # REQ2 & !ACK2 & !MSK2
        # REQ1 & !ACK1 & !MSK1
        # REQ0 & !ACK0 & !MSK0
        # USTAT;

 !ISTAT =  < oe> !CS & WE
       < xsum>   {force invert}
       < set_out> CS & ISTAT {FF output is reset }
       < ck_out> !CS & WE
       < set_in> !RST  {interrupt is masked on reset}
       < ck_in> !WE & !CS
       < sum>  REQ3 & !ACK3 & !MSK3
          # REQ2 & !ACK2 & !MSK2
          # REQ1 & !ACK1 & !MSK1
          # REQ0 & !ACK0 & !MSK0
          # USTAT;
```

Appendix B. PLD ToolKit Source Code
Cascadable Interrupt Controller-Lower Element (continued)

```
!PVEC2 = < oe> !CS & WE
     < xsum>  {force invert}
     < ck_out> !CS & WE
     < sum> USTAT
     < set_in> !RST  {interrupt is masked on reset}
     < ck_in> !WE & !CS;

!PVEC1 = < oe> !CS & WE
     < xsum>  {force invert}
     < ck_out> !CS & WE
     < sum>  !ACK3 & REQ3 & !MSK3 & !USTAT
       #  !ACK2 & REQ2 & !MSK2 & !USTAT
       #  RVEC1 & USTAT
     < set_in> !RST  {interrupt is masked on reset}
     < ck_in> !WE & !CS;

!PVEC0 = < oe> !CS & WE
     < xsum>  {force invert}
     < ck_out> !CS & WE
     < sum>  !ACK3 & REQ3 & !MSK3 & !USTAT
       # !ACK1 & REQ1 & !MSK1 & MSK2 & !USTAT
       # !MSK1 & !ACK1 & REQ1 & !REQ2 & !USTAT
       #  RVEC0 & USTAT
     < set_in> !RST  {interrupt is masked on reset}
     < ck_in> !WE & !CS;

!ACK3 = < oe>
     < clr_out> !CS & WE & !PVEC2 & PVEC1 & PVEC0 & ISTB & !ACK3 {FF output is set }
     < set_out> CS & ACK3 & !REQ3; {FF output is reset }

!ACK2 = < oe>
     < clr_out> !CS & WE & !PVEC2 & PVEC1 & !PVEC0 & ISTB & !ACK2 {FF output is set }
     < set_out> CS & ACK2 & !REQ2; {FF output is reset }

!ACK1 = < oe>
     < clr_out> !CS & WE & !PVEC2 & !PVEC1 & PVEC0 & ISTB & !ACK1 {FF output is set }
     < set_out> CS & ACK1 & !REQ1; {FF output is reset }

!ACK0 = < oe>
     < clr_out> !CS & WE & !PVEC2 & !PVEC1 & !PVEC0 & ISTB & !ACK0 {FF output is set }
     < set_out> CS & ACK0 & !REQ0; {FF output is reset }

!ISTB = < oe>
     < clr_out> ISTAT & !ISTB {FF output is set }
     < set_out> CS & ISTB; {FF output is reset }
```

Appendix C. PLD ToolKit Source Code
Cascadable Interrupt Controller-Upper Element

```
{Cascaded Interrupt Controller - Upper Element}
CY7C331;  {declare device type}

CONFIGURE;

CS(node =  4),              {pin 4, chip select}
WE(node =  5),              {pin 5, write enable}
RST(node =  6),             {pin 6, reset}
REQ3(node =  9),            {pin 9, interrupt request channel 3}
REQ2(node =  10),           {pin 10, interrupt request channel 2}
REQ1(node =  11),           {pin 11, interrupt request channel 1}
REQ0(node =  12),           {pin 12, interrupt request channel 0}

PVEC3(node =  28),          {pin 28, data bus 3 - always zero}
PVEC2(node =  26),          {pin 26, data bus 2 - always zero}
PVEC1(node =  24),          {pin 24, data bus 1 - always zero}
PVEC0(node =  20),          {pin 20, data bus 0 - always zero}
ACK3(node =  25),           {pin 25, acknowledge channel 3}
ACK2(node =  23),           {pin 23, acknowledge channel 2}
ACK1(node =  19),           {pin 19, acknowledge channel 1}
ACK0(node =  17),           {pin 17, acknowledge channel 0}
MSK3(node =  34,SRC =  28),   {shared input mux for pin 28}
MSK2(node =  33,SRC =  26),   {shared input mux for pin 26}
MSK1(node =  32,SRC =  24),   {shared input mux for pin 24}
MSK0(node =  31,SRC =  20),   {shared input mux for pin 20}

ISTB(node =  27),           {pin 27, internal strobe}

USTAT(node =  18),          {pin 18, interrupt status output}
ISENSE(node =  30,SRC =  18),  {shared input mux for pin 18}
                  {internal interrupt sense to generate input for ISTB}

RVEC1(node =  16),          {pin 16, ripple vector bit 1 output}
RVEC0(node =  15),          {pin 15, ripple vector bit 0 output}

EQUATIONS;
 !PVEC3 =  < set_out>  {always zero}
       < set_in> !RST  {interrupt is masked on reset}
       < ck_in> !WE & !CS;

 !PVEC2 =  < set_out>  {always zero}
       < set_in> !RST  {interrupt is masked on reset}
       < ck_in> !WE & !CS;

 !PVEC1 =  < xsum>   {force invert}
       < ck_out> !CS & WE
       < sum>   !ACK3 & REQ3 & !MSK3
          #   !ACK2 & REQ2 & !MSK2
       < set_in> !RST  {interrupt is masked on reset}
       < ck_in> !WE & !CS;

 !PVEC0 =  < xsum>   {force invert}
       < ck_out> !CS & WE
```

Appendix C. PLD ToolKit Source Code
Cascadable Interrupt Controller-Upper Element (continued)

```
      < sum>  !ACK3 & REQ3 & !MSK3
        # !ACK1 & REQ1 & !MSK1 & MSK2 #  !MSK1 & !ACK1 & REQ1 & !REQ2
      < set_in> !RST  {interrupt is masked on reset}
      < ck_in> !WE & !CS;

!ACK3 =  < oe>
     < clr_out> !CS & WE & PVEC1 & PVEC0 & ISTB & !ACK3 {FF output is set }
     < set_out> CS & ACK3 & !REQ3; {FF output is reset }

!ACK2 =  < oe>
     < clr_out> !CS & WE & PVEC1 & !PVEC0 & ISTB & !ACK2 {FF output is set }
     < set_out> CS & ACK2 & !REQ2; {FF output is reset }

!ACK1 =  < oe>
     < clr_out> !CS & WE & !PVEC1 & PVEC0 & ISTB & !ACK1 {FF output is set }
     < set_out> CS & ACK1 & !REQ1; {FF output is reset }

!ACK0 =  < oe>
     < clr_out> !CS & WE & !PVEC1 & !PVEC0 & ISTB & !ACK0 {FF output is set }
     < set_out> CS & ACK0 & !REQ0; {FF output is reset }

!USTAT =  < oe>
      < xsum>  {force invert}
      < set_out>  {make FF transparent}
      < clr_out>  {make FF transparent}
      < sum>  REQ3 & !ACK3 & !MSK3
        # REQ2 & !ACK2 & !MSK2
        # REQ1 & !ACK1 & !MSK1
        # REQ0 & !ACK0 & !MSK0
      < ck_in> !CS & WE
      < clr_in> CS & ISENSE;

!RVEC1 =  < oe>
      < xsum>  {force invert}
      < set_out>  {make FF transparent}
      < clr_out>  {make FF transparent}
      < sum>  !ACK3 & REQ3 & !MSK3
        #  !ACK2 & REQ2 & !MSK2;

!RVEC0 =  < oe>
      < xsum>  {force invert}
      < set_out>  {make FF transparent}
      < clr_out>  {make FF transparent}
      < ck_out> !CS & WE
      < sum>  !ACK3 & REQ3 & !MSK3
        # !ACK1 & REQ1 & !MSK1 & MSK2
        # !ACK1 & REQ1 & !MSK1 & !REQ2;

!ISTB =  < oe>
     < clr_out> ISENSE & !ISTB {FF output is set }
     < set_out> CS & ISTB; {FF output is reset }
```

CYPRESS SEMICONDUCTOR

Using the CY7C330 as a Multi-channel Mbus Arbiter

This application note discusses the use of the CY7C330 as a bus arbiter for an Mbus system based on the Cypress SPARC CY7C600 RISC processor. The CY7C330 is a high-speed synchronous erasable programmable logic device (EPLD) optimized for finite state machine (FSM) applications.

The Cypress SPARC system utilizes a CY7C601 RISC processor, a CY7C602 floating point unit (FPU), four CY7C604 cache controller and memory management units (CMU), and eight CY7C157 16K x 16 cache RAMs for a 256-Kbyte cache. The arbiter uses a combination of techniques to resolve Mbus access contention for a system with four CMU bus masters. *Figure 1* shows a block diagram of the Mbus system.

CY7C330 Brief Description

The CY7C330 is a 66-MHz, high-performance PLD with 11 input latches, 17,000 programmable bits, four buried state registers, and 12 user-configurable output macrocells. It is manufactured using a CMOS 0.8-micron, double-metal processing technology that is UV erasable. The CY7C330 comes in 28-pin, 300-mil dual in-line and LCC/PLCC packages. You can partition it into multiple functional blocks, as shown in this application. (See *Figure 1* in "Understanding the CY7C330 Synchronous EPLD" for a block diagram of the CY7C330.)

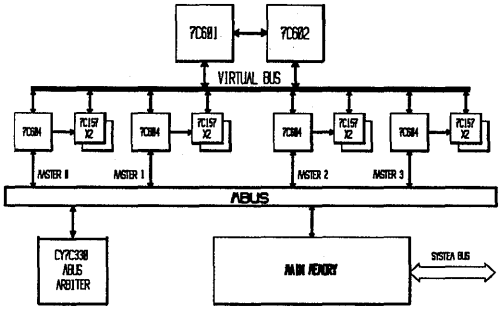


Figure 1. Mbus System Block Diagram

Mbus Description

The Mbus is a system bus defined to be a SPARC standard main memory interface for the Cypress CY7C604 SPARC cache/memory management unit. The M in Mbus stands for module and emphasizes the multi-processor module support that SPARC offers.

The Mbus is a high-speed synchronous, 64-bit, multiplexed address/data bus that operates at the CY7C601's clock rate. Mbus accesses are initiated by a master and responded to by a slave. Generally, a bus transaction takes place between a master and main memory, but in the case of direct data intervention, transactions can occur between masters.

The handshake between the CY7C604 CMMU and the arbiter utilizes a request line (MRQ0-3) and a grant line (MGT0-3) for each master. A busy line (MBB) is common to all masters and indicates that the bus is in use.

Figure 2 shows the multiple Mbus request sequence. By design, bus mastership and resolution of multiple requests are performed outside the realm of Mbus and SPARC. This allows you to implement the arbitration scheme that best fits your system requirements. The application example presented here describes only one such implementation.

Mbus transfers are synchronous with respect to the system clock. The data transactions across the bus consist of a single-clock-period address phase and a multiple-clock-period data phase. The bus transfers data in word (64-bit), multi-word burst, or atomic-load-store formats. All signals are valid and sampled on the system clock's rising edge. The address phase is validated by the memory address strobe (/MAS) signal, which denotes the start of the actual data transfer. Bus states are indicated by three status lines and convey the current bus operation as well as error status. *Figure 3* shows Mbus data transfer waveforms.

Timing Considerations

To meet the Mbus timing specifications, the arbitrator must be able to: accept a request, resolve any access contention, and grant bus rights to a master, all in a single Mbus clock cycle. In this application, a 66-MHz CY7C330 implements the arbiter, whose input registers run at the same 33-MHz clock rate as the CY7C601 and CY7C604s. This speed allows the arbiter inputs to meet the Mbus masters' timing requirements. The output registers (including the state machine) are clocked at twice the rate of the bus masters (66 MHz), enabling the arbiter to sample requests with the input latches on one Mbus clock cycle's rising edge, transfer from one state to another, and grant access before the Mbus clock's next rising edge. *Figure 4* illustrates the timing relationship between Master 0 (CY7C604 at 33 MHz) and the 66 MHz CY7C330 arbiter.

Arbitration Scheme

You can employ several resolution techniques for the arbitration function. Fixed priority, rotating priority, least recently used (LRU), and random priority prove successful, although each has its own faults. A fixed priority, for instance, favors one requester more than the others. Rotating priority provides a simple but not always fair approach to arbitration. An LRU arbitration scheme represents the fairest form of contention resolution but requires a highly complex implementation. The random technique does not allow predictable arbitration results and could result in performance problems.

A combination of methods minimizes the associated problems. The circuit presented here, for example, employs both a random and a fixed priority scheme. The random scheme uses a 2-bit counter that increments every clock cycle and varies the priority accordingly.

You can set the priority function such that the processor can specify which master has the highest priority; the processor does this by loading a value into the CY7C330 via a store instruction. To support the processor in this function, the interface to the processor must provide a latched and decoded chip select, along

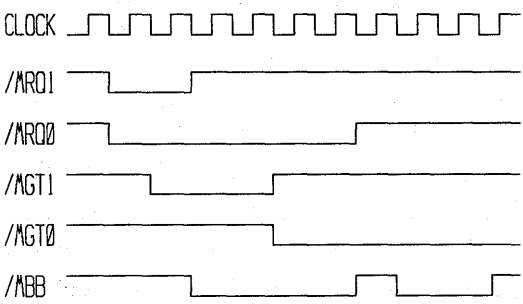


Figure 2. Mbus Multiple Request Sequence

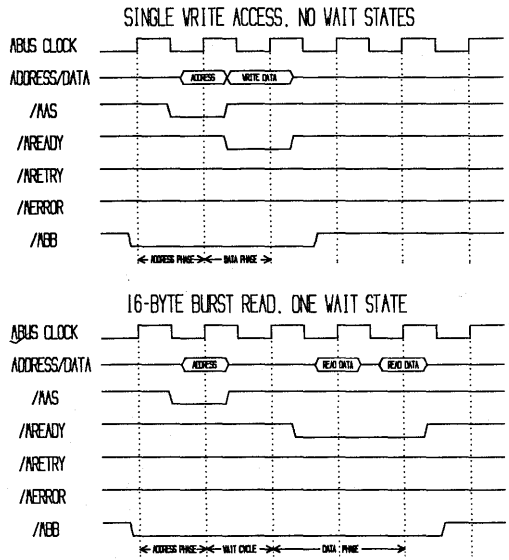


Figure 3. Mbus Data Transfer Waveforms

with a latched write enable connected directly to the arbiter. The priority function can be of value if the preset highest-priority Mbus master is fetching a program's critical data from main memory. The remaining channels follow a preset priority defined in *Table 1*.

The Random Priority Counter employs the same priority scheme used for preset priority and operates only when the latched priority is disabled by the priority selection block via the EN signal.

Design Partitioning

The arbiter design is partitioned into four functional blocks that are designed separately (*Figure 5*). The first block is the priority latch, which is a synchronous register using the decoded and latched chip select (/CS)

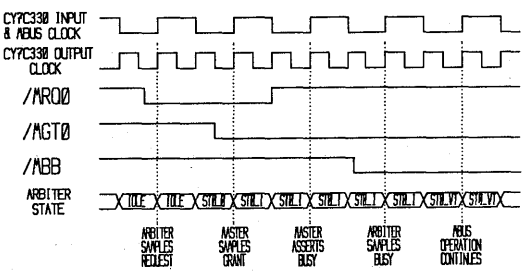


Figure 4. CY7C604 & CY7C330 Timing for Master 0

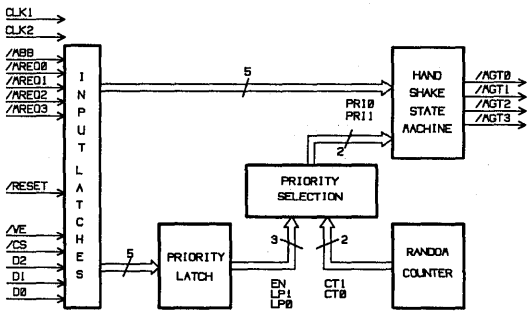


Figure 5. Arbiter Block Diagram

and write enable (/WE) signals from the CY7C601 to generate an enable signal.

The priority latch accepts three data lines from the processor bus (one for the priority enable and two for the high-priority bus master's value). The latch loads the values into dedicated registers.

The random counter, a minor portion of the design, is a free-running counter that supplies a 2-bit binary value to the priority-select block. The count changes every output clock (CLK1) cycle and provides a "seed" for the random priority function.

The priority-select block chooses between the priority latch outputs (LP0 - 1) and the random counter value (CT0 - 1) using the EN signal as the selection criteria. The two outputs (PRI0 - 1) feed to the handshake state machine and arbitrate between bus masters when more than one simultaneous request occurs.

The handshake state machine monitors the request (MRQ0 - 3) and busy (MBB) inputs and generates the grant (MGT0 - 3) signals that give an Mbus master ownership of the bus.

Priority Latch, Select and Random Counter

As described previously, the priority latch is a synchronous register loaded by the processor. When the active-Low write enable (/WE) and chip select (/CS) signals are both Low, the latch loads three data bits from the bus to the three macrocells dedicated to the priority latch. When either /WE or /CS are inactive (High), each register's output value is continuously reloaded every clock cycle, thus retaining the proper value. The equations for the priority latch are:

EN= /CS */WE*D2 + /CS*/WE*D1 + /CS*/WE*D0 + EN*WE + LP1*WE + LP0*WE + EN*CS; + LP1*CS; +LP0*CS;

EN = LP1= LP0;

The random counter is simply a 2-bit counter that changes state every output clock (CLK1) transition. The counter clears when /RESET is Low and counts in a

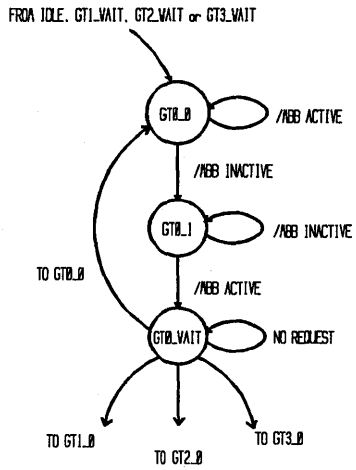


Figure 6. Bus Master 0 State Diagram.

0-1-2-3 sequence. The equations for the random counter are:

CT1 = CT0 = + CT1*/CT0 + /CT0; + /CT1*CT0;

The priority selection block selects between the priority latch and the random counter. This block is a registered multiplexer that loads its register outputs with the priority latch value if EN = 1, or the counter's current state if EN = 0. The outputs are updated every clock and fed to the handshake state machine.

Handshake State Machine

The handshake state machine controls Mbus handshake and arbitration. The machine cycles through 13 discrete states in performing its function. On power-up or reset, the state machine enters the idle state, waiting for a bus request. Upon receiving a request (/MRQ0, for instance), the machine enters a wait mode (state GT0_0). In wait mode, the arbiter looks for busy (/MBB) to go inactive, while driving the /MGT0 output active. When /MBB goes inactive, the machine goes to state GT0_1 and holds /MGT0 active, while waiting for the granted master to assert /MBB. When /MBB is

Table 1. Mbus Channel Priorities

Latched Value	PRIORITY FIRST	2ND	3RD	LOWEST
11	master3	master2	master1	master4
10	master2	master1	master0	master3
01	master1	master0	master3	master2
00	master0	master3	master2	master1

detected, the machine goes to state GT0_WAIT and looks for another request. The MGT0 grant line is held active during and after the sequence, allowing the master to maintain bus ownership until another master requests ownership.

Figure 6 shows the bus master 0 state diagram and the request/grant handshake. The operation is identical for each of the four bus masters.

The equations for the handshake state machine can be produced from a state transition table that also includes the arbiter's priority encoding. The table can be reduced to a manageable number of minterms using a public-domain optimizer called McBOOLE (see the *Reference*). *Appendix A* shows the state transition table. The sum-of-products format equations are then merged into the Cypress PLD ToolKit design file with the priority-latch, random-counter, and priority-selection equations. The PLD ToolKit design file appears in *Appendix B*.

Design Verification

The CY7C330 four-channel Mbus arbiter design was entered and verified using the PLD ToolKit. Design verification was performed using the PLD ToolKit's interactive simulator. A mouse was used with pop-down menus to create the circuit stimuli by drawing the waveform on the graphics screen for a each CY7C330 node or pin. The SIMULATE command was then selected, and the response waveforms were visually inspected, giving a high degree of confidence in the design's function before programming a part.

Reference

"McBOOLE: A New Procedure For Exact Logic Minimization," M.R. Dagenias, V.K. Agarwal, N.C. Rumin, *IEEE transactions on CAD of Circuit and Systems*, vol. CAD-5, N.1, January 1986, p.229.

Appendix A. Mbus Handshake/Arbiter State Transition Table.

```
/*STATE TABLE FOR MBUS ARBITER HANDSHAKE STATE MACHINE -names:
MBB,MRQ3,MRQ2,MRQ1,MRQ0,PRI1,PRI0,ST3,ST2,ST1,ST0,MGT3,MGT2,MGT1,MGT0;     input
ST3,ST2,ST1,ST0,MGT3,MGT2,MGT1,MGT0;   output
*/
/*  PRESENT      NEXT
    STATE        STATE
    (INPUTS)     (OUTPUTS)
    ------------------------------------------------------------------------
              MMMMPP  MMMM          MMMM
              MRRRRRRSSSSGGGG       SSSSGGGG
              BQQQQ11TTTTTTTT       TTTTTTTT
              B32101032103210       32103210*/
              X1111XX00000000       00000000      /*WAIT FOR MRQx */
              X1110XX00X0XXXX       01000001      /*GOTO GT0 */
              X1101XX00X0XXXX       01000010      /*GOTO GT1 */
              X1011XX00X0XXXX       10000100      /*GOTO GT2 */
              X0111XX00X0XXXX       10001000      /*GOTO GT3 */
              X00000000X0XXXX       01000001      /*GOTO GT0 */
              X00010000X0XXXX       10001000      /*GOTO GT3 */
              X00100000X0XXXX       01000001      /*GOTO GT0 */
              X00110000X0XXXX       10001000      /*GOTO GT3 */
              X01000000X0XXXX       01000001      /*GOTO GT0 */
              X01010000X0XXXX       10001000      /*GOTO GT3 */
              X01100000X0XXXX       01000001      /*GOTO GT0 */
              X10000000X0XXXX       01000001      /*GOTO GT0 */
              X10010000X0XXXX       10000100      /*GOTO GT2 */
              X10100000X0XXXX       01000001      /*GOTO GT0 */
              X11000000X0XXXX       01000001      /*GOTO GT0 */
              X00000100X0XXXX       01000010      /*GOTO GT1 */
              X00010100X0XXXX       01000010      /*GOTO GT1 */
              X00100100X0XXXX       01000001      /*GOTO GT0 */
              X00110100X0XXXX       10001000      /*GOTO GT3 */
              X01000100X0XXXX       01000010      /*GOTO GT1 */
              X01010100X0XXXX       01000010      /*GOTO GT1 */
              X01100100X0XXXX       01000001      /*GOTO GT0 */
              X10000100X0XXXX       01000010      /*GOTO GT1 */
              X10010100X0XXXX       01000010      /*GOTO GT1 */
              X10100100X0XXXX       01000001      /*GOTO GT0 */
              X11000100X0XXXX       01000010      /*GOTO GT1 */
              X00001000X0XXXX       10000100      /*GOTO GT2 */
              X00011000X0XXXX       10000100      /*GOTO GT2 */
              X00101000X0XXXX       10000100      /*GOTO GT2 */
              X00111000X0XXXX       10000100      /*GOTO GT2 */

              X01001000X0XXXX       01000010      /*GOTO GT1 */
              X01011000X0XXXX       01000010      /*GOTO GT1 */
              X01101000X0XXXX       01000001      /*GOTO GT0 */
              X10001000X0XXXX       10000100      /*GOTO GT2 */
              X10011000X0XXXX       10000100      /*GOTO GT2 */
              X10101000X0XXXX       10000100      /*GOTO GT2 */
              X11001000X0XXXX       01000010      /*GOTO GT1 */
              X00001100X0XXXX       10001000      /*GOTO GT3 */
              X00011100X0XXXX       10001000      /*GOTO GT3 */
              X00101100X0XXXX       10001000      /*GOTO GT3 */
```

Appendix A. Mbus Handshake/Arbiter State Transition Table.

```
X00111100X0XXXX   10001000   /*GOTO GT3 */
X01001100X0XXXX   10001000   /*GOTO GT3 */
X01011100X0XXXX   10001000   /*GOTO GT3 */
X01101100X0XXXX   10001000   /*GOTO GT3 */
X10001100X0XXXX   10000100   /*GOTO GT2 */
X10011100X0XXXX   10000100   /*GOTO GT2 */
X10101100X0XXXX   10000100   /*GOTO GT2 */
X11001100X0XXXX   01000010   /*GOTO GT1 */
                             /*CH 0 STATES */
0XXXXXX01000001   01000001   /*GT0_0,  WAIT ON MBB= 1 IN GT0_0*/
1XXXXXX01000001   00010001   /*GT0_0,  GOTO GT0_1 */
1XXXXXX00010001   00010001   /*GT0_1,  WAIT ON MBB= 0 */
0XXXXXX00010001   00100001   /*GT0_1,  GOTO GT0_WAIT */
X1111XX00100001   00100001   /*GT0_WAIT */
                             /*CH 1 STATES */
0XXXXXX01000010   01000010   /*GT1_0,  WAIT ON MBB= 1 IN GT1_0*/
1XXXXXX01000010   00010010   /*GT1_0,  GOTO GT1_1 */
1XXXXXX00010010   00010010   /*GT1_1,  WAIT ON MBB= 0 */
0XXXXXX00010010   00100010   /*GT1_1,  GOTO GT1_WAIT */
X1111XX00100010   00100010   /*GT1_WAIT */
                             /*CH 2 STATES */
0XXXXXX10000100   10000100   /*GT2_0,  WAIT ON MBB= 1 IN GT2_0*/
1XXXXXX10000100   00010100   /*GT2_0,  GOTO GT2_1 */
1XXXXXX00010100   00010100   /*GT2_1,  WAIT ON MBB = 0 */
0XXXXXX00010100   00100100   /*GT2_1,  GOTO GT2_WAIT */
X1111XX00100100   00100100   /*GT2_WAIT */
                             /*CH 3 STATES */
0XXXXXX10001000   10001000   /*GT3_0,  WAIT ON MBB= 1 IN GT3_0*/
1XXXXXX10001000   00011000   /*GT3_0,  GOTO GT3_1 */
1XXXXXX00011000   00011000   /*GT3_1,  WAIT ON MBB = 0 */
0XXXXXX00011000   00101000   /*GT3_1,  GOTO GT3_WAIT */
X1111XX00101000   00101000   /*GT3_WAIT */
```

Appendix B. PLD ToolKit Source File for Mbus Arbiter

```
CY7C330;
                    {DESIGN FILE: FOUR CHANNEL MBUS ARBITRATION UNIT WITH
                    RANDOM PRIORITY COUNTERS AND SYNCHRONOUS PRIORITY ENABLE}
CONFIGURE;

                    {INPUTS}

CLK1,               {Output Clock 2x CLK2 }
CLK2,               {Input Clock = MBUS System Clock }
!RESET,             {Reset, Active Low }
MBB,                {MBUS Busy, Active Low}
MRQ0,               {MBUS Channel 0 Request, Active Low}
MRQ1,               {MBUS Channel 1 Request, Active Low}
MRQ2,               {MBUS Channel 2 Request, Active Low}
MRQ3(node=9),       {MBUS Channel 3 Request, Active Low}
CS,                 {Decoded Processor Chip Select}
WE,                 {Processor Write Enable}
D0,                 {Data Bus Bit 0, Lalched Priority Bit 0}
D1,                 {Data Bus Bit 1, Latched Priority Bit 1}
D2,                 {Data Bus Bit 2, Latched Priority Enable Bit}

                    {OUTPUTS}

!MGT0(node=15),     {MBUS Channel 0 Grant, Active Low}
!MGT1,              {MBUS Channel 1 Grant, Active Low}
!MGT2,              {MBUS Channel 2 Grant, Active Low}
!MGT3,              {MBUS Channel 3 Grant, Active Low}
!EN,                {Settable Priority Enable Bit}
!PRI0(node=23),     {Priority Selection Bit 0}
!PRI1,              {Priority Selection Bit 1}
!CT0,               {Random Counter Bit 0}
!CT1,               {Random Counter Bit 1}
!LP0,               {Latched Priority Bit 0}
!LP1,               {Latched Priority Bit 1}
INT_RST(node=29),   {Sync Reset Node}
ST0(node=31),       {State Variable Bit 0}
ST1,                {State Variable Bit 1}
ST2,                {State Variable Bit 2}
ST3,                {State Variable Bit 3}
                    {End of configuration section}

EQUATIONS;

INT_RST = RESET;
                    {MBUS Request/Grant Handshake State Machine Equations}

ST3 = <sum>  /MRQ3*MRQ1*MRQ0*/PRI1*/ST3*/ST2*/ST0
        + /MRQ3*PRI1*PRI0*/ST3*/ST2*/ST0
        + /MRQ3*MRQ0*/PRI1*/PRI0*/ST3*/ST2*/ST0
        + /MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*/ST0
        + /MRQ2*PRI1*/PRI0*/ST3*/ST2*/ST0
        + MRQ3*/MRQ2*MRQ1*MRQ0*/ST3*/ST2*/ST0
        + MRQ3*/MRQ2*MRQ0*/PRI0*/ST3*/ST2*/ST0
        + MRQ3*/MRQ2*PRI1*/ST3*/ST2*/ST0
        + /MBB*ST3*/ST2*/ST1*/ST0*/MGT3*MGT2*/MGT1*/MGT0
        + /MBB*ST3*/ST2*/ST1*/ST0*MGT3*/MGT2*/MGT1*/MGT0;
```

Appendix B. PLD ToolKit Source File for Mbus Arbiter

```
ST2 = <sum>  MRQ2*/MRQ1*PRI1*/PRI0*/ST3*/ST2*/ST0
         + MRQ2*MRQ1*/MRQ0*/PRI0*/ST3*/ST2*/ST0
         + /MRQ1*/PRI1*PRI0*/ST3*/ST2*/ST0
         + /MRQ0*/PRI1*/PRI0*/ST3*/ST2*/ST0
         + MRQ1*/MRQ0*/PRI1*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*/MRQ1*MRQ0*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*/MRQ1*PRI1*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*MRQ1*/MRQ0*/ST3*/ST2*/ST0
         + /MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*/MGT1*MGT0
         + /MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*MGT1*/MGT0;
ST1 = <sum>  /MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*MGT1*/MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*/MGT3*MGT2*/MGT1*/MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*MGT3*/MGT2*/MGT1*/MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*/MGT1*MGT0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*/MGT3*/MGT2*MGT1*/MGT0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*/MGT3*MGT2*/MGT1*/MGT0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*/MGT3*/MGT2*/MGT1*MGT0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*MGT3*/MGT2*/MGT1*/MGT0;
ST0 = <sum>  MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*/MGT1*MGT0
         + MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*MGT1*/MGT0
         + MBB*/ST3*/ST2*/ST1*ST0*MGT3*/MGT2*/MGT1*/MGT0
         + MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*MGT1*/MGT0
         + MBB*ST3*/ST2*/ST1*/ST0*/MGT3*MGT2*/MGT1*/MGT0
         + MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*/MGT1*MGT0
         + MBB*ST3*/ST2*/ST1*/ST0*MGT3*/MGT2*/MGT1*/MGT0;
MGT3=<oe>
         <sum>  /MRQ3*MRQ1*MRQ0*/PRI1*/ST3*/ST2*/ST0
         + /MRQ3*PRI1*PRI0*/ST3*/ST2*/ST0
          + /MRQ3*MRQ0*/PRI1*/PRI0*/ST3*/ST2*/ST0
         + /MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*/ST0
         + MBB*/ST3*/ST2*/ST1*ST0*MGT3*/MGT2*/MGT1*/MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*MGT3*/MGT2*/MGT1*/MGT0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*MGT3*/MGT2*/MGT1*/MGT0
         + MBB*ST3*/ST2*/ST1*/ST0*MGT3*/MGT2*/MGT1*/MGT0
          +/MBB*ST3*/ST2*/ST1*/ST0*MGT3*/MGT2*/MGT1*/MGT0;
MGT2  =  <oe>
        <sum>  MBB*/ST3*/ST2*/ST1*ST0*/MGT3*MGT2*/MGT1*/MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*/MGT3*MGT2*/MGT1*/MGT0
         + /MRQ2*PRI1*/PRI0*/ST3*/ST2*/ST0
         + MRQ3*/MRQ2*MRQ1*MRQ0*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*/MGT3*MGT2*/MGT1*/MGT0
         + MRQ3*/MRQ2*MRQ0*/PRI0*/ST3*/ST2*/ST0
         + MRQ3*/MRQ2*PRI1*/ST3*/ST2*/ST0
         + MBB*ST3*/ST2*/ST1*/ST0*/MGT3*MGT2*/MGT1*/MGT0
         + /MBB*ST3*/ST2*/ST1*/ST0*/MGT3*MGT2*/MGT1*/MGT0;
MGT1  =  <oe>
        <sum>  MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*MGT1*/MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*MGT1*/MGT0
         + MRQ2*/MRQ1*PRI1*/PRI0*/ST3*/ST2*/ST0
         + /MRQ1*/PRI1*PRI0*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*/MRQ1*MRQ0*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*/MRQ1*PRI1*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*/MGT3*/MGT2*MGT1*/MGT0
         + /MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*MGT1*/MGT0
         + MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*MGT1*/MGT0;
```

Appendix B. PLD ToolKit Source File for Mbus Arbiter

```
MGT0  =  <oe>
       <sum>  MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*/MGT1*MGT0
         + /MBB*/ST3*/ST2*/ST1*ST0*/MGT3*/MGT2*/MGT1*MGT0
         + MRQ2*MRQ1*/MRQ0*/PRI0*/ST3*/ST2*/ST0
         + /MRQ0*/PRI1*/PRI0*/ST3*/ST2*/ST0
         + MRQ1*/MRQ0*/PRI1*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*MRQ1*/MRQ0*/ST3*/ST2*/ST0
         + MRQ3*MRQ2*MRQ1*MRQ0*/ST3*/ST2*ST1*/ST0*/MGT3*/MGT2*/MGT1*MGT0
         + /MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*/MGT1*MGT0
         + MBB*/ST3*ST2*/ST1*/ST0*/MGT3*/MGT2*/MGT1*MGT0;

              {Random Counter Equations}

CT1  =  <oe>
       <sum>  CT1*/CT0
         + /CT1*CT0;

CT0  =  <oe>
       <sum>   /CT0;

              {Latched Priority Equations}


EN   =  <oe>
       <sum>  /CS*/WE*D2
         + EN*WE
         + EN*CS;


LP1  =  <oe>
       <sum>  /CS*/WE*D1
         + LP1*WE
         + LP1*CS;


LP0  =  <oe>
       <sum>  /CS*/WE*D0
         + LP0*WE
         + LP0*CS;


              {Priority Selection Latch}


PRI1  =  <oe>
       <sum>  /EN*CT1
           + EN*LP1;


PRI0  =  <oe>
       <sum>  /EN*CT0
         + EN*LP0;

              {End of file}
```

Using the CY7C331 as a Waveform Generator

This application note demonstrates the ability of the Cypress CY7C331 CMOS Erasable Programmable Logic Device (EPLD) to implement a design requiring multiple clocks, input registers, buried registers, and independent control of individual registers' set and reset inputs. Combined with this design flexibility, the CY7C331 provides high-speed performance—an unprecedented combination.

The application example described in this application note shows how to use the CY7C331 as a programmable waveform generator.

CY7C331 Background

The CY7C331 is a member of the Cypress slimline 28-pin family of high-performance CMOS EPLDs, which are characterized by high speed, increased I/O, and high integration. The CY7C331 has a highly flexible architecture that supports asynchronous and general-purpose glue-logic integration applications.

The CY7C331 has a 192-product-term array and 12 I/O-logic macrocells. Each macrocell has two D-type flip-flops with asynchronous set, reset, and bypass capability. You can individually program the flip-flops' clock, set, and reset inputs, as well as each macrocell's logic polarity and output enable control. The CY7C331 easily supports combinatorial and registered inputs, along with buried states.

The ability to bury registers and associated gates is highly desirable because it helps increase the number of usable gates in an EPLD. Typically, if you use an I/O pin as an input, you waste the output register and its supporting product term structure. This loss occurs because conventional devices provide only one macrocell feedback

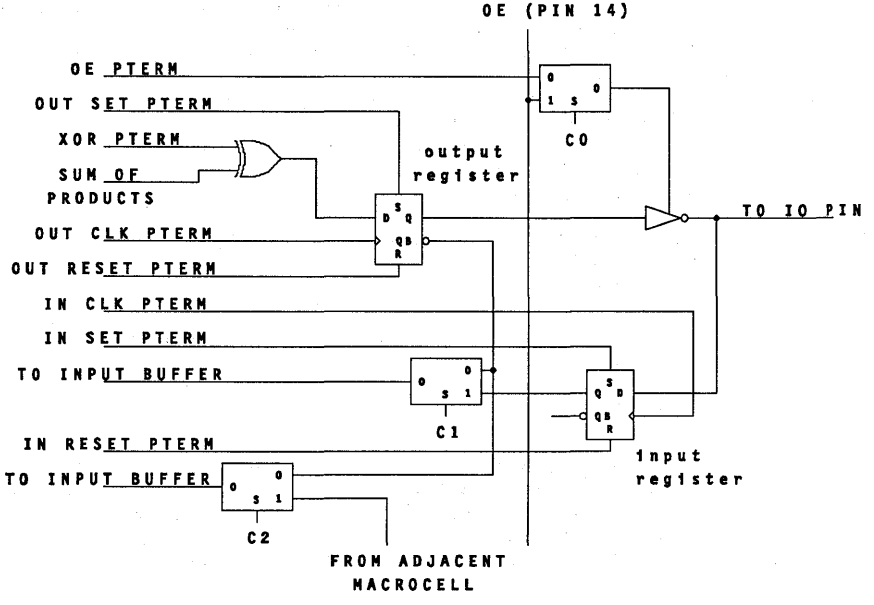


Figure 1. The CY7C331 I/O Macrocell and Shared Input Mux

path. Using this path as an input makes it impossible to feed the contents of the register back into the array.

The CY7C331's dual-muxing structure eliminates this limitation by allowing you to use the shared input mux (*Figure 1*) as an I/O path into the array, while simultaneously feeding back the register contents using the separate macrocell feedback mux. Because you can make the CY7C331's output register transparent by asserting both the register's set and clear nodes, you can also achieve simultaneous combinatorial feedback. Using this feature, you can implement bidirectional I/O in both registered and combinatorial configurations.

Configuring the CY7C331

Figure 2 lists PLD ToolKit source code that configures a CY7C331 I/O macrocell as bidirectional, with feedback from the output. The I/O pin corresponding to the macrocell is labeled IO_PIN, and the path from the I/O pin to the macrocell is IN_PATH. The code includes explanatory comments.

Note that the source code assigns IO_PIN to node 28 and IN_PATH to node 34, with pin 28 as a source. In the PLD ToolKit simulator, you must add the input waveform on the trace corresponding to node 28, even though that trace is named IO_PIN. IN_PATH's node 34 is a read-only node. This is true even if you configure IO_PIN as a buried register, and IN_PATH is always an input. The reason is that node 34 is just a mux, and the register associated with the input belongs to node (pin) 28. If you want to see the output register's value when the pin is an input, you can create a view node for the mux node. This arrangement allows you to probe several different places inside a macrocell (see the *Reference* for more information on view nodes).

The CY7C331 as a Function Generator

Waveform generators are useful in a variety of applications, primarily in the test and diagnostic areas. Any time you need to create high-speed digital waveforms, a programmable waveform generator is the ideal solution. The CY7C331 design described here allows you to generate waveforms of frequencies greater than 30 MHz.

This waveform generator builds waveforms with respect to a system clock called SYS_CLK. To use the generator, you load into LOW_REG(2:0) the number of

```
{*************************************************************************************}
CY7C331;        {The first line of code selects the device }
CONFIGURE;      {In this section pin and node names are specified, along with configuration information}

INCLK, OUTCLK, /INCLR, /INSET, OE1, /OE2, INPUT, /OUTCLR(NODE=9), /OUTSET,
    {The input names are listed above. Pin 1 will be the input clock, pin 2 will be the output clock. Pins 3 and
4 will be the input register's clear and set signals respectively. Pins 5 and 6 will be output enables, OE1 is high
asserted, /OE2 is low asserted. Pin 7 is a straight input. We skip pin 8 because it is Vss. Pins 9 and 10 will be
the input register's clear and set signals.}
IO_PIN(NODE=28, IREG), IN_PATH(NODE=34, SRC=28), OUT(NODE=27),
    {Pin 28 is the actual bidirectional pin. The IREG attribute specifies that the input to the array comes from
the output register, rather than the pin. Node 34 is the shared input mux for nodes 27 and 28. IN_PATH is the
input path to the array from pin 28. Pin 27 is a simple output.}
EQUATIONS;      {This is where the array is specified.}

IO_PIN =        <SUM> INPUT  {When IO_PIN is an output, it follows Pin 7.}
                <SET_OUT> OUTSET
                <CLR_OUT> OUTCLR
                <CLK_OUT> OUTCLK
                <OE> OE1 * OE2          {Outputs are enabled when OE_1 is high, and /OE_2 is low.}
                <CLK_IN>  INCLK
                <CLR_IN> INCLR
                <SET_IN> INSET;

OUT =           <OE>            {Listing the connective alone sets the product term to "1", always asserted.}
                <SET_OUT>       {When both the set and reset product terms are asserted, the register }
                <CLR_OUT>       {becomes transparent. Thus, this is a combinatorial output.}
                <SUM> IN_PATH;  {This output always shows the value of the input register at pin 28.}
                                {If the register is in combinatorial mode, the value on pin 28 will be shown.}
```

Figure 2. PLD ToolKit Source Code for a Bidirectional Pin With Feedback

SYS_CLK cycles that you want the output waveform (OUT_WAVE) to remain Low. HI_REG(2:0) contains the number of SYS_CLK cycles that you want OUT_WAVE to be High. For this implementation, the values must be between 2 and 7.

When the START signal is asserted, OUT_WAVE goes low, and LOW_REG(2:0) is loaded into a counter. When the count is almost 0, the signal TERM_CNT is deasserted, then reasserted when the count reaches 0. This toggles OUT_WAVE and loads a second counter with the value in HI_REG(2:0). The cycle repeats, alternating between HI_REG(2:0) and LOW_REG(2:0) until SYS_CLK is withheld, or new values are loaded into HI_REG(2:0) and LOW_REG(2:0), and START is reissued. *Figure 3* depicts the waveforms for this design.

HI_REG(2:0) and LOW_REG(2:0) are loaded using /DS and ADDR(7:0). You can specify any address for these registers. In this example, HI_REG(2:0) is at ADDR(7:0) = 00 Hex, and LOW_REG(2:0) is at ADDR(7:0) = 01 Hex.

LOW_CLK_IN is the clock input for LOW_REG(2:0). The clock results from decoding the active low /DS (data strobe) and ADDR(7:0) = 01 Hex. HI_CLK_IN is similarly decoded from /DS and ADDR(7:0) = 00 Hex.

LOW_CNT_(2:0) and HI_CNT_(2:0) form two 3-bit counters. These counters are loaded with the contents of the LOW_REG(2:0) and HI_REG(2:0) registers, respectively, via each flip-flop's individual set and reset. LOW_CNT_(2:0) is loaded when /TERM_CNT is Low and OUT_WAVE is High. Similarly, HI_CNT_(2:0) is loaded when /TERM_CNT is Low and OUT_WAVE is Low. SYS_CLK clocks both counters.

/TERM_CNT is also clocked by SYS_CLK and detects when either of the counters equals 1. When this occurs, /TERM_CNT goes Low for one clock, then goes High again. /TERM_CNT's rising edge clocks OUT_WAVE, which toggles on every clock.

Implementing this design requires two separate 3-bit input registers, decoding logic for the input-register clocks, two separate 3-bit counters, logic, and two miscellaneous registers. All the counter flip-flops must be individually settable or resettable. In addition, there are four separate clocking functions. *Figure 4* shows an implementation of this design using small-scale integration.

This type of design is usually difficult to implement in a PLD. The flip-flops in most PLDs permit neither the use of the individual set and reset inputs nor separate clocking. Because the CY7C331 has these features, however, it implements the design effortlessly.

PLD ToolKit Implementation

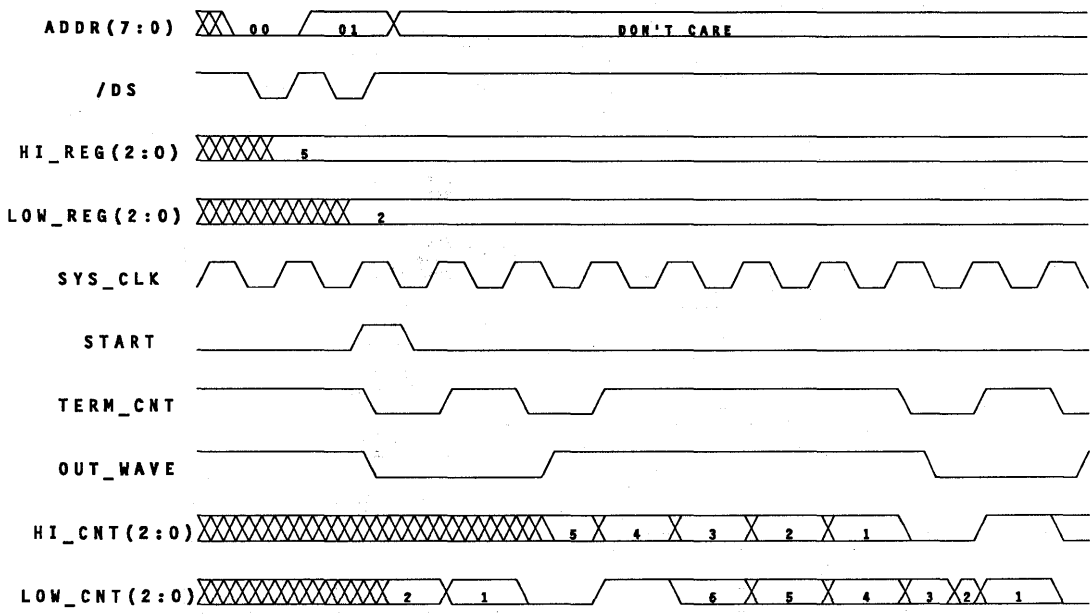


Figure 3. Waveform Generator Internal and External Timing

Appendix A contains the Cypress PLD ToolKit source code for the waveform generator. Two aspects of the code require some clarification: the pin assignments and polarity.

The pin assignments for nodes (pins) 1 through 14 are straightforward. Pin 8 has been skipped because it is a V_{SS} pin. Otherwise, these pins are the CY7C331's combinatorial inputs and thus require no configuration information.

OUT_WAVE is assigned to pin 16. "IOP" following the node assignment indicates that the feedback mux is programmed to feed back the OUT_WAVE register's Q output. Because this is the default, it does not need to be specified, but it is included here for documentation purposes. The same is true for TERM_CNT, /HI_CNT_0, and /LOW_CNT_1.

Notice that HI_IN_1 and LOW_IN_0 have the attribute "IREG" listed after the node assignment. This attribute specifies that these pins are dedicated inputs; the feedback mux selects the Q output of the input register associated with the pin, as opposed to the output register's Q output. This is an override of the default discussed above.

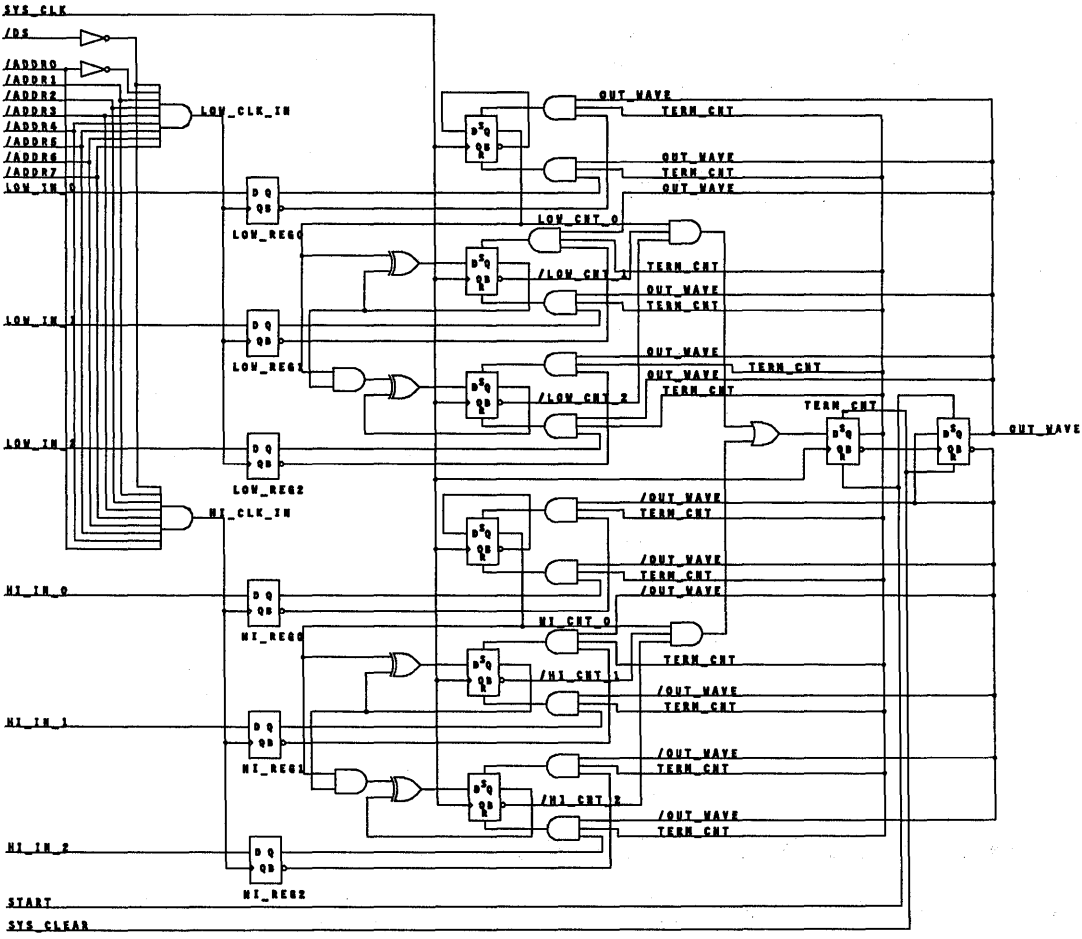


Figure 4. Schematic of the Waveform Generator

The rest of the assignments are of the same form as /HI_CNT_2 and HI_IN_2. /HI_CNT_2 is assigned to node 18, with an attribute of IOP. As mentioned earlier, this configures the feedback mux to select feedback from /HI_CNT_2 as the array input. HI_IN_2 is assigned to node 30, which is an additional mux that serves as an input path from the input register on either pin 18 or 17. The notation "SRC = 18" specifies that HI_IN_2 is assigned to the input register on pin 18. The default is that the even pin is always selected, and thus "SRC = 18" is included primarily for documentation purposes. This method for utilizing both a pin's input and output registers is used four times in this design. In each case, the output register is buried (not accessible to the pin). *Figure 5* shows the CY7C331 footprint with all external pin signals labeled.

A close look at the file in *Appendix A* might also raise questions concerning polarity conventions in the PLD ToolKit. Polarity on inputs is fairly straightforward. Note that the "/" in /START denotes a Low-asserted signal. When START appears in the EQUATIONS section (refer to /OUT_WAVE and /TERM_CNT equations) without the "/", the signal is interpreted as /START being asserted. Thus, when /START = 0, the OUT_WAVE register is set.

The output feedback polarity can cause more confusion. Polarity on the CY7C331 is programmed using the XOR in the array. Thus, when TERM_CNT is specified in the CONFIGURATION section, the output register is actually /TERM_CNT, because an inverter lies between the

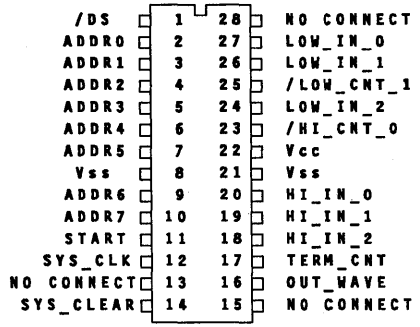


Figure 5. Footprint of the CY7C331 Waveform Generator

register output and the pin. Further, when you set TERM_CNT, the pin is Low. How, then, do you specify that TERM_CNT is asserted when it appears on the right of an equation? You refer to the polarity present on the pin. Thus, in the /OUT_WAVE equation's <CK_OUT> portion, TERM_CNT is specified. This means that /OUT_WAVE is clocked when pin 17 (TERM_CNT) exhibits a rising edge.

Reference

PLD ToolKit Manual, Chapter 4.3. Available from Cypress Semiconductor.

Appendix A. PLD ToolKit Code for the Waveform Generator

```
CY7C331;

CONFIGURE;
/DS,                                          {Low asserted data strobe}
ADDR0, ADDR1, ADDR2, ADDR3, ADDR4, ADDR5,     {address bits 0,1,2,3,4,5,}
ADDR6(NODE=9), ADDR7,                         {address bits 6 and 7}
/START,                                       {start sequence}
SYS_CLK,                                      {counter clock}
SYS_CLEAR(NODE=14),                           {initialize OUT_WAVE,TERM_CNT to a quiescent state}
OUT_WAVE(NODE=16,IOP),                        {output wave form}
TERM_CNT(NODE=17,IOP),                        {terminal count decode register}
/HI_CNT_2(NODE=18,IOP),                       {high counter bit 2, a buried register}
HI_IN_2(NODE=30,SRC=18),                      {high register input bit 2}
HI_IN_1(NODE=19,IREG),                        {high counter input bit 1}
/HI_CNT_1(NODE=20,IOP),                       {high counter bit 1, a buried register}
HI_IN_0(NODE=31,SRC=20),                      {pin 20 acts as high register input bit 0}
/HI_CNT_0(NODE=23,IOP),                       {high counter bit 0}
/LOW_CNT_2(NODE=24,IOP),                      {low counter bit 2, a buried register}
LOW_IN_2(NODE=32,SRC=24),                     {pin 24 is low register input bit 2}
/LOW_CNT_1(NODE=25,IOP),                      {low counter bit 1}
LOW_IN_1(NODE=33,SRC=26),                     {pin 26 acts as low register input bit 1}
/LOW_CNT_0(NODE=26,IOP),                      {low counter bit 1, a buried register}
LOW_IN_0(NODE=27,IREG),                       {low register input bit 0}

EQUATIONS;

LOW_CNT_0 :=        <SUM> /LOW_CNT_0
                    <CK_OUT> SYS_CLK
                    <CK_IN> DS*ADDR0*/ADDR1*/ADDR2*/ADDR3*/ADDR4*/ADDR5*/ADDR6*/ADDR7
                    <SET_OUT> /LOW_IN_0 * /OUT_WAVE * /TERM_CNT
                    <CLR_OUT> LOW_IN_0 * /OUT_WAVE * /TERM_CNT;

/LOW_IN_0 =         <CK_IN> DS*ADDR0*/ADDR1*/ADDR2*/ADDR3*/ADDR4*/ADDR5*/ADDR6*/ADDR7;

LOW_CNT_1 := <SUM> LOW_CNT_1
                    <XSUM> LOW_CNT_0
                    <SET_OUT> /LOW_IN_1 * /OUT_WAVE * /TERM_CNT
                    <CLR_OUT> LOW_IN_1 * /OUT_WAVE * /TERM_CNT
                    <CK_OUT> SYS_CLK
                    <OE>;

LOW_CNT_2 := <SUM> LOW_CNT_2
                    <XSUM> LOW_CNT_0 * LOW_CNT_1
                    <SET_OUT> /LOW_IN_2 * /OUT_WAVE * /TERM_CNT
                    <CLR_OUT> LOW_IN_2 * /OUT_WAVE * /TERM_CNT
                    <CK_OUT> SYS_CLK
                    <CK_IN> DS*ADDR0*/ADDR1*/ADDR2*/ADDR3*/ADDR4*/ADDR5*/ADDR6*/ADDR7;

/OUT_WAVE :=        <SUM> OUT_WAVE
                    <CK_OUT> TERM_CNT
                    <SET_OUT> START
                    <CLR_OUT> SYS_CLEAR
                    <OE>;
```

Appendix A. PLD ToolKit Code for the Waveform Generator

```
/TERM_CNT := <SUM> /LOW_CNT_0 * LOW_CNT_1 * LOW_CNT_2
             <SUM> /HI_CNT_0 * HI_CNT_1 * HI_CNT_2
             <CK_OUT> SYS_CLK
             <CLR_OUT> START
             <SET_OUT> SYS_CLEAR
             <OE>;

HI_CNT_0 :=  <SUM> /HI_CNT_0
             <CK_OUT> SYS_CLK
             <OE>
             <CLR_OUT> HI_IN_0 * OUT_WAVE * /TERM_CNT
             <SET_OUT> /HI_IN_0 * OUT_WAVE * /TERM_CNT;

HI_CNT_1 :=  <SUM> HI_CNT_1
             <XSUM> HI_CNT_0
             <SET_OUT> /HI_IN_1*OUT_WAVE*/TERM_CNT
             <CLR_OUT> HI_IN_1*OUT_WAVE*/TERM_CNT
             <CK_OUT> SYS_CLK
             <CK_IN> DS*/ADDR0*/ADDR1*/ADDR2*/ADDR3*/ADDR4*/ADDR5*/ADDR6*/ADDR7;

/HI_IN_1 =   <CK_IN> DS*/ADDR0*/ADDR1*/ADDR2*/ADDR3*/ADDR4*/ADDR5*/ADDR6*/ADDR7;

HI_CNT_2 :=  <SUM> HI_CNT_2
             <XSUM> HI_CNT_1*HI_CNT_0
             <SET_OUT> /HI_IN_2*OUT_WAVE*/TERM_CNT
             <CLR_OUT> HI_IN_2*OUT_WAVE*/TERM_CNT
             <CK_OUT> SYS_CLK
             <CK_IN> DS*/ADDR0*/ADDR1*/ADDR2*/ADDR3*/ADDR4*/ADDR5*/ADDR6*/ADDR7;
```

CYPRESS SEMICONDUCTOR

CY7C331 Application Example: Asynchronous, Self-Timed VMEbus Requester

This application note describes how to use the Cypress CY7C331 CMOS erasable programmable logic device (EPLD) to support asynchronous, self-timed designs. The CY7C331 is ideal for implementing asynchronous, self-timed, and general-purpose logic integration applications. The application example described here is an asynchronous, self-timed VMEbus requester .

The CY7C331 is a member of the Cypress slim-line, 28-pin family of high-performance CMOS EPLDs. Family members are characterized by high speed, increased I/O, and high integration. The CY7C331 has a highly flexible architecture with a 192-product-term logic array and 12 I/O-logic macrocells. Each macrocell provides two D flip-flops with asynchronous set, reset, and bypass capability. The flip-flop's clock, set, and reset inputs are individually programmable, as are each macrocell's logic polarity and output-enable control. The CY7C331 easily supports combinatorial and registered inputs and outputs and buried states.

Additionally, the CY7C331 has the uncommon ability to self-time asynchronous, sequential applications. A self-timed design performs a sequential task without the presence of a clock to synchronize each step in the sequence. This design approach usually results in higher performance compared to synchronous designs. The main application for self-timing is in high-performance I/O interfaces. The CY7C331 supports self-timed designs because its clock inputs are programmable, internal timing relationships are well-controlled, and metastable resolution is ultra-fast.

The VMEbus is a common, high-performance asynchronous bus. The VMEbus request function is asynchronously initiated and sequential. In addition to showing the CY7C331's ability to handle asynchronous, self-timed tasks, this application example demonstrates the use of many unique CY7C331 features.

CY7C331 Brief Description

The CY7C331 is available in a 28-pin slim-line (300 mil wide) plastic or windowed DIP and in 28-pin PLCC and LCC packages. The windowed version is UV erasable and reprogrammable, and the plastic DIP, PLCC, and LCC versions are one-time programmable. The CY7C331 is available with T_{PD} and T_{CO} specified at 20 ns max and with register set-up times of 12 or 2 ns, depending on whether the register connects to an input pin or to the device's logic array. Other commercial and military speed grades are also available.

The CY7C331 is based on a programmable sum-of-products (AND-OR) logic-array architecture. The logic array consists of 192 programmable product terms, each having as input the true and complement versions of 31 logic inputs. The product terms connect to one of twelve I/O logic macrocells, and each of these macrocells connects to a device pin. The product terms are allocated with a variable distribution to the macrocells.

The CY7C331 provides 13 combinatorial inputs to the array from dedicated input pins, one of which (pin 14) can also be used as an output-enable control. The macrocells and six shared input muxes each provide an input to the array. A shared input mux selects the input from one of two adjacent macrocells (*Figure1*).

The CY7C331's I/O-logic macrocell sums array product terms, selectively inverts the sum, and provides the result to the D input of a D flip-flop. The flip-flop's output (Q) connects through an inverting three-state buffer to a device pin and can be fed back to the array. The I/O macrocell also provides a second D flip-flop that latches data from the same device pin. This flip-flop's Q output connects to the macrocell input-select mux and to the shared-input mux (see *Figure1* in "Using the CY7C331 as a Waveform Generator"). Both flip-flops have asynchronous set (S) and reset (R) inputs, as well as bypass capability. A flip-flop bypasses the D input to Q when S and R are both High. Separate product terms drive both flip-flops' clock, S, and R inputs.

A multi-input OR gate sums the product terms. The number of product terms input to the OR gate depends on the macrocell (*Figure1*). A dual-input XOR gate selectively inverts the sum. The XOR gate's second input is a product term that controls selective inversion. You can control a macrocell's output enable (OE) by

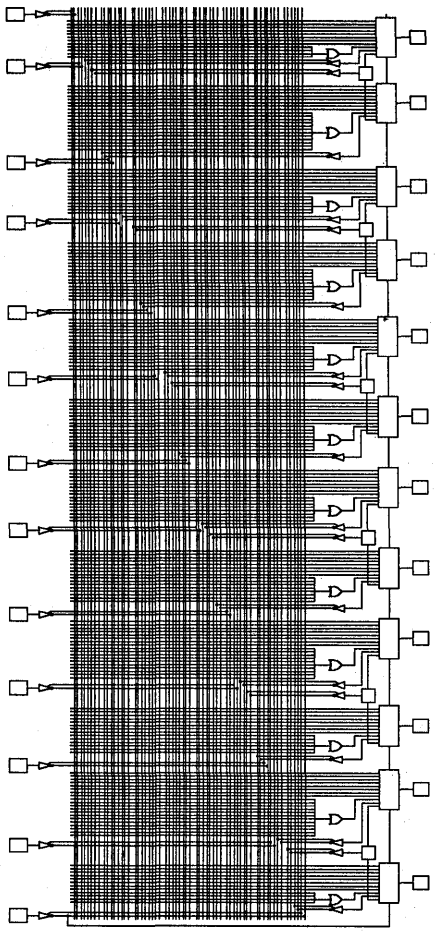


Figure 1. Cypress CY7C331 Block Diagram

using pin 14 or a product term. The OE mux selects one of these two options. Another mux, the FB mux, selects the macrocell array input. Each OE, FB, and shared-input feedback mux has an associated programmable configuration bit that controls mux selection.

CY7C331 Self-Timed Capability

The main application for self-timed functions is in high-performance I/O interfaces, where clocking restrictions prevent performance requirements from being satisfied. These applications might not have an available clock, the clock might be too slow, or synchronization time might have to be minimized.

A self-timed design implements a state machine without the presence of a clock to synchronize each state transition. The implementation of a self-timed design must meet two basic requirements:

1. It must time and perform state transitions.
2. It must synchronize asynchronous inputs.

As in any state machine, a self-timed design must meet minimum state flip-flop set-up times before performing a state transition. Without the benefit of a clock, the design must generate self-timing clocks based on the state data change due to a state transition itself. Thus, clock initiation and data changes are coincident, and the design must delay a clock to allow data to settle and meet minimum set-up time requirements.

The simplest example of self-timing appears in *Figure 3*. This circuit clocks a logic 1 into a D flip-flop on the input's rising edge. The design works if the clock delay time is long enough to allow the data input to be set up. This simple circuit illustrates how the CY7C331 supports self-timed designs; the CY7C331 allows you to program the timing relationship between the flip-flop's D-input logic and clock input logic to guarantee satisfaction of minimum set-up time requirements. The CY7C331 synchronizes asynchronous inputs in the same manner, except that the set-up time is longer to allow for metastable resolution. The CY7C331 can also perform self-timed synchronization because metastable resolution is ultra-fast.

The approach used in the CY7C331 to self-time state transitions is to delay a clock signal by passing it through the logic array one additional time; this arrangement allows data to meet set-up time requirements. To guarantee that this approach works, the extra delay in the clock path must be programmed to delay the clock as long as possible (*Figure 4*). In general, a self-timed design should set up data as fast as possible and delay the clock long enough to guarantee that data is set up. But delay time in the CY7C331 is sensitive to the logic function programmed. Guaranteeing that data is set up as fast as possible restricts the logic functions the device can perform. You can avoid this limitation by placing restrictions on the clock path. You can program any logic function if the clock delay path is slow enough.

To perform self-timed synchronization, the clock is delayed by two extra passes to provide the extra delay required for metastable resolution (*Figure 5*). Program both clock delay elements to be as slow as possible so you can configure any logic function. With these restrictions, the mean time to failure (MTF) due to a metastable condition is greater than 10 years.

Clock Delay Programming

In the CY7C331, a product term generates an output transition from Low to High faster than from High to Low. A transition caused by a single input and a single product term is faster than those caused by multiple inputs and/or product terms. The shortest delay time through a CY7C331 occurs when a single input

triggers a single product term to transition from Low to High. The slowest clock path results from placing restrictions on how the extra level of clock delay is programmed. These restrictions are:

- The clock delay should use a logic path through multiple product terms, OR gates, and XOR gates to a bypassed flip-flop.
- Clock delay logic should make product term outputs transition from High to Low.
- All product terms to the OR gate should be programmed identically to implement clock logic. The OR gate should have the same or more inputs than associated data-path OR gates.
- The programmable XOR input should be set Low.

The clock delay element shown in *Figure 4* illustrates each of the four programming restrictions.

Self-Timed VMEbus Requester

Bus requesters are used in common bus systems that support multiple processors controlling bus transfers. A processor that controls bus transfers is typically referred to as a bus master. The bus requester requests permission for a master to control the data bus and indicates to the master when data bus control has been granted. The VMEbus supports multiple bus masters.

A self-timed design approach for a VMEbus requester is appropriate because the VMEbus is asynchronous and offers high performance. The bus-request function is asynchronously initiated and is sequential. A self-timed design self-synchronizes to initiate the request and self-times the rest of the request sequence at CY7C331 device speed. A synchronous approach requires an external clock to synchronize and time the sequence, for which the VMEbus provides a 16-MHz system clock. However, a CY7C331 self-timed design provides much higher performance than a synchronous design using the system clock.

VME Background

The VMEbus is defined to support multiple bus masters, although only one master can control the bus at a time. The VMEbus provides an arbitration subsystem in which a central bus arbiter determines which master is granted the data bus. Each master contains a bus requester to request control of the bus from the arbiter.

The arbitration subsystem is supported on the VMEbus with six bused lines and four daisy-chained lines. All these lines are active Low, which is indicated by a "-" suffix on a line name. The bused lines are Bus Busy (BBSY-), Bus Clear (BCLR-), and Bus Request 3 - 0 (BR3- through BR0-).

When the daisy-chained lines enter a board, they are designated Bus Grant 3-0 In (BG3IN- through BG0IN-), and when leaving are designated Bus Grant 3 - 0 Out (BG3OUT- through BG0OUT-). (The terms BRx-, BGxIN-, and BGxOUT- are used when references are not to a specific line or lines; x is any value from 0 to 3.) The highest priority is allocated to number 3 lines and lowest to number 0 lines. The BGxOUT- lines that leave a board in slot n enter the board in slot n+1 as BGxIN- lines. The bus arbiter must always reside in the first slot of a VMEbus-based system to initiate BGxOUT- generation.

All masters in the system drive BBSY- when they have control of the bus. Within each bus-grant daisy chain, all masters drive the same BRx- line. Multiple masters on a bus grant daisy chain can request the data bus at the same time by simultaneously driving their associated BRx- lines. When this occurs, the requester furthest up in the daisy chain gets the bus grant. The remaining master(s) on the daisy chain can continue to assert BRx- until they receive a bus grant.

A simple VMEbus requester initiates a request after detecting an on-board request (OBR). (A simplified bus-request state diagram and timing diagram appear in *Figures6* and *7*.) The requester then drives the BRx- line active and waits for the associated BGxIN- line to become active. Once the requester detects BGxIN- active, BBSY- and the appropriate DMA Grant line (DMAGRx-) are driven active, while BRx- is released to inactive. The active DMAGRx line indicates to an on-board master that it has the bus and can perform a a data transfer.

While data is being transferred, the bus master asserts the Data Transfer (DTR-) input to the CY7C331 bus requester. When the master has finished using the bus, the DTR input is deasserted. The requester then releases the bus by deasserting BBSY- and OBG. Even if one of the other on-board masters wants the bus, the requester deasserts BBSY- and waits for a new BGxIN- before granting the bus to this master. This extra overhead allows other requesters that might be further up the daisy chain to obtain the bus between on-board bus requests.

If the bus grant input (BGxIN-) becomes active while none of the on-board request lines are active, the requester must pass the request down the daisy chain. This is accomplished by asserting the bus grant out (BGxOUT-) signal.

The VMEbus specification includes a few timing and requester design restrictions. A VMEbus requester must satisfy the two timing requirements displayed in *Figure6*. BBSY- must be driven for a minimum of 90 ns, and the release of BRx- must occur at least 30 ns before BBSY- is released. The primary design requirements are that BBSY- and BRx- must use open-collector

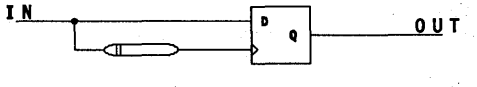


Figure 2. A Self-Timed Element

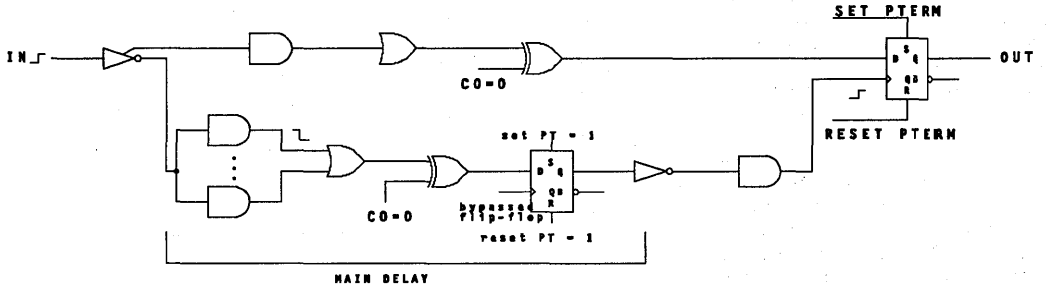


Figure 3. CY7C331 Self-Timed Element

drivers, and BGxOUT- must never glitch during operation. The restriction on BGxOUT- ensures avoidance of inadvertent bus grants.

Requester Design

The requester supports overlapped bus requests. It also releases the data bus every transfer cycle to allow the central arbiter to grant the bus to a higher-priority requester, if one exists.

The CY7C331 VMEbus requester supports three on-board DMA request lines (DMARQ2- through DMARQ0-). All the DMARQx- lines can generate a bus request on the BRx- line. The requester supports three on-board grant lines (DMAGR2- through DMAGR0-), one for each request line. When a bus grant is received on BGxIN-, the requester must determine which DMAGRx- line to activate. The requester prioritizes the DMARQx- lines and grants the bus to the highest priority request; DMARQ0- has the highest priority and DMARQ2- the lowest. The selected DMAGRx- line is not activated until the previous data transfer is complete.

If any of the DMARQx- lines are active when a bus grant is received, the requester drives BBSY- active. For overlapped operation, BBSY- is released as soon as possible to facilitate the next bus arbitration. BBSY- is not released, however, until the following criteria are met: BBSY- is driven for at least 90 ns, BGxIN- is inactive, and the previous data transfer is complete (DTR- is deasserted). If none of the DMARQx- lines is requesting the bus when a grant is received, the requester passes the grant onto BGxOUT- for the next requester on the daisy chain. The requester also recognizes a system reset (SYSRESET-) and initializes the device appropriately.

A logic diagram of a self-timed VMEbus requester using the CY7C331 appears in *Figure8.* BRx- is the OR of the DMARQx- lines.

Requester Operation

If any DMARQx line becomes active, BRx- becomes active, signifying to the arbiter that one of the masters on this board wants the data bus. An external open-collector driver drives BRx-.

Self-timed operation begins when the incoming BGxIN- line becomes active. The three on-board DMA request lines (DMARQ2- through DMARQ0-) are self-synchronized to the BGxIN- line. BGxIN's falling edge serves as a clock to register the DMARQx- lines and toggle a flip-flop from High to Low to initiate an inter-

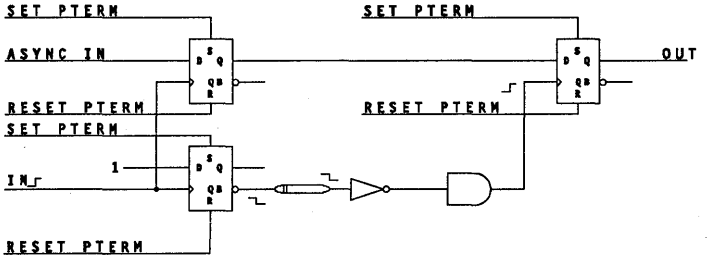


Figure 4. CY7C331 Self-Synchronizing Element

nal, self-timed clock signal (STCP). The DMARQx- lines must be synchronized, because BGxIN- can be activated when any BRx- line becomes active or when BBSY- is released. For example, if DMARQ0- causes the associated BRx- to initiate bus arbitration, and DMARQ2- attempts to become active at the same time BGxIN- becomes active, DMARQ2's resulting state could be an indeterminate metastable condition that needs time for resolution. The pair of internal clock delays provides this time before the DMAGR2- output register samples the state of DMARQ2-.

Two CY7C331 delay elements delay the internal, self-timed clock signal to provide enough time to self-synchronize the requests. The requests are prioritized during the clock delay time. The resulting delayed clock (STCP2) then asserts BBSY- if any of the DMARQx- lines are active. If none are active, the BGxOUT- line is asserted to send the grant to the next requester in the daisy chain. Using the delayed clock to generate BBSY- and BGxOUT- guarantees that both lines are synchronized and cannot glitch.

BBSY- is driven onto the bus with an external open-collector driver. The prioritized requests are clocked into registers to create the DMAGRx- signals on the delayed STCP's rising edge, if the previous data

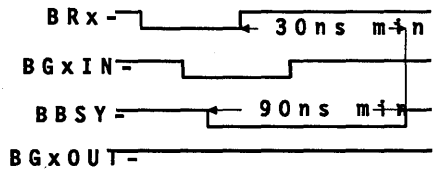


Figure 6. VME Arbitration Timing

transfer has completed, or on the rising edge of DTR- when the data transfer completes. An internal flip-flop toggles at the same time. The flip-flop output indicates transfer completion (TC).

The registered BBSY- line feeds into an external 90-ns delay line to guarantee that BBSY- is active for the minimum required time. The delay mechanism should be designed such that the delay circuit has no effect if the data transfer requires more than 90 ns to complete. One way to implement this feature is to use a one-shot triggered by the falling edge of the CY7C331's BBSY- signal. The one-shot's output is ORed with the BBSY- signal from the CY7C331 to generate the

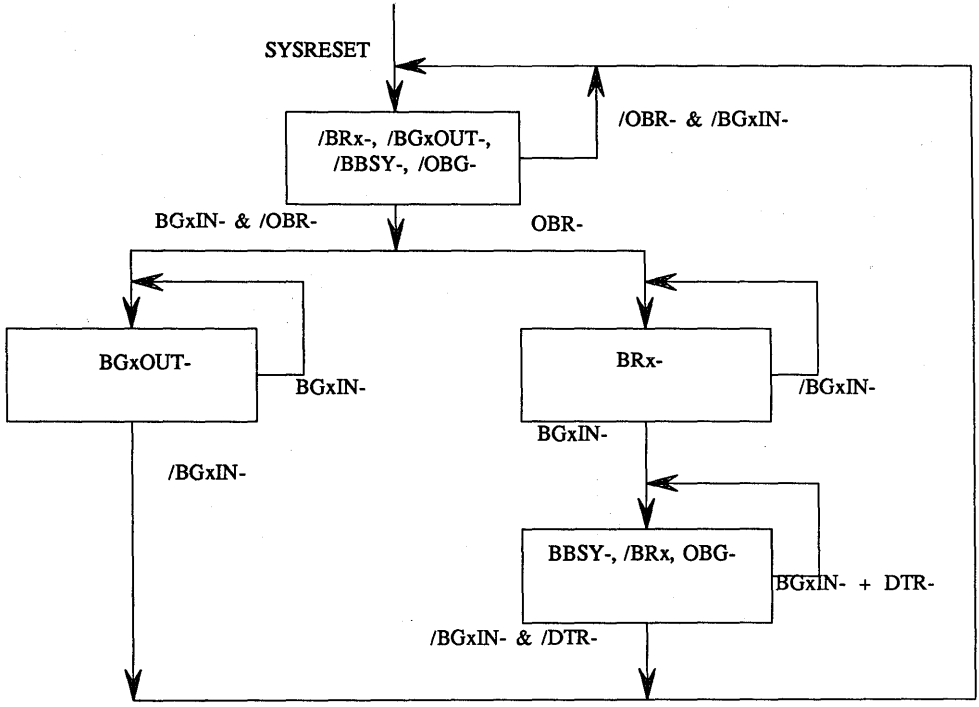


Figure 5. VME Bus Requester State Diagram

BBSY- signal to the VMEbus. The VME BBSY- signal is inactivated when the 90-ns delay has elapsed, provided that TC is True and DTR- and BGxIN- are inactive. The requester is initialized for another self-timed operation at the same time. The requester also initializes when the SYSRESET input is asserted.

This design uses the 90-ns delay circuit because an absolute delay is required to meet the VME specification. A self-timed delay can yield only relative results because there is no way to determine how many delay levels are required to obtain a 90-ns delay. Any one delay is usually much faster than the worst-case specification, but the delay might be that slow. You can emulate the delay on-chip by creating a digital delay, but accuracy would be poor because you would have to synchronize BBSY- to an absolute time base, such as the 16-MHz system clock.

The CY7C331 can emulate the external open-collector drivers, but the emulation would not meet the VMEbus specification's drive requirements. To emulate an open-collector driver, use the signal output to the external driver to drive the output enable of an on-board, inverting, three-state driver (with the input tied High).

CY7C331 Implementation

The bus requester can be implemented and simulated using the source code in *Appendix A*, generated via the Cypress PLD ToolKit software package. A close examination of the code reveals how many of the CY7C331's features are utilized.

The DMARQx- lines use two CY7C331 pins for each line—one combinatorial and one registered. The registered input pins are used to conserve output logic for other functions. The three macrocells associated with the registered inputs also perform the internal self-timed clock generation and delay functions; most other PLDs require six outputs to implement these functions. In addition, the CY7C331's individually programmable clocks allow the input register flip-flops to be clocked on BGxIN's falling edge.

BBSY is assumed to be the input to the external delay line, and the CY7C331 input BBSY90 is assumed to connect to the delay line output.

The source code defines the self-timed clock generation and delay logic needed to meet the requirements of CY7C331 self-synchronization.

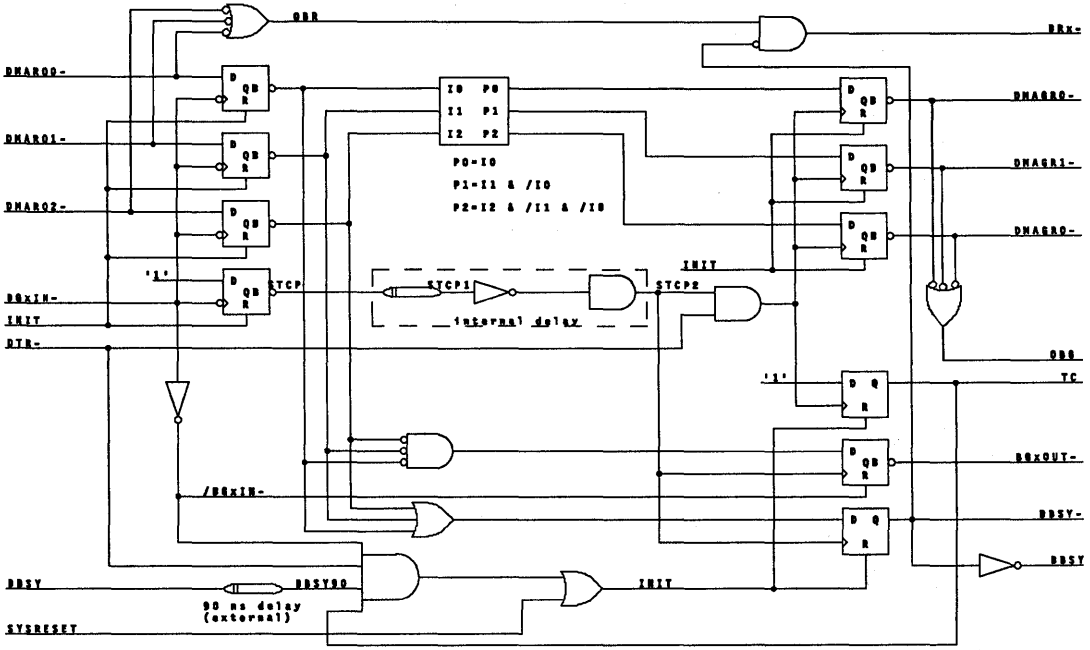


Figure 7. Self-Timed VMEbus Requester

Appendix A. PLD ToolKit Source Code for VMEbus Requester

```
CY7C331;

{ Norman Taffe
  Cypress Semiconductor
  6/20/1990
  Cypress PLD Toolkit
  VME Bus Requester
}

CONFIGURE;

DMARQ2(node= 1),
DMARQ1(node= 2),          { On-board Request Lines }
DMARQ0(node=3),
BGxIN(node= 4),           { VME Bus Grant Input }
SYSRESET(node= 6),
BBSY90(node= 7),          { Externally delayed BBSY signal}
DTR(node= 9),             { Signifies a Data Transfer in progress }
node14(node=14),
/INIT(node= 15),          { Requester initialize signal }
/OBG(node= 16,ireg),      { Signals board that it has the bus }
/STCP(node= 17),          { Self timed CLK input register }
/BBSY(node= 18),          { Assert Bus Busy when taking the bus }
/BGxOUT(node= 19,ireg),   { Send Bus Grant down the daisy chain if not wanted }
/BRx(node= 20,ireg),      { Signal arbiter that this board wants the bus }
/DMAGR0(node= 23),
/DMAGR1(node= 24),        { On-board grant lines }
/DMAGR2(node=25),
/RDMARQ1(node= 26),
/RDMARQ2(node= 27),       { Registered On-Board Request lines }
/RDMARQ0(node= 28,ireg),
STCP2(node= 33),          { Second delay stage of self timed clock }
STCP1(node= 34,src= 27),  { First delay stage of self timed clock }
TC(node= 30,SRC= 17),     { Resets the INIT signal }


EQUATIONS;

INIT =      < OE>
       < SET_OUT>
       < CLR_OUT>
       <SUM>  BGxIN*BBSY90*TC*DTR
       <SUM>  /SYSRESET;


STCP =      < CK_OUT>  RDMARQ1 & DTR      {Output Register is used for TC}
       <CLR_OUT>  INIT
       < CK_IN>  /BGxIN
       <CLR_IN>  INIT
       <SUM>;

BBSY =      < OE>
       <CK_OUT>  RDMARQ1
       < CLR_OUT>  INIT
```

```
        <SUM>  RDMARQ0
        <SUM>  /STCP1
        <SUM>  /STCP2;


BGxOUT =  < OE>
        <CK_OUT>  RDMARQ1
        <CLR_OUT>  BGxIN
        <SUM>  /RDMARQ0*STCP1*STCP2;

BRx =  < OE>
        < SET_OUT>
        < CLR_OUT>
        < XSUM>
        <SUM>  DMARQ2*DMARQ1*DMARQ0
 < SUM>  BBSY;

DMAGR0 =  < OE>
        <CK_OUT>  RDMARQ1
        < CLR_OUT>  INIT
        <SUM>  RDMARQ0;

DMAGR1 =  < OE>
        <CK_OUT>  RDMARQ1
        < CLR_OUT>  INIT
        <SUM>  /RDMARQ0*/STCP2;

DMAGR2 =  < OE>
        <CK_OUT>  RDMARQ1
        <CLR_OUT>  INIT
        <SUM>  /RDMARQ0*/STCP1*STCP2;

RDMARQ1 =  < SET_OUT>
        < CLR_OUT>                { output register for STCP2 }
        < CK_IN>  /BGxIN          { Note that XSUM is set to zero and }
        < CLR_IN>  INIT           { p-term transitions are from high }
        < SUM>  RDMARQ2           { to low, to maximize self-timed delay  }
        <SUM>  RDMARQ2
        <SUM>  RDMARQ2
        < SUM>  RDMARQ2           { Use all 6 p-terms to add to delay }
        <SUM>  RDMARQ2
        <SUM>  RDMARQ2;


RDMARQ2 =  < SET_OUT>
        < CLR_OUT>                { output register for STCP1 }
        < CK_IN>  /BGxIN          { Note that XSUM is set to zero and }
        < CLR_IN>  INIT           { p-term transitions are from high }
        < SUM>  TC                { to low, to maximize self-timed delay }
        <SUM>  TC
        <SUM>  TC
        <SUM>  TC
        <SUM>  TC
        < SUM>  TC                { Use all 12 p-terms to add to delay }
        <SUM>  TC
        <SUM>  TC
        <SUM>  TC
```

```
        <SUM>  TC
        <SUM>  TC
        <SUM>  TC;

RDMARQ0 =  <CK_IN>  /BGxIN
        <CLR_IN>  INIT
        <XSUM>;

OBG =   < OE>
     < CLR_OUT>
     < SET_OUT>
     < XSUM>
     < SUM>  DMAGR0
     < SUM>  DMAGR1
     < SUM>  DMAGR2;
```

CYPRESS SEMICONDUCTOR

Understanding the CY7C361

The Cypress CY7C361 UV-erasable PLD employs a revolutionary architecture that allows internal speeds as high as 125 MHz. The part comes in a 28-pin, 300-mil DIP and a 28-pin (P)LCC. The CY7C361 has eight input pins with macrocells, four bidirectional pins with input macrocells, one clock input with doubler, six "pure" outputs, and four Mealy macrocell outputs. Internally, there are 32 state registers.

Control-logic clocks usually run at twice the system-clock frequency in high-performance systems. Thus, for a 33-MHz system, a CY7C330 running at 66 MHz works fine. But 40-MHz RISC CPUs are now available, and even faster clock rates are right around the corner. Because control logic often does not stabilize until late in the design cycle, a PLD solution beyond 66 MHz is needed. The CY7C361 is that solution.

How does the CY7C361 achieve speeds up to 125 MHz? Through a combination of state-of-the-art process technology, circuit design, and architectural innovation (see *Figure1*).

Traditional Architectures

To understand how the CY7C361 achieves its high level of performance, consider some common PLD architectures and their limitations.

The PAL

Figure 2 shows a simplified block diagram of a traditional PAL architecture. When you implement a state machine in a PAL, two components contribute to the worst case f_{MAX}. The first is t_S, which is the delay through the AND array and the fixed OR plus the register set-up time. The second factor is t_{CF}, which is the clock-to-feedback time. Although you cannot measure t_{CF} directly, it is slightly less than the clock-to-output time, t_{CO}. The maximum frequency for a state machine implemented in this device is:

$$f_{MAX} = 1/(t_S + t_{CF}) \qquad \text{Eq. 1}$$

Substituting the minimum t_S and the maximum t_{CF} yields the worst case f_{MAX}. Typical numbers for these parameters are t_S = 18 ns and t_{CF} = 13 ns. The

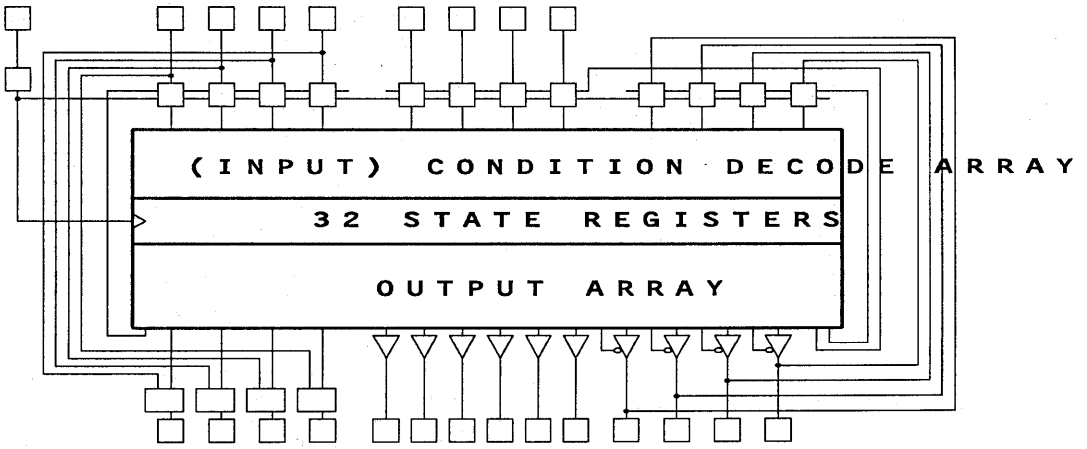


Figure 1. Block Diagram of the CY7C361

dominant parameter is always ts, primarily due to the delay through the AND array. Thus, the key to improving f_{MAX} lies in minimizing the propagation delay through the AND array.

The FPLS

Figure 3 shows a simplified block diagram of an FPLS—another architecture commonly used to implement state machines. The method for computing f_{MAX} for the FPLS is similar to that for the PLD. In this case ts is approximately twice the corresponding value for a PAL, because the FPLS value includes delays through two arrays (the AND and OR) instead of just one. This makes ts even more of a dominant parameter in the f_{MAX} calculation. The higher ts value is the tradeoff for some extra flexibility in implementing the state machine, due to having an OR array rather than a fixed-OR scheme.

General Limitations

Another major barrier to speed in both PAL and FPLS state machine designs is the design methodology itself. Traditionally, efficiency of state machine implementation has been the overriding concern for designers. The goal was to use as few flip-flops as possible. In such devices, the required number of states (S) is encoded into N flip-flops, where N is the smallest integer such that $S<=2^N$. The actual control signals must be decoded from the state machine inputs and registers. This adds extra latency time.

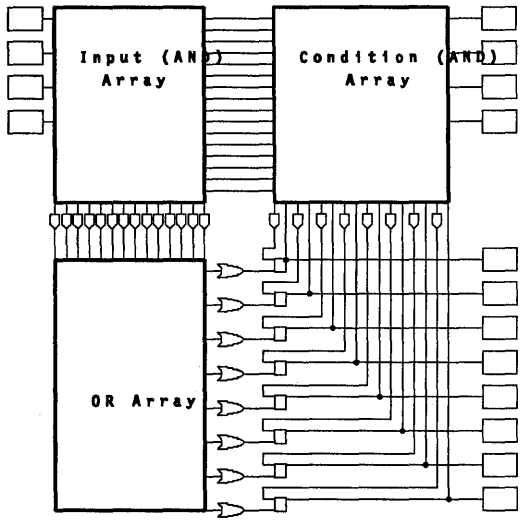


Figure 3. A Simplified FPLS Block Diagram

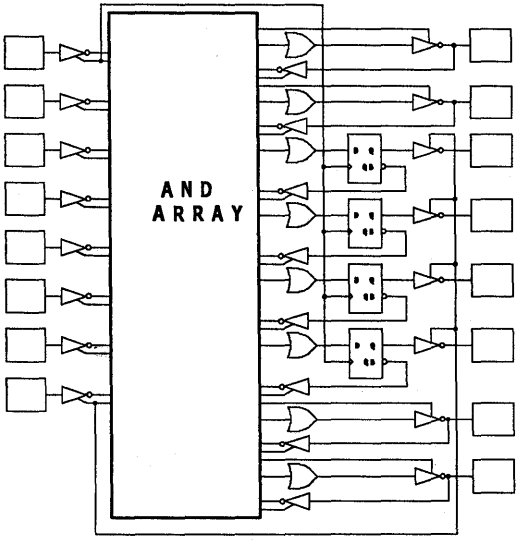


Figure 2. A Simplified PAL Block Diagram

With the advent of high-density PLDs and the shrinking of cycle times, the minimal-flip-flop strategy is no longer viable. A Petrie net (see *Reference*) or token-passing methodology suits high-speed state machine design better. In the token-passing methodology, each state has its own register, and these registers are directly connected, as in a shift register. Passing from one register to the next, a token signifies the present state by its position. Branching results from passing the token to a new process, which is enabled by an input condition.

This approach removes the necessity for the encoding/decoding logic in the traditional state machine in two ways: because the token passes directly from one state register to the next, and usually the control signals can be taken directly from the state-register outputs.

The CY7C361 Architecture

Cypress Semiconductor developed the CY7C361 architecture by modifying and streamlining the architectures discussed earlier. Direct connections between state register macrocells allow implementation of the token-passing methodology. But the CY7C361 removes the FPLS's primary speed barrier—that all inputs and

feedback propagate through two arrays before reaching the registers. The CY7C361 removes the barrier by placing the state register macrocells between the two arrays, with feedback going directly to the input array. This strategy cuts ts in half and minimizes t_{CF}, along with providing state-decoding logic on chip, if needed.

Even reduced by half, however, ts is still a dominant factor in the f_{MAX} calculation shown in Eq. 1. But because the propagation delay through a programmable array is directly proportional to the array's size, streamlining the array further reduces ts. In the CY7C361, state-register feedback accounts for 64 array inputs; actual chip inputs account for only 24 array inputs. 88 inputs makes for a fairly large array. How can the array size be reduced without sacrificing inputs or state registers?

You can implement most state machines with four or fewer registers, and you can usually break larger state machines into several smaller processes that pass control back and forth. And although all registers need to have access to the inputs, in most cases feedback is local to a specific process.

The CY7C361 takes advantage of these facts by implementing feedback in stages. The state-register macrocells have been separated into eight groups of four, each with its own local reset. In each of the groups, one register has feedback available to all 32 registers, one register has feedback available to a group of 16 registers, and the other two registers have feedback available to eight registers. You can break a large state machine into several processes, with local feedback used within an individual process. Arbitration among the smaller processes is accomplished with global feedback or the direct connections between adjacent state macrocells. This distribution allows the effective array size to shrink to 56 input lines, without sacrificing the number of inputs. *Figure4* illustrates the concept.

The Condition Decoder

Array size has two components, the number of inputs and the number of product terms. In PAL architecture design, one of the most critical tradeoffs is speed vs. the number of product terms. Thus, the second part of the challenge of minimizing ts is to provide only as many product terms as needed to implement state machines. Most PALs offer a minimum of seven product terms per register. (The CY7C330 offers a range between nine and 19.) However, seven product terms for each of 32 registers (224 product terms total) is obviously not the answer for a high-speed device.

Because the CY7C361 is designed specifically for state machine applications, Cypress Semiconductor analyzed state machine operations to find out what logical functions are necessary for state machine implementation. Cypress found that all state machine operations fall into two classes: entering a new state from one of several states based on a condition or leaving the present state for one of several other states based on a condition.

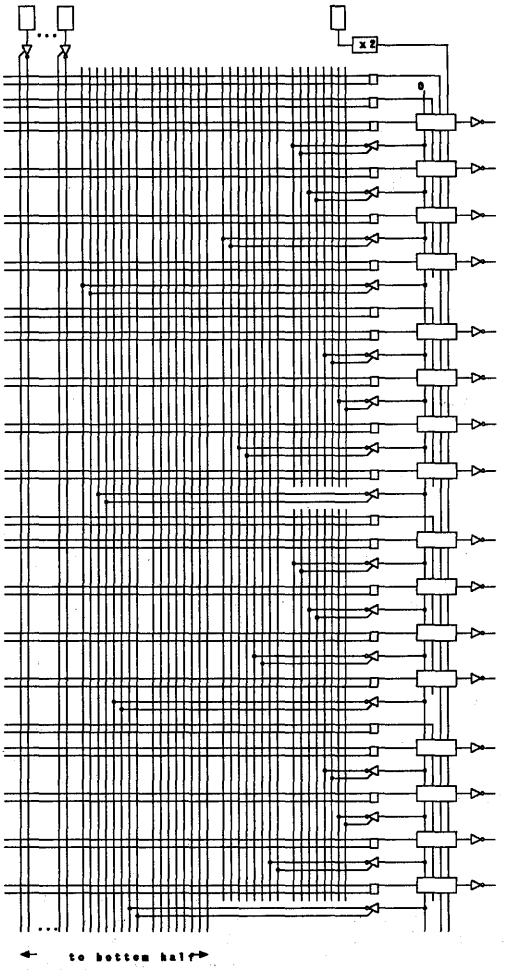


Figure 4. Global vs. Local Feedback

As illustrated in *Figure 5*, both operations lend themselves to the form:

(a+b+c..+n) & (N..& X & Y & Z) Eq. 2

Thus, in state machine applications, the standard sum of products construct can be replaced with the more efficient construct shown in *Figure 6*. The CY7C361 uses the version pictured on the right, because that implementation is faster in CMOS. This circuit is called the Condition Decoder.

Replacing the standard sums of products with condition decoders reduces the number of terms in the array to 64 for state registers, plus 16 terms used for local resets, and two terms for the global reset. This permits a total input array size of 56 x 82. This is small

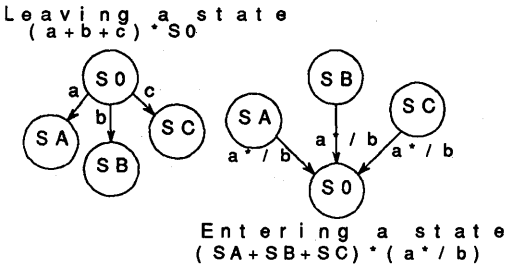


Figure 5. The Two Kinds of State Machine Operations

enough to make possible a $t_S + t_{CF}$ of 8 ns or under. A t_P of 8 ns means an f_{MAX} of 125 MHz.

The Output Array

The CY7C361's output array is OR based. The state macrocell outputs are driven in complemented form only, and the pure and bidirectional outputs are product terms. This structure results in a logical NOR for the overall output array function because:

$$!A \,\&\, !B = !(A + B) \qquad \text{Eq. 3}$$

The output enables are Low asserted and fed by product terms. When taken with the complemented inputs to the output array, the output enables are an OR function of any state register output(s). Mealy output terms are NAND terms (more on this later). Each of the output types appears in *Figure7.*

State Macrocells

As mentioned earlier, the CY7C361 has 32 state macrocells. The state macrocells each have a single condition-decoder input, and they all share the same clock and a global reset-condition decoder. For each group of four state macrocells there is also a local reset-condition decoder. Additionally, each state macrocell has a C_IN input and a C_OUT output that connect macrocells to their adjacent macrocells. In addition to C_OUT, the macrocell output is driven directly back to

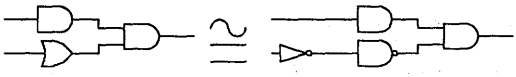


Figure 6. The Condition Decoder

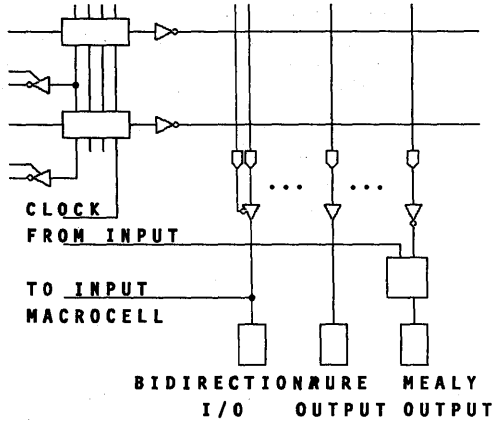


Figure 7. Portions of the Output Array and Output Types

the input array, in both true and complement form, and to the output array in complement form only.

There are three possible configurations for the state macrocell: START, TOGGLE, and TERMINATE.

The START Configuration

Figure 8 shows the CY7C361 state macrocell in its START configuration, which causes the macrocell to act like a one-shot circuit. Configuration bit C2 selects whether C_IN is a logic 0 or C_OUT from the previous macrocell. The activating signal is the logical OR of the C_IN signal and the condition decoder. When these signals activate the input, the macrocell output is asserted for one clock period only. You can use this macrocell configuration to start a process or as a state in an unbranching sequence.

The TOGGLE Configuration

The CY7C361 state macrocell in its TOGGLE configuration acts like a toggle flip-flop. Once again, configuration bit C2 selects whether C_IN is a logic 0 or C_OUT from the previous macrocell. The activating signal is the logical OR of the C_IN signal and the condition decoder. While this input is active, the macrocell changes state on the rising edge of every clock. If the input is not active, the macrocell retains its state (see *Figure9*). You can use the TOGGLE configuration for binary counters and other traditional state machine implementations.

The (Wait Until) TERMINATE Configuration

The third configuration of the CY7C361 state macrocell is TERMINATE. The TERMINATE configuration differs from those already described in that you *must* configure C2 such that C_IN is the C_OUT of the previous macrocell. Asserting C_IN activates the circuit, causing the output to become asserted on the next

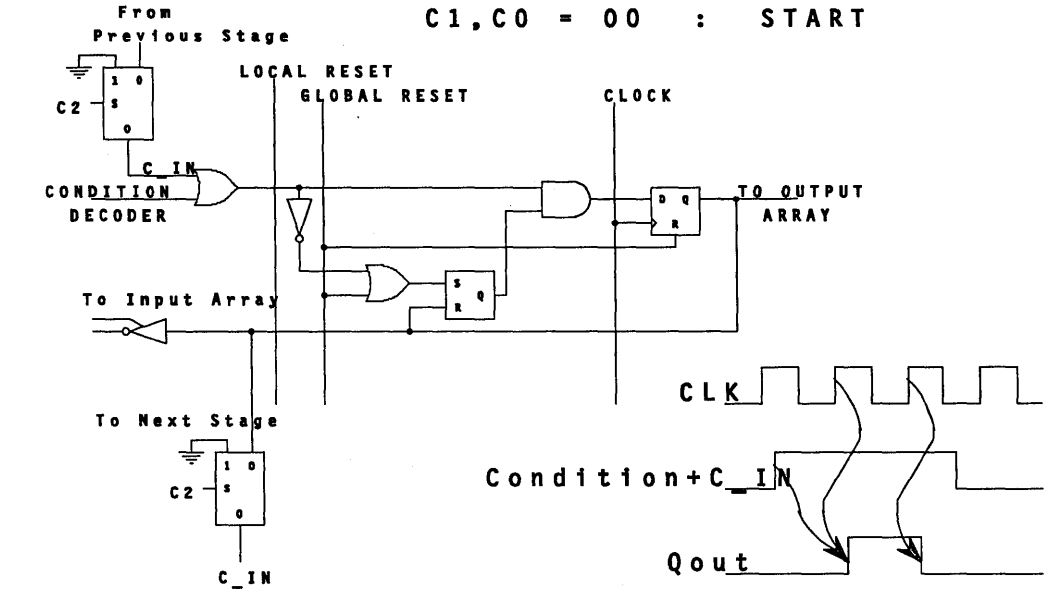


Figure 8. The START Macrocell Configuration

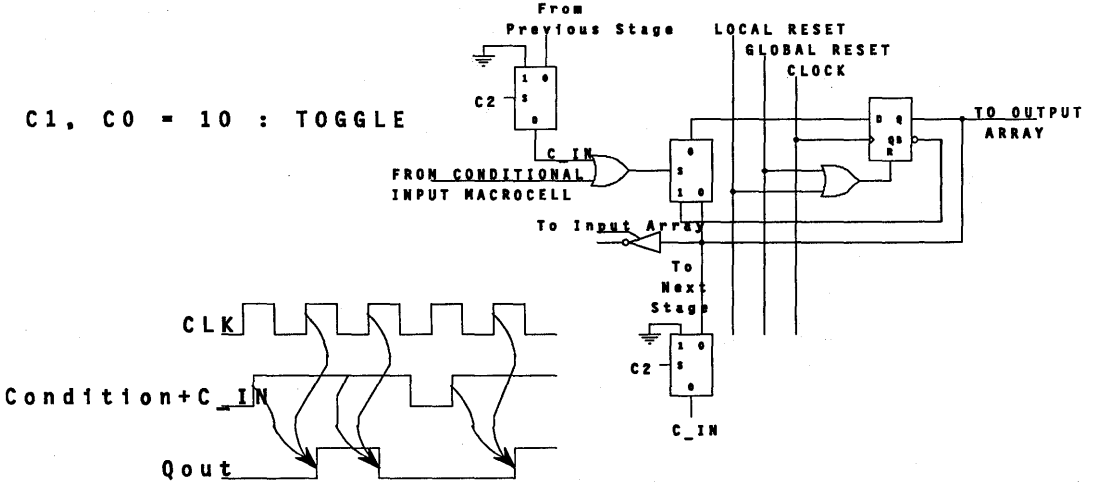


Figure 9. The TOGGLE Macrocell Configuration

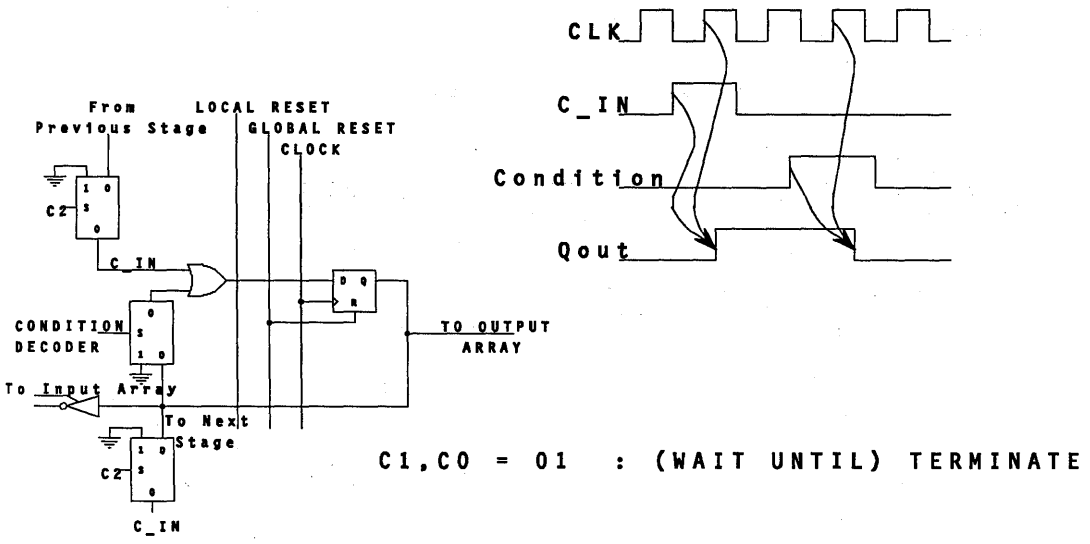


Figure 10. The TERMINATE Macrocell Configuration

rising clock edge. The macrocell remains in this state until the condition decoder becomes asserted. This causes the output to be deasserted (terminated) on the clock's next rising edge. *Figure 10* shows this configuration. You can use TERMINATE to insert wait states in a process.

The Input Macrocell

The CY7C361 contains 12 input macrocells, of which four are straight inputs, four are inputs from pins that are also connected to the Mealy macrocells, and four constitute the input path of the bidirectional pins.

The input macrocell of the CY7C361 avoids metastability problems and provides for flexibility in the timing of inputs. Metastability has always been a problem in asynchronous systems, but with cycle times shrinking dramatically, metastability is becoming more of an issue in high-speed synchronous design as well. The CY7C361 inputs are designed to be very metastability resistant. That is, metastability occurs rarely, and when it does happen, the device resolves it quickly.

The input macrocell (*Figure 11*) has three possible configurations. The first (default) is a nonregistered configuration that can be used to cascade the CY7C361 with other high-speed devices. If you use the CY7C361 inputs in this mode, however, be careful not to violate the state registers' set-up and hold time specs. This timing is tight, and violating either spec could lead to metastability conditions in the state macrocells. The second configuration of the input macrocell is single-registered mode, which you can use for pipelining inputs. The last configuration is double-registered mode. It is used to synchronize asynchronous signals. *Note that the clock for the registered modes is the same as the internal state clock.* Thus, if you enable the clock doubler, the input registers are clocked at twice the frequency of the external clock. The input-macrocell configuration-bit settings appear in *Table 1*.

The input macrocells also have a clock-enable function. Each of the three groups of macrocells has its own input enable, which come from three product terms in the output array. Because the enables are Low asserted, and the output array inputs are all in complemented

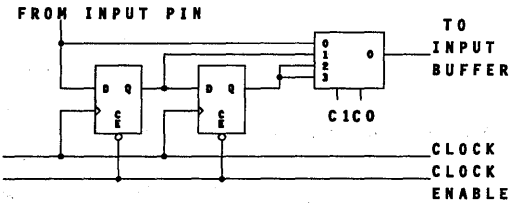


Figure 11. The CY7C361 Input Macrocell

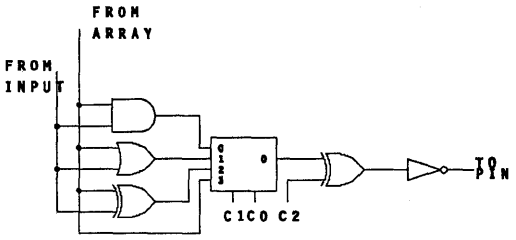


Figure 12. The CY7C361 Mealy Macrocell

form, the logical function of these clock enables is an OR.

The Mealy Macrocell

The CY7C361 provides four macrocells that allow you to build Mealy machines—one of the two general classes of state machine. (The other is the Moore machine, in which outputs depend only on the present state of the registers.) In a Mealy machine, outputs depend on both present state information and the state machine inputs. The CY7C361's Mealy macrocell appears in *Figure 12*.

The Mealy macrocells have two inputs, one from the output array and the other from a dedicated input pin (before the macrocell). As mentioned earlier, the array outputs have an extra inverter in the path. This is because the Mealy macrocells have programmable polarity; the default configuration has an inverter in the path. These two inverters cancel each other, effectively making the Mealy macrocell's straight output a logical NOR, just like the other outputs.

In addition to programmable polarity, the Mealy outputs offer configurations of output only; AND of input and output; OR of input and output; or XOR of input and output. *Table 2* shows the Mealy macrocell configuration bit settings.

Table 1. Input Modes for the CY7C361

C1	C0	Input Macrocell Mode
0	0	Combinatorial
0	1	Single Registered (Pipeline Mode)
1	X	Double Registered (Synchronizer Mode)

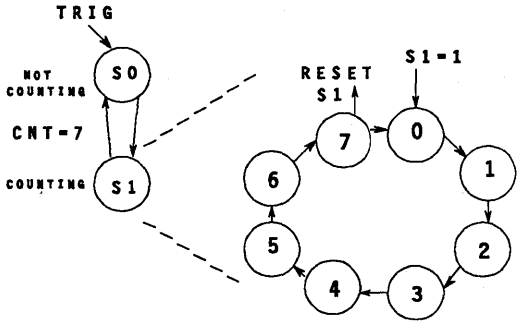


Figure 13. State Diagram of the Pulse-Triggered Counter

Example: Pulse-Triggered Counter

An example application of the CY7C361 illustrates the use of most of the features discussed in this application note. In this circuit, an asynchronous trigger, !TRIG, starts a 3-bit binary counter. Because only one count cycle can be triggered at a time, the circuit ignores any !TRIG pulses received in the middle of a count cycle. !TRIG is synchronized using the double-registered input macrocell configuration.

The counter outputs, COUNT(2:0), cycle from 0 to 7 and reset to 0. Note that each instance of COUNT(2:0) is High asserted. The outputs are thus assigned to Mealy macrocells, where the polarity can be controlled.

When the circuit accepts a !TRIG pulse, the fourth Mealy output is used to generate an acknowledge signal, !TRGACK. The clock for this circuit is called CLKIN, and the clock doubler circuit is activated. !RST is a Low-asserted global reset signal. Because the timing of !RST is not critical for this example, the input macrocell is configured as combinatorial.

Table 2. Mealy Macrocell Configuration Settings

C2	C1	C0	Mealy Configuration
0	0	0	pin = input NAND array out
0	0	1	pin = input NOR array out
0	1	0	pin = input XNOR array out
0	1	1	pin = INV array out
1	0	0	pin = input AND array out
1	0	1	pin = input OR array out
1	1	0	pin = input XOR array out
1	1	1	pin = array out

Two state machines implement this design. The first, a supervisory machine, consists of two states: S0 and S1. S0 is a START macrocell, triggered by !TRIG. S1 is a (wait until) TERMINATE macrocell, with C_OUT from S0 connected to C_IN. Thus, !TRIG initiates a token in S0, which then passes the token to S1. S1 acts as an enable for the counter, which is implemented using three TOGGLE macrocells, C(2:0). The outputs COUNT(2:0) come directly from C(2:0). When the counter reaches 7, S1's terminate condition is met and the circuit is ready for the next !TRIG. *Figure 13* shows the state diagram for this example.

!TRGACK is to be asserted only when the circuit receives !TRIG and S1 is not asserted. The Mealy macrocell is chosen so that !TRIG is the input, and it is programmed as an OR gate. The output term is !S1.

Figure 14 shows the physical implementation of this circuit, and *Appendix A* lists the PLD ToolKit source file. The source file's CONFIGURE section must list configuration information for all the state machine's macrocells. For the CY7C361, this includes input macrocells, state macrocells, Mealy macrocells, and the clock doubler.

This example uses two input macrocell configurations. When configuring an input register, the default is combinatorial. To override the default, specify IREG for a single register or IIREG for double registers. In this example, !RST is assigned to node (pin) 12 and configured as single registered. !TRIG is assigned to node (pin) 13 and configured as double registered.

CLKIN is assigned to node 4, the dedicated clock input. Node 74 is the clock doubler. The default is that the doubling function is not enabled. To enable it, assign the DBL_CLK attribute to node 74.

The internal state macrocells for this design are S(1:0) and C(2:0). S0 is assigned to node 32 and configured as a start macrocell by specifying the START attribute. START is the default, but it is specified here for completeness. S1 is assigned to node 33, and configured as a terminate macrocell with the attribute

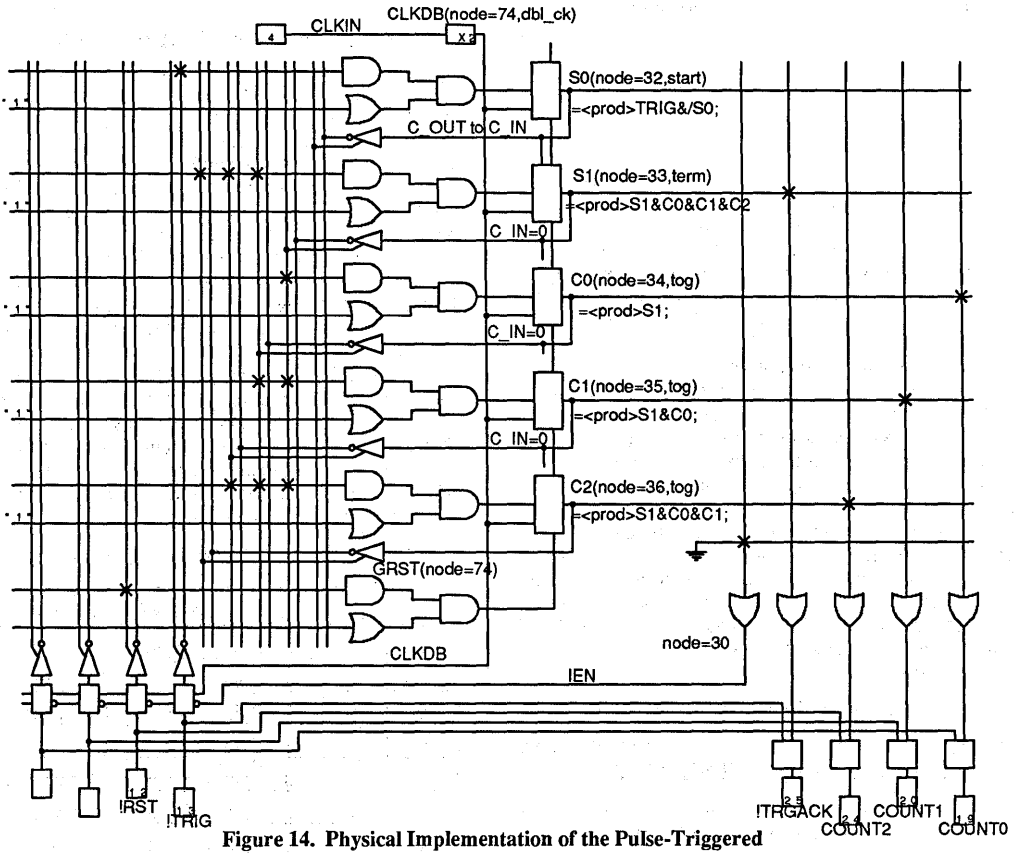


Figure 14. Physical Implementation of the Pulse-Triggered Counter

TERM. CIN must also be specified for every macrocell configured as terminate. The C(0:2) macrocells are assigned to nodes 34, 35, and 36. Because these macrocells make up a counter, they are configured as toggle macrocells by specifying the TOG attribute.

This example uses all four Mealy macrocells. The COUNT(2:0) macrocells are assigned to nodes 19, 20, and 24. In this case, no logical function is used. This is the default, although it is not configuration 00. The Mealy macrocells are used to make the outputs High asserted. The toggle macrocells, C(2:0), are inverted going into the output array, then inverted again going into the Mealy macrocell. The macrocell contains one other inverter, which the attribute NINV bypasses.

!TRGACK, the OR function of !S1 and !TRIG, is assigned to the fourth Mealy macrocell at node 25. The OR attribute is specified, and the NINV attribute bypasses the inverter in the Mealy macrocell path.

IEN and GLBRST are both internal nodes. IEN is assigned to node 30, which is the input-enable term for pins 10 - 13. GLBRST is the global reset term for all internal registers. Node 73, OFF in this example, is used for tying anything in the output array Low.

The source file's EQUATIONS section contains equations for both the condition-decode array and the output array. The legal connectives for the condition-decode array are <PROD> for the AND product term and <INV_PROD> for the NAND (or OR) product term. This example uses only <PROD> terms, and the logic/miser bits are automatically programmed to enable the condition decoders.

The two legal connectives for the output array are < INV_SUM> , which is used for all outputs and input enables, and <INV_OE>, which is used for the output enables on the bidirectional pins. The output-array connectives are somewhat confusing unless you remember that the entire output array is an OR array, and all the state inputs are Low-asserted only. Thus, if you wanted to assert an output only while S1 is asserted, the equation would use <INV_OE> !S1. Because the output enable is Low asserted, the sum of !S1 and nothing else serves in this example.

The equations are fairly straightforward. TRIG triggers S0. S1 is terminated by C2, C1, C0 = 7. The toggle macrocells, C(2:0), are configured as a toggle counter with S1 as an enable. GLBRST is assigned to RST.

The input clock enable is always enabled. Because it is Low asserted, IEN is assigned to node 73, GND. The counter outputs, COUNT(2:0), connect directly to the outputs of the toggle macrocells, C(2:0), which are inverted. TRGACK is assigned to pin 25—the Mealy macrocell connected to pin 13, which is !TRIG. The array input is !S1. The NOR function is selected in the source file's CONFIGURE section.

Reference

Murata, Tadao, "Petrie Nets: Properties, Analysis, and Applications" (*Proceedings of the IEEE*, VOL. 77, NO. 4, April 1989)

Appendix A. PLD ToolKit Source File for Pulse-Triggered Counter

```
CY7C361;

CONFIGURE;

CLKIN(node=4),                          {system clock}

!RST(node=12,ireg),                     {low asserted reset, no input register}

!TRIG(iireg),                           {asynchronous trigger, iireg means double registered}

IEN(node=30),                           {node 30 is the clock enable for inputs 10-13}

CLKDB(node=74,dbl_clk),                 {node 74 is the clock doubler, enabled here}

COUNT0(node=19,ninv),                   {nodes 19,20,24,25 are the mealy outputs}

COUNT1(ninv),                           {COUNT(2:0) are not inverted, no logical function is used}

COUNT2(node=24,ninv),

!TRGACK(or,ninv),                       {!TRGACK is an OR function of pin 13 (!TRIG), and the
                                         output assigned below, no invert of output is performed}

S0(node=32,start),                      {state register 32, configured as START (one shot)}

S1(cin,term),                           {state register 33, C_IN enabled,
                                         configured as (wait until) TERMINATE}

C0(tog), C1(tog), C2(tog),              {state registers 34,35,36, configured as TOG (toggle flops)
                                         the internal counter}

GLBRST(node=64),                        {node 64 is the global reset condition decoder}

OFF(node=73)                            {node 73 is used to tie signals low}

EQUATIONS;

S0 = <prod> TRIG;                       {S0 is triggered by TRG}

S1 = <prod> C0 * C1 * C2;               {S1 is triggered by C_IN from S0, released when C(2:0)=7}

C0 = <prod> S1;                         {C0, least significant bit of counter, enabled by S1}

C1 = <prod> C0 * S1;                    {C1, middle bit of counter, enabled by S1 and C0}

C2 = <prod> C0 * C1 * S1;               {C2, most significant bit of counter, enabled by S1, C0, C1}

GLBRST = <prod> RST;                    {RST selected as a global reset}

COUNT0 = <inv_sum> /C0;                 {counter outputs, connected to /C0, /C1, /C2 respectively}

COUNT1 = <inv_sum> /C1;                 {inverted once more before mealy macrocell, }

COUNT2 = <inv_sum> /C2;                 {high asserted on pins}

IEN = <inv_sum> /OFF;                   {input clocks always enabled}

TRGACK = <inv_sum> /S1;                 {TRGACK is a mealy and of TRG and /S1}
```

Using the CY7C361 as an Mbus Arbiter

This application note discusses the use of the CY7C361 as a bus arbiter for a Cypress SPARC CY7C600 RISC-processor Mbus system. The Cypress CY7C361 is a very high-speed synchronous Erasable Programmable Logic Device (EPLD) optimized for state machine applications. The Cypress SPARC system utilizes a CY7C601 40-MHz RISC processor, a CY7C602 Floating Point Unit (FPU), four CY7C604 Cache Controller and Memory Management Units (CMU), and eight CY7C157 16K x 16 cache RAMs make up a 256-Kb cache. The arbiter resolves Mbus access contention for a system with four CMU bus masters. Refer to *Figure 1* for a block diagram of the Mbus system.

CY7C361 Brief Description

The CY7C361 is a high-performance PLD with 32 state macrocells, a condition-decode array, an output array, 12 input macrocells for eight dedicated inputs and four bidirectional inputs, six dedicated outputs and four Mealy output macrocells. The CY7C361 also has a clock-doubler circuit, which allows up to 125-MHz internal operation. Packaged in a 28-pin, 300-mil DIP or LCC/PLCC package, the CY7C361 is manufactured using a CMOS 0.8-micron, double-metal-processing technology that is UV erasable. Please consult the Cypress application note, "Understanding the CY7C361," for an in-depth description of the CY7C361 architecture.

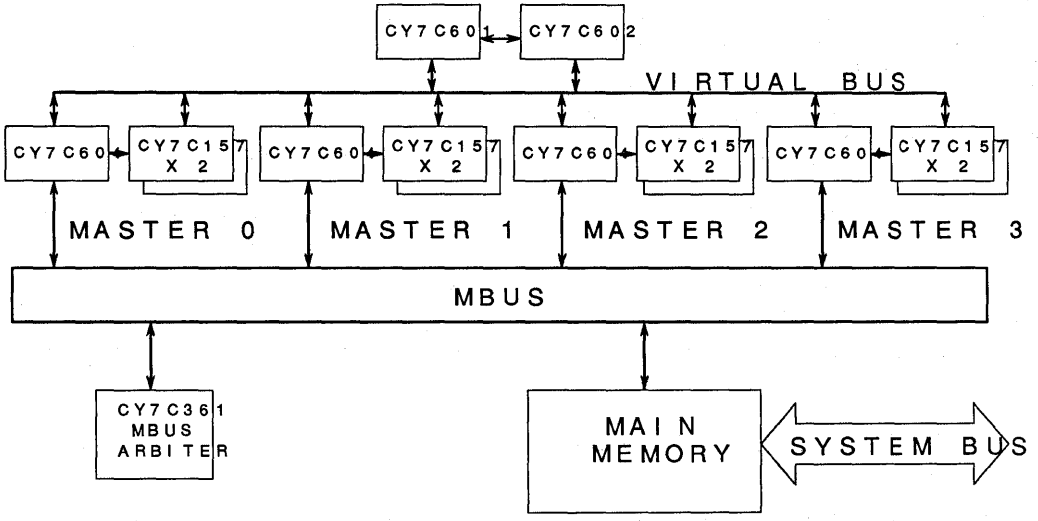


Figure 1. Mbus System Block Diagram

Mbus Description

The Mbus is a SPARC standard main-memory interface for the Cypress SPARC Cache/Memory Management Unit device (the CY7C604). The M in Mbus stands for module and emphasizes the multiprocessor module support that SPARC offers. It is a high-speed, synchronous, 64-bit, multiplexed address and data bus that operates at the CY7C601's clock rate.

Mbus accesses are initiated by a master and responded to by a slave. Generally a bus transaction takes place between a master and main memory, but in the case of direct data intervention, transactions can occur between masters. The handshake between the CY7C604 CMU and the arbiter consists of a request line (/MBR(3:0)) and a grant line (/MBG(3:0)) for each master. A busy line (/MBB) is common to all masters and indicates that the bus is in use.

Mbus arbitration uses the following procedure: A master asserts its request line. The arbiter decides whether to grant the request. The request is granted when the arbiter asserts the master's corresponding grant. As soon as the grant is received, the master can deassert its request. The newly granted master watches the /MBB (busy) signal. When /MBB is deasserted, the new master must drive /MBB Low on the next clock cycle to take control of the bus or risk losing its chance for mastership. The new master can now start its bus transaction. When the transaction is completed, the master deasserts /MBB. The arbiter continues to assert the grant until another request is received. This allows the master to perform multiple transactions without repeating the arbitration sequence. Refer to *Figure2* for the Mbus multiple request sequence.

Mbus transfers are synchronous with respect to the system clock. The data transactions across the bus consist of a single-clock-period address phase and a multiple-clock-period data phase. Data transfers can occur in word (64 bit), multi-word-burst, or atomic-load-store formats. All signals are valid and sampled on the system clock's rising edge. The Memory Address Strobe (/MAS) signal validates the address phase and denotes the start of the actual data transfer. Three status lines indicate bus states and convey the current bus operation, as well as error status. *Figure 3* shows the Mbus data-transfer waveforms.

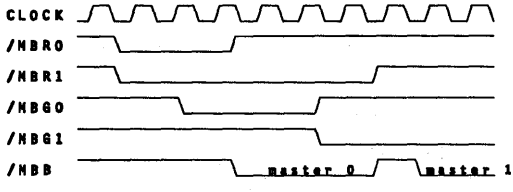


Figure 2. Mbus Multiple Request Sequence

By design, the details of bus mastership and resolution of multiple requests are handled outside the realm of Mbus and SPARC. This approach allows you to implement any arbitration scheme that suits the system requirements.

Two arbitration schemes fit the Mbus specification. The simplest is concurrent arbitration, in which the bus is granted to a master, and the master performs its bus transaction. If requests are pending when the master completes its transaction, the bus is re-arbitrated, and the new master takes over. In this arbitration scheme, the current master's grant is asserted during the bus transaction and deasserted after the transaction is finished. The bus arbitration happens between bus transactions, causing several cycles of latency between transfers.

The second arbitration scheme is pre-arbitration, which is more efficient on Mbus but trickier to implement. In pre-arbitration, the arbitration happens before the previous bus cycle completes. The bus is granted to a master, and the master starts its transaction. If other requests are pending, the grant is withdrawn and the bus is rearbitrated. Once the new master receives its grant, it waits until /MBB is deasserted and then takes control of the bus by asserting /MBB on the next cycle. At this point the bus can be arbitrated again. This means that as long as requests are pending, /MBB is inactive for, at most, one cycle at a time. This takes more work to implement because the arbiter must have

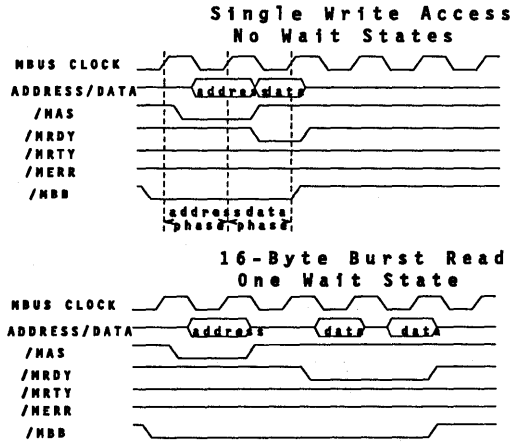


Figure 3. Mbus Data Transfer Waveforms

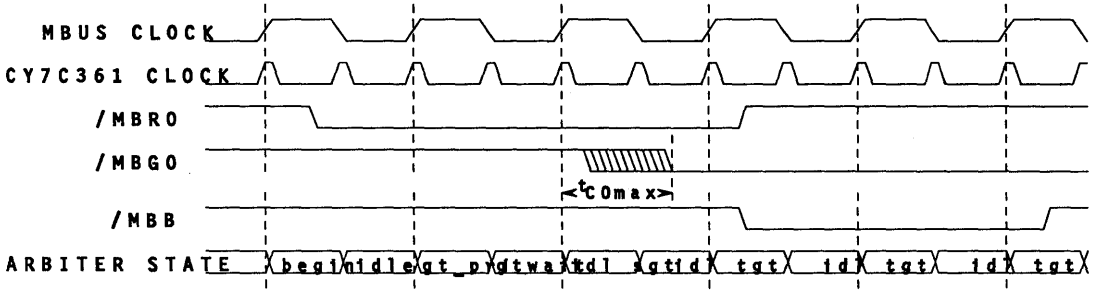


Figure 4. CY7C604 & CY7C361 Timing for Master 0

two different arbitration modes: one for when the bus is idle, (e.g., immediately after power up), the second for normal operation.

The arbiter described in this application note uses the pre-arbitration scheme.

Timing Considerations

Because this arbiter uses pre-arbitration, the arbiter need not be able to accept a request, resolve access contention, and grant bus rights to a master in a single Mbus clock cycle. In this application, the arbiter's clock input runs at the same 40-MHz clock rate used by the CY7C601 and the CY7C604s. This clock rate allows the arbiter inputs to meet the timing requirements of the Mbus masters. Internally, the CY7C361 that implements the arbiter is clocked at 80 MHz by virtue of the on-chip clock doubler.

Figure 4 shows the timing relationship between master 0 (a CY7C604 at 40 MHz) and the CY7C361 arbiter. This figure also illustrates the first arbitration cycle after a reset.

Arbitration Scheme

With the arbitration function left to the designer, there are several resolution techniques you can employ. Fixed priority, rotating priority, least recently used, and random priority are all contention-resolution schemes that have proven successful, yet each has its own faults. A fixed priority, for instance, favors one requester more than the others. Rotating priority provides a simple, but not always fair approach to arbitration. A least recently used arbitration scheme represents the fairest form of contention resolution but requires a highly complex implementation. The random technique does not guarantee arbitration results. To help simplify this example, it uses fixed priority, with master 0 having the highest priority, and master 3 the lowest.

Design Partitioning

The design is partitioned into three functional blocks (*Figure 5*). The first block is the condition decoder or input array. The second block is the handshake state machine, which keeps track of requests (/MBR(3:0)) and the bus-busy input (/MBB). The CY7C361's state macrocells implement this block. The third block is implemented in the output array, which generates the grant (/MBG(3:0)) signals that give an Mbus master ownership of the bus.

Handshake State Machine

Because the condition decode array uses the feedback from the handshake state machine, consider the state machine first. The machine controls Mbus handshake and arbitration. The arbiter cycles through 26 discrete states in performing its function and thus takes 26 state macrocells to implement. The first two states, BEGIN and IDLE, can be thought of as supervisory states. Their state diagram appears in *Figure 6*. Each master has its own grant sequence of six states for a total of 24 (*Figure7*).

You need to consider a number of issues when fitting a design into the CY7C361. The first is the placement of the state macrocells with relation to each other. All (wait until) TERMINATE macrocells must be preceded by another macrocell to provide a way to pass the token into the cell.

Because no grants are asserted on power-up or reset, the machine starts a token in BEGIN and passes the token to IDLE on the next clock. IDLE is a (wait until) TERMINATE macrocell. This means that the previous state macrocell (BEGIN in this case) passes a token to IDLE, which keeps the token until certain conditions are met (more on this shortly).

When the machine receives a request (/MBRn, where n = 0, 1, 2, 3), priority is decided and a token is created in a START macrocell called GTn_PRI. Because the priority selection happens as a condition to GTn_PRI, a condition for GT0_PRI is that /MBR0 is asserted. Further, a condition for GT3_PRI is that /MBR3 is asserted and not the other requests.

The token from GTn_PRI is immediately passed to TERMINATE macrocell GTn_WAIT. The GTn_PRI macrocell must be placed first, because it is the START macrocell that creates the token. GTn_PRI is immediately followed by the GTn_WAIT macrocell, which is essentially a timing loop that waits until the bus can be

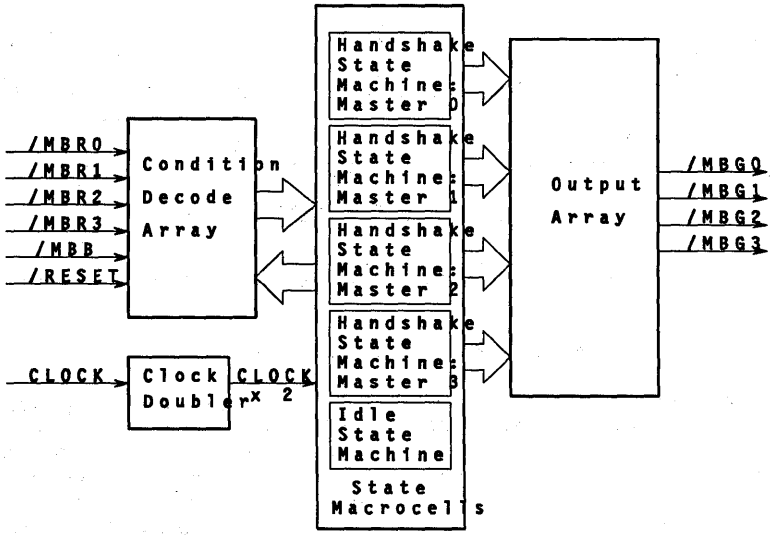


Figure 5. Arbiter Block Diagram

arbitrated. GTn_WAIT terminates under two conditions: when IDLE is active, meaning that no bus activity is occurring or when /MBB goes active, meaning that the bus can be arbitrated. At this point GTn_WAIT passes the token to one of two processes.

If IDLE is still active, the token passes to IDLn_SGT. This is another START macrocell, which passes the token to the adjacent TERMINATE macrocell, called IDLn_TGT. These two states produce the /MBGn signal. IDLn_TGT and IDLE both terminate when /MBB is asserted, (meaning that master n is now controlling the bus), and any GTn_WAIT state is asserted (meaning that another request is pending).

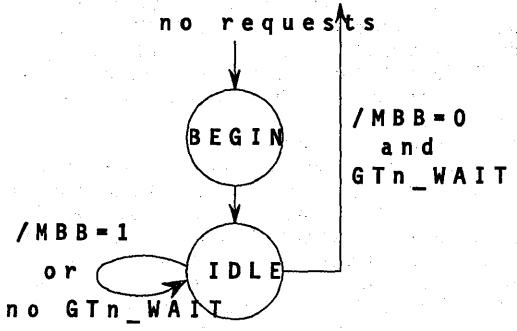


Figure 6. Supervisory State Machine: Begin/Idle

This sequence is only used for the first two arbitrations immediately after a reset, or if there has been a lapse in bus requests. The sequence finishes as soon as the master takes control of the bus.

If IDLE is not active, GTn_WAIT passes the token to NRMn_SGT. This is a START macrocell, which passes the token to the adjacent TERMINATE macrocell called NRMn_TGT. Like the equivalent IDLn_SGT and IDLn_TGT states, these two states produce the /MBGn signal. However, NRMn_TGT terminates when /MBB is deasserted (signaling that the current bus transaction has completed and master n will control the bus next), and any GTn_WAIT state is asserted (indicating another pending request). The next grant is asserted as soon as master n asserts /MBB.

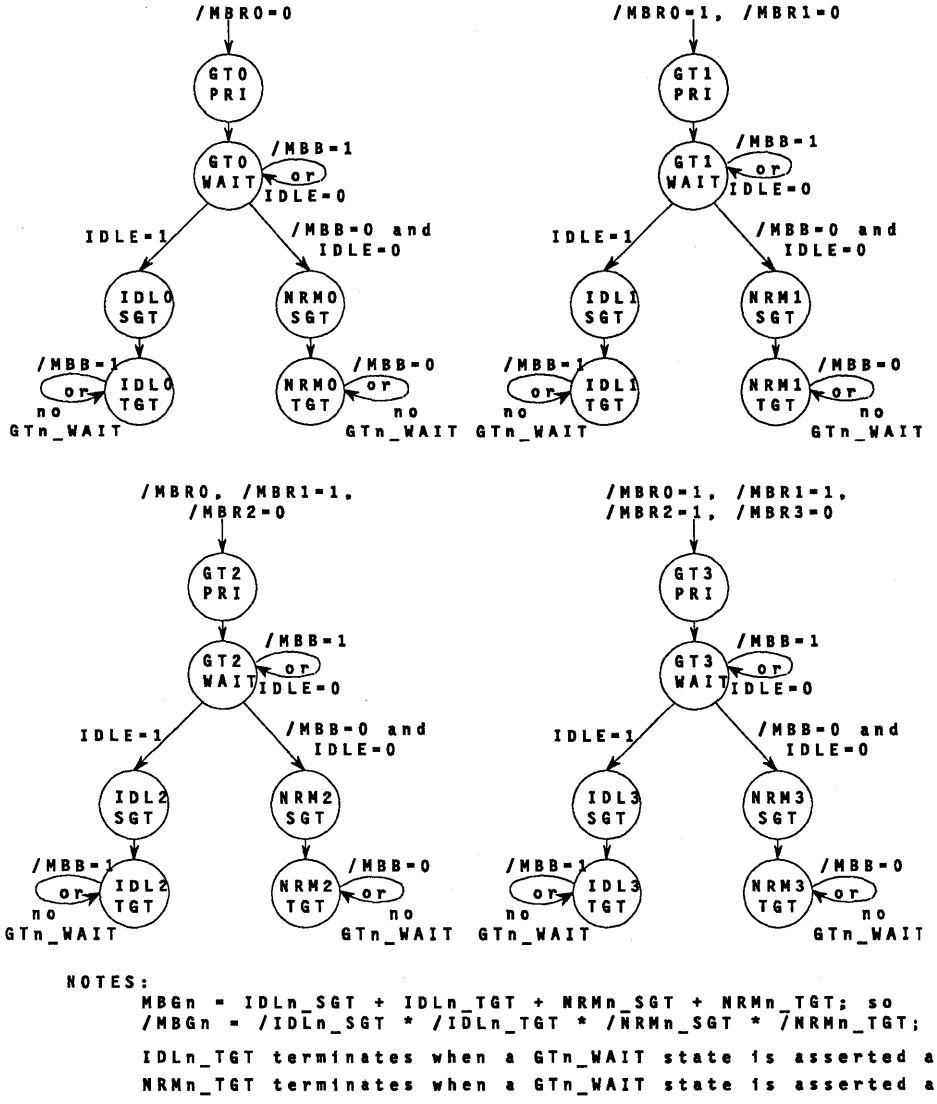
This sequence is the normal mode of operation. It terminates as soon as the previous bus transaction completes. On the next cycle, the granted master takes control of the bus while the arbiter is issuing the next grant.

Note that both modes allow for bus parking, which allows a master to do multiple bus transactions without re-arbitrating, so long as no other requests are pending. This is why GTn_WAIT is used to terminate the grant line.

Figure 8 shows the waveforms for two consecutive arbitrations, the first starting from the IDLE state and using the idle mode, the second proceeding in normal operation mode.

The Condition Decode Array

The condition decode array implements the control logic for the handshake state machine. This array's inputs consist of the true and complement of all input pins along with the true and complement of the state



NOTES:

MBGn = IDLn_SGT + IDLn_TGT + NRMn_SGT + NRMn_TGT; so
/MBGn = /IDLn_SGT * /IDLn_TGT * /NRMn_SGT * /NRMn_TGT;

IDLn_TGT terminates when a GTn_WAIT state is asserted and /MBB=0.

NRMn_TGT terminates when a GTn_WAIT state is asserted and /MBB=1.

Figure 7. State Diagram of the Mbus Arbiter

macrocell feedback. The state macrocell feedback terms are not all global inputs, however. Every fourth macrocell has global feedback, and every third macrocell has feedback to 16 of the 32 macrocells. The rest of the macrocells have local feedback within their group of eight. Placement of the states so that there is adequate feedback is one of the biggest challenges when using the CY7C361. The global feedback in the Mbus arbiter comes from the GTn_WAIT states.

The other array inputs are /MBB, /RESET, and /MBR(3:0). The /MBR(3:0) inputs are single registered. Because the input registers have very small set-up and hold times (t_{SU} = 2 ns, t_H = 3 ns), metastable events are unlikely. The combinatorial nature of /RESET makes it suitable to work as a non-registered signal. /MBB, on the other hand, is double registered because of its importance to the design's internal processes. If you can guarantee that /MBB will not change until after the falling edge of the system clock, you can single-register /MBB for slightly better performance.

All the array inputs mentioned above are Low asserted. You specify this fact in Cypress PLD ToolKit syntax by the preceding slash when you define the signals in the source file's configuration section. In the source file's equation section, because all signals are treated as High asserted, specifying MBB in an equation is the same as saying that /MBB = 0 on the pin.

Each state macrocell has two possible inputs. One input comes from the previous macrocell. You enable this input by specifying CIN in the macrocell's configuration. A TERMINATE macrocell requires the CIN designation; otherwise the macrocell cannot be set.

The other state macrocell input comes from a condition decoder, which is the output of the condition-decoder array. A condition decoder is the output of an AND function, with inputs from a NAND (or INV_PROD) term and an AND (or PROD) term. OR functions are accomplished by inverting the inputs to the INV_PROD term.

The BEGIN state starts a token after a reset, when none of the requests (/MBR(3:0)) are asserted. This state passes the token to IDLE, where it stays until terminated by the combination of a GTn_WAIT state and the /MBB line being activated. The PLD ToolKit code for this is:

```
BEGIN =   <PROD> /MBR0 * /MBR1
            * /MBR2 * /MBR3;

IDLE =    <PROD>   MBB
        <INV_PROD> /GT0_WAIT * /GT1_WAIT
            * /GT2_WAIT * /GT3_WAIT;
```

Keep in mind that, because BEGIN is a START macrocell, BEGIN is asserted for one cycle after its condition is true. Because IDLE is a TERMINATE

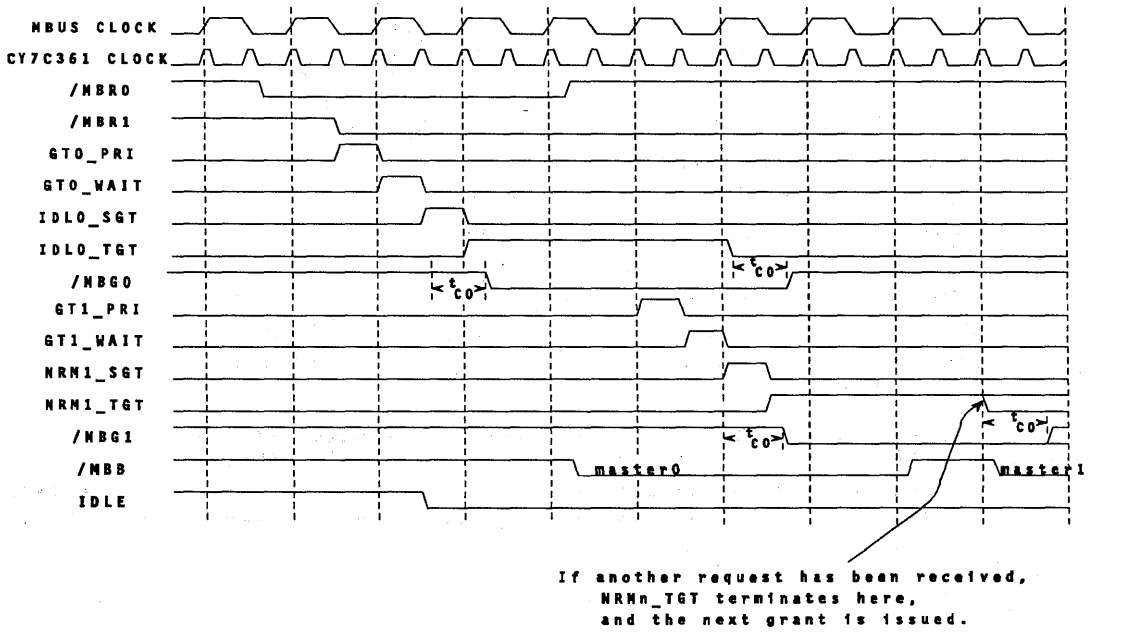


Figure 8. Handshake State Machine Waveforms

macrocell, it is asserted after the previous macrocell is asserted; IDLE is deasserted when its condition is true.

The machine enters the GTn_PRI state if this state's request is the highest priority received. GTn_PRI passes the token directly to GTn_WAIT, which terminates when either /MBB goes active (meaning that a bus transaction has started) or IDLE is active (indicating that no bus transactions have taken place). The PLD ToolKit code for GT3_PRI and GT3_WAIT is:

```
GT3_PRI =     <PROD> /MBR0 * /MBR1
                  * /MBR2 * MBR3;

GT3_WAIT =    <PROD>
              <INV_PROD> /MBB  *  /IDLE;
```

Remember that <INV_PROD> is a NAND term, and therefore the expression above means that GT3_WAIT is terminated when MBB or IDLE is asserted (according to the DeMorgan theorem). Also note that the GT3_WAIT equation specifies <PROD> without an expression. This automatically sets the AND term to a logical 1. If the equation did not include <PROD>, PLD ToolKit would assume that <PROD> equals 0, which would cause the condition decoder to always be false and GT3_WAIT to never terminate.

At the same time that GTn_WAIT is terminated, a new token is started in one of two macrocells. In the first case, IDLE is asserted, and the token is created in IDLn_SGT, a START macrocell. The condition for entering IDLn_SGT is that IDLE and the corresponding GTn_WAIT state are asserted. IDLn_SGT then passes the token to the TERMINATE macrocell IDLn_TGT. IDLn_TGT terminates when MBB is asserted and one of the other handshake processes is in its GTn_WAIT state. The equations are:

```
IDL3_SGT =    <PROD> GT3_WAIT  *  IDLE;

IDL3_TGT =    <PROD>    MBB
              <INV_PROD>  /GT0_WAIT
                 * /GT1_WAIT * /GT2_WAIT;
```

This sequence only happens immediately after a reset or if there has been a lapse in bus requests. The sequence finishes as soon as the master takes control of the bus.

The second state the machine can enter from GTn_WAIT is NRMn_SGT. This state is entered when the corresponding GTn_WAIT is asserted, MBB is asserted, and IDLE is not asserted. NRMn_SGT (a START macrocell) passes the token to NRMn_TGT, which is a TERMINATE macrocell. NRMn_TGT terminates when MBB is deasserted and one of the other GTn_WAIT states is asserted. The equations are:

```
NRM3_SGT  = <PROD> GT3_WAIT  *  MBB  *
/IDLE;

NRM3_TGT  =  <PROD>    /MBB
             <INV_PROD>  /GT0_WAIT
                * /GT1_WAIT * /GT2_WAIT;
```

This sequence is the normal mode of operation. The sequence terminates as soon as the previous bus transaction finishes. On the next cycle, the granted master takes control of the bus, while the arbiter is issuing the next grant.

The Output Array

The CY7C361's output array has the complements of all the state registers as its inputs. The terms are NAND based and connected directly to the output pins or Mealy macrocells, which makes the outputs an OR function of the state macrocells. /MBGn (where n = 0, 1, 2, 3) is produced by ORing states IDLn_SGT, IDLn_TGT, NRMn_SGT, and NRMn_TGT.

Design Verification

The entire CY7C361 Mbus arbiter design was entered using the Cypress PLD ToolKit and verified using the PLD ToolKit's interactive simulator. Working with a mouse and pop-down menus, the designer created the circuit stimuli by drawing waveforms on a graphics screen for a each CY7C361 node or pin. The PLD ToolKit's SIMULATE command then displays the response waveforms, promoting a high degree of confidence in the design's operation before programming a part. The PLD ToolKit source file can be found in *Appendix A.*

Reference

SPARC Mbus Interface Specification, Revision 1.1, Published March 29, 1990 by Sun Microsystems.

Appendix A. PLD ToolKit Source File for Mbus Arbiter

```
CY7C361;                                {Mbus Arbiter using the CY7C361}

CONFIGURE;                              {Configuration of macrocells}
                                        {Inputs}
CLK  (NODE=4),

/MBB  (IIREG),                          {MBus Busy can be single or double registered,
                                        depending on the system}
/RESET,
/MBR0  (NODE=10,  IREG),                {MBus Request from master 0, single registered}

/MBR1  (IREG),                          {MBus Request from master 1, single registered}

/MBR2  (IREG),                          {MBus Request from master 2, single registered}

/MBR3  (IREG),                          {MBus Request from master 3, single registered}

                                        {Outputs}
/MBG0  (NODE=16),                       {MBus Grant for master 0, low asserted output}

/MBG1,                                  {MBus Grant for master 1, low asserted output}

/MBG2,                                  {MBus Grant for master 2, low asserted output}

/MBG3  (NODE=28),                       {MBus Grant for master 3, low asserted output}

                                        {internal node configuration}
CKEN1  (NODE=30),  CKEN2,               {clock enables for input macrocells}

BEGIN  (NODE=34,  START),               {begin and idle are supervisory states}
IDLE  (TERM,  CIN),                     {idle is situated on a global feedback macrocell}

GT0_PRI  (NODE=38,  START),             {prioritization is a condition to this state, 0 is highest priority}
GT0_WAIT  (TERM,CIN),                   {waits until a grant can be issued, this macrocell has global feedback}

NRM0_SGT  (START),                      {These 2 states are the actual grant states for /MBG0 }
NRM0_TGT  (TERM,CIN),                   {during normal operation}

GT1_PRI  (START),                       {prioritization is a condition to this state, 1 is second highest priority}
GT1_WAIT  (TERM,CIN),                   {waits until a grant can be issued, this macrocell has global feedback}

NRM1_SGT  (START),                      {These 2 states are the actual grant states for /MBG1}
NRM1_TGT  (TERM,CIN),                   {during normal operation}

GT2_PRI  (START),                       {prioritization is a condition to this state, 2 is third highest priority}
GT2_WAIT  (TERM,CIN),                   {waits until a grant can be issued, this macrocell has global feedback}

NRM2_SGT  (START),                      {These 2 states are the actual grant states for /MBG2}
NRM2_TGT  (TERM,CIN),                   {during normal operation}

GT3_PRI  (START),                       {prioritization is a condition to this state, 3 is lowest priority}
GT3_WAIT  (TERM,CIN),                   {waits until a grant can be issued, this macrocell has global feedback}

NRM3_SGT  (START),                      {These 2 states are the actual grant states for /MBG3}
NRM3_TGT  (TERM,CIN),                   {during normal operation}

IDL0_SGT  (START),                      {These 2 states are the actual grant states for /MBG0}
IDL0_TGT  (TERM,CIN),                   {for the first two transactions after an idle state}
```

Appendix A. PLD ToolKit Source File for the Mbus Arbiter

```
                                        {internal node configuration data--continued}
IDL1_SGT (START),                       {These 2 states are the actual grant states for /MBG1}
IDL1_TGT  (TERM,CIN),                   {for the first two transactions after an idle state}

IDL2_SGT (START),                       {These 2 states are the actual grant states for /MBG2}
IDL2_TGT  (TERM,CIN),                   {for the first two transactions after an idle state}

IDL3_SGT (START),                       {These 2 states are the actual grant states for /MBG3}
IDL3_TGT  (TERM,CIN),                   {for the first two transactions after an idle state}

GRST  (NODE=64),                        {Global ReSeT node}

OFF  (NODE=73),                         {internal reference point}

CLK2X (NODE=74, DBL_CLK),               {the internal clock doubler is enabled}

EQUATIONS;                              {the equations for the part are specified here}

                                        {the condition decode array}
GRST =      <PROD>   RESET;

GT0_PRI =   <PROD>   MBR0;              {start macrocell, master 0 has highest priority}

GT0_WAIT =  <PROD>                      {terminate macrocell, <PROD> enables condition decoder}
            <INV_PROD> /MBB * /IDLE;        {gt0_wait terminates if MBB or IDLE are asserted}

NRM0_SGT = <PROD> GT0_WAIT * MBB * /IDLE; {grant is issued if request is highest order pending, and}
                                        {MBB is asserted and IDLE is not asserted}

NRM0_TGT =  <PROD> /MBB                                 {grant terminates when MBB is deasserted,}
            <INV_PROD> /GT1_WAIT * /GT2_WAIT * /GT3_WAIT; {and a request is pending}

IDL0_SGT =  <PROD> GT0_WAIT * IDLE;     {grant is issued if request is highest order pending, and
                                        {IDLE is asserted}

IDL0_TGT =  <PROD> MBB                                  {grant terminates when MBB is asserted, }
            <INV_PROD> /GT1_WAIT * /GT2_WAIT * /GT3_WAIT; {and a request is pending}


GT1_PRI =   <PROD> /MBR0 * MBR1;        {start macrocell, master 1 has second highest priority}

GT1_WAIT =  <PROD>                      {terminate macrocell}
            <INV_PROD> /MBB * /IDLE;        {gt1_wait terminates if MBB or IDLE are asserted}

NRM1_SGT = <PROD> GT1_WAIT * MBB * /IDLE; {grant is issued if request is highest order pending, and}
                                        {MBB is asserted and IDLE is not asserted}

NRM1_TGT = <PROD>   /MBB                {grant terminates when MBB is deasserted,}
           <INV_PROD> /GT0_WAIT * /GT2_WAIT * /GT3_WAIT; {and a request is pending}

IDL1_SGT =  <PROD> GT1_WAIT * IDLE;     {grant is issued if request is highest order pending, and
                                        {IDLE is asserted}

IDL1_TGT =  <PROD> MBB                  {grant terminates when MBB is asserted,}
            <INV_PROD> /GT0_WAIT * /GT2_WAIT * /GT3_WAIT; {and a request is pending}
```

Appendix A. PLD ToolKit Source File for the Mbus Arbiter

```
                                        {condition decode array equations--continued}
GT2_PRI  =   <PROD> /MBR0 * /MBR1 * MBR2;   {start macrocell, master 2 has third highest priority}

GT2_WAIT =   <PROD>                         {terminate macrocell}
             <INV_PROD> /MBB * /IDLE;       {gt2_wait terminates if MBB or IDLE are asserted}

NRM2_SGT = <PROD> GT2_WAIT * MBB * /IDLE; {grant is issued if request is highest order pending, and}
                                           {MBB is asserted and IDLE is not asserted}

NRM2_TGT = <PROD> /MBB                      {grant terminates when MBB is deasserted,}
           <INV_PROD> /GT0_WAIT * /GT1_WAIT * /GT3_WAIT; {and a request is pending}

IDL2_SGT =   <PROD> GT2_WAIT * IDLE;        {grant is issued if request is highest order pending, and
                                            {IDLE is asserted}

IDL2_TGT =   <PROD> MBB                     {grant terminates when MBB is asserted,}
             <INV_PROD> /GT0_WAIT * /GT1_WAIT * /GT3_WAIT; {and a request is pending}

GT3_PRI  =   <PROD> /MBR0 * /MBR1 * /MBR2 * MBR3; {start macrocell, master 3 has lowest priority}

GT3_WAIT =   <PROD>                         {terminate macrocell}
             <INV_PROD> /MBB * /IDLE;       {gt3_wait terminates if MBB or IDLE are asserted}

NRM3_SGT = <PROD> GT3_WAIT * MBB * /IDLE; {grant is issued if request is highest order pending, and}
                                           {MBB is asserted and IDLE is not asserted}

NRM3_TGT = <PROD> /MBB                      {grant terminates when MBB is deasserted,}
           <INV_PROD> /GT0_WAIT * /GT1_WAIT * /GT2_WAIT; {and a request is pending}

IDL3_SGT =   <PROD> GT3_WAIT * IDLE;        {grant is issued if request is highest order pending, and
                                            {IDLE is asserted}

IDL3_TGT =   <PROD> MBB                     {grant terminates when MBB is asserted,}
             <INV_PROD> /GT0_WAIT * /GT1_WAIT * /GT2_WAIT; {and a request is pending}

BEGIN =      <PROD> /MBR0 * /MBR1 * /MBR2 * /MBR3; {begin asserts when there are no requests,
                                                    usually after a reset}

IDLE  =      <PROD> MBB                                {idle terminates when MBB is asserted,}
             <INV_PROD> /GT0_WAIT * /GT1_WAIT * /GT2_WAIT * /GT3_WAIT;
                                                    {and a request is pending}

                                            {output array equations}
CKEN1 = <INV_SUM> /OFF;                     {input register clocks are always enabled}
CKEN2 = <INV_SUM> /OFF;

MBG0 =       < INV_SUM> /NRM0_SGT * /NRM0_TGT * /IDL0_SGT * /IDL0_TGT; {each grant is made up
                                                                        of these four states}
MBG1 =       <INV_SUM> /NRM1_SGT * /NRM1_TGT * /IDL1_SGT * /IDL1_TGT
;
MBG2 =       <INV_SUM> /NRM2_SGT * /NRM2_TGT * /IDL2_SGT * /IDL2_TGT;{the output pins are
                                                                       inverted for the low }

MBG3 =       <INV_SUM> /NRM3_SGT * /NRM3_TGT * /IDL3_SGT * /IDL3_TGT; {asserted grants}

                                        {end of file};
```

CYPRESS SEMICONDUCTOR

TMS320C30/VME Signal Conditioner Using the CY7C361

The design documented in this application note shows how to use the Cypress CY7C361 to work with the TMS320C30 ('C30) digital signal processor from Texas Instruments and a VME interface. The design uses a single CY7C361 to perform 'C30 interrupt signal conditioning as well as VME DTACK (Data Transfer ACKnowledge) generation. The CY7C361 performs these functions at a cost that is generally lower than would otherwise be possible.

This application note provides a brief introduction to the CY7C361 and the methods you can use to implement two different functions in the device. This design contains six different state machines and uses 30 of 32 available macrocells.

CY7C361 Description

The CY7C361 is a 28-pin, 125-MHz state machine EPLD. It contains 32 macrocells, eight dedicated inputs, four bidirectional pins, six dedicated output pins, and four Mealy output pins (which you can use as fast combinatorial outputs).

The CY7C361 is based on a token-passing state machine methodology, which is distinctly different from what you might consider the "normal" method of designing state machines (e.g., encoding states). The CY7C361's token-passing scheme effects a logical, streamlined state machine design methodology. In this scheme, each macrocell typically corresponds to a state. It is possible, however, to encode states in this device. But associating each macrocell with a state generally obviates the need to decode the macrocell outputs to determine the machine's present state, which eliminates the need for a state table.

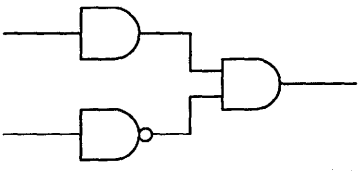


Figure 1. The CY7C361 Condition Decoder

You can configure each CY7C361 state macrocell in one of three ways. First, in the START configuration, the macrocell's output pulses High for exactly one cycle when either of two conditions are asserted: the C_IN signal from the previous macrocell or the output of the CY7C361's condition decoder. The start configuration is useful for starting a sequence.

The second macrocell configuration is the (Wait Until) TERMINATE configuration. In this configuration, the output goes High when C_IN is received from the previous macrocell and remains High until the condition decoder's output is asserted. You can use this configuration to indicate, for example, "I am performing this function now," where "I" is a state machine implemented in the CY7C361.

In the third macrocell configuration, the TOGGLE configuration, the macrocell output toggles so long as the C_IN from the previous macrocell or the condition decoder 's output is asserted. This configuration is useful for counters.

The condition decoder differs from the traditional sum-of-products decoder in conventional PLDs. The CY7C361 condition decoder efficiently facilitates either entering a state from one of several states based on a condition or leaving a state based on a condition. The condition decoder performs, in effect, an AND on an AND and an OR (*Figure1*).

To keep the CY7C361's speed very high, the macrocell feedback structure is divided into three different feedback types: feedback that routes to all cells in a group of eight, feedback that routes to all cells in a group of sixteen, and feedback that routes to all 32 macrocells. It is important to be aware of these feedback groups when designing with the CY7C361.

Interrupt Signal Conditioning for TMS320C30

The 'C30 has four external interrupt lines (/INT0 through /INT3, referred to here as /INTx). These lines are active Low, and have specific timing requirements relative to the internal 'C30 clocks, H1 and H3. These internal clocks derive from the 33-MHz CLKIN signal and have a period twice as long as CLKIN's period. The

interrupt lines must be asserted (pulled Low) for at least one but less than two periods of the H1/H3 clock. If /INTx is asserted for less than one period, the 'C30 does not respond to the interrupt. If /INTx is asserted for two or more periods, the 'C30 responds to the interrupt twice. Because H1 and H3 have a 60-ns period, the interrupt pulses must be 60 to 120 ns long.

Design Goals

Figure 2 presents the pertinent timing information for the interrupt signals. CLKIN is the main 'C30 clock and also drives the CY7C361. DBL_CLK is the internal CY7C361 clock; it runs at twice the frequency of CLKIN because the VME DTACK circuitry requires timing resolution to 15-ns intervals. Consequently, the CY7C361's internal clock doubler is turned on.

Note that an interrupt pulse width of 105 ns (or seven double-clock periods) has been chosen. This design detects a falling edge on the external interrupt pulses (/XINTx), and the CY7C361 provides a conditioned 105-ns active-Low pulse on /INTx. /XINTx must toggle high and then Low again to initiate another /INTx pulse. Double registered for metastable hardness, all four external interrupt pulses are asynchronous with respect to each other and CLKIN.

Implementation in the CY7C361

Although this design's logic is relatively simple, fitting the 'C30 interrupt conditioning circuitry as well as the VME DTACK generation circuitry in the same device poses a space challenge. The DTACK circuitry requires eight macrocells, leaving 24 macrocells available for the 'C30 interrupt circuitry. The method used to generate the conditioned interrupt pulses is similar to the pulse-triggered counter described in the application note, "Understanding the CY7C361."

The 'C30/VME design requires modifications to the counter for two reasons: First, the pulse length required in this implementation is less than the counter's maximum length, and because there is no way to preset a macrocell's state, the counter must be reset upon reaching the desired count (in this case, a count of seven cycles). Second, two of the interrupt conditioning circuits must be designed a little differently to take into account the fact that not all macrocell feedback terms are available to all other macrocells.

Figure 3 presents a handy tool you can use to efficiently allocate CY7C361 resources. This diagram shows all 32 macrocells, and their respective node numbers, for use in Cypress's PLD ToolKit. Above each group of four macrocells appears the internal node number for that group's local reset. This reset is used to reset the counters at the end of the count sequence. *Figure 3* also shows the CY7C361's feedback structure. Note that each group of eight macrocells has four macrocells that are connected only to the local group-of-eight feedback path. Two more are visible to all macrocells in the group of sixteen. The last two macrocells are visible to all 32 registers.

As a brief review of the pulse generator's operation, consider that each interrupt has an associated conditioner circuit, and each contains two supervisory states and a 3-bit counter. The supervisory states are

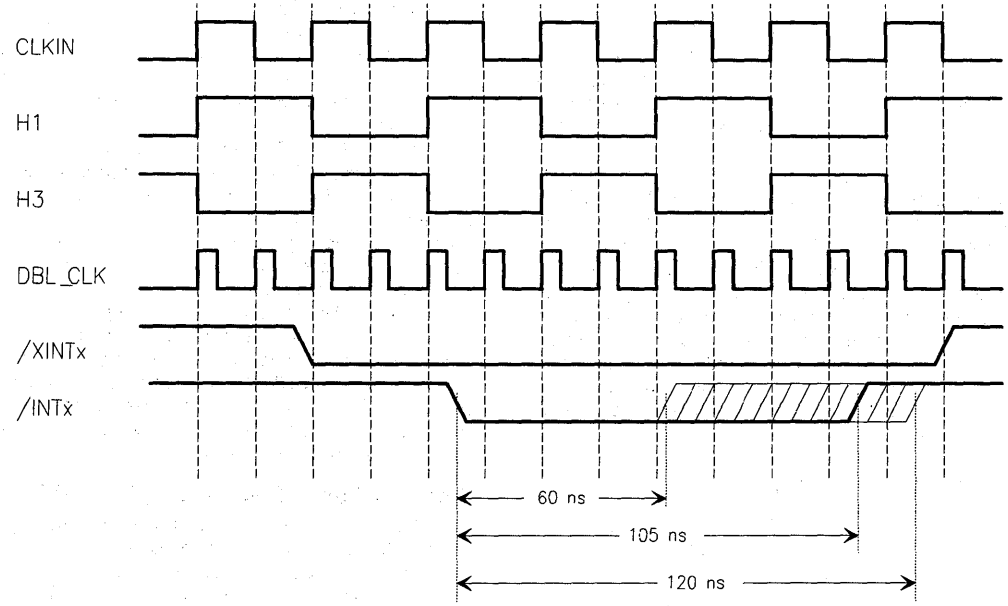


Figure 2. TMS320C30 External Interrupt Timing

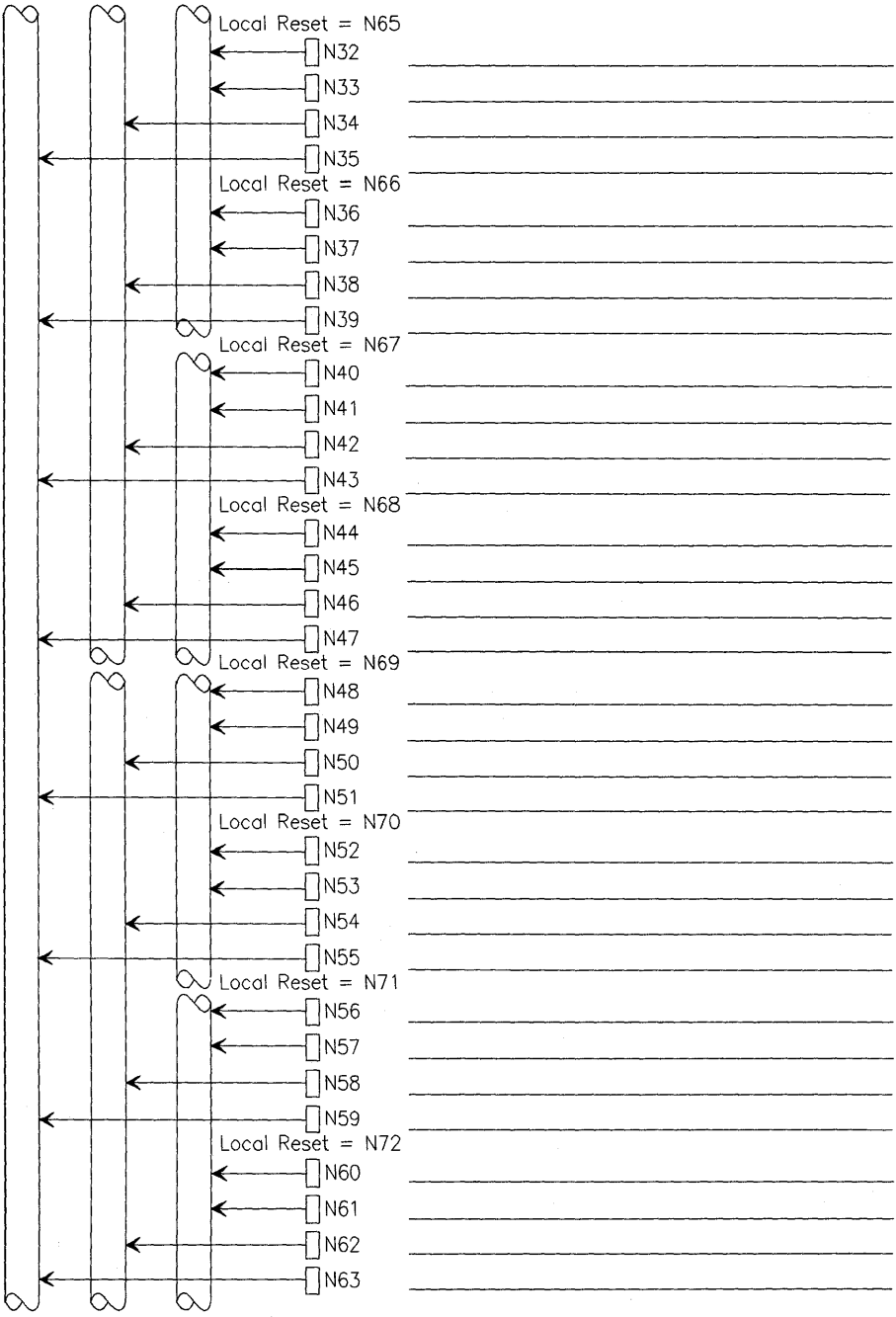


Figure 3. CY7C361 Resource Allocation Chart

called INTx_S0 and INTx_S1, where x = 0, 1, 2, or 3 for interrupt 0, 1, 2, or 3. INTx_S0 is in the START configuration and sends a C_IN to INTx_S1 when a falling edge is detected on /XINTx. INTx_S1 is in the (Wait Until) TERMINATE configuration. Upon receipt of C_IN from INTx_S0, INTxS1 asserts its output, which remains asserted until the count is reached. The counter is implemented in three macrocells (labeled INTx_C0, INTx_C1, and INTx_C2), each of which is configured in the TOGGLE configuration. When INTx_S1 is asserted, the counter starts counting. When the counter macrocells' local reset, INTx_RST, detects the final count, it resets the counter. Simultaneously, INTx_S1 deasserts. The output (/INTx) is decoded directly from INTx_S1 and also deasserts.

A few words of caution: First, note that INTx_S1 must be the next node after INTx_S0, because a C_IN from the INTx_S0 macrocell triggers INTx_S1. Second, counter macrocells cannot be shared between any two interrupt conditioner circuits, because INT1_RST could then reset a bit in INT2's counter (for example), resulting in an inaccurate pulse width that could prevent a response to interrupt 2.

The CY7C361 resource allocation chart for this section of the design appears in *Figure4*. As shown, this design occupies 22 of 24 available macrocells. The INT1 and INT2 counters (located in nodes 40 through 43 and nodes 44 through 47, respectively) each have an extra associated macrocell called INT1_MON and INT2_MON. The machine requires these extra states because not all counter bits are visible to their respective supervisory states. *Appendix A* lists the PLD ToolKit source code for this design.

A representation of the PLD ToolKit simulator output for the generic case appears in *Figure5*. Because /XINTx is double registered, INTx_S0 asserts two cycles after /XINTx is clocked in. INTx_S1 becomes asserted on the next CLKDB cycle, which starts the counter. For INT0 and INT3, when the count equals 5, INTx_S1 deasserts, and INTx_RST resets the counter macrocells. For INT1 and INT2, when the count equals 5, INTx_MON become asserted, which terminates INTx_S1. INTx_MON then triggers INTx_RST, and the counter is reset. Note that /XINTx can remain Low for any arbitrary time period, and only one count cycle is triggered.

VME DTACK Generation

DTACK, the VMEbus's Data Transfer ACKnowledge signal, is active Low. On a VME write, data is transferred from the master to the slave's local memory. On a read, data is read from the slave's local memory and placed on the bus. In either case, the bus takes a known amount of time to perform the transfer. The slave uses DTACK to inform the master that enough time has passed for the transfer to have taken place. Thus, on a VME write, the slave asserts DTACK when the data has been stored to local memory. On a VME read, the slave asserts DTACK when the data the slave has placed on the bus is valid.

Design Goals

The DTACK generator presented here handles two different speeds of slave local memory. The desired timing appears in *Figure6*. When other circuitry on the board decodes a valid address for local memory from the master, the speed of the addressed memory determines which of two signals is asserted. If the slower memory on the board was addressed, /GMSEL0 is asserted. For the faster memory, /GMSEL1 is asserted. These signals are mutually exclusive because they indicate two different address spaces.

The design goal is to detect a transition on either /GMSEL0 or /GMSEL1 and produce an appropriately delayed active-High DTACK signal. This signal must remain asserted until the triggering /GMSEL line toggles back High. Note that on the VMEbus, DTACK is active Low. It is being shown active High in this application note to demonstrate how to use a Mealy output to control a signal's polarity.

Implementation in the CY7C361

In operation, when the 'C30/VME circuit detects a falling edge on either /GMSEL line, a counter begins to count. This 4-bit counter is located in a single group of four macrocells (denoted by DTACK_Cx, where x = 0, 1, 2, 3) as shown in *Figure7*. Another bank of four macrocells implements the supervisory states for /GMSEL0 and /GMSEL1. Each case requires two supervisory states.

SELx_S0 (where x = 1, 2) is in the START configuration and is triggered when the counter reaches the desired number. SELx_S1 is configured in the (Wait Until) TERMINATE configuration, is triggered by SELx_S0, and terminates when a rising edge is detected on /GMSELx. Because /GMSEL0 and /GMSEL1 are never active at the same time, the same counter can produce both delays.

When /GMSEL0 is active, the machine requires a 135-ns delay. To produce this delay, the counter counts to 7. When combined with one clock delay for getting data into the CY7C361 and one clock delay for asserting DTACK, the count of 7 results in 9 clocks at 15 ns each, or 135 ns. The assertion of /GMSEL1 requires a 75-ns delay, produced by a count of 3 when combined with one clock delay for getting data into the CY7C361 and one clock delay for asserting DTACK. Note that you could easily modify the counter to produce delays as high as 255 ns.

When the desired count is reached, SELx_S0 starts (which triggers SELx_S1) and DTACKRST is asserted, thus resetting the counter. SELx_S1 remains active until terminated by detection of a rising edge on /GMSELx. DTACK is asserted when either SEL1_S1 or SEL2_S1 is active. *Appendix A* shows the PLD ToolKit source file for this section of the design. A representation of the

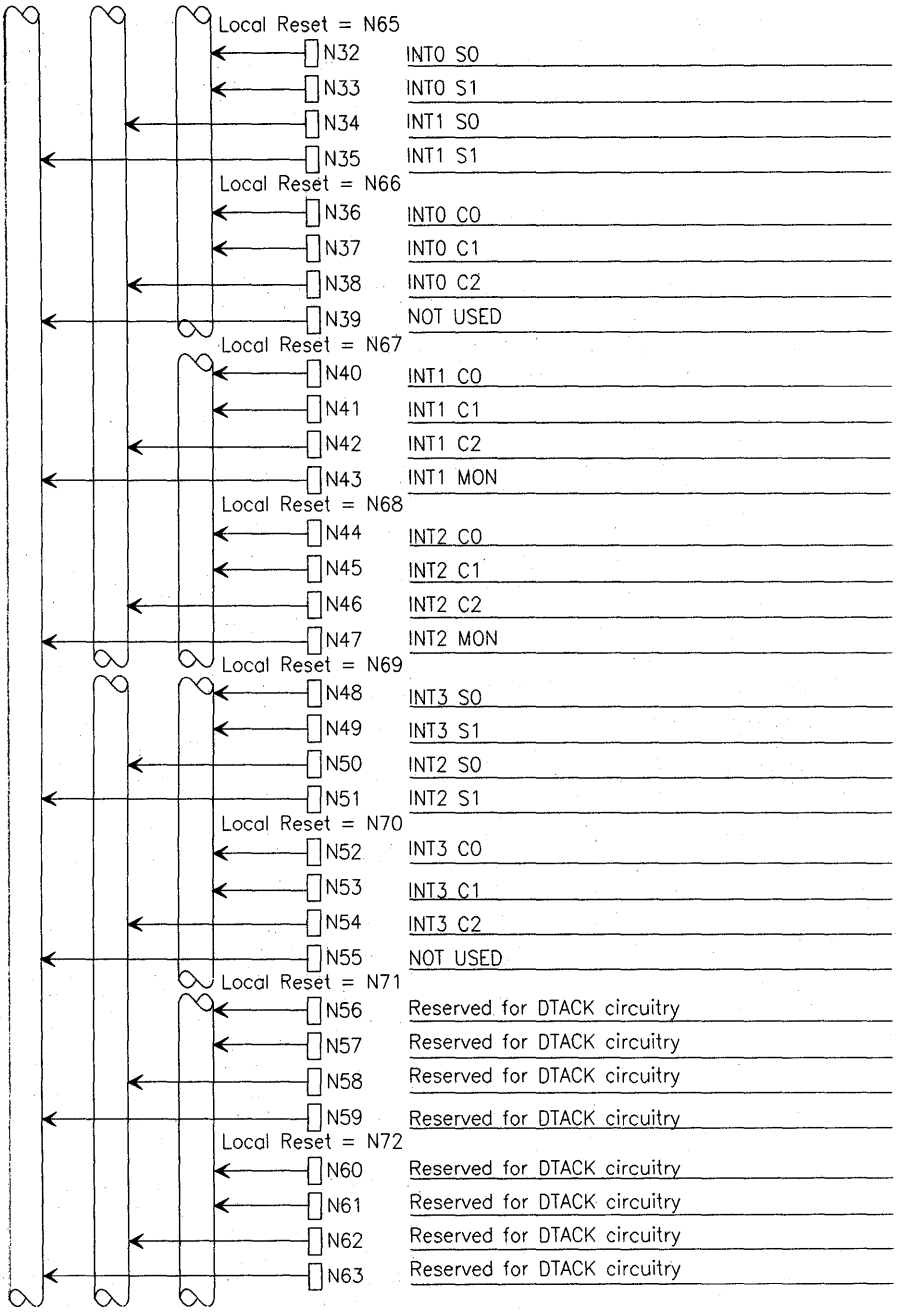


Figure 4. Resource Allocation Chart for 'C30 Interrupt Conditioner Circuit

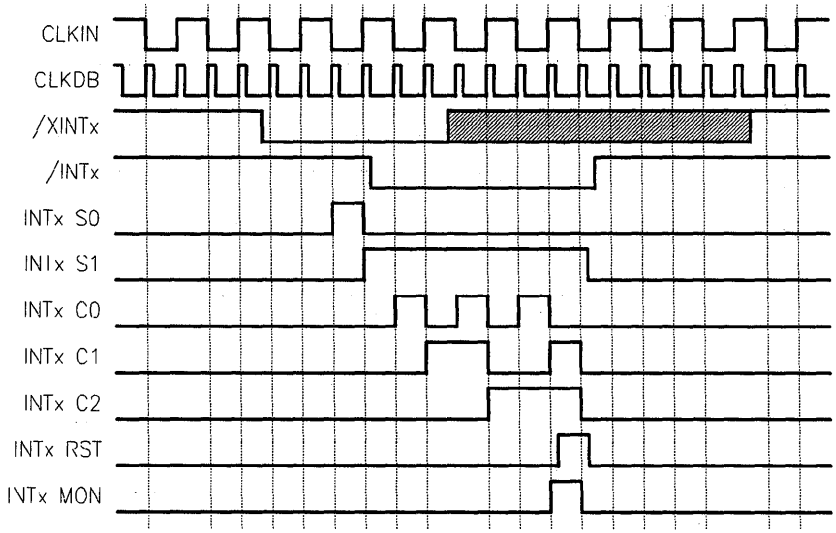


Figure 5. Output Signals for the Interrupt Controller

PLD ToolKit simulator output is presented in *Figures8* and *9*.

CY7C361 Configuration Information

Both the configuration and equations parts of the PLD ToolKit source file in *Appendix A* contain a section for miscellaneous functions. These lines of code configure a variety of CY7C361 characteristics.

The configuration section contains a required line that sets up the internal clock:

CLKDB(node=74, dbl_clk),

CLKDB is the name assigned to the CY7C361's internal clock. Node 74 is the clock's internal node number. "dbl_clk" is an attribute that turns on the CY7C361's clock doubler. If you do not want double-clock operation, simply leave out the "dbl_clk" attribute.

Three lines of code configure the input enables:

ien1(node=29),
ien2(node=30),
ien3(node=31),

"ienx" is the name assigned to the nodes. Node 29 enables inputs I0 through I3. Node 30 enables I4 through I7. Node 31 enables the inputs of the bidirectional pins (B0 through B3).

One line of code provides a way to tie signals to ground:

GND(node=73),

You can use this code, for example, to permanently enable inputs, which the equations section demonstrates:

ien1 = <inv_sum> /gnd;
ien2 = <inv_sum> /gnd;
ien3 = <inv_sum> /gnd;

Acknowledgment

The author thanks Steve Heinrichs and Scott Mindemann for permitting their idea to be used here and for providing information regarding the TMS320C30's interrupt structure and timing.

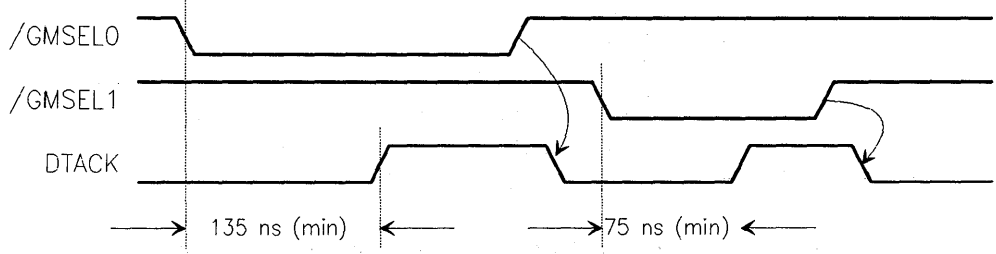


Figure 6. Timing Requirements for DTACK Circuitry

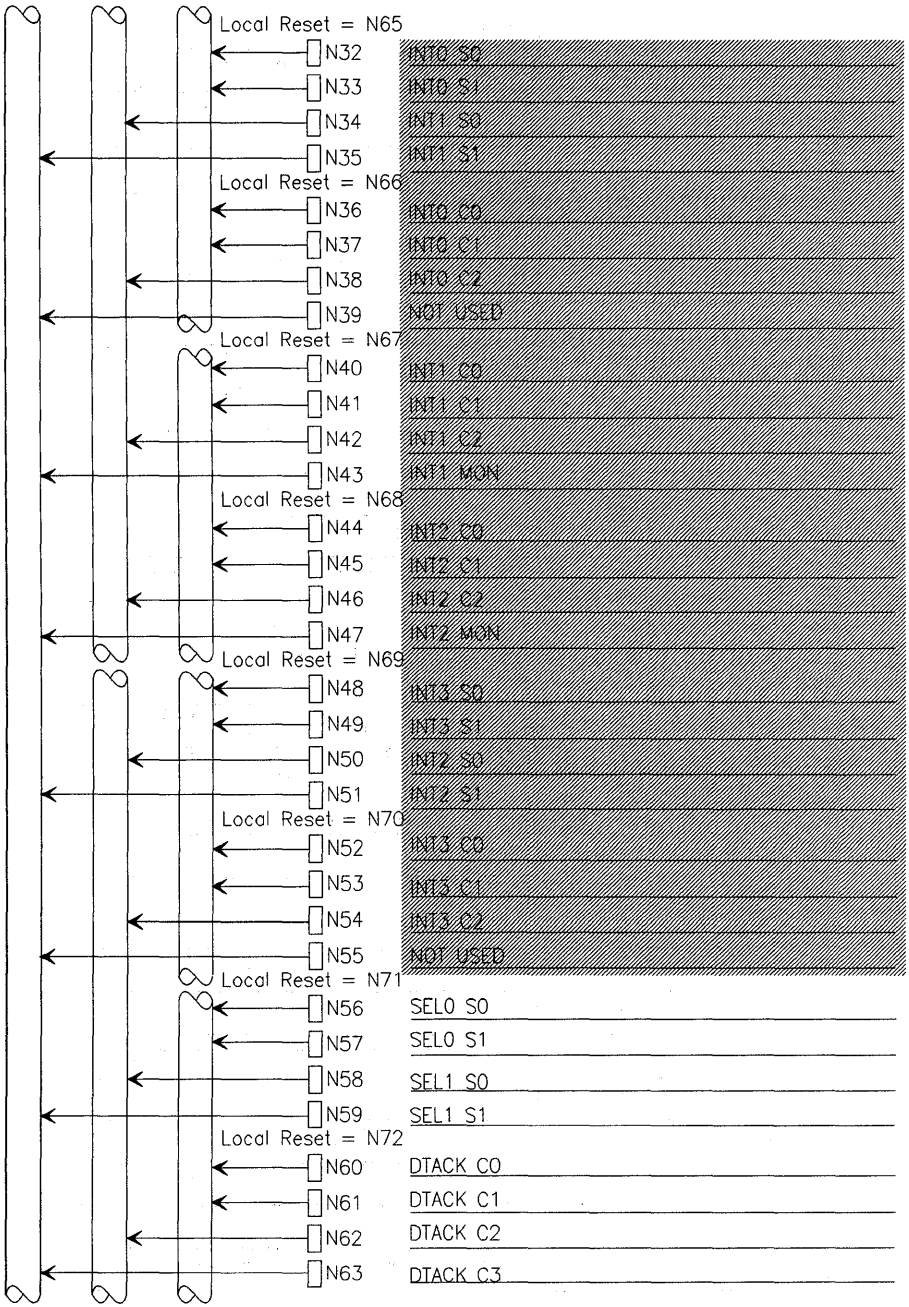


Figure 7. Resource Allocation for the DTACK Circuitry

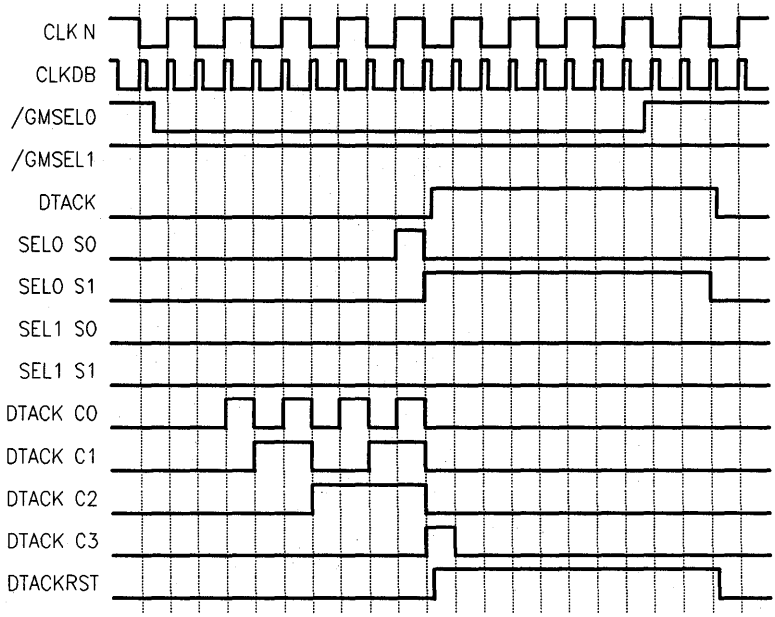


Figure 8. DTACK Timing for /GMSEL0

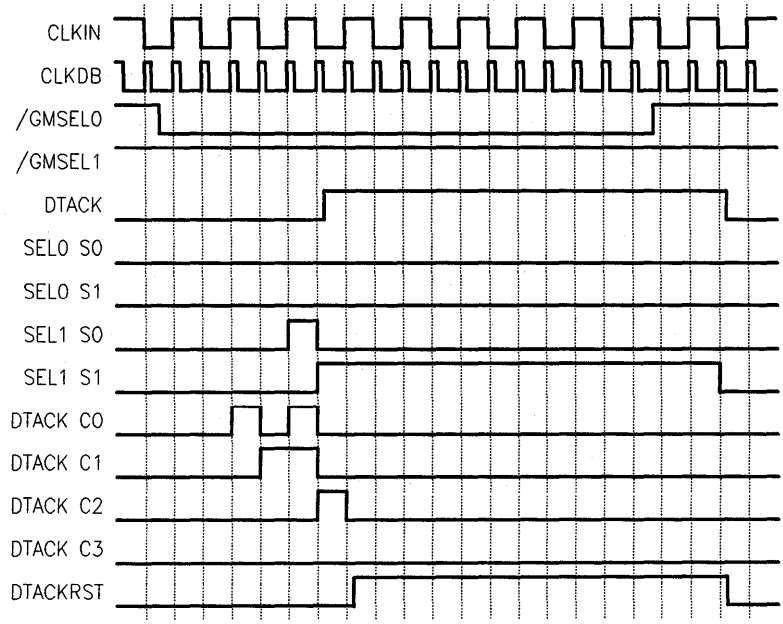


Figure 9. DTACK Timing for /GMSEL1

Appendix A. PLD ToolKit Source File for TMS320C30/VME Signal Conditioner/Generator

```
CY7C361;
    {PLD Toolkit source code listing for TMS320C30/VME signal conditioner/generator}

CONFIGURE;

{These lines are miscellaneous setup for the 361.}
        CLKDB(node=74,   dbl_clk),          {Required line. In this case, the clock doubler is ON.}

        ien1(node=29),                      {Nodes 29, 30, and 31 are the input enables for the}
        ien2(node=30),                      {dedicated inputs and the bidirectional inputs used on}
        ien3(node=31),                      {this device.}

        GND(node=73),                       {Used to permanently assert an internal signal.}

        CLKIN(node=4),                      {Clock signal. Feeds both the 361 and TMS320C30.}

        GLBL_RST(node=64),                  {Naming the internal global reset node}

        /RESET(node=9),                     {Pin that will be used to assert GLBL_RST}

{This section is the configuration of the interrupt logic for INT0.}
        /XINT0(node=1,   iireg),            {An interrupt trigger input. Pin 1, double registered.}

        /INT0(node=28),                     {Massaged interrupt pulse, output on pin 28.}

        INT0_S0(node=32,   start),          {State S0 means "not counting", start configuration.}
        INT0_S1(cin, term),                 {State S1 means "counting", (Wait Until) Terminate}
                                            {configuration, triggered by C_IN from INT0_S0.}

        INT0_C0(node=36,   tog),            {C0 , C1, and C2 are the counter bits. They count to}
        INT0_C1(tog),                       {6 and are locally reset to 000 binary, all configured in}
        INT0_C2(tog),                       {the toggle configuration}

        INT0_RST(node=66),                  {Local reset for INT0 counter. Resets when C2C1C0 =  110.}

{This section is the configuration of the interrupt logic for INT1.}
        /XINT1(node=2,   iireg),            {An interrupt trigger input. Pin 2, double registered.}

        /INT1(node=27),                     {Massaged interrupt pulse, output on pin 27.}

        INT1_S0(node=34,   start),          {State S0 means "not counting", start configuration.}
        INT1_S1(cin, term),                 {State S1 means "counting", (Wait Until) Terminate}
                                            {configuration, triggered by C_IN from INT1_S0.}

        INT1_C0(node=40,   tog),            {C0 , C1, and C2 are the counter bits. They count to 5}
        INT1_C1(tog),                       {which triggers INT1_MON, which then causes them to be reset}
        INT1_C2(tog),                       {to 000 binary, all configured inthe toggle configuration}

        INT1_MON(start),                    {This is the monitor bit for the INT1 counter. Configured}
                                            {as Start and triggered by C2C1C0 = 101}

        INT1_RST(node=67),                  {Local reset for INT1 counter. Resets when INT1_MON is}
                                            {asserted.}
```

Appendix A. PLD ToolKit Source File for TMS320C30/VME Signal Conditioner/Generator (Continued)

```
{This section is the configuration of the interrupt logic for INT2. All comments from INT1 apply here.}
        /XINT2(node=3,  iireg),

        /INT2(node=26),

        INT2_S0(node=50,  start),
        INT2_S1(cin, term),

        INT2_C0(node=44,  tog),
        INT2_C1(tog),
        INT2_C2(tog),

        INT2_MON(start),

        INT2_RST(node=68),


{This section is the configuration of the interrupt logic for INT3. All comments from INT0 apply here.}
        /XINT3(node=5,  iireg),

        /INT3(node=25),

        INT3_S0(node=48,  start),
        INT3_S1(cin, term),

        INT3_C0(node=52,  tog),
        INT3_C1(tog),
        INT3_C2(tog),

        INT3_RST(node=70),


{Configuration of the DTACK circuitry.}
        /GMSEL0(node=10,    iireg),     {Input, pin 10, double registered}
        /GMSEL1(node=11,    iireg),     {Input, pin 11, double registered}
        DTACK(node=19,    ninv),        {Output, pin 19. Mealy output is used to get non-inverting}
                                        {output.}

        SEL0_S0(node=56,    start),     {Supervisory state for /GMSEL0.}
        SEL0_S1(cin,  term),            {Supervisory state for /GMSEL0.}

        SEL1_S0(start),                 {Supervisory state for /GMSEL1.}
        SEL1_S1(cin,  term),            {Supervisory state for /GMSEL1.}


        DTACK_C0(tog),                  {LSB of the DTACK delay counter.}
        DTACK_C1(tog),
        DTACK_C2(tog),
        DTACK_C3(tog),                  {MSB of the DTACK delay counter.}

        DTACKRST(node=72)               {Reset term for the DTACK delay counter.}
```

Appendix A. PLD ToolKit Source File for TMS320C30/VME Signal Conditioner/Generator (Continued)

```
EQUATIONS;

{This section makes sure the inputs are always ENABLED and connects RESET to the internal Global Reset.}
        ien1 =<inv_sum> /gnd;
        ien2= < inv_sum> /gnd;
        ien3 =<inv_sum> /gnd;
        GLBL_RST = <prod> RESET;

{Equations for the interrupt logic for /INT0.}
        INT0_S0 = < inv_prod>     /XINT0                {Start configuration, triggers on falling}
                < prod> ;                               {edge of /XINT0. < prod> is included}
                                                        {to set the AND term to logic 1. If this}
                                                        {is not done, the condition decoder is}
                                                        {always logic 0.}

        INT0_S1 = <prod> INT0_C2 * INT0_C1 * /INT0_C0;  {(Wait Until) Terminate configuration,}
                                                        {triggered by C_IN from above, }
                                                        {terminated by a count of 6.}

        INT0_C0 = <prod> INT0_S1;                       {LSB of the counter}
        INT0_C1 = <prod> INT0_C0 * INT0_S1;
        INT0_C2 = <prod> INT0_C0 * INT0_C1 * INT0_S1;   {MSB of the counter}

        INT0 = <inv_sum> /INT0_S1;                      {Output equation. Effectively,}
                                                        {/INT0 is asserted while INT0_S1 is}
                                                        {asserted.}

        INT0_RST = <prod> INT0_C2 * INT0_C1 * /INT0_C0; {Local reset term.}

        XINT0 = <inv_oe>;                               {Included since XINT0 is an input to a}
                                                        {bidirectional pin. This turns off the}
                                                        {output buffer, making the pin an input.}

{Equations for the interrupt logic for /INT1.}
        INT1_S0 = < inv_prod>     /XINT1                {Start configuration, triggers on falling}
                <prod>;                                 {edge of /XINT1. <prod> is included}
                                                        {to set the AND term to logic 1. If this}
                                                        {is not done, the condition decoder is}
                                                        {always logic 0.}

        INT1_S1 = <prod> INT1_MON;                      {triggered by C_IN from above,}
                                                        {terminated when INT1_MON is asserted.}

        INT1_C0 = <prod> INT1_S1;                       {LSB of the counter}
        INT1_C1 = <prod> INT1_C0 * INT1_S1;
        INT1_C2 = <prod> INT1_C0 * INT1_C1 * INT1_S1;   {MSB of the counter}

        INT1_MON = <prod> INT1_C2 * /INT1_C1 * INT1_C0; {This is the monitor bit. In order to make}
                                                        {the output be 7 clocks long, this must}
                                                        {be triggered at 6 clocks (eg, a count of 5.)}

        INT1 = <inv_sum> /INT1_S1;                      {Output equation. Effectively,}
                                                        {/INT1 is asserted while INT1_S1 is}
                                                        {asserted.}
        INT1_RST = <prod> INT1_MON;                     {Counter is reset when INT1_MON is}
                                                        {asserted.}
```

Appendix A. PLD ToolKit Source File for TMS320C30/VME Signal Conditioner/Generator (Continued)

```
XINT1  =  <inv_oe>;                          {Included since XINT1 is an input to a}
                                             {bidirectional pin. This turns off the}
                                             {output buffer, making the pin an input}

{Equations for the interrupt logic for /INT2. All comments from /INT1 apply here.}
INT2_S0 = < inv_prod>     /XINT2
          <prod>;
INT2_S1  =  <prod>  INT2_MON;

INT2_C0  =  <prod>  INT2_S1;
INT2_C1  =  <prod>  INT2_C0 * INT2_S1;
INT2_C2  =  <prod>  INT2_C0 * INT2_C1 * INT2_S1;

INT2_MON  =  <prod>  INT2_C2 * /INT2_C1 * INT2_C0;

INT2  =  <inv_sum>  /INT2_S1;

INT2_RST  =  <prod>  INT2_MON;

{Equations for the interrupt logic for /INT3. All comments from /INT0 apply here.}
INT3_S0  =  <inv_prod>  /XINT3
            <prod>;
INT3_S1  =  <prod>  INT3_C2 * INT3_C1 * /INT3_C0;

INT3_C0  =  <prod>  INT3_S1;
INT3_C1  =  <prod>  INT3_C0 * INT3_S1;
INT3_C2  =  <prod>  INT3_C0 * INT3_C1 * INT3_S1;

INT3  =  <inv_sum>  /INT3_S1;

INT3_RST  =  <prod>  INT3_C2 * INT3_C1 * /INT3_C0;

{Equations for the DTACK counter}
DTACK_C0  =  <inv_prod>  /GMSEL0 * /GMSEL1          {LSB toggles when /GMSEL0 OR}
             <prod>;                                {/GMSEL1 is active.}
DTACK_C1  =<prod>  DTACK_C0
             <inv_prod>  /GMSEL0 * /GMSEL1;

DTACK_C2  =  <prod>  DTACK_C0 * DTACK_C1
             <inv_prod>  /GMSEL0 * /GMSEL1;
DTACK_C3  =  <prod>  DTACK_C0 * DTACK_C1 * DTACK_C2
             <inv_prod>  /GMSEL0 * /GMSEL1;

{Equations for DTACK pulse generation.}
SEL0_S0  =  <prod>  /DTACK_C3 * DTACK_C2 * DTACK_C1 * /DTACK_C0 * /GMSEL1;
SEL0_S1  =  <prod>  /GMSEL0;

SEL1_S0  =  <prod>  /DTACK_C3 * /DTACK_C2 * DTACK_C1 * /DTACK_C0 * /GMSEL0;
SEL1_S1  =  <prod>  /GMSEL1;

DTACKRST  =  <inv_prod>  /SEL0_S1 * /SEL1_S1
             <prod>;

DTACK  =  <inv_sum>  /SEL0_S1 * /SEL1_S1;
```

CYPRESS SEMICONDUCTOR

DMA Control Using the CY7C342 MAX EPLD

This application note details the use of a CY7C342 MAX EPLD as a general-purpose DMA controller for a 16-bit microprocessor-based system. The design showcases the versatility and density you can achieve with this type of device, as well as demonstrating a modular and hierarchical design approach utilizing the MAX+PLUS development system's schematic capture and textual design entry capabilities.

The Cypress Multiple Array MatriX (MAX) EPLDs are a reprogrammable, user configurable, high-density, high-performance family of logic devices that suit a host of applications. The MAX architecture allows you to replace large numbers of small- and medium-scale integration (74XXX series) parts, as well as programmable logic devices (PLDs) with a single CY7C340 MAX family device.

CY7C342 Description

The CY7C342 EPLD is functionally equivalent to 4000 - 5000 logic gates. A block diagram of the chip appears in *Figure 1*. The CY7C342 offers high performance, reprogrammability, excellent design tool support, and fast design turn around. The device has 128 flexible macrocells arranged into eight groups called Logic Array Blocks (LABs). You can configure individual macrocells for combinatorial or registered operation, supporting product term control of XOR input, register preset, register clear, asynchronous clock, synchronous clock, and output enable.

Additionally, you can augment macrocell product terms by the use of expander product terms (expanders). This array of inverted-AND product terms feeds back to the macrocell inputs as well as their own inputs, allowing the implementation of large sum-of-products structures and cross-coupled NAND registers.

Each LAB contains 16 macrocells and 32 expanders and functions much like a small EPLD. Signals are routed between the eight LABs via the programmable interconnect array (PIA), which allows you to partition a design across several LABs. The PIA's routing resources feature a single uniform delay, which minimizes the overall impact on device performance. For a more complete description of the features of the CY7C340 MAX family, refer to the Cypress Semiconductor *Databook*.

MAX+PLUS Description

The MAX+PLUS development system is a computer aided design environment used to implement designs with the Cypress CY7C340 EPLD family. MAX+PLUS offers an integrated approach to design entry, design verification, and device programming. Running on an IBM PC/AT-compatible platform,

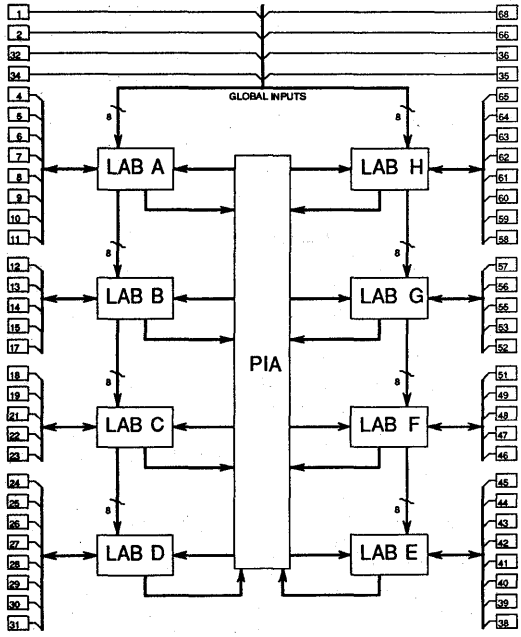


Figure 1. CY7C342 Block Diagram

Table 1. DMAC Modules

DMAC Modules	Design Method
CPU Decoder	Text Entry
Control Register	Schematic Entry
Address Counter	Schematic Entry
Word Counter	Schematic Entry
Three-State Buffers	Schematic Entry
Output Multiplexers	Schematic Entry
Cycle Controller	Text Entry

MAX+PLUS provides all the tools necessary to quickly and efficiently convert complex logic designs into functional silicon.

You enter designs in MAX+PLUS using a powerful hierarchical graphic editor that supports both schematic capture and high-level text definition. The graphical editor gives you the ability to capture schematics utilizing standard 74XXX TTL macrofunctions, generic logic primitives, or user-defined custom functions. You can also use a high-level text definition written in the Advanced Hardware Description Language (AHDL) to define the function of an entire design or just a portion of a design. By incorporating schematic capture, Boolean equation, state machine, or truth table design entry methods, you are free to choose the technique that best fits your application.

When the MAX+PLUS design entry is complete, the design is compiled. The compiler performs several tasks as it goes through the design database. The compiler performs a minimization function on the logic, fits the design into a device, creates files for the simulator, and provides an object file for programming a device. After compiling the design, you can use the interactive, event-driven timing simulator to determine the design's function and worst case timing characteristics.

Application Description

The application illustrated here is a general-purpose, 16-bit direct memory access controller (DMAC). A DMAC shares the address, data, and control buses with the central processing unit (CPU) and acts as a bus master in place of the CPU when granted bus ownership. Generally, a DMAC has access to all system resources, including memory, memory-mapped I/O, and I/O. *Figure2* shows a typical system block diagram with a 16-bit CPU, RAM and ROM memory, I/O devices, and the DMAC.

When an I/O device requires data from memory or needs to transfer data to memory, it must request service from the DMAC by asserting a DMA request (DREQ). The DMAC (when configured and enabled) then requests ownership of the system bus by activating its hold request output (HREQ). The DMAC then waits until it receives a hold acknowledge (HLDA) from the CPU. Next, the DMAC enables its 23 address

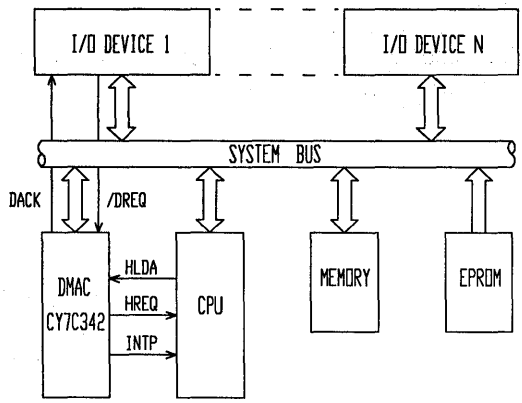


Figure 2. Typical System Block Diagram

outputs to access a specific memory location. Depending on the DMAC's programmed configuration, it either reads the memory location's contents and writes the data to the I/O device or reads data from the I/O device and writes the data into the referenced memory location.

Data transfers between memory and I/O devices can occur as single-word operations or as bursts of words under CPU program control. A 16-bit counter is initialized and maintained to control the number of words being transferred. This counter is decremented every transfer, allowing a count of zero to generate an interrupt. The DMAC also handles memory interface timing and transfer control, because the DMAC essentially functions as the processor when in control of the bus.

Design Partitioning

The MAX+PLUS development system is a hierarchical tool that allows you to partition your design into functional blocks, with each block designed as a separate module. The DMAC implemented here is partitioned into seven functional modules. *Table 1* lists the modules' names and the design method utilized. In addition to the main modules, the DMAC uses some glue logic, which effectively ties the design together.

Figure 3 shows the DMAC block diagram. Each module was constructed utilizing the design technique which best suits the application. The modules are described here in detail including an explanation of the design methodology chosen for each implementation.

CPU Decoder Module

Because the DMAC acts as a peripheral until it has control of the system bus, the DMAC must respond to several I/O commands from the CPU. These commands are essential to configure the DMAC properly and to successfully transfer data. The CPU decoder module

receives interface signals from the CPU and decodes them into write strobes and a read enable. The write strobes latch incoming parameters from the data bus into the parameters' respective locations (e.g., control register, word counter, etc.). The read enable and address line MA1 allow internally selected registers to be multiplexed onto the data bus. The DMAC is configured by the processor via I/O instructions and responds to the addresses as shown in *Table 2*.

The CPU interface's function is defined using AHDL. The resulting ASCII text file describes the CPU decoder's behavior without determining the Boolean equation s or using the schematic-capture/graphic-design entry method. Because MAX+PLUS AHDL provides several different ways to specify a module's operation, a truth table is used to describe the CPU decoder to clearly express which outputs are active when specific inputs are asserted. Refer to *Appendix A* for the CPU decoder AHDL text file.

Control Register Module

The control register module configures the DMAC and controls the DMAC's operation. The CPU writes to the control register module, which has control bits to enable or disable the DMAC, enable an interrupt when the word count equals zero, clear the word counter, enable burst or single-byte transfers, and define the transfer direction (memory to I/O or I/O to memory). The bit definitions for each DMAC function appear in *Table 3*.

The processor can read the DMAC's current status and configuration. The bit definitions (*Table 4*) are essentially the same as the those for the control word.

MAX+PLUS 's schematic capture capability is used to implement the control register function (*Figure 4*). This register stores control and configuration information from the CPU, with the exception of the clear word counter bit. When written by the CPU as a logic 1, this bit uses an additional flip-flop to clear itself and the 16-bit counter.

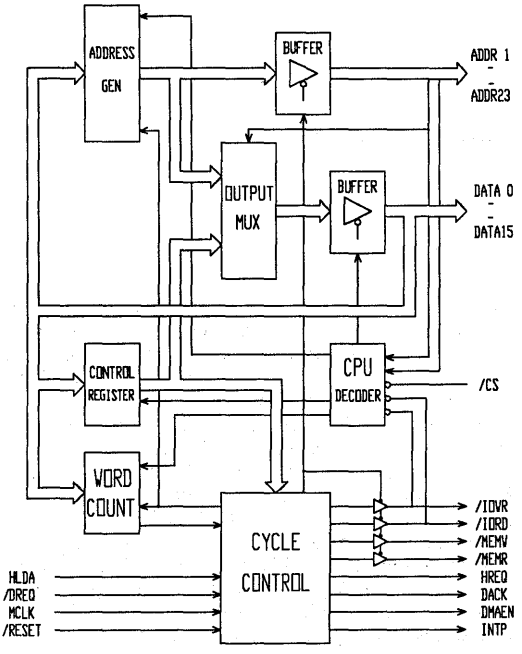


Figure 3. DMAC Block Diagram

Table 2. DMAC I/O Addresses

A2	A1	CS	IORD	IOWR	OPERATION
X	X	1	X	X	No Operation
0	0	0	1	0	Write Control Register
0	1	0	1	0	Write Word Count
1	0	0	1	0	Write Low Mem Addr
X	0	0	0	1	Read Low Mem Addr
1	1	0	1	0	Write High Mem Addr
X	1	0	0	1	Read High Mem Addr and DMAC Status

Address Generator Module

The address generator module is a 23-bit synchronous counter that provides the system memory address for the data transfer operation. The CPU must initialize this counter to the 23-bit value that corresponds to the transfer's starting memory address. As

Table 3. DMAC Control Register Bit Definitions

BIT	DEFINITION
0	Enable DMA Controller (0 = Disabled, 1=Enabled)
1	Enable Interrupt (0=Disabled, 1=Enabled)
2	Clear Word Counter (1 Clears WordCounter and Bit 2 to zero)
3	Burst/Single Word Transfer (0=Single Transfer, 1=Burst Transfer)
4	Transfer Direction (0=Memory to I/O, 1=I/O to Memory)
5-15	Not Used

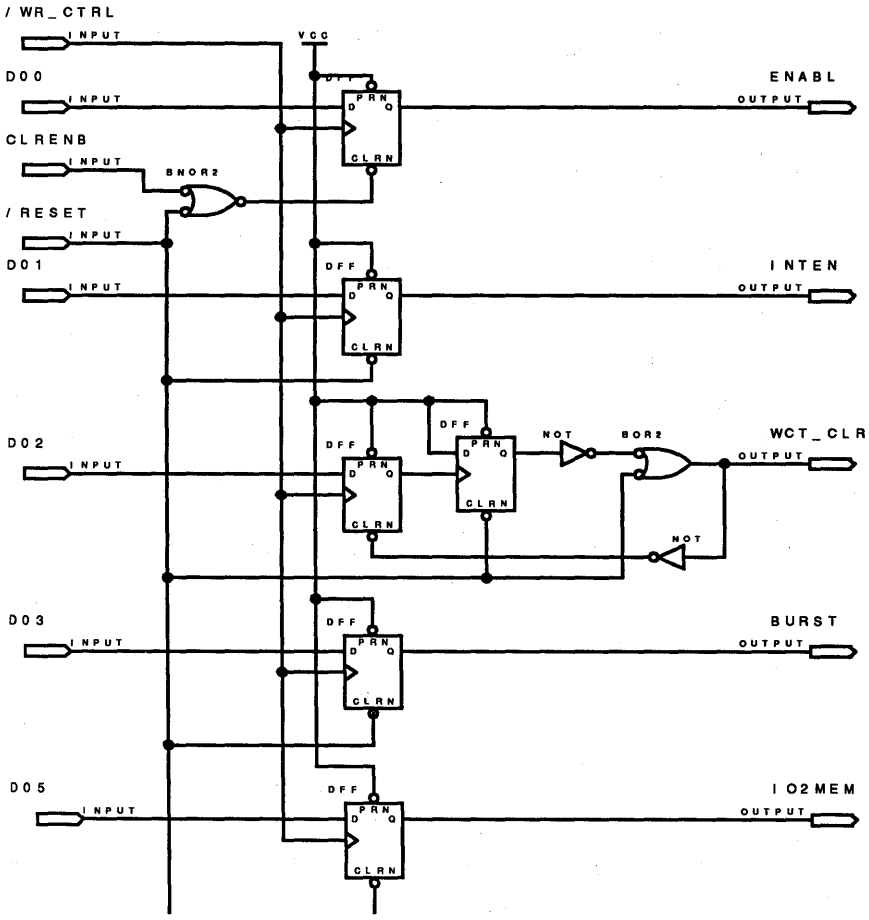


Figure 4. Control Register Schematic

Table 4. DMAC Status Register Definitions

BIT	DEFINITION
8	DMA Controller Enabled (0 = Disabled, 1=Enabled)
9	Interrupt Enabled (0=Disabled, 1=Enabled)
10	Not Used = 0
11	Burst Transfer Mode (0=Disabled, 1=Enabled)
13	Transfer Direction (0=Memory to I/O, 1=I/O to Memory)
14,15	Not Used

memory transactions take place, the counter is incremented at the end of every memory operation to guarantee that the address is set for the next transfer. The counter is incremented under the control of the cycle controller module. The CPU can read the 23-bit address with two I/O read operations, one for the lower 16 bits and another for the upper 7 bits.

Using the 74XXX TTL macrofunctions available in MAX+PLUS , the address generation function is implemented with six 4-bit, 74161-equivalent counters. These counters are arranged so that when each 4-bit counter increments to a binary count of 1111, its ripple carry output (RCO) enables the next higher 4-bit counter via the enable P (ENP) and enable T (ENT) inputs. The CPU parallel loads these counters via the data bus in two operations, one for the lower 16 bits and one for the upper 7 bits. *Figure5* shows the address generator schematic diagram.

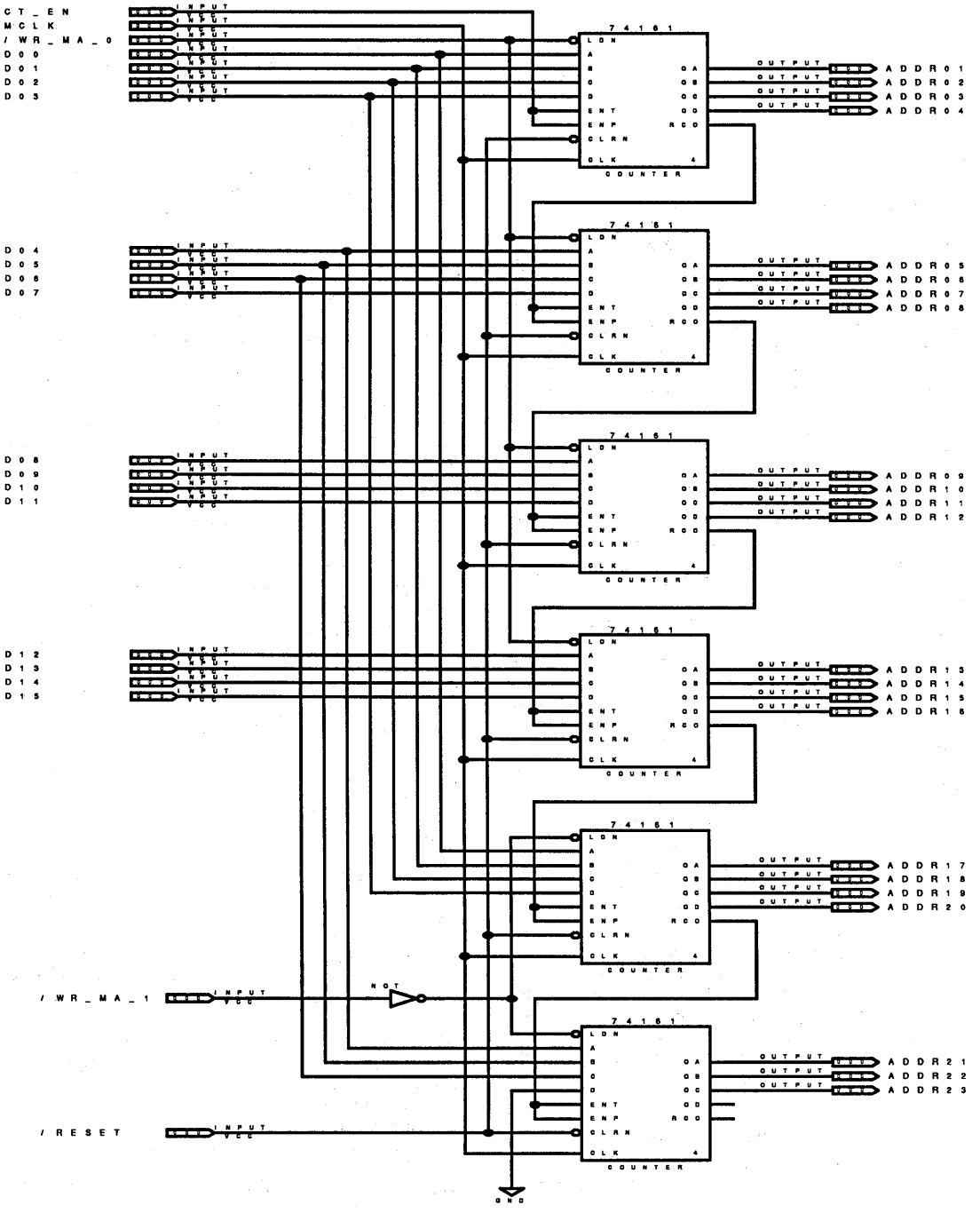


Figure 5. Address Generator Schematic

Word Counter Module

Because each transfer operation requires a word count, a 16-bit word counter monitors the number of words to be transferred. The CPU initializes this counter to a value representing one less than the total number of words to be transferred. This value allows the counter to reach zero before the last transfer and terminate the operation at the proper time, with the correct number of words transferred.

The 16-bit word counter is constructed utilizing four 74161 synchronous counter macrofunctions. Initialization of the word count occurs during set up of the transfer operation. The 16-bit word count value is 1's complemented (inverted) as it is loaded into the counter s. The counter is actually incremented instead of decremented, and the RCO output of the last counter indicates that the word count has reached zero. This arrangement requires only four macrocells or 3 percent of the CY7C342's resources, and thus allows use of the 74161 macrofunction. *Figure 6* shows the word counter schematic.

Three-State Buffers

When the CPU has ownership of the system bus, the DMAC's address, memory, and I/O control lines are in a high-impedance state. The data bus must also remain in a high-impedance state unless the CPU is reading the DMAC's internal register.

An octal three-state buffer implements the high-impedance interface function. The buffer uses eight Tri buffers from the macrocell library with the enables all tied together. These octal buffers correspond to the output portion of the CY7C342's I/O pins. The cycle control module output, DMAEN, enables the address, memory, and I/O control outputs. The RD_ENAB signal from the CPU decoder module enables the data outputs.

Output Multiplexer Module

The CPU must have access to the DMAC's internal registers to monitor operation. Because the MAX family of devices does not support an internal three-state bus, an alternative technique is employed when driving I/O pins from multiple sources within the device. Four 74157 2:1 multiplexer macrofunctions are placed in front of the 16 I/O pins for the data bus. With address input A1 connected to each 74157 mux's select input, either the address generator's lower 16 bits (when A1 = 0) or upper 7 bits (including the DMAC status information) are driven onto the data bus during a CPU read operation. *Figure 7* shows the output multiplexer schematic with octal buffers.

Cycle Control Module

The cycle control module controls DMAC memory and I/O operations. This finite state machine (FSM) handles the hold request (HREQ) and hold acknowledge (HLDA) handshake with the CPU. It also accepts DMA requests (DREQ) from I/O devices and acknowledges the requests with the output DACK. The FSM generates all memory and I/O interface timing, as well as the address/control output enables, counter increment/decrement operations, and interrupt generation (when enabled). The FSM is specified as a text design file and uses the AHDL state machine syntax with CASE statements to define the operation in each state. *Figure 8* shows the cycle controller state diagram, and *Appendix B* lists the cycle controller AHDL text design file.

Examining the state variable declarations in the cycle controller text design file reveals that all the internal control signals, handshake lines, and external interface signals are encoded into the FSM's state definitions. This text allows for a clear definition of each state using the fewest number of macrocells. However, this method can sometimes result in each macrocell having a complex Boolean expression that requires additional expanders. Thus, you might find it beneficial in some cases to allow MAX+PLUS to define an FSM's state definitions utilizing more macrocells, which can reduce the number of expanders. This approach relieves you from the responsibility of manually assigning state bits and often results in better performance.

Design Compilation

Upon completing each DMAC module, you can compile the module to eliminate any errors. If a design module contains an error, MAX+PLUS flags the error and takes you to the error's location in the schematic or text file. As each module compiles successfully, MAX+PLUS automatically generates a symbol representing the module design. These symbols are then incorporated in the DMAC's top-level schematic (*Appendix D*).

Once all DMAC blocks are integrated into the design, you can perform top-level compilation of the DMAC. You can compile the DMAC design from within MAX+PLUS 's graphic editor or from the compiler itself.

The compiler follows a series of steps during the compilation process ranging from netlist extraction to the creation of the object file used by the device programmer. First, the design processor creates a compiler netlist file (.CNF file) and tests for any design rule violations (output shorts, syntax errors in AHDL files, inputs and outputs not used, etc.). Next, the compiler generates a hierarchy interconnect file (.HIF file), which details the design's hierarchical interconnections. The database builder then flattens the design into a single level, maintaining the original design's function and connectivity. A logic synthesizer determines Boolean expressions for each logic function and primitive, allowing for the sum-of-products form required by the MAX EPLD architecture. Proprietary minimization algorithms remove redundant logic and reduce the number of required product terms.

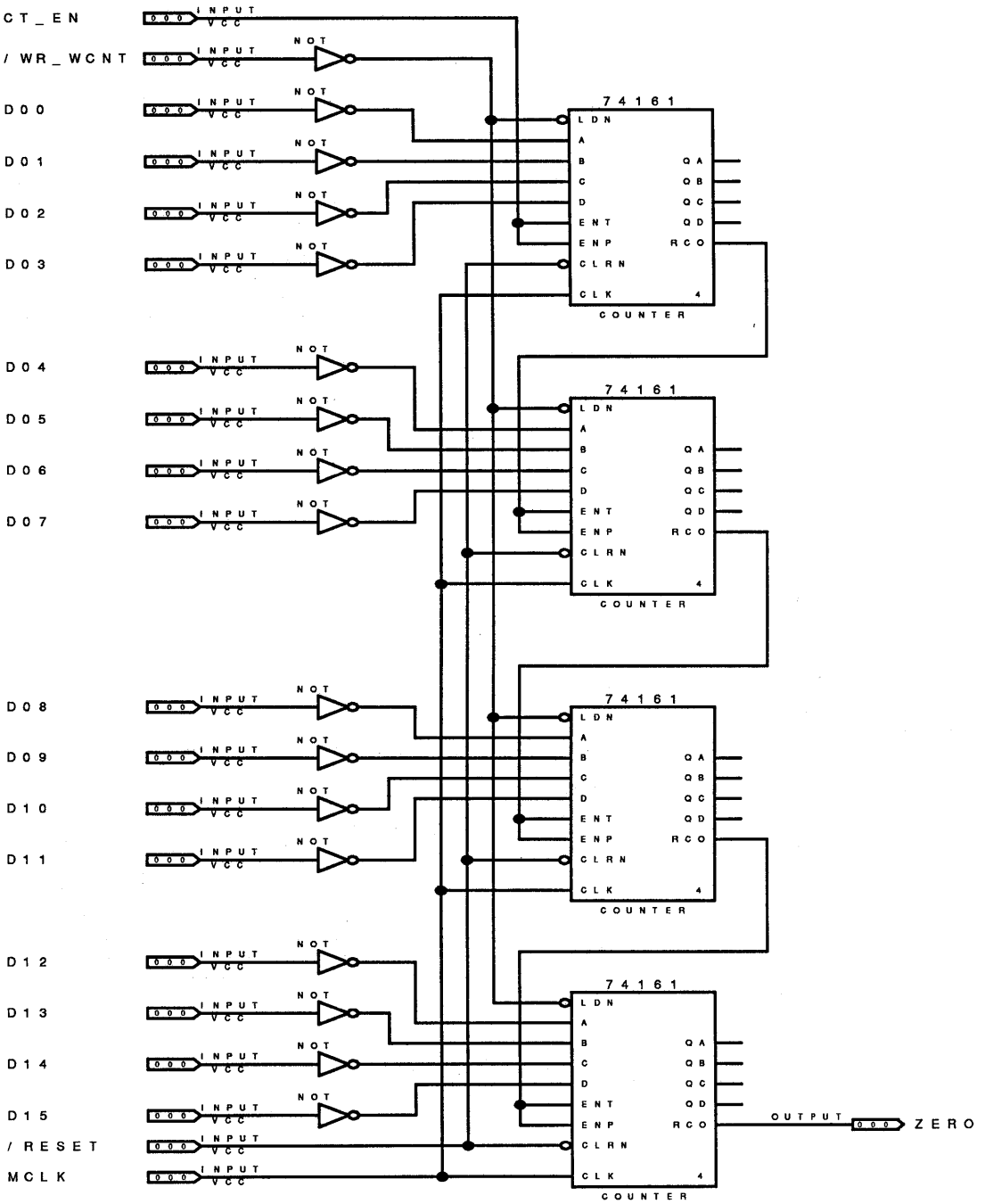


Figure 6. Word Counter Schematic

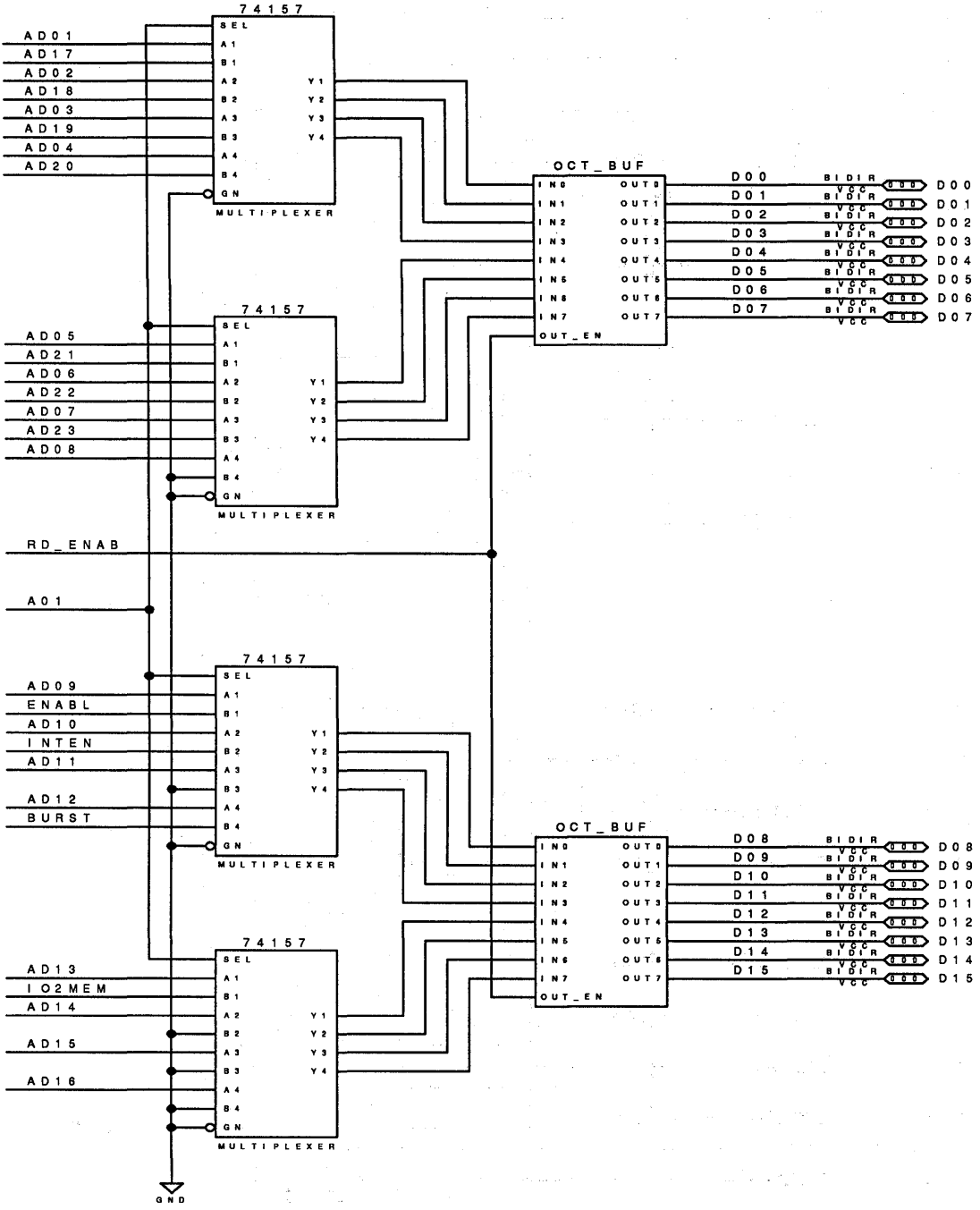


Figure 7. Output Mux Schematic W/Octal Buffers

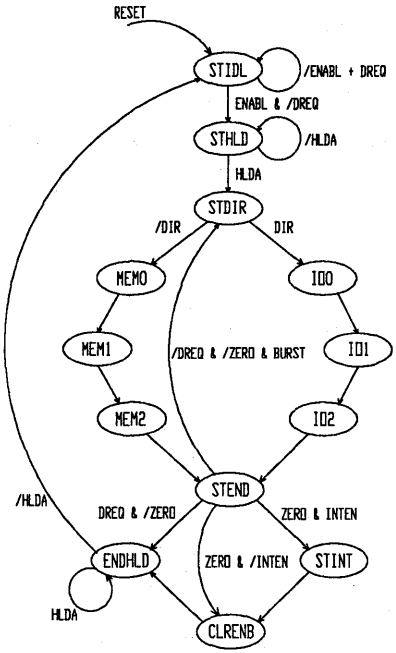


Figure 8. Cycle Controller State Diagram

A rule-based expert system then groups the design's logical requirements into a balanced number of macrocells and expanders. MAX+PLUS 's fitter allocates resources within the device, selecting the best macrocell location, pin assignments (if not already assigned), and interconnection paths. After a successful fit, MAX+PLUS creates a programmer object file (.POF file), which you can use to program a device. Whether the design compiled successfully or not, MAX+PLUS also creates a report file (.RPT file) that details the utilization of macrocells, expanders, interconnects, and I/O pins.

If the design compiled properly, you can program a device or verify the design through simulation. If the design did not compile, the compiler provides warnings and error messages to aid you in correcting any problems. Refer to the Cypress MAX+PLUS *User's Guide* for a complete list of error messages and recommended corrective actions.

Upon initial compilation of the DMAC design, the MAX+PLUS fitter determined that two of the LABs required more connections from the PIA than are available. The macrocell interconnection cross reference in the report file (*Appendix C*) revealed that portions of the CPU decode function were implemented with expanders in several LABs. This arrangement required the routing of many CPU decoder module inputs to each LAB. The compiler chose this approach to conserve macrocells, enhance performance, and prevent an additional PIA delay. However, this extra delay affects operation only when the CPU writes to or reads from the DMAC. Because the CPU typically requires a slow I/O operation to access the DMAC, the extra delay would cause no significant performance reduction.

Using the graphic editor, the DMAC schematic is altered by placing an MCELL buffer between each output of the CPU Decoder block and the destination. Placing the MCELL buffer after a module forces MAX+PLUS to place the logic function preceding the buffer in a macrocell. The module's output is then routed to all LABs via the PIA, resulting in an additional delay but requiring fewer interconnects. The design was recompiled and successfully fit into a CY7C342.

Design Verification

Design verification is an important step in the development of any programmable logic function and can be accomplished in several different ways. One way is to take a programmed device, insert it in a circuit, and observe it's behavior. This "plug and chug" technique works fine for simple devices performing well-defined functions; it does not work well for large, complex designs with major portions of the logic buried within the device.

The second, far more sophisticated approach, is to model the programmable logic function's behavior and simulate the operation before the part goes on a board. This design verification procedure is recommended for all but the most elementary designs to determine that the function and timing characteristics obtained match the system's requirements.

The MAX+PLUS simulator provides fast, easy design verification. It allows you to input circuit stimuli from either a vector or waveform file (.VEC and .SCF files, respectively). Output information can be stored in a table and compared during later simulation sessions or viewed in the waveform editor. You can create batch operations to automate the simulation process.

The simulator is accurate to 100 ps and features glitch, oscillation, and set-up/hold monitoring on every internal node within the selected device. This capability allows you to monitor the device's operation from a functional standpoint, utilizing worst-case timing parameters to guarantee proper operation in the application.

Although you can obtain "standard" DMA controllers, often they are not a good fit for the specific system you are designing. A custom gate array solution, with its high non-recurring engineering charges and long processing delays, is difficult to justify when compared to MAX EPLDs.

The Cypress CY7C340 family of EPLDs offer you capabilities far beyond those of earlier PLD generations. Through the use of the powerful tools in the MAX+PLUS development system, you can complete complex designs in less time, using fewer components, and achieve lower system cost than ever before.

Appendix A. CPU Decoder AHDL Text File

```
TITLE "CPU Address Decoder";              % CYPRESS SEMICONDUCTOR INC. %

%*****************************************************************************
                    CPU Interface Decoder for DMA Controller
*****************************************************************************%

SUBDESIGN cpu_decd (

%*****************************************************************************
                              Decoder Inputs
*****************************************************************************%
  ma2,   % Address Bit 2 %
  ma1,   % Address Bit 1 %
  /cs,   % Chip Select %
  /iowr, % I/O Write Signal %
  /iord  % I/O Read Signal %
  : INPUT;

%*****************************************************************************
                                                        Write Strobe Outputs
*****************************************************************************%
  /wr_ctrl,   % Control Register Write Strobe %
  /wr_wcnt,   % Word Count Write Strobe %
  /wr_ma_0,   % Lower Memory Address Write Strobe %
  /wr_ma_1,   % Upper Memory Address Write Strobe %
  rd_enabl    % Output Multiplexer Read Enable %
  : OUTPUT;
)

BEGIN
  TABLE
   ma2,ma1,/iord,/iowr,/cs = > /wr_ctrl,/wr_wcnt,/wr_ma_0,/wr_ma_1,rd_enabl;
   %-------------------------------------------------------------------%
    x, x,  0,   1,  0 = >   1,    1,    1,    1,    1;
    0, 0,  1,   0,  0 = >   0,    1,    1,    1,    0;
    0, 1,  1,   0,  0 = >   1,    0,    1,    1,    0;
    1, 0,  1,   0,  0 = >   1,    1,    0,    1,    0;
    1, 1,  1,   0,  0 = >   1,    1,    1,    0,    0;
  END TABLE;
END;
```

Appendix B. Cycle Controller AHDL Design File

```
TITLE "Cycle Controller";                    % CYPRESS SEMICONDUCTOR INC. %

%*****************************************************************************
                        MEMORY and I/O Cycle Controller
*****************************************************************************%

SUBDESIGN  cyc_ctrl(

%*****************************************************************************
                                cyc_ctrl Input Definitions
*****************************************************************************%
 reset,  % reset Input Active High %
 dreq,   % DMA Request Input %
 hlda,   % CPU Hold Acknowledge %
 zero,   % Word Counter Borrow output %
 enabl,  % DMA Enable Input %
 inten,  % Interrupt Enable %
 dir,    % Direction Bit: dir =  0 =  MEM TO I/O, dir =  1 =  I/O TO MEM %
 burst,  % Burst Enable Bit:  burst =  0 =  single xfers,
                              burst =  1 =  multiple xfers %
 clock   % System Clock %
 : INPUT;

%*****************************************************************************
                                cyc_ctrl Output Definitions
*****************************************************************************%
 count,      % Count Enable Bit %
 memw,       % Memory Write %
 memr,       % Memory Read %
 iowr,       % I/O Write %
 iord,       % I/O Read %
 dack,       % DMA Acknowledge %
 dmaen,      % DMA Address Enable %
 hreq,       % CPU Hold Request %
 setint,     % SET Interrupt Output %
 clrenb      % Clear DMA Enable bit %
 : OUTPUT;
)
```

Appendix B. Cycle Controller AHDL Design File (Continued)

```
VARIABLE
  cyc_ctrl : MACHINE OF BITS (q[10..0])
           WITH STATES ( stidl  = B"00000000000",
                         sthld  = B"00000001000",
                         stdir  = B"00000011000",
                         mem0   = B"00100111000",
                         mem1   = B"00110111000",
                         mem2   = B"00000111001",
                         io0    = B"00001111000",
                         io1    = B"01001111000",
                         io2    = B"10000111001",
                         stend  = B"10000011000",
                         stint  = B"00000001100",
                         endhld = B"10000000000",
                         clenb  = B"00000000010" );

BEGIN

    cyc_ctrl.clk = clock;    % system clock %
  cyc_ctrl.reset = reset;  % system reset %
       memw  = cyc_ctrl.q[9];
       memr  = cyc_ctrl.q[8];
       iowr  = cyc_ctrl.q[7];
       iord  = cyc_ctrl.q[6];
       dack  = cyc_ctrl.q[5];
       dmaen = cyc_ctrl.q[4];
       hreq  = cyc_ctrl.q[3];
       setint = cyc_ctrl.q[2];
       clrenb = cyc_ctrl.q[1];
       count = cyc_ctrl.q[0];

%  Q10 is a state variable to make all state definitions unique %

   CASE (cyc_ctrl) IS
     WHEN stidl = >          % Wait for Enable and Request %
       IF enabl & !dreq THEN cyc_ctrl = sthld;
       END IF;

     WHEN sthld  = >         % Wait for Hold Acknowledge %
       IF hlda THEN cyc_ctrl = stdir;
       END IF;



     WHEN stdir  = >         % Determine which direction %
       IF  dir THEN cyc_ctrl = io0; % I/O to Memory %
       ELSE cyc_ctrl = mem0;       % Memory to I/O %
       END IF;

     WHEN mem0  = >          % Memory Read and I/O Write %
          cyc_ctrl = mem1;

     WHEN mem1  = >
          cyc_ctrl = mem2;
```

Appendix B. Cycle Controller AHDL Design File (Continued)

```
    WHEN mem2 = >
        cyc_ctrl = stend;

    WHEN io0 = >          % I/O Read and Memory Write %
        cyc_ctrl = io1;

    WHEN io1 = >
        cyc_ctrl = io2;

    WHEN io2 = >
        cyc_ctrl = stend;

    WHEN stend = >          % Determine what to do next %
      IF !dreq & !zero & burst THEN cyc_ctrl = stdir;
      ELSIF dreq & !zero  THEN cyc_ctrl = endhld;
      ELSIF zero & inten  THEN cyc_ctrl = stint;
      ELSIF zero & !inten THEN cyc_ctrl = clenb;
      END IF;

    WHEN stint = >          % Set Interrupt, if Enabled %
      cyc_ctrl = clenb;

    WHEN clenb = >          % Clear Enable and Counters %
      cyc_ctrl = endhld;

    WHEN endhld = > % Wait for end of HOLD/ACK Sequence %
      IF !hlda THEN cyc_ctrl = stidl;
      END IF;

  END CASE;
END;
```

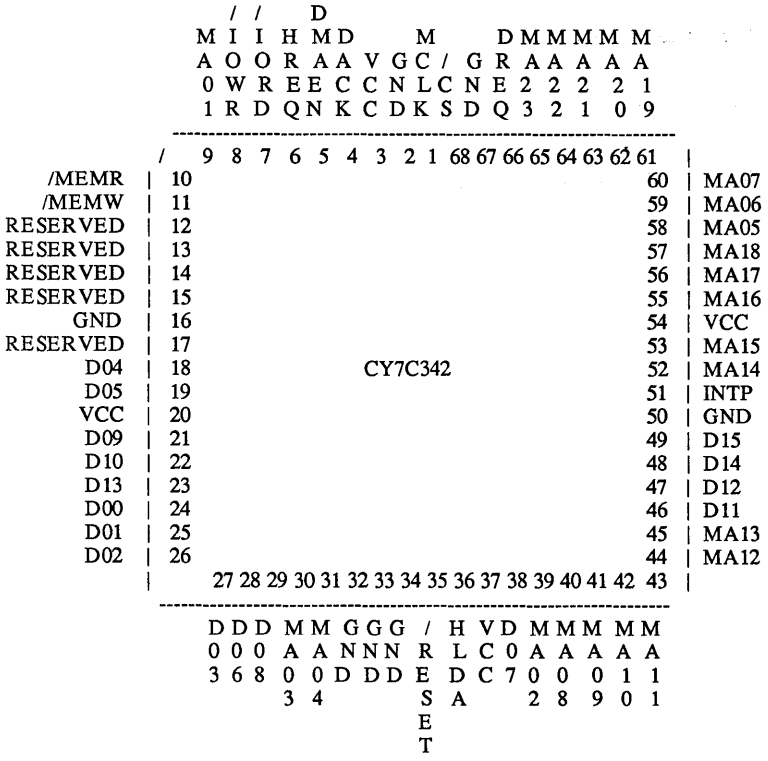
Appendix C. DMAC Report File

C:\MAX_WORK\DMAC_APP\DMAC.RPT

MAX+PLUS Compiler Report File
Version 2.03C 01/12/90

***** Design compiled without errors

Title: DMA CONTROLLER
Company: Cypress Semiconductor
Designer: Joe Engineer
Rev: A
Date: 12:25a 4-14-1990
Turbo: ON
Security: OFF



Appendix C. DMAC Report File (Continued)

C:\MAX_WORK\DMAC_APP\DMAC.RPT

** RESOURCE USAGE **

Logic Array Block	Macrocells	I/O Pins	Expanders	External Interconnect
A: MC1 - MC16	8/16(50%)	8/ 8(100%)	0/32(0%)	5/24(20%)
B: MC17 - MC32	0/16(0%)	0/ 5(0%)	0/32(0%)	0/24(0%)
C: MC33 - MC48	5/16(31%)	5/ 5(100%)	0/32(0%)	9/24(37%)
D: MC49 - MC64	8/16(50%)	8/ 8(100%)	0/32(0%)	16/24(66%)
E: MC65 - MC80	14/16(87%)	8/ 8(100%)	4/32(12%)	24/24(100%)
F: MC81 - MC96	7/16(43%)	5/ 5(100%)	0/32(0%)	9/24(37%)
G: MC97 - MC112	5/16(31%)	5/ 5(100%)	0/32(0%)	22/24(91%)
H: MC113 - MC128	8/16(50%)	8/ 8(100%)	0/32(0%)	24/24(100%)

Total dedicated input pins used:	5/ 8 (62%)
Total I/O pins used:	47/ 52 (90%)
Total macrocells used:	55/128 (42%)
Total expanders used:	4/256 (1%)
Total input pins required:	5
Total output pins required:	27
Total bidirectional pins required:	20
Total macrocells required:	55
Total expanders in database:	4
Synthesized macrocells:	0/128 (0%)

C:\MAX_WORK\DMAC_APP\DMAC.RPT

** FILE HIERARCHY **

|OCT_BUF:167|
|CYC_CTRL:106|
|CTRL_REG:73|
|CPU_DECD:75|
|ADDR_GEN:123|
ADDR_GEN:123	74161:66
ADDR_GEN:123	74161:71
ADDR_GEN:123	74161:70
ADDR_GEN:123	74161:69
ADDR_GEN:123	74161:68
ADDR_GEN:123	74161:67
WORD_CNT:125
WORD_CNT:125	74161:66
WORD_CNT:125	74161:69
WORD_CNT:125	74161:68
WORD_CNT:125	74161:67
74157:130
74157:132
74157:133
74157:134
OCT_BUF:56
OCT_BUF:55
OCT_BUF:35
OCT_BUF:168

Appendix C. DMAC Report File (Continued)

C:\ MAX_WORK\ DMAC_APP\ DMAC.RPT

** INPUTS **

Pin	MCell	LAB	Primitive	Expanders Total	Expanders Shared	Fan-In INP	Fan-In FBK	Name
68	-	-	INPUT	0	0	0	0	/CS
66	-	-	INPUT	0	0	0	0	DREQ
24	(49)	(D)	INPUT	0	0	0	0	D00
25	(50)	(D)	INPUT	0	0	0	0	D01
26	(51)	(D)	INPUT	0	0	0	0	D02
27	(52)	(D)	INPUT	0	0	0	0	D03
18	(33)	(C)	INPUT	0	0	0	0	D04
19	(34)	(C)	INPUT	0	0	0	0	D05
28	(53)	(D)	INPUT	0	0	0	0	D06
38	(65)	(E)	INPUT	0	0	0	0	D07
29	(54)	(D)	INPUT	0	0	0	0	D08
21	(35)	(C)	INPUT	0	0	0	0	D09
22	(36)	(C)	INPUT	0	0	0	0	D10
46	(81)	(F)	INPUT	0	0	0	0	D11
47	(82)	(F)	INPUT	0	0	0	0	D12
23	(37)	(C)	INPUT	0	0	0	0	D13
48	(83)	(F)	INPUT	0	0	0	0	D14
49	(84)	(F)	INPUT	0	0	0	0	D15
36	-	-	INPUT	0	0	0	0	HLDA
7	(4)	(A)	INPUT	0	0	0	0	/IORD
8	(5)	(A)	INPUT	0	0	0	0	/IOWR
9	(6)	(A)	INPUT	0	0	0	0	MA01
39	(66)	(E)	INPUT	0	0	0	0	MA02
1	-	-	INPUT	0	0	0	0	MCLK
35	-	-	INPUT	0	0	0	0	/RESET

Appendix C. DMAC Report File (Continued)

C:\ MAX_WORK\ DMAC_APP\ DMAC.RPT

** OUTPUTS **

Pin	MCell	LAB	Primitive	Expanders Total	Expanders Shared	Fan-In INP	Fan-In FBK	Name
4	1	A	DFF+	0	0	2	7	DACK
5	2	A	DFF+	0	0	4	7	DMAEN
24	49	D	OR2	0	0	1	3	D00
25	50	D	OR2	0	0	1	3	D01
26	51	D	OR2	0	0	1	3	D02
27	52	D	OR2	0	0	1	3	D03
18	33	C	OR2	0	0	1	3	D04
19	34	C	OR2	0	0	1	3	D05
28	53	D	OR2	0	0	1	3	D06
38	65	E	OR2	0	0	1	2	D07
29	54	D	OR2	0	0	1	2	D08
21	35	C	OR2	0	0	1	2	D09
22	36	C	OR2	0	0	1	2	D10
46	81	F	OR2	0	0	1	3	D11
47	82	F	OR2	0	0	1	3	D12
23	37	C	OR2	0	0	1	2	D13
48	83	F	OR2	0	0	1	2	D14
49	84	F	OR2	0	0	1	2	D15
6	3	A	DFF+	0	0	3	7	HREQ
51	85	F	OUTPUT	0	0	0	0	INTP
7	4	A	DFF+	0	0	2	8	/IORD
8	5	A	DFF+	0	0	2	7	/IOWR
9	6	A	DFF+	0	0	3	3	MA01
39	66	E	DFF+	0	0	3	4	MA02
30	55	D	DFF+	0	0	3	5	MA03
31	56	D	DFF+	0	0	3	6	MA04
58	113	H	DFF+	0	0	3	7	MA05
59	114	H	DFF+	0	0	3	8	MA06
60	115	H	DFF+	0	0	3	9	MA07
40	67	E	DFF+	0	0	3	10	MA08
41	68	E	DFF+	0	0	3	11	MA09
42	69	E	DFF+	0	0	3	12	MA10
43	70	E	DFF+	0	0	3	13	MA11
44	71	E	DFF+	0	0	3	14	MA12
45	72	E	DFF+	0	0	3	15	MA13
52	97	G	DFF+	0	0	3	16	MA14
53	98	G	DFF+	0	0	3	17	MA15
55	99	G	DFF+	0	0	3	18	MA16
56	100	G	DFF+	0	0	3	19	MA17
57	101	G	DFF+	0	0	3	20	MA18
61	116	H	DFF+	0	0	3	21	MA19
62	117	H	DFF+	0	0	3	22	MA20
63	118	H	DFF+	0	0	3	23	MA21
64	119	H	DFF+	0	0	3	24	MA22
65	120	H	DFF+	0	0	3	25	MA23
10	7	A	DFF+	0	0	2	9	/MEMR
11	8	A	DFF+	0	0	2	7	/MEMW

Appendix C. DMAC Report File (Continued)

C:\MAX_WORK\DMAC_APP\DMAC.RPT

** BURIED LOGIC **

Pin	MCell	LAB	Primitive	Expanders Total	Expanders Shared	Fan-In INP	Fan-In FBK	Name
-	96	F	DFF	0	0	2	1	|CTRL_REG:73|:9
-	95	F	DFF	0	0	2	1	|CTRL_REG:73|:33
-	80	E	DFF+	0	0	2	7	|CYC_CTRL:106|q0
-	79	E	DFF+	4	0	4	8	|CYC_CTRL:106|q10
-	78	E	MCELL	0	0	3	0	:152
-	77	E	MCELL	0	0	5	0	:153
-	76	E	MCELL	0	0	5	0	:155
-	75	E	MCELL	0	0	5	0	:156

C:\MAX_WORK\DMAC_APP\DMAC.RPT

** STATE MACHINE ASSIGNMENTS **

```
|CYC_CTRL:106|cyc_ctrl:    MACHINE
    OF BITS ( _MC079, _MC008, _MC007, _MC005, _MC004, _MC001, _MC002, _MC003, _MC080 ) WITH
      STATES (
        stidl  =  B"000000000",
        sthld  =  B"000000010",
        stdir  =  B"000000110",
        mem0   =  B"001001110",
        mem1   =  B"001101110",
        mem2   =  B"000001111",
          io0  =  B"000011110",
          io1  =  B"010011110",
          io2  =  B"100001111",
        stend  =  B"100000110",
        stint  =  B"000000010",
       endhld  =  B"100000000",
        clenb  =  B"000000000"
);
```

CYPRESS SEMICONDUCTOR

Interfacing PROMs and RAMs to a High-Speed DSP Chip using MAX

This application note describes how to interface Cypress CY7C128A Static RAMs and CY7C291A PROMs to the AT&T DSP16A using the CY7C343 64-Macrocell MAX EPLD. This design illustrates MAX's ability to integrate SSI and MSI logic for system cost and space savings.

The DSP16A includes a parallel port, with associated strobe signals, which is available for interfacing to external memory. The parallel port needs an external address generator when interfacing to RAM or PROM, and an EPLD of MAX's density suits this purpose.

Design Description

The conventional method of attaching external memory to the DSP16A is through its external memory interface, which comprises the following signals: a 16-bit ROM address bus; a 16-bit ROM data bus; and a clock-out signal, CKO, which cycles at 25 ns. Bear in mind two constraints when using this external memory interface. First, it allows only memory reads; second, it requires extremely fast PROM speeds. With the DSP16A-25, for example, you must use PROMs with 7-ns address access times (clock cycle - address delay - data set up = 25 - 5 - 13 = 7 ns). Memory devices at this performance level are very expensive.

The DSP16A parallel port provides a non-zero-wait-state alternative to the device's external memory interface, which accommodates both read and write memory accesses. A non-zero-wait-state external memory subsystem is appropriate because the DSP16A has 2K words of RAM on chip, and the external memory can download data or coefficients to the on-chip memory prior to time-critical computation (or equivalently, to upload data or results from the on-chip memory following time-critical computation). The design outlined in this application note requires four cycles (100 ns) to load a starting address, three cycles (75 ns) to perform a write operation, and five cycles (125 ns) to perform a read operation.

The DSP16A's 16-bit bidirectional parallel port includes three associated signals:

- PSEL, peripheral select—indicates which one of two logical ports, pdx0 or pdx1, is used during the current parallel I/O transaction
- PIDS, parallel input data strobe—asserted during a read transaction
- PODS, parallel output data strobe—asserted during a write transaction.

You can program the pulse width of both PIDS and PODS to be from one to four times the processor's cycle time (abbreviated as T). The pulse width is controlled by two bits in the DSP16A's parallel I/O control (PIOC) register. Two other bits in the PIOC define PIDS and PODS as either active (output signal) or passive (input signal). This design assumes that PIDS and PODS are in the active mode and that all 16 bits of the parallel port bus are configured to be bidirectional (PIOC's status/control bit equals 0).

Because the DSP16A parallel port lacks an address bus, it is necessary to create an external one. In this design example, the CY7C343 MAX implements an address generator and an address decoder. When fully utilized, this generator/decoder addresses up to 16K words of mixed ROM and RAM.

Design Details

Figure1 shows the block diagram for this design. In addition to the PROMs, SRAMs and MAX chip, the design requires a discrete 74F08 AND gate (more on this later). Note that the MAX chip generates four BANK/ and four BANK signals. The BANK/ lines control the CY7C128A SRAMs' active-Low chip enable, and the BANK lines connect to the CY7C291A PROMs' active-High chip select. You can modify the number of SRAMs versus the number of PROMs just by changing the MAX design, because the bank signals' timing is the same for both the active-High (BANK) and the active-Low (BANK/) version.

Figure2 shows a schematic of the CY7C343 logic. The four 74163s make up a 16-bit preloadable, auto-incrementing up counter. The 74138 decodes the eight memory-chip enables and chip selects, which are conditioned with the PSEL signal. SCLK is the symbol that

forces the MAX+PLUS compiler to use synchronous clocking.

The DSP16A's physical parallel I/O port connects to the two logical ports, pdx0 and pdx1, which distinguish between address and data transfers. When writing code for the DSP16A, you issue an external memory address from pdx1. This causes PSEL to go High and enables the load function on the 74163s; PODS' rising edge clocks the address from the parallel bus (PB00 - PB13) into the 74163s. The code for the DSP16A then reads data in through pdx0 or writes data out of pdx0.

In the case of a read, PSEL goes Low, which disables the 74163s' load function and enables the bank signal to the memories. Because PIDS is Low, the memories are output enabled, data returns to the DSP16A on the parallel bus, and PIDS's rising edge increments the 74163s.

In the case of a write, PSEL again goes Low. Because PODS is Low, the memories are write enabled, data is written from the DSP16A parallel bus, and PODS's rising edge increments the 74163s. *Appendix A* lists a code fragment that performs the read and write operations described here.

This design's conceptual operation is relatively straightforward. The challenge is to design the address-ing logic in MAX, determine the proper pulse widths for the PIDS and PODS strobes, and find memories with the appropriate speed.

Timing Diagrams

The timing diagrams for this design appear in *Figures 3* and *4*, with the corresponding timing parameters in *Tables 1, 2,* and *3*. Specifically, *Figure 3* shows an address load and back-to-back data writes. *Figure4* shows an address load and a single data read.

The CY7C343-30 timing parameters shown in these illustrations were calculated from the internal switching characteristics described in the device's data sheet; the MAX+PLUS simulator verified the signal timing. The address-generator circuit was captured with MAX+PLUS 's graphic editor, then compiled and simulated.

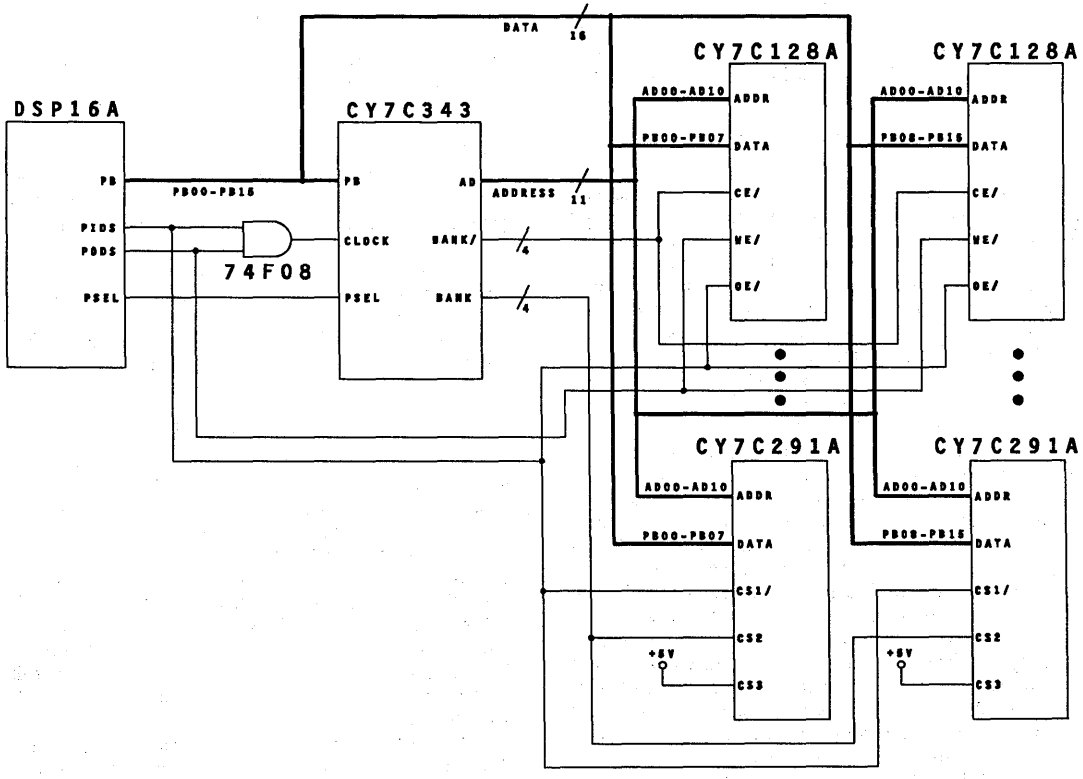


Figure 1. Block Diagram

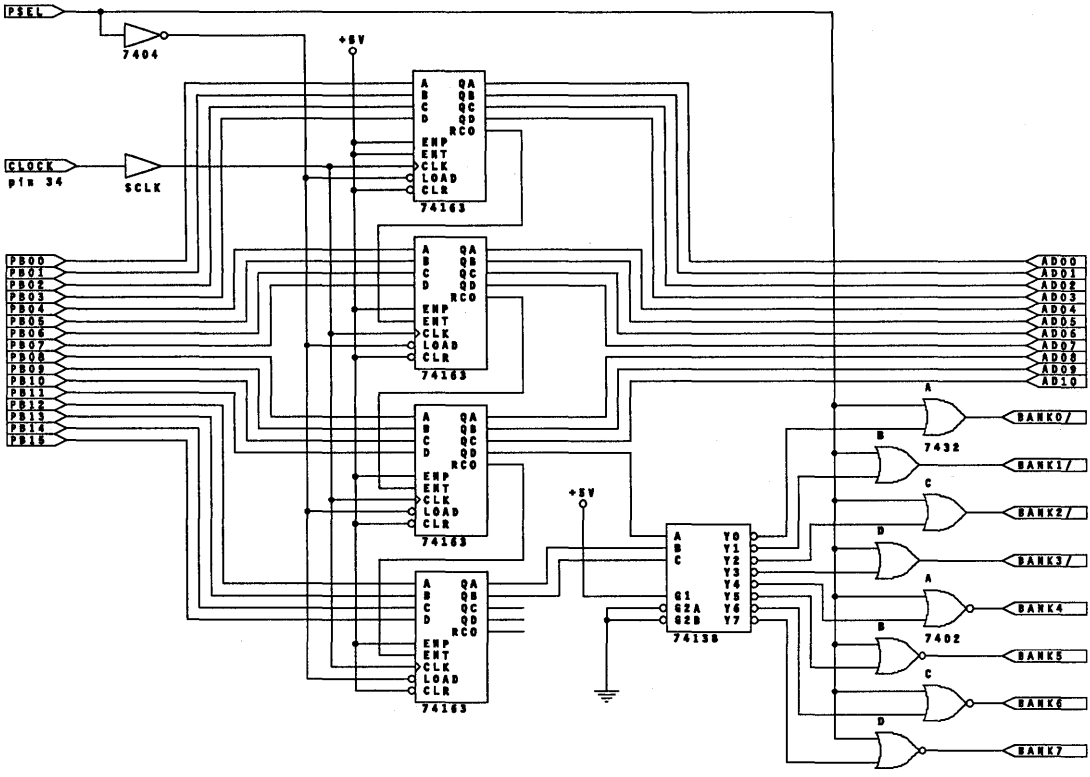


Figure 2. CY7C343 Schematic Diagram

Note that you could include inside the MAX chip the AND gate that uses PIDS and PODS to create the clock signal. This forces the MAX+PLUS compiler to use asynchronous clocking, however, pushing the I/O input hold time out beyond the 10 ns provided by the DSP16A (see P13 in the timing diagrams and parameter listings). Using an external AND gate allows the macrocell clocking to be synchronous, which eliminates the hold-time problem. The maximum propagation delay through the 74F08 AND gate is taken to be 6 ns (P14).

To create the design, you must determine the length of the PODS strobe during the address-load cycle. The critical requirement is the 34-ns 74163 set-up time (P15). Because the PD bus is valid 25 ns after PODS goes active (P13 in *Figure 3*), PODS must be programmed for a pulse width of 3T to meet the set-up time requirement. After the load cycle, PODS must go inactive for at least one cycle so that an address load takes 4T or 100 ns.

Similarly, it is necessary to determine the length of the PODS strobe during a data write. Using the CY7C128A-20 SRAM, the critical requirement is the 15-ns interval from chip-enable Low to the end of the write. With PODS programmed to 2T, the chip-enable-Low-to-write-end interval is guaranteed to be 18 ns (P21). A write to SRAM thus takes 3T, or 75 ns. As shown in *Figure 3*, this configuration also provides a write-enable pulse width of 50 ns (P22), data-set-up-to-

Table 1. DSP16A Parallel I/O Read-Cycle Specs

CKO High to PIDS Low = 15 ns max [P1]
CKO High to PIDS High = 15 ns max [P2]
PIDS Low to PSEL valid = 10 ns max [P3]
PIDS High to PSEL invalid (PSEL hold) = 25 ns min [P4]
PB valid before PIDS High (data set up) = 15 ns min [P5]
PIDS High to PB invalid (data hold) = 0 ns min [P6]

write-end time of 25 ns (P23), data-hold-from-write-end time of 10 ns (P24), address-set-up-to-write-end time of 53 ns (P25), write cycle time of 53 ns (P26), address-set-up-to-write-start time of 35 ns (P27), and address-hold-from-write-end time of 0 ns (P28).

The next requirement to be determined is the length of the PIDS strobe during a read operation. For the CY7C291A-20 EPROM, the critical parameter is the chip-select-active-to-data-valid time of 15 ns (see P29 in *Figure 4*). To meet this requirement, PIDS must be programmed to 4T, thus providing a total EPROM read cycle time of 5T, or 125 ns. This also provides an address-to-output-valid time of 69.5 ns (P30).

Now it is necessary to verify that this read cycle timing also meets the CY7C128A-20 SRAM's requirements. Again, the critical parameter is the chip-select-active-to-data-valid time, but for the SRAM this parameter is 20 ns (P31). As shown earlier, programming PIDS to 4T suffices for this operation, as well, so that an SRAM read cycle is also 5T, or 125 ns. As for the EPROM, this configuration provides the SRAM an address-to-data-valid time of 69.5 ns (P32) and an output-enable-to-data-valid time of 69.6 (P33).

Table 2. DSP16A Parallel I/O Write-Cycle Specs

CKO High to PSEL valid = 8 ns max [P7]
PSEL valid before PODS Low = 2.5 ns min [P8]
PODS High to PSEL invalid (PSEL hold) = 12.5 ns min [P9]
CKO Low to PODS Low = 8 ns max [P10]
CKO Low to PODS High = 8 ns max [P11]
PODS Low to PB valid = 25 ns max [P12]
PODS High to PB invalid (data hold) = 10 ns min [P13]

This MAX design is I/O intensive, rather than macrocell intensive. Although the design uses 96 percent of the I/O pins, it uses only 34% of the macrocells. Still, the CY7C343 in this example integrates five 16-pin and two 14-pin TTL MSI packages into a 44-pin PLCC

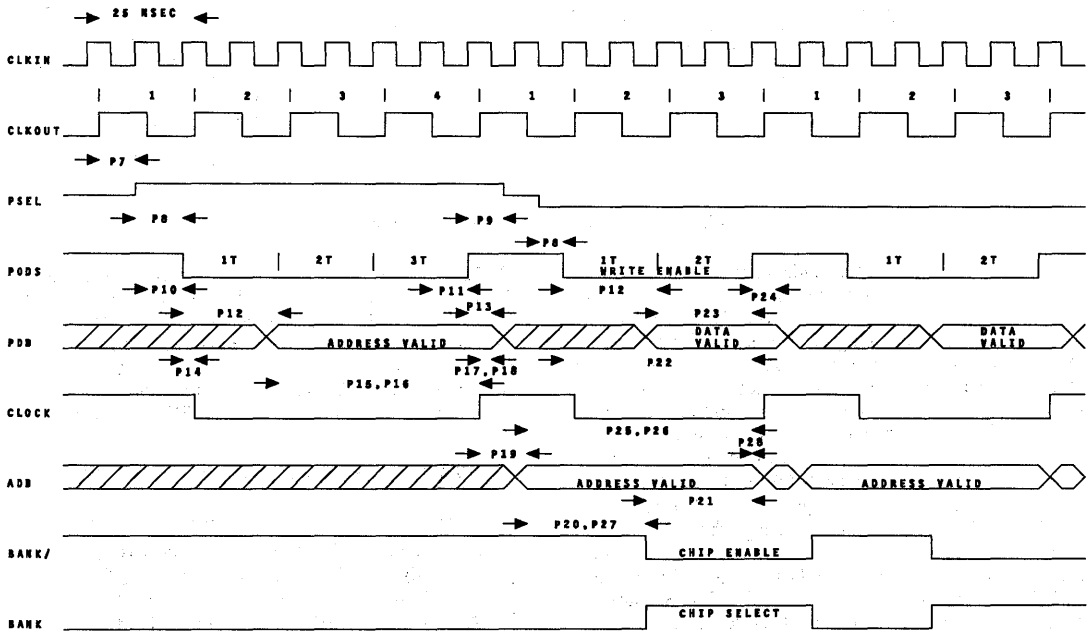


Figure 3. Address Load, Data Write, Data Write

package. In situations where space is at a premium, MAX is a very powerful integration tool.

A discrete implementation of the circuit outlined here requires approximately 1.6 square inches of board space, while the CY7C343 requires approximately 0.5 square inches. And with the CY7C343, you also have the advantage of design flexibility, especially when the CY7C343 is socketed. You could, for instance, change the MAX design to include up- and down-count addressing or accommodate higher density PROMs and SRAMs. MAX EPLDs are available in windowed packages for erasure under UV light; to make a change, you simply redesign, recompile, and reprogram the chips.

Because one MAX part replaces seven TTL parts, the MAX implementation offers inherently higher reliability. Inventory overhead is reduced, and the CY7C343 consumes 155 mA worst case versus 311 mA worst case for the FTTL parts it replaces.

Acknowledgments

The AT&T application note "Interfacing External RAM to the WE DSP16 Family of Digital Signal Processors" outlines a parallel-port implementation using discrete logic. Thanks to Daniel Yasi and Jim Flynn of AT&T.

Table 3. Critical CY7C343 Parameters

74163 set-up time (I/O input) =
Tio + Tpia + Tlad + Trsu - Tin - Tics =
5 + 16 + 14 + 8 - 7 - 2 = 34 ns [P15]
74163 set-up time (dedicated input) =
Tin + Tlad + Trsu - Tin - Tics =
7 + 14 + 8 - 7 - 2 = 20 ns [P16]
74163 hold time (I/O input) =
Tin + Tics - Tio - Tpia - Tlad + Trh =
7 + 2 - 5 - 16 - 14 + 8 =
-18 ns, assume 0 ns [P17]
74163 hold time (dedicated input) =
Tin + Tics - Tin - Tlad + Trh =
7 + 2 - 7 - 14 + 8 =
-4 ns, assume 0 ns [P18]
74163 clock-to-output time =
Tin + Tics + Trd + Tod =
7 + 2 + 2 + 5 = 16 ns [P19]
74138 propagation time =
Tfd + Tpia + Tlad + Tcomb + Tod - Tod =
1 + 16 + 14 + 4 + 5 - 5 = 35 ns [P20]

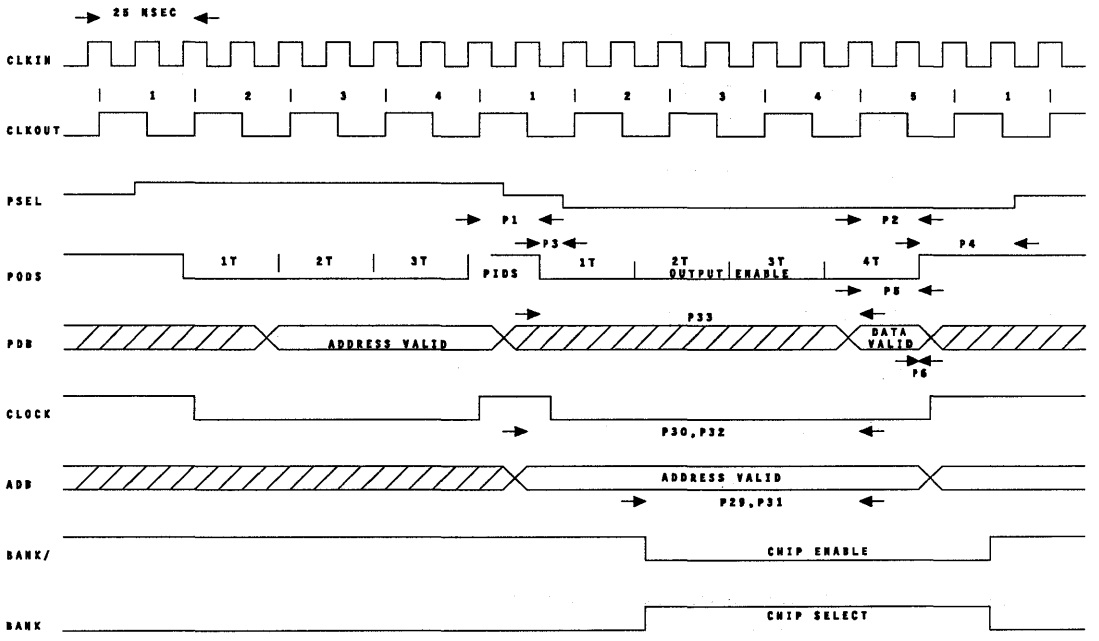


Figure 4. Address Load, Data Read

Appendix A. Code Fragment

This DSP16A program reads two words from external memory, multiplies them together, and writes the result back to external memory. Note that there is a latency of one read cycle during active read because of a double-buffering process. When a read statement is encountered, the data on which the program operates is taken from the on-chip input register. As the data is being taken from the input register, a read transaction is initiated on the physical port so that, at the end of the read cycle, the correct value is in the input register.

This program example is pathological because it does not make use of any pipelining or parallelism, of which the DSP16A is capable. Ordinarily, a large block of data would be downloaded, processed, and uploaded, not just a few words. However, this example does illustrate the steps necessary to address, read, and write external memory through the parallel port.

```
/* Issue Address                                          */

pioc=0x5800     /* not status/control mode (parallel bus 16-bit   */
                /* bidirectional), PIDS and PODS are outputs      */
                /* (active mode), PIDS and PODS strobe width      */
                /* equals three times the processor cycle time    */
                /* or 3T                                          */

pdx1=0x00       /* address external memory location 0x00          */

/* Read Data                                              */

pioc=0x7800     /* not s/c mode, PIDS and PODS active, PIDS       */
                /* and PODS strobe width equals 4T                */

a0=pdx0         /* first read not valid, discard                  */

a0=pdx0         /* second read valid, read first location of      */
                /* external memory into accumulator 0             */

a1=pdx0         /* read second location of external memory into   */
                /* accumulator 1                                  */

/* Process Data                                           */

x=a0            /* put first word into x register                 */

y=a1            /* put second word into y register                */

p=x*y           /* multiply words, result in p register           */

a0=p            /* put product back into a0                       */

/* Write Data                                             */

pioc=3800       /* not s/c mode, PIDS and PODS active, PIDS       */
                /* and PODS strobe width equals 2T                */

pdx0=a0         /* write product to third location in external    */
                /* memory                                         */
```

CYPRESS SEMICONDUCTOR

FIFO RAM Controller With Programmable Flags

This application note describes a scalable FIFO (first in, first out) RAM controller that provides all the control circuitry necessary to make a deep FIFO. The design uses off-the-shelf dual-port static RAMs (Cypress CY7C130s, for example). The controller also features an array of programmable flags that you can tailor to the specific needs of your project.

FIFOs are often used to buffer data transfers. The increasing volumes of data that must be manipulated and transferred between systems has prompted the need for large FIFOs.

FIFO RAM Controller Architecture

The FIFO RAM controller is implemented here in two stages. The first stage illustrates the architecture of the controller by implementing a shallow, 8-word-deep FIFO using a dual-port RAM. The second stage expands this scalable architecture to implement an 8-Kword-deep FIFO.

Typically, FIFOs are based on a dual-port RAM structure. This structure includes a memory cell that can be written to and read from at the same time. These devices are relatively inexpensive and provide the kind of asynchronous operation essential to a FIFO.

The design includes four primary sections: counter logic/address generation, flag generation, overflow control, and memory (*Figure 1*).

Data is written into the left port of the dual-port SRAM with an address supplied from the write counter. Data is read from the right port of the dual-port SRAM using an address supplied from the read counter.

The core of the design is the dual-port memory. A Cypress CY7C130 1K x 8 dual-port SRAM is used here. (Refer to "Understanding Dual-Port RAMs" in the Logic section of this book for more information on dual-port RAMs.) To make a wide FIFO, you add as many CY7C130s as necessary and address them in parallel. As mentioned earlier, you can implement deep FIFOs (even deeper than 8 Kwords) by scaling this design properly.

8-Word FIFO RAM Controller Operation

For the simple 8-word design, the signal names and their definitions are:

/MR — Master Reset

/SI — Shift in is the external signal used to write data into the FIFO

/SO — Shift out is the external signal that requests a read from the FIFO

RDADDR(3:0) — The dual-port read address, connected to A(3:0)R on the CY7C130

WRADDR(3:0) — The dual-port write address, connected to A(3:0)L on the CY7C130

FULL — The flag that indicates when the FIFO is full

ALMOST_FULL — The flag that indicates when the FIFO is 75 percent full

ALMOST_EMPTY — The flag that indicates when the FIFO is 25 percent full

EMPTY — The flag that indicates that the FIFO is empty

/SIINT — Internal shift-in signal for the dual-port RAM; connected to R/WL and /CEL of the CY7C130

/SOINT — Internal shift-out signal for the dual-port RAM; connected to /OER and /CER of the CY7C130

DATAIN(7:0) — Input data lines connected to I/OL(0:7) on the CY7C130

DATAOUT(7:0) — Output data lines connected to I/OR(0:7) on the CY7C130

BUSY_IN, BUSY_OUT — The busy flags on the dual-port RAM should be used to indicate when data can safely be shifted into or out of the device

Asserting /MR initializes the FIFO. This signal resets the write counter and read counter, so that they both point to location 0000. A master reset also clears the address latches and causes the EMPTY and ALMOST_EMPTY flags to be asserted.

The inverse of the FULL flag enables /SIINT, so that when the FIFO is full, no more data can be shifted in. This gated SI signal is connected to the R/WL and /CEL pins of the CY7C130. When /SI is asserted, and the FIFO has room, the dual-port RAM's left port is enabled and

put into read mode. Data on I/OL(7:0) is read into the FIFO. /OEL is permanently disabled.

Applying the first /SI pulse causes data from the I/OL(7:0) lines to be latched into memory. When the read is completed, the EMPTY flag is deasserted, indicating that there is data in the FIFO that can be shifted out. The ALMOST_EMPTY flag stays asserted if the FIFO contains two or fewer valid words (25 percent full). ALMOST_EMPTY deasserts when there are three or more valid data words in the FIFO.

If six consecutive shift-in cycles are completed without a shift-out cycle, the ALMOST_FULL flag is asserted. This means that the FIFO is 75 percent full. After eight consecutive shift-in cycles without a shift-out cycle, the FULL flag is asserted, signalling that the FIFO is full, and no more data can be shifted in.

The inverse of the EMPTY flag enables /SOINT, so that when the FIFO is empty, invalid data is not read. This gated SO signal is connected to the /OER and /CER pins of the CY7C130. When /SO is asserted, and the FIFO contains valid data, the data is driven to the I/OR(7:0) pins. R/WR is tied high, forcing the right port to read mode.

If the FIFO is full and a shift-out cycle is completed, the FULL flag is deasserted, because the FIFO is no longer full. Once there are less than six words in the FIFO (less than 75 percent full), the ALMOST_FULL flag is deasserted. When there are only two words left in the FIFO, the ALMOST_EMPTY flag is asserted. If all the valid words in the FIFO have been read, the EMPTY flag is asserted. Waveforms for this circuit appear in *Figure 2*.

Due to the asynchronous nature of dual-port RAMs, the FIFO can be written to or read from at any time (unless, of course, the FIFO is full or empty). It is a good idea to monitor the flags when /SI and /SO are both deasserted, however, to give the internal logic time to settle. The flags can be safely monitored at any time if you can guarantee that only one operation is performed at a time. Note that flags are updated when /SI and /SO are High.

Counter/Address Generation Logic

The two counters in this design serve as a read counter and a write counter (*Figure 3*). The write counter provides an address to the memory port that is being written to. This counter is incremented every time an /SI (shift in) pulse is received, until the FIFO is full. When the FIFO is full, the FULL flag is asserted and the counter is inhibited, preventing data overflow. When the FIFO is no longer full due to a read, the FULL flag deasserts and the write counter is enabled again.

The read counter provides the address to the memory port that is being read from. This counter is incremented

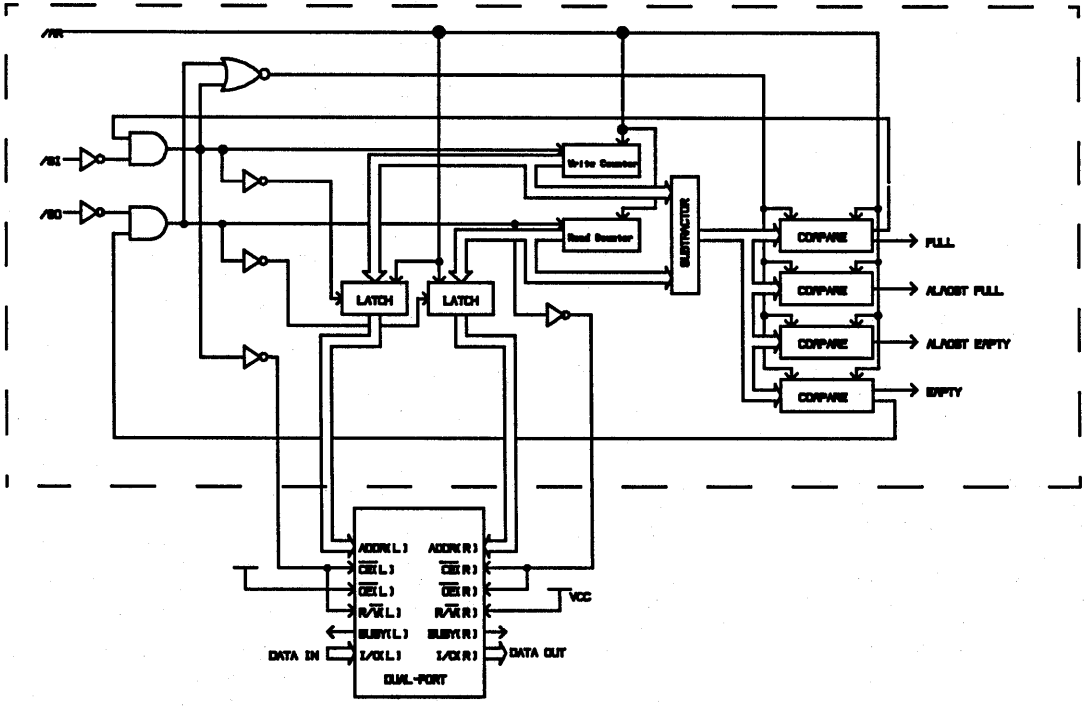


Figure 1. Block Diagram of a FIFO RAM Controller

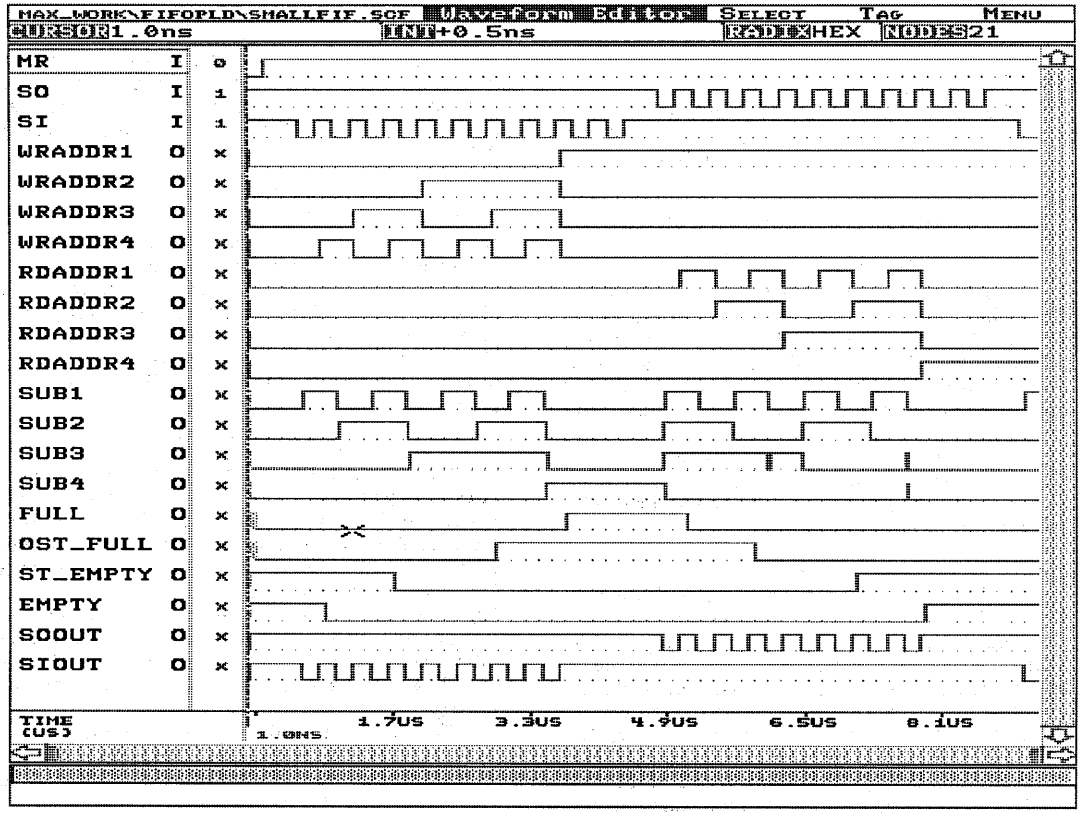


Figure 2. Sample Waveforms for the 8 Word FIFO Example

every time an /SO pulse is received, until the FIFO is empty. When the FIFO is empty, the EMPTY flag is asserted, and the read counter is inhibited until some valid data has been written to the FIFO.

Both the read and write counters are forced to 0000 when /MR is asserted.

/SI and /SO are Low-asserted signals, as are their internal counterparts /SIINT and /SOINT. The counters are incremented on the High-to-Low transition of /SIINT or /SOINT. Note that SIINT and SOINT clock the counters. The counters must be clocked on the signal's rising edge, but the external signals are Low asserted; an inverter is thus required. With this scheme, the current address is always stable before the next /SI or /SO pulse.

The address lines are latched to keep the address stable during the Low-asserted /SI or /SO pulse. When /SI or /SO deasserts, the latches enable the new address to the dual-port RAM. In simple form, the counter is incremented on the falling edge of /SI or /SO, and the new address is latched-out on the rising edge of /SI or /SO. Note that the rising edge of the /SI or /SO signal propagates through the CY7C342 PLD used to implement this design; the signal continues out to the dual-port RAM before the new address appears, thereby allowing the correct address at the rising edge of /SI or /SO before changing to the next address.

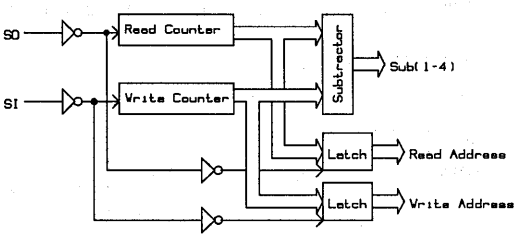


Figure 3. Counter/Address Generation Logic and

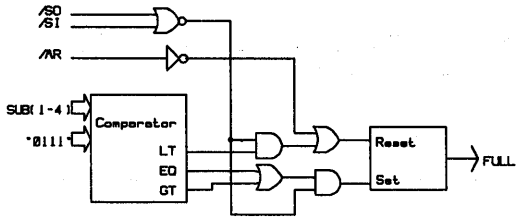


Figure 4. The FULL Flag Circuit

Flag Control Logic

The flag control logic consists of several comparators and a subtraction circuit, which keeps track of the number of valid data words currently in the FIFO. This is done by subtracting the value in the read counter from the value in the write counter. The output of the subtraction circuit is compared with values you supply. The flags result from these comparisons.

Because this is a PLD solution, you can customize the flags to suit your needs. This example includes only four flags, but you can create as many flags as needed to check for any number of words.

The FULL Flag

The FULL flag logic appears in *Figure 4*. The comparator takes its A port inputs from the subtractor. Its B port inputs come from tying the first 3 bits high and the fourth to ground (0111 binary or 7 decimal). The value of 0111 is used instead of 1000 (8 decimal) because the first address is 0000, not 0001, and thus the subtract circuit's output is actually 1 less than the number of valid words in the FIFO. The 0111 value also allows the flag to be processed concurrently with other activities. In other words, the FULL flag is asserted on the eighth valid word, rather than waiting until the ninth, when it is too late. Again, you can customize the flag to be asserted whenever you need the information.

The flag comes from an RS latch. The latch is set only when the comparator indicates that the subtract logic value is equal to or greater than 0111 (7). The comparator's greater-than output is not strictly necessary here, but it is included as a safeguard. The latch's set input is gated by a combination of /SIINT and /SOINT that is asserted only when neither /SI or /SO are asserted. This arrangement ensures that the flags do not change until the counters both settle. Remember that the counters change on the falling edge of /SIINT and /SOINT, and thus settle by the time /SI or /SO is deasserted.

The reset input of the FULL flag latch is fed by the master reset (/MR), the comparator's less-than output and the gating signal discussed above. In other words, the FULL flag is reset when the number of valid words is less than 8 and both /SI and /SO are deasserted, or when an /MR pulse is received.

The ALMOST_FULL Flag

For the ALMOST_FULL flag (*Figure 5*), the comparator takes its A port inputs from the subtract circuit; the comparator's B port inputs are tied to 0101 (5). The operation of this flag is almost identical the that of the FULL flag, except that the ALMOST_FULL flag is asserted when there are 6 or more valid words in the FIFO. This is where the comparator's greater-than output comes into play. Note that the ALMOST_FULL flag is always asserted while the FULL flag is asserted.

The ALMOST_EMPTY Flag

The ALMOST_EMPTY flag comparator (*Figure 6*) takes its A port inputs from the subtractor and its B port inputs from the value 0010 (2). The /MR signal feeds the set input of the RS latch, because when the part is reset, the FIFO is empty. Because this means there are less than 2 valid data words in the FIFO, the ALMOST_EMPTY flag is asserted. The flag is also set by the comparator's less-than or equal-to outputs, gated by the deasserted /SIINT and /SOINT, as before. Consequently, whenever the subtractor gives a value less than or equal to 0010 (2), the flag is set.

The reset of the ALMOST_EMPTY flag is fed by the comparator's greater-than output, gated with /SIINT and /SOINT. Thus, when the subtractor indicates a value greater than 0010 (2), the flag is reset.

The EMPTY Flag

The EMPTY flag comparator (*Figure 7*) takes its A port input from the subtractor. The comparator's B port input is tied to 0000. The RS latch is set with the /MR signal or when the subtractor value is 0000, which indicates that the FIFO is empty. Any other value at the subtract circuit clears the EMPTY flag. Again, /SIINT and /SOINT gate all signals except /MR.

Overflow Control

The controller contains a simple mechanism to prevent data overflow and underflow — the condition

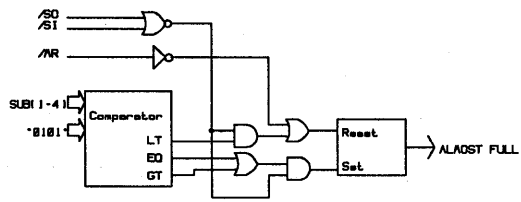


Figure 5. The ALMOST_FULL Flag Circuit

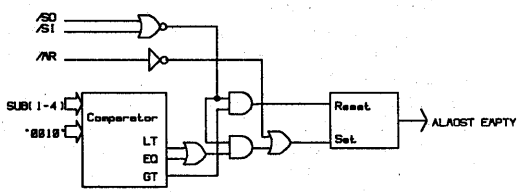


Figure 6. The ALMOST_EMPTY Flag Circuit

where an address that does not contain current, valid data is read. Specifically, when the FULL flag goes active, its inverse is ANDed with the SI signal to block any further shift-in pulses from entering the system. This keeps the write address counter from incrementing and the dual-port RAM from accepting any more data. When a shift-out pulse is received, the FULL flag resets, and data can be shifted into the FIFO again.

When the FIFO is empty, the inverse of the EMPTY flag is ANDed with the SO signal to block any further shift-out operations. This keeps the read address counter from incrementing and disallows any further reads from the dual-port RAM. When the next shift-in pulse is received, the EMPTY flag resets, and data can be shifted out again.

Referring back to *Figure 2*, notice that 10 total /SI pulses are sent, but only eight /SIINT pulses are generated.

Scaling the Controller to 8 Kwords

So far this application note has analyzed the basic functions of a FIFO RAM controller with a simplified example. Next, consider a real-world example of an 8-Kword-deep FIFO.

The counters are expanded to provide a 13-bit address that drives the dual-port memories. The comparators and subtract circuit are also expanded to 13 bits. To facilitate the large design, the comparators are eliminated, and the desired toggle point is decoded using a 13-input AND structure. Put the decoded output into S/R of the NORLTCH (a latch made of NORs). The memory core now consists of four CY7C132 2K X 8 dual-port RAMS.

The CY7C344 PLD is used to implement the design's first stage. The final implementation scales into the CY7C342 in much the same way as the design itself. The CY7C344 is a high-density (1250 gate equivalent) 28-pin, 32-macrocell PLD. The design fits nicely into this device, as shown in the design report file in *Appendix D*. All the macrocells are used, while still leaving five inputs, two outputs, and half the available p-terms unused.

The MAX+PLUS Tools

The FIFO RAM controllers described here have been implemented using the MAX+PLUS design package. This well-integrated, user-friendly package provides schematic capture, a high-level design language, simulation, and programming facilities. MAX+PLUS also supports hierarchical designs.

The schematic capture utility features an easy-to-use, mouse-driven, pull-down menu format. Most of the macrofunctions used in the design come from the MAX+PLUS macrofunction library. This library includes everything from basic logic components, such as a two-input AND gate (AND2), to an 8-bit counter (8COUNT) and 7400 Series logic. These robust libraries facilitate quick, efficient schematic generation.

The MAX+PLUS package provides a high-level design language, AHDL (Advanced Hardware Design Language). AHDL allows you to create a textual description of your design using Boolean equations, truth tables, and state machine syntax.

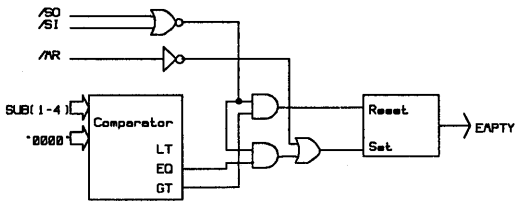


Figure 7. The EMPTY Flag Circuit

If you need a new macrofunction, you can easily create exactly what you need using schematic capture, AHDL, or a combination of both. When the new macrofunction is completed, a symbol for it can be created automatically.

Creating a Macrofunction

Because the MAX+PLUS library contains no subtractor macrofunction, the subtract circuit is created using AHDL:

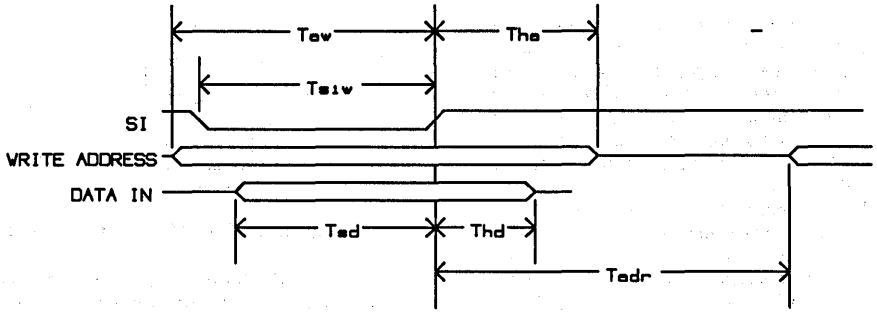
```
subdesign subtract
(
wrcnt[12..0], rdcnt[12..0] : input;
sub[12..0] : output;
)
begin
(„sub[12..0])=(0,1,wrcnt[12..0])-(0,0,rdcnt[12..0]);
end;
```

The first line of code specifies that this subdesign, or macrofunction, is called "subtract." This is an arbitrary choice. Next, the inputs and outputs are defined as being 13 bits wide (0 through 12). Finally, the required function is defined between the begin and end statements. The term "0,1," before wrcnt[12..0] assures that wrcnt is always bigger than 0,0,rdcnt. It is like adding another significant digit. Thus, when a subtract occurs, this macrofunction always returns a positive number. This number represents the magnitude difference between the write counter and the read counter. The subtract function returns a 2's complement number; for simplicity of design, the result should therefore always be positive. If the result were negative, the comparators would have to be very complex. For convenience, the MAX+PLUS Text Editor is used to create this file, which is called SUBTRACT.TDF.

After the function is implemented, the design is complied using the COMPILE function in the pull-down menu. The logic is automatically generated and a symbol created that can now be placed in the design. The compiler is run at this stage to turn the subtractor into a functioning macrocell, but do not run the compiler on the entire design until you have finished total design capture.

Design Verification and Simulation

After the entire design has been entered, the MAX+PLUS compiler checks the design to detect any



Parameter	Description	Min	Max
Tsw	Address Set-Up to Write End	20	
Tha	Address Hold from Write End	2	
Tsiw	Shift In Pulse Width	20	
Tsd	Data Set-Up to Write End	15	
Thd	Data Hold from Write End	35	
Tsdr	Rising Edge SI to Next Address		65

Figure 8 Simple FIFO Timing

design rule violations. When an error is detected, the MAX+PLUS software is so well integrated that it jumps to the schematic or text editor and highlights the error. This feature is extremely helpful during the initial debug phase of the design. The compiler also creates all the files necessary for design simulation and device programming.

Simulation

MAX+PLUS provides a simulation package that allows you to test your design. You can define waveforms using tabular entry format or a waveform editor. Vectors entered in the tabular format can be converted and displayed as waveforms. The MAX+PLUS simulator performs both timing and logic simulation. The MAX+PLUS simulation facility generated the timing diagram shown in *Figure 2*.

The code used to generate the waveforms is created in the MAX+PLUS text editor. The top part of the file defines the inputs and outputs. START and STOP commands permit you to start and stop the simulation at a given time. INTERVAL defines the time between the lines of code; an interval of 200 means the inputs change every 200 ns. The rest of the file consists of columnized entries for /MR, /SO, and /SI. *Appendix A* lists the vector file for the 8-word FIFO RAM controller, and *Appendix C* contains the vector file for the 8-Kword controller.

Simulating a design as large as the 8-Kword FIFO RAM controller takes a prohibitively long period of time (16,000 cycles), without some kind of complex simulation capability. The MAX+PLUS simulator provides such features through the implementation of command and vector files. The vector file used to exercise the flags for the 8K FIFO appears in *Appendix C*.

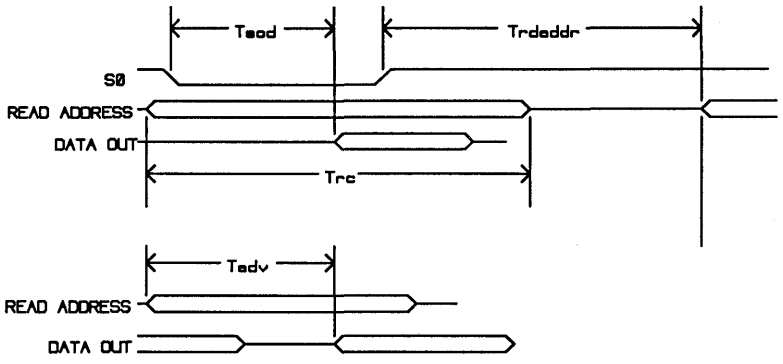
Note that a vector file can run without a command file. A typical vector file contains the START, STOP, and INTERVAL statements explained earlier. Additionally, a PATTERN statement sets up the inputs and the subsequent pattern of 1s and 0s provides the desired input stimulation. This simple compilation of rows of 1s and 0s quickly exercises a design. You can enter the vector file in the text editor; the filename must end with a .VEC extension.

When you are ready to simulate, you use the pull-down menu to activate "FILE," then "VECTOR input," followed by "VECTOR file (.VEC)." This sequence of commands fetches the vector file as input stimulus. Now activate the SIMULATE command. You are prompted for the length of the simulation, and simulation begins. When the simulation is completed, activate the WAVEFORM command to display the results.

To further aid in complex simulation, you can create a command file, which includes the instructions needed to execute a simulation sequence. *Appendix B* lists the command file used to exercise the flag operation for the 8K FIFO. The command file works in conjunction with a vector file (*Appendix C*).

The first line of the command file in *Appendix B* defines that the GROUP command accepts hex input. Next, the VECTOR command uses the given vector file as a source of stimulation. GROUP shortens the description of a group of inputs/nodes by allowing you to describe them in hex format instead of defining every input individually in binary format. You can type 07FE, for example, instead of 0000011111111110.

The "SIMULATE 2000ns" command gives the amount of time the simulator runs for this section of the



Parameter	Description	Min	Max
Tsod	Shift Out to Data Valid		65
Trdaddr	Shift Out to Valid Address		65
Trc	Read Cycle Time	95	
Tsdv	Valid Address to Data Valid		25

Figure 9 Read Cycle

command file. In this example, the simulation starts by running for 2000 ns. As shown in the COMMAND & VECTOR files, this resets (MR) the device and allows a couple of shift-in operations (SI) to occur. Next, a FORCE STICK command forces the RDCNT and WRCNT to a desired value. "SIMULATE +100ns" allows this value to be implemented. The + in front of the number means that this value is for an additional amount of time, or incremental. The absence of the + indicates an absolute time.

Now a FORCE UNSTICK command allows the nodes to simulate freely. The next "SIMULATE +1800ns" allows for two SIs followed by two SOs. This sequence sets the counter to a desired value just before the ALMOST_FULL flag activates; the sequence also provides SIs to activate the counter and SOs to deactivate it.

The rest of the code sets counters next to the value that toggles flags and activates and deactivates the counters. These two files save a great deal of simulation effort and compute time.

Timing Analysis

The MAX+PLUS software offers valuable timing information. The timing data from the MAX+PLUS simulation and the information in the CY7C132 data sheet allow complete timing analysis of the FIFO RAM controller. *Figures 8* and *9* show the controller's timing waveforms. The pertinent timing parameters and their values are:

t_{AW} = 20 ns: Dual-port address set-up to write end
t_{HA} = 2 ns: Dual-port address hold from write end
t_{PWE} = 20 ns: Dual-port /WE pulse width
t_{SD} = 15 ns: Dual-port data set-up to write end
t_{HD} = 0 ns: Dual-port data hold to write end
t_{ADR} = 30 ns: CY7C344 /SI to next clock
t_{ACE} = 30 ns: Dual-port /CE Low to data out
t_{RDADDR} = 30 ns: CY7C344 /SO to valid address
t_{RC} = 25 ns: Dual-port read cycle time
t_{AA} = 25 ns: Dual-port valid address to data valid

Timing simulation data from the VECTOR and COMMAND files indicates the worst-case timing from /SI or /SO until flags are stable is approximately 95 ns. This value, coupled with the normal timing parameters for shifting data in and out of the dual-port RAM, gives a 200-ns FIFO RAM controller system cycle time.

Because this is a very complex design, you can change the logic to add or delete features. Note that you can use the minimum /SI or /SI pulse width, but keep the overall period at approximately 200 ns.

Programming Support

When the design is complete and fully exercised, a device can be programmed using the MAX+PLUS programmer module in conjunction with the QuickPro II programmer hardware. The small design is fitted to a CY7C344, and the expanded design is fitted to a CY7C342.

With the proper software and adapters, the QuickPro II is versatile enough to program all the MAX devices, as well as every PROM and PLD Cypress manufactures. The QuickPro II is connected to a PC via a parallel port, leaving the slots in your PC available for other peripherals.

The complete MAX+PLUS package contains the MAX+PLUS software, the QuickPro II programmer and software, and adapter sockets for the entire MAX family.

The designs presented here have been verified by simulation.

Appendix A. Simulation File: 8 Word FIFO RAM Controller

```
SIMPLE EXAMPLE SIMULATION CODE

START 0;
STOP 9000;
INTERVAL 200;
OUTPUTS WRADDR1 WRADDR2 WRADDR3 WRADDR4 RDADDR1 RDADDR2 RDADDR3 RDADDR4 SUB1 SUB2 SUB3 SUB4 FULL ALMOST_FULL ALMOST_EMPTY EMPTY FULL SOOUT SIOUT ;
INPUTS MR SO SI;
PATTERN
0 1 1
%RESET%
1 1 1
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
% SHIFTED IN 8-BYTES %
1 1 0
1 1 1
1 1 0
1 1 1
1 1 1
% SHIFTED 2 EXTRA TIMES TO ENSURE PROPER FLAG OPERATION %
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
% SHIFTED OUT 8-BYTES %
1 0 1
1 1 1
1 0 1
1 1 1
1 1 1
% SHIFTED OUT 2 EXTRA TIMES TO ENSURE PROPER FLAG OPERATION %
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
1 1 0
1 1 1
% SHIFTED IN 8-BYTES%
1 1 0
1 1 1
1 1 0
1 1 1
% SHIFTED IN 2 EXTRA BYTES TO ENSURE PROPER FLAG OPERATION%
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
1 0 1
1 1 1
%SHIFTED IN 8-BYTES%
1 0 1
1 1 1
1 0 1
1 1 1
%SHIFTED 2 EXTRA TIMES TO ENSURE PROPER FLAG OPERATION%
1 1 1 ;
```

Appendix B. Simulation Command File: 8K FIFO RAM Controller

```
                    /* This file executes preloads to simplify
                    /* simulation of the FIFO RAM Controller
RADIX HEX
VECTOR BIGFIF.VEC
GROUP CREATE RDCNT = RCNT13 RCNT12 RCNT11 RCNT10 RCNT9 RCNT8 RCNT7 RCNT6 RCNT5 RCNT4
RCNT3 RCNT2 RCNT1
GROUP CREATE WRCNT = WCNT13 WCNT12 WCNT11 WCNT10 WCNT9 WCNT8
                     WCNT7 WCNT6 WCNT5 WCNT4 WCNT3 WCNT2 WCNT1

SIMULATE 2000NS              /* initialize, turn EMPTY flag off
FORCE STICK WRCNT=07FE       /* force controller to ALMOST EMPTY BOUNDARY
FORCE STICK RDCNT=0000
SIMULATE +100NS              /* preload
FORCE UNSTICK WRCNT          /* relieve force
FORCE UNSTICK RDCNT
SIMULATE +1800NS             /* do several writes to turn flag on
                                    /* do several reads to turn flag off

FORCE STICK WRCNT=0FFE       /* force controller to ALMOST FULL boundary
FORCE STICK RDCNT=0000
SIMULATE +100NS              /* preload
FORCE UNSTICK WRCNT          /* relieve force
FORCE UNSTICK RDCNT
SIMULATE +1800NS             /* do several writes to turn flag on
                                    /* do several reads to turn flag off

    FORCE STICK WRCNT=1FFE/* force controller to FULL boundary
    FORCE STICK RDCNT=0000
    SIMULATE +100NS/* preload
    FORCE UNSTICK WRCNT  /* relieve force
    FORCE UNSTICK RDCNT
    SIMULATE +1800NS/* do a write to turn flag on, attempt more
    /* do several reads to turn flag off
```

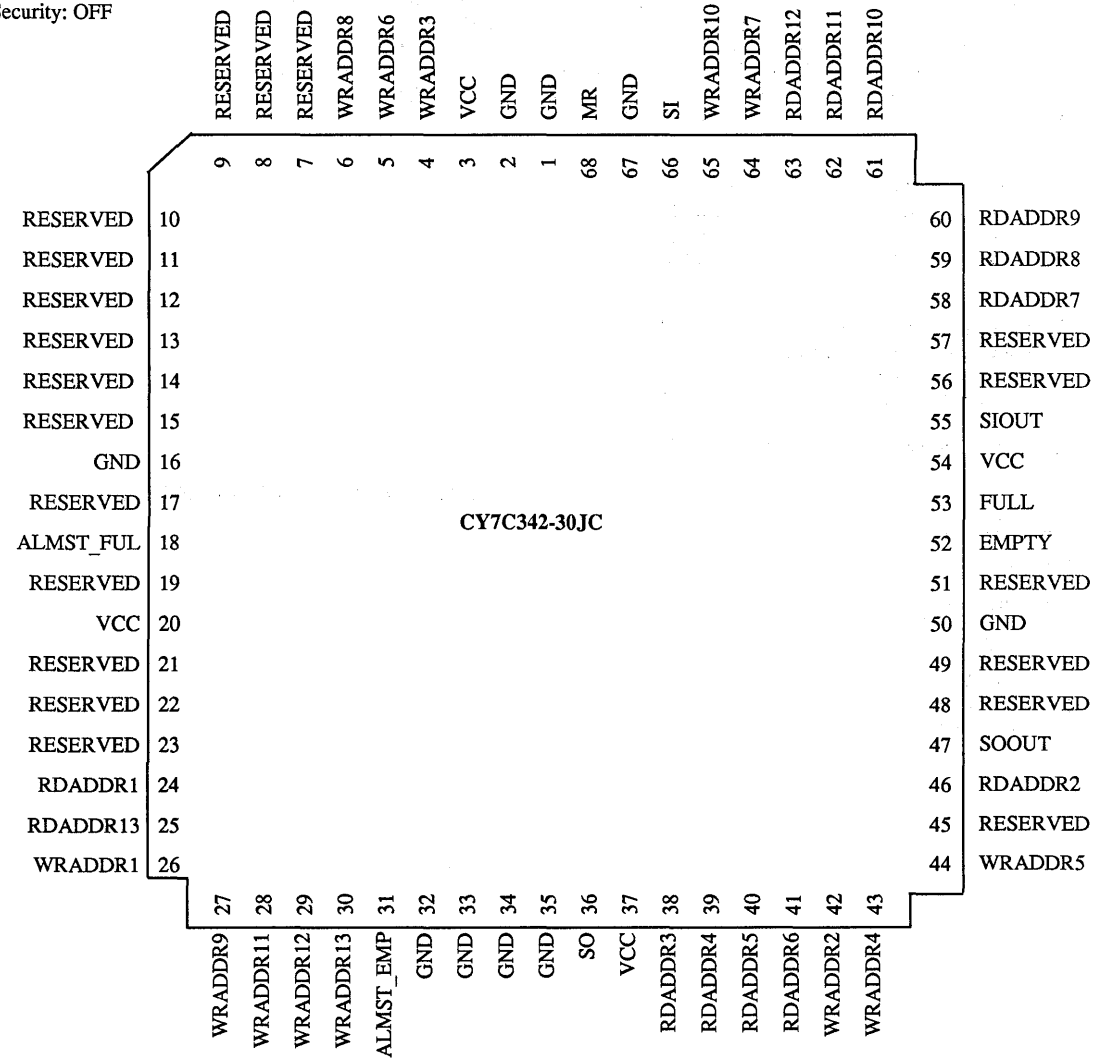
Appendix C. Simulation File: 8K Word FIFO RAM Controller

```
% Simulation file for the FIFO RAM Controller
START 0;
STOP 399;
INTERVAL 100;
% Each cycle is 100 ns long

OUTPUTS     WCNT13 WCNT12 WCNT11 WCNT10
            WCNT9 WCNT8 WCNT7 WCNT6
            WCNT5 WCNT4 WCNT3 WCNT2
            WCNT1 RCNT13 RCNT12 RCNT11
            RCNT10 RCNT9 RCNT8 RCNT7
            RCNT6 RCNT5 RCNT4 RCNT3
            RCNT2 RCNT1 WRADDR13
            WRADDR12 WRADDR11
            WRADDR10 WRADDR9 WRADDR8
            WRADDR7 WRADDR6 WRADDR5
            WRADDR4 WRADDR3 WRADDR2
            WRADDR1 RDADDR13 RDADDR12
            RDADDR11 RDADDR10 RDADDR9
            RDADDR8  RDADDR7 RDADDR6
            RDADDR5 RDADDR4 RDADDR3
            RDADDR2 RDADDR1
            FULL ALMST_EMP ALMST_FUL
            EMPTY SIOUT SOOUT ;
INPUTS      MR SI SO;
PATTERN
0 1 1
% master reset and initialization
1 1 1
1 1 1
1 1 1;

START 400;
STOP 7500;
INTERVAL 100;
INPUTS  MR SI SO;
PATTERN 1  0  1
     1  0  1
     1  1  1
     1  1  1
     1  0  1
     1  0  1
     1  1  1
     1  1  1
     1  1  0
     1  1  0
     1  1  1
     1  1  1
     1  1  0
     1  1  0
     1  1  1
     1  1  1
     1  1  1
     1  1  1
     1  1  1
     1  0  1
     1  1  1
     1  1  1
     1  0  1
     1  0  1
     1  1  1
     1  1  1
     1  0  1
     1  0  1
     1  1  1
     1  1  1
     1  1  0
     1  1  0
     1  1  1
     1  1  1
     1  1  0
     1  1  0
     1  1  1
     1  1  1;
```

Appendix D. Report File: 8K Word FIFO RAM Controller

C:\MAX_WORK\FIFOPLD\BIGFIF.RPT

MAX+PLUS Compiler Report File
Version 2.50C 7/18/90
***** Design compiled without errors
Title: DESIGN NAME
Company: CYPRESS SEMICONDUCTOR
Designer: MIKE LEWIS
Rev: A
Date: 6:41p 11-01-1990
Turbo: ON
Security: OFF



Appendix D. Report File: 8K Word FIFO RAM Controller

C:\MAX_WORK\FIFOPLD\BIGFIF.RPT

** RESOURCE USAGE **

Logic Array Block	Macrocells	I/O Pins	Expanders	External Interconnect
A: MC1 - MC16	16/16(100%)	3/ 8(37%)	16/32(50%)	15/24(62%)
B: MC17 - MC32	0/16(0%)	0/ 5(0%)	0/32(0%)	0/24(0%)
C: MC33 - MC48	1/16(6%)	1/ 5(20%)	16/32(50%)	14/24(58%)
D: MC49 - MC64	10/16(62%)	8/ 8(100%)	9/32(28%)	12/24(50%)
E: MC65 - MC80	16/16(100%)	7/ 8(87%)	10/32(31%)	14/24(58%)
F: MC81 - MC96	16/16(100%)	2/ 5(40%)	32/32(100%)	13/24(54%)
G: MC97 - MC112	3/16(18%)	3/ 5(60%)	32/32(100%)	14/24(58%)
H: MC113 - MC128	16/16(100%)	8/ 8(100%)	12/32(37%)	15/24(62%)

Total dedicated input pins used:	3/ 8 (37%)
Total I/O pins used:	32/ 52 (61%)
Total macrocells used:	78/128 (60%)
Total expanders used:	127/256 (49%)

Total input pins required:	3
Total output pins required:	32
Total bidirectional pins required:	0
Total macrocells required:	78
Total expanders in database:	74

Synthesized macrocells: 0/128 (0%)

C:\MAX_WORK\FIFOPLD\BIGFIF.RPT

** FILE HIERARCHY **

|8COUNT:284|
|NORLTCH:353|
|NORLTCH:344|
|NORLTCH:141|
|NORLTCH:135|
|BIGSUB:261|
|8COUNT:193|
|8COUNT:194|
|8COUNT:283|

Appendix D. Report File: 8K Word FIFO RAM Controller

C:\MAX_WORK\FIFOPLD\BIGFIF.RPT

** INPUTS **

Pin	MCell	LAB	Primitive	Expanders Total	Expanders Shared	Fan-In INP	Fan-In FBK	Name
68	-	-	INPUT	0	0	0	0	MR
66	-	-	INPUT	0	0	0	0	SI
36	-	-	INPUT	0	0	0	0	SO

C:\MAX_WORK\FIFOPLD\BIGFIF.RPT

** OUTPUTS **

Pin	MCell	LAB	Primitive	Expanders Total	Expanders Shared	Fan-In INP	Fan-In FBK	Name
31	56	D	OUTPUT	5	3	1	3	ALMST_EMP
18	33	C	OUTPUT	16	2	1	14	ALMST_FUL
52	97	G	OUTPUT	16	16	1	14	EMPTY
53	98	G	OUTPUT	16	16	1	14	FULL
24	49	D	DFF	0	0	1	2	RDADDR1
46	81	F	DFF	0	0	1	2	RDADDR2
38	65	E	DFF	0	0	1	2	RDADDR3
39	66	E	DFF	0	0	1	2	RDADDR4
40	67	E	DFF	0	0	1	2	RDADDR5
41	68	E	DFF	0	0	1	2	RDADDR6
58	113	H	DFF	0	0	1	2	RDADDR7
59	114	H	DFF	0	0	1	2	RDADDR8
60	115	H	DFF	0	0	1	2	RDADDR9
61	116	H	DFF	0	0	1	2	RDADDR10
62	117	H	DFF	0	0	1	2	RDADDR11
63	118	H	DFF	0	0	1	2	RDADDR12
25	50	D	DFF	0	0	1	2	RDADDR13
55	99	G	MCELL	16	16	2	14	SIOUT
47	82	F	MCELL	16	16	2	14	SOOUT
26	51	D	DFF	0	0	1	2	WRADDR1
42	69	E	DFF	0	0	1	2	WRADDR2
4	1	A	DFF	0	0	1	2	WRADDR3
43	70	E	DFF	0	0	1	2	WRADDR4
44	71	E	DFF	0	0	1	2	WRADDR5
5	2	A	DFF	0	0	1	2	WRADDR6
64	119	H	DFF	0	0	1	2	WRADDR7
6	3	A	DFF	0	0	1	2	WRADDR8
27	52	D	DFF	0	0	1	2	WRADDR9
65	120	H	DFF	0	0	1	2	WRADDR10
28	53	D	DFF	0	0	1	2	WRADDR11
29	54	D	DFF	0	0	1	2	WRADDR12
30	55	D	DFF	0	0	1	2	WRADDR13

Appendix D. Report File: 8K Word FIFO RAM Controller

C:\MAX_WORK\FIFOPLD\BIGFIF.RPT

** BURIED LOGIC **

Pin	MCell	LAB	Primitive	Expanders Total	Expanders Shared	Fan-In INP	Fan-In FBK	Name
-	64	D	SOFT	2	1	0	2	|BIGSUB:261|:71
-	80	E	SOFT	1	1	0	4	|BIGSUB:261|:82
-	79	E	SOFT	0	0	0	2	|BIGSUB:261|:87
-	78	E	SOFT	0	0	0	2	|BIGSUB:261|:90
-	77	E	SOFT	3	2	0	6	|BIGSUB:261|:94
-	76	E	SOFT	3	2	0	6	|BIGSUB:261|:97
-	75	E	SOFT	2	1	0	3	|BIGSUB:261|:107
-	74	E	SOFT	3	2	0	5	|BIGSUB:261|:119
-	73	E	SOFT	4	4	0	7	|BIGSUB:261|:131
(45)	72	E	SOFT	4	4	0	7	|BIGSUB:261|:134
-	128	H	SOFT	2	1	0	3	|BIGSUB:261|:144
-	127	H	SOFT	3	2	0	5	|BIGSUB:261|:156
-	126	H	SOFT	4	4	0	7	|BIGSUB:261|:168
-	125	H	SOFT	4	4	0	7	|BIGSUB:261|:171
-	124	H	SOFT	2	1	0	3	|BIGSUB:261|:181
-	123	H	SOFT	3	2	0	5	|BIGSUB:261|:193
-	122	H	SOFT	4	4	0	7	|BIGSUB:261|:205
-	121	H	SOFT	4	4	0	7	|BIGSUB:261|:208
-	63	D	SOFT	2	0	0	3	|BIGSUB:261|:218
-	96	F	DFF	16	16	2	14	rcnt1
-	95	F	DFF	16	16	2	15	rcnt2
-	94	F	DFF	16	16	2	16	rcnt3
-	93	F	DFF	16	16	2	17	rcnt4
-	92	F	DFF	16	16	2	18	rcnt5
-	91	F	DFF	16	16	2	19	rcnt6
-	90	F	DFF	16	16	2	20	rcnt7
-	89	F	DFF	16	16	2	21	rcnt8
-	88	F	DFF	16	16	2	22	rcnt9
-	87	F	DFF	16	16	2	23	rcnt10
-	86	F	DFF	16	16	2	24	rcnt11
(48)	83	F	DFF	16	16	2	25	rcnt12
(49)	84	F	DFF	16	16	2	26	rcnt13
-	16	A	DFF	16	16	2	14	WCNT1
-	15	A	DFF	16	16	2	15	WCNT2
-	14	A	DFF	16	16	2	16	WCNT3
-	13	A	DFF	16	16	2	17	WCNT4
-	12	A	DFF	16	16	2	18	WCNT5
-	11	A	DFF	16	16	2	19	WCNT6
-	10	A	DFF	16	16	2	20	WCNT7
-	9	A	DFF	16	16	2	21	WCNT8
(7)	4	A	DFF	16	16	2	22	wcnt9
(8)	5	A	DFF	16	16	2	23	wcnt10
(9)	6	A	DFF	16	16	2	24	wcnt11
(10)	7	A	DFF	16	16	2	25	wcnt12
(11)	8	A	DFF	16	16	2	26	wcnt13
(51)	85	F	MCELL	32	32	3	13	:242

Section Contents

CYPRESS SEMICONDUCTOR

Understanding Small FIFOs

FIFO is an acronym for first in first out. In digital electronics, a FIFO is a buffer memory organized such that the first data entered into the memory is the first data read from the memory.

Software FIFOs

Software FIFOs are used extensively in computer programs where tasks are placed in queues waiting for execution. Data is also exchanged in a similar manner. In programmer's language, the process (program) that puts data into the memory is the "producer" and the program that takes data out of the memory is the "consumer." In a conventional RAM, the producer and the consumer cannot access the memory simultaneously. It is the programmer's responsibility to ensure that contention does not occur.

Data transfer via a shared memory is a standard programming technique, but it is not feasible to have the processor in the data path for data rates greater than 5 Mbytes/s. Higher data rates require use of direct memory access (DMA), a FIFO, or some combination of these two hardware techniques.

In system design, once procedures are standardized and verified in software, hardware can replace the software. The benefits are improved performance, reduction of software, ease of design, and usually reduced costs.

A Recent History of Small FIFOs.

The first monolithic FIFO was the 3341 introduced by Fairchild in 1969. This 64 X 4 (64 words of 4 bits each) device was fabricated using P-channel MOS transistors, required +5V, -12V, and ground supplies, had a maximum operating frequency of 1 MHz, and was packaged in a 16-pin DIP. The 3341 was second-sourced in P-channel MOS by TI and AMD. In 1973, Fairchild introduced the 9403 FIFO, which was also 64 X 4, but had both serial and parallel inputs and outputs and was fabricated using bipolar technology. Signetics licensed the 9403 from Fairchild.

In 1979, MMI and NSC came out with an "improved 3341," which they called the 401 and 402. These bipolar devices operated at maximum read and write frequencies of 15 MHz. The 401 was 64 X 4 with a no connect on pin 1, where the 3341 had -12V. Thus, it was a pin-for-pin, improved-performance, functionally equivalent replacement for the 3341. The 402 was 64 X 5 and packaged in an 18-pin DIP. Otherwise, it was functionally identical to the 401.

The Register Array Architecture

The early FIFOs use what is called a register array architecture , in which the FIFO is implemented as an array of parallel flip-flop registers; the outputs of each flip-flop drive the inputs of the next. In parallel with the data registers is a single-bit register whose depth equals that of the FIFO. When data is written to the FIFO, a "valid data bit" is entered into this register; when data is read, the bit is cleared.

Control logic causes the data word at the FIFO input to propagate to either the FIFO output or to the empty location next to a valid word. Control logic also prevents writing into a full FIFO (overrun) but not explicitly reading from an empty FIFO (underrun). The parallel data words are physically propagated serially (i.e., sequentially in time) through the FIFO registers.

Figure 1 shows the input stage in the register array architecture. This input stage is a 1-word by m-bit-wide parallel shift register controlled by the input handshaking signals SI (Shift In) and IR (Input Ready). For generality, *Figure 1* illustrates an m-bit-wide, N-word-deep FIFO.

The output stage is also a 1-word by m-bit-wide parallel shift register, which is controlled by the output handshaking signals SO (Shift Out) and OR (Output Ready).

Of the total of 64 parallel registers, the middle 62 are controlled by signals derived from SI, IR, SO, and OR.

Each word in the FIFO has an associated flag bit that tells whether the data stored in that word is valid. The usual convention is to set the bit to a One when the data is written and to clear it when the data is read.

Fallthrough and Bubblethrough

The preceding statements regarding input and output stages are not precisely correct under two special conditions, which occur when the FIFO is empty or full.

When the FIFO is empty, the data must enter the input stage, propagate through the register array, and enter the output stage. The time required to do this is called the fallthrough time and it limits the output data rate.

When the FIFO is full and one word is read, all the remaining words must move down one word. The empty location propagates from the output to the input. The time required to do this is called the bubblethrough time, and it limits the input data rate.

FIFOs that use the register array architecture have large fallthrough and bubblethrough times (1.6 µs), which seriously reduce the FIFO's maximum throughput rates. In addition, the maximum input and output frequencies are a function of the number of words in the FIFO. This phenomenon is called fullness sensitivity.

Maximum Throughput Calculations

The fallthrough and bubblethrough times limit a FIFO's maximum throughput when the FIFO is empty and full, respectively. The minimum throughput period ($T_{min.}$) corresponding to the stand-alone period (t_A) and the fallthrough time (t_F) is:

$$T_{min.} = t_A + t_F$$

Converting to frequency yields

$$\frac{1}{F_{max.}} = \frac{1}{F_A} + t_F$$

where $F_{max.}$ is the FIFO's maximum throughput frequency and F_A is the maximum stand-alone operating frequency.

Rearranging and solving for $F_{max.}$ yields

$$F_{max.} = \frac{1}{\frac{1}{F_A} + t_F} \qquad \textit{Eq. 1}$$

Thus, the larger the fallthrough time, the lower the maximum throughput frequency. Also, note that if the fallthrough time can be made equal to zero, then $F_{max.} = F_A$. This means that you can operate the FIFO continously at its maximum specified input and output frequencies. Today's FIFOs, referred to as zero-fallthrough-time FIFOs, all use a dual-port RAM cell and pointers to address the RAM.

For register-array-architecture FIFOs, t_F = 1.6 µs and FA = 15 MHz. Substiting these values into Equation 1 yields $F_{max.}$ = 599.88 KHz, which is considerably less that the 15 MHz specified on the data sheet.

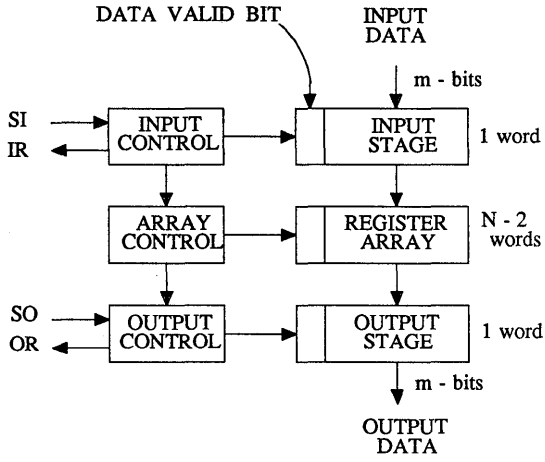


Figure 1. Register Array Architecture

Fullness Sensitivity

The number of words that can be written into the FIFO at the maximum input data rate (15 MHz) during an interval equal to the fallthrough time is:

$$\frac{F_{in}}{F_{fallthrough}} = \frac{15 \times 10^6}{\frac{1}{1.6 \times 10^{-6}}} = 24 \text{ words}$$

The bubblethrough time is the same as the fallthrough time because the same logic is used. Therefore, 24 words can be read from a full FIFO during an interval equal to the bubblethrough time.

This means that the FIFO can operate at its maximum data rate (15 MHz) only when the FIFO is between 24 and 40 words full (noting that 64 - 24 = 40 words). In other words, the maximum frequency at which the FIFO can operate is a function of the FIFO's fullness. It must contain 32 ± 8 words to operate at 15 MHz. To avoid fullness sensitivity, the FIFO must operate at or slower than the frequency corresponding to the fallthrough/bubblethrough time. This frequency is 625 KHz.

FIFO Depth Expansion

Figure 2 shows the interconnection of two 64 X 4 FIFOs to form a 128 X 4 FIFO. This depth expansion technique is called cascading. Data transfers in parallel between the two FIFOs, under control of the handshaking signals.

Observe that the first FIFO's OR output becomes the second FIFO's SI input. Similarly, the second FIFO's IR output becomes the first's SO input. Thus,

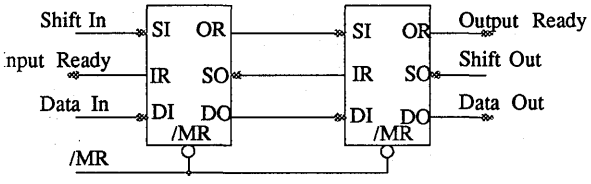


Figure 2. 128 x 4 FIFO

the bubblethrough/fallthrough times add serially when the FIFOs are connected together. The maximum throughput that can be handled by N FIFOs cascaded together is:

$$F_{max.} = \frac{1}{\frac{1}{F_A} + N\,t_F} \qquad \textit{Eq. 2}$$

To make a wider word as well as a deeper FIFO, connect the FIFOs as illustrated in *Figure3*. To use this technique, you must generate composite IR and OR signals using two external AND gates (e.g., 74LS08), which compensate for variations in the signals' propagation delays from device to device.

If a Low-to-High transition occurs on the SI pin when the IR pin is not High, the FIFO ignores the transition and does not sample the data. Similarly, if a Low-to-High transition occurs on the SO pin when the OR pin is not High, the FIFO ignores the transition and does not output new data. The old data, however, remains in the output register.

Dual-Port RAM Architecture

In 1983, Synertek introduced a 1024 X 8 dual-port RAM, and Mostek introduced a large (512 X 9) FIFO. Both products were based upon a dual-port RAM cell. Adding two read and two write transistors to the conventional two-transistor RAM cell makes the read and write functions independent. This change obviously increases the RAM cell's size, but simpler control logic and greatly improved performance more than compensate for the size disadvantage. A schematic of the dual-port RAM cell appears in *Figure4*.

In 1985, Cypress introduced the CY7C401, CY7C402, CY7C403, CY7C404, and CY3341 small FIFOs, all of which use the dual-port RAM architecture. The CY7C401, CY7C402, and CY3341 are pin-compatible, improved-performance, functional equivalents of the industry-standard FIFOs. The CY7C403 and CY7C404 are Cypress proprietary parts. They feature output-enable controls so that the data outputs can either be enabled or placed in a high-impedance state.

Figure 5 shows a simplified block diagram of a RAM-based small FIFO. This architecture applies to the CY3341, the CY7C401 through CY7C404, and the CY7C408/CY7C409 FIFOs, although the latter's flag generation logic is not shown.

The architecture is that of a dual-port RAM accessed by two pointers — one each for read and write — implemented as address registers. The input data and the output data do not reside in input or output registers, as in the register-array architecture. Instead, the pointers address the memory locations of the input and output data. Comparators control the IR and OR lines to prevent overflow and underflow, as well as to

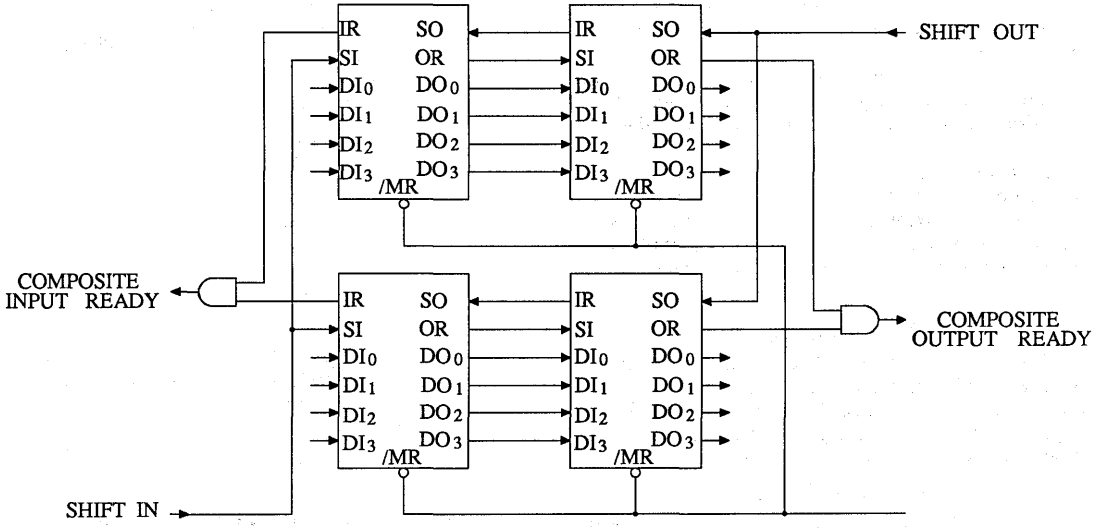


Figure 3. 64 x 8 FIFO

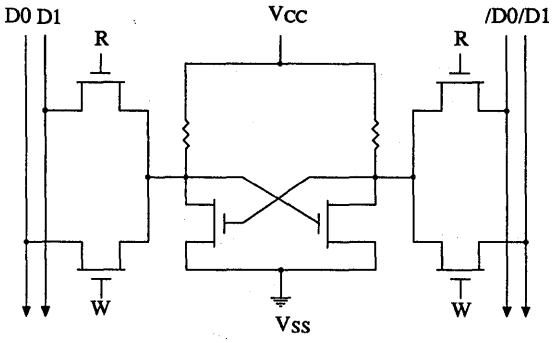


Figure 4. Dual Port RAM Cell Schematic

generate the Almost Full/Empty and Half Full flags (AFE and HF; for the CY7C408/409 only).

This architecture reduces the fallthrough/bubblethrough time to 65 nanoseconds — 24.6 times faster than the register array version's 1.6 µs. This dramatic improvement occurs because the fallthrough/bubblethrough time now represents the time to update the pointers, not the time for data to propagate through the memory array. In fact, the fallthrough/bubblethrough terminology does not really apply to RAM-based FIFOs. Data sheets include these terms to emphasize that the RAM-based FIFOs are functionally equivalent to their register-array-based ancestors.

FIFO Reset

Upon power up, the FIFO must be reset by pulling the Master Reset (/MR, active Low) pin Low. This asynchronously resets the internal read and write pointers, causes the FIFO to go to the empty state, and causes the data outputs to go Low. The IR pin goes High, signifying that the FIFO is not full, and the OR pin stays Low, signifying that the FIFO is empty. Note that when the /MR pin is activated, all data within the FIFO is lost.

If adjacent signals couple to the /MR signal, it can experience short negative noise pulses that can partially reset the FIFO. When this happens, the FIFO locks up and must be properly re-initialized with a /MR pulse that meets the minimum pulse width, as specified in the data sheet.

Handshaking

The small FIFOs use what is called the classical two-edge handshaking mechanism. It is the most efficient method of exchanging information when acknowledgement of a signal's receipt is required. This is true because every signal transition conveys information.

Input Handshaking

As illustrated in *Figure 6*, when the IR signal goes from Low to High, external logic should cause a Low to High transition on the SI input. To acknowledge receipt of SI High, the FIFO lowers the IR output within time t_{DLIR}, the delay from SI High to IR Low.

Nothing further happens until the producer lowers the SI signal. Then, if the FIFO has at least one empty location, the device raises the IR output within time t_{DHIR}, the delay from SI Low to IR High.

Output Handshaking

The output handshake timing appears in *Figure 7*. Note the similarities between *Figures 6* and *7*. When the OR (Output Ready) signal goes from Low to High, external logic should apply the same transition on the SO (Shift Out) input. To acknowledge receipt of the SO High, the FIFO lowers the OR output within time t_{DLOR}, the delay from SO High to OR Low.

Nothing further happens until the consumer lowers the SO signal. Then, if the FIFO contains at least one word of data, the device raises its OR output within time t_{DHOR}, the delay from SO Low to OR High.

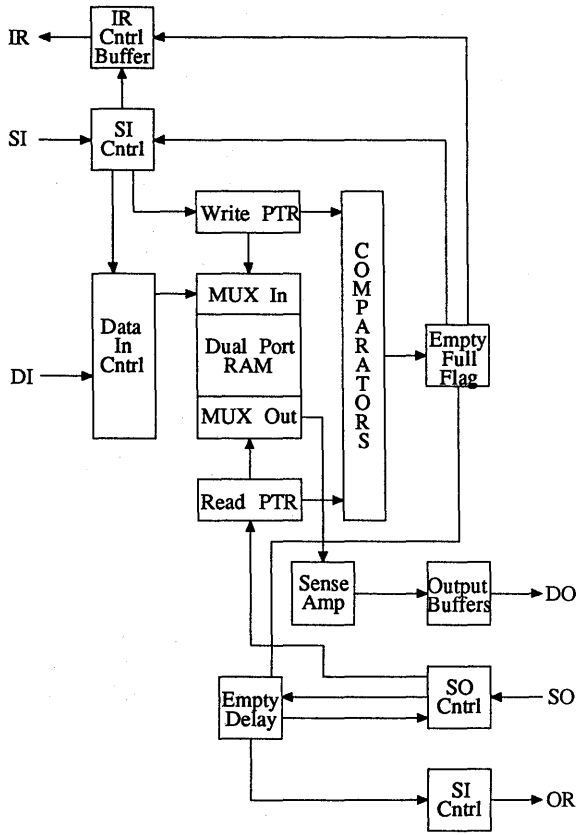


Figure 5. RAM Based FIFO Architechture

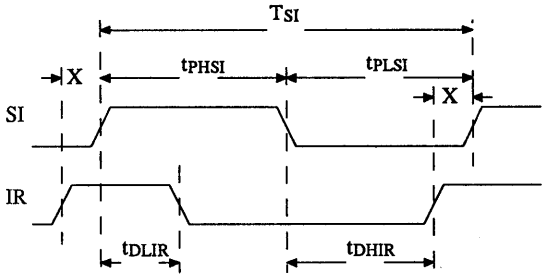


Figure 6. Input Handshaking

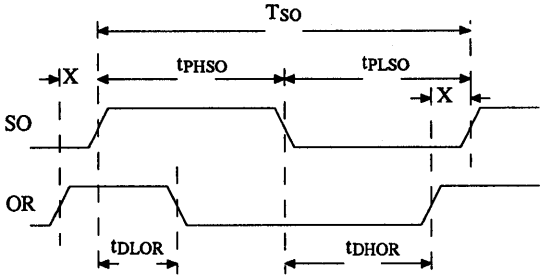


Figure 7. Output Handshaking

Handshaking Frequency Calculations

The maximum SI and SO frequencies can be calculated in several ways. One way is to calculate the minimum SI and SO periods by adding the minimum High and Low pulse widths from the data sheet.

For SI:

$$\frac{1}{f_{SI}} = T_{SI} = t_{PHSI} + t_{PLSI} \qquad Eq.\ 3$$

where f_{SI} is the SI frequency, T_{SI} is the SI period, t_{PHSI} is the SI High time, and t_{PLSI} is the SI Low time.

For SO:

$$\frac{1}{f_{SO}} = T_{SO} = t_{PHSO} + t_{PLSO} \qquad Eq.\ 4$$

where f_{SO} is the SO frequency, T_{SO} is the SO period, t_{PHSO} is the SO High time, and t_{PLSO} is the SO Low time.

Examining the data sheets for the small FIFOs verifies that Equations 3 and 4 are satisfied. Note as well that $t_{PHSI} = t_{PHSO}$, and $t_{PLSI} = t_{PLSO}$, which are required for cascadability.

A second method is to calculate the minimum SI and SO periods from *Figures 6* and *7*. Beginning with the rising edge of SI:

$$T_{SI} = t_{DLIR} + (t_{PHSI} - t_{DLIR}) + t_{DHIR} + X \qquad Eq.\ 5$$
$$= t_{PHSO} + t_{DHIR} + X$$

where X is the response time of the external logic.

In a similar manner, beginning with the rising edge of SO:

$$T_{SO} = t_{DLOR} + (t_{PHSO} - t_{DLOR}) + t_{DHOR} + X \qquad Eq.\ 6$$
$$= t_{PHSO} + t_{DHOR} + X$$

Examining the FIFO data sheets verifies that Equations 5 and 6 are satisfied. Note that $t_{PHSI} = t_{PHSO}$, and $t_{DHIR} = t_{DHOR}$, assuring cascadability. Additionally

$$T_{SI} > t_{PHSI} + t_{DHIR} \qquad Eq.\ 7$$

and

$$T_{SO} > t_{PHSO} + t_{DHOR} \qquad Eq.\ 8$$

The difference (X) between the T_{SI}/T_{SO} period and the sums represented by Equations 5 and 6 are the times within which the external logic must respond to achieve the performance specified on the data sheet.

These analyses prove that you can operate the small FIFOs at the maximum SI/SO frequencies, as guaranteed by the worst-case AC parameters on the data sheets.

Under nominal conditions (V_{CC} = 5V, T_a = 25°C), t_{DLIR} = 10 ns, t_{DHIR} = 10 ns, and t_{PHSI} = 5 ns for the CY7C408/409.

FIFO Operation

The FIFO samples input data during the SI signal's Low-to-High transition if and only if the IR output is High. Internally, the IR signal is logically ANDed with the SI input. As a result, if external logic generates a positive SI pulse when IR is Low, the FIFO ignores the pulse. Therefore, the data does not appear at the FIFO outputs, and the FIFO appears to drop words, when in fact the words were never entered.

Input Data

As explained previously, a rising edge on SI causes a falling edge on IR. Nothing further happens as long as SI is held High. The internal write pointer is incremented on the SI signal's High-to-Low transition. If the FIFO is not full, after the write pointer settles the IR signal goes High, indicating that more room is available. If the IR signal does not go High within t_{DHIR} (delay, SI Low to IR High), the IR Low signifies that the FIFO is full.

Output Data

Output data appears at the data output pins, then the OR output signal goes from Low to High, signifying that the data is valid. Internally, the OR signal is logically ANDed with the SO input.

As a result, if external logic generates a positive SO pulse when OR is Low, the FIFO ignores the pulse. Therefore, the read pointer is not incremented, and the same data is read, assuming that external logic samples the output data on SO's rising edge; this makes the FIFO appear to pick up words. In fact, the device that generates the SO pulse is reading the words more than once.

As explained previously, a rising edge on SO causes a falling edge on OR. The internal read pointer is incre-

mented on the SO signal's High-to-Low transition. The read pointer now settles, and an interval equivalent to an SRAM's address access time passes. Then, if the FIFO contains at least one word of data, the OR signal goes High, signifying that more data is available. If the OR signal does not go High within t_{DHOR} (delay, SO Low to OR High), the OR Low indicates that the FIFO is empty.

Data Timing

Examination of *Figure 8* shows the minimum SI period to be:

$$T_{SI} = t_{SSI} + t_{HSI} \qquad \textit{Eq. 9}$$

Comparing these parameters' values in the data sheets for the small FIFOs reveals that the maximum input data frequency is considerably greater than the frequency represented by Equation 3. This is because the control signals go into the IC as well as come out of it, which requires more time than simply presenting the input data to be sampled. In other words, the maximum input frequency is limited by the propagation delay of the control signal path, not the data path.

Examination of *Figure 9* shows the minimum SO period to be:

$$T_{SO} = t_{PHSO} + t_{DHOR} \qquad \textit{Eq. 10}$$

Comparison of these parameters' values in the data sheets for the small FIFOs reveals that this maximum output data frequency is considerably greater than the frequency represented by Equation 4, for the same reason given for the input data frequency.

Output Data Set-Up Time

The difference between t_{PLSO} and t_{DHOR} is the set-up time for the output data, t_{SOR}. This is true because, when cascading FIFOs, the output data must be available a set-up time before the OR signal goes from Low to High. The data sheets for the small FIFOs specify t_{SOR} as 0 ns (min.), but you can also calculate it yourself. Simply subtracting the data sheet values t_{PLSO} - t_{DHOR} does not give a reasonable answer, because t_{PLSO} is specified as a minimum and t_{DHOR} as a maximum. Instead, calculate the maximum value of t_{PLSO} using the relationship

$$t_{PLSO}\,(\text{max.}) = \frac{1}{F_o} - t_{PHSO} \qquad \textit{Eq. 11}$$

where F_o is the output frequency.

Then calculate the set-up time as

$$t_{SOR} = t_{PLSO}\,(\text{max.}) - t_{DHOR}\,(\text{max.}) \qquad \textit{Eq. 12}$$

where t_{DHOR} (max.) is the maximum value from the data sheet.

Table 1 summarizes the data and calculations using Equations 11 and 12 for the CY7C408 and CY7C409.

Output data should be sampled using a positive-edge-triggered flip-flop or register such as the 74AS374 or equivalent. Clock the register with the SO signal.

Table 1. Output Data Setup to OR

CY7C408/409-xx

Parameter	-15	-25	-35
F_O	15	25	35
$1/F_O$	67	40	29
t_{PLSO} (min)	25	24	17
t_{PHSO} (min)	23	11	9
t_{PLSO} (max)	44	29	20
t_{PHSO} (max)	40	23	16
t_{SOR} (calc)	4	6	4

Operation at the Boundary Conditions

When FIFOs are connected in parallel to make a wider word, under certain conditions, they might individually ignore a read or write request. The system-level symptom of this problem is byte mis-alignment. When a single FIFO is operating alone, the words are

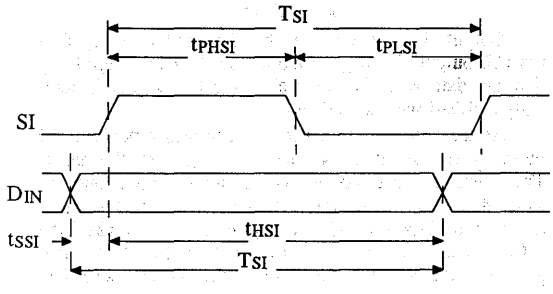


Figure 8. Input Data Timing

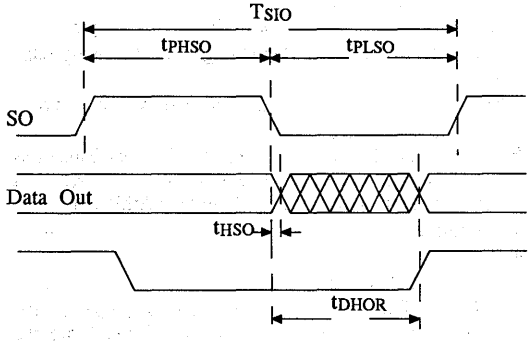


Figure 9. Output Data Timing

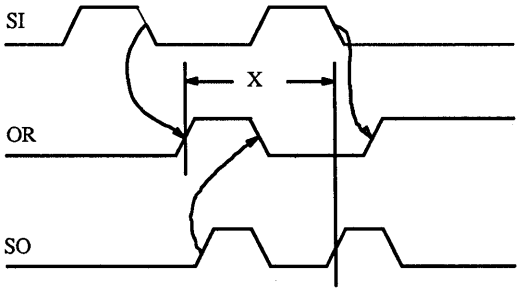


Figure 10. Forbidden Window

simply missing. They were either not written or not read.

The problem occurs at the empty condition, when a write is immediately followed by a read, and at the full condition, when a read is immediately followed by a write.

Operation at the Empty Boundary

Consider first a FIFO that has been reset and is empty. Read operations are inhibited by internal logic, so that the read pointer is not incremented, but all zeros are read at the data outputs. In the general case, the read and write signals are asynchronous.

Upon completion of a write operation, the FIFO's internal state goes from empty to empty + 1. During this interval, a read operation might not be recognized. If the read precedes the write, the read is ignored; if the read follows the write, the read is executed. Between these conditions, the FIFO must decide whether to recognize the read. During this aperature of uncertainty, you cannot determine whether the read will be ignored or not. With one FIFO, this behavior is acceptable. If two or more FIFOs are connected in parallel to make a wider word, however, some might ignore the read, and others might not.

Operation at the Full Boundary

A similar condition occurs when a single FIFO becomes full. Write operations are inhibited by internal logic. A read operation immediately followed by a write operation causes the FIFO to go from full to full - 1 and back to full. During the time the FIFO is going from full to full - 1, the write operation might not be recognized. The same aperature of uncertainty exists, because the FIFO takes a finite amount of time to change internal states. If a write command arrives at this instant, it might be ignored.

The most obvious solution to the aperture-of-uncertainty problems is to not perform the operation at the boundary condition. That is, (1) do not perform a read immediately after writing the first word into an empty FIFO, and (2) do not perform a write immediately after reading from a full FIFO.

With the small FIFOs, this is easier said than done. This is because, with the exception of the CY7C408/409, they do not have full or empty flags. However, the other FIFOs do have handshaking signals, and it has been shown that the output data is available before OR's Low-to-High transition. So long as the consumer generates a Low-to-High transition on SO only when there is a Low-to-High transition on OR, proper operation at the empty boundary (as well as everywhere else) is guaranteed.

Similarly, if the producer generates a Low-to-High transition on SI only when there is a Low-to-High signal transition on IR, proper operation at the full boundary (as well as everywhere else) is guaranteed.

A Caveat for the CY7C401, 402, 403, and 404

In addition to the aperture of uncertainty, note that the CY7C401 - 404 have a forbidden window of 40 ns during which they recognize only one SO pulse. This window (*Figure 10*) is measured from OR's rising edge when the first word is written into an empty FIFO to the rising edge of the second SO pulse. The forbidden window is a consideration only at high speeds (25 MHz), when a second output system clock could cause a second SO pulse within 40 ns of the first OR transition.

One way around this situation is to detect the empty and full conditions and delay the appropriate clock (SI or SO) the required amount of time. If the FIFO is empty, OR does not go High within a fallthrough time after SO goes Low; this condition can be sensed and used to indicate EMPTY. Similarly, if the FIFO is full, IR does not go High within a bubblethrough time after SI goes Low, and this condition can indicate FULL.

Interfacing to the FIFO

This section deals with issues regarding interfacing to the small FIFOs. The two areas of concern are (1) voltage sensitivity on the SI and SO inputs, and (2) metastability when the handshaking signals are used and the SI and SO signals are derived from independent frequency sources. The following information applies to all of the small FIFOs.

High-Gain Inputs

The FIFO data sheets specify the minimum positive SI and SO pulse widths as 9 ns for the 35-MHz SI/SO versions of the CY7C408/409 and 11 to 20 ns for the other speed grades of all the small FIFOs. At room temperature and nominal (5V) V_{cc}, the FIFO operates reliably with SI/SO pulses as short as 5 ns, as measured at the input threshold level (approximately 1.5V). These FIFOs respond to such short pulses because the Cypress high-performance CMOS process yields circuits that have very thin gate oxides. This characteristic permits the transistors to have high gains and, consequently, require very little energy to change state.

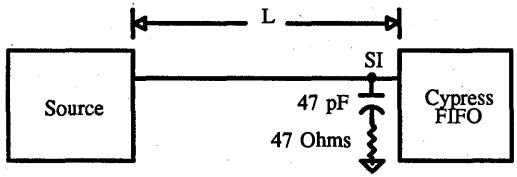


Figure 11. Recommended Termination Network

Termination networks are recommended on the SI and SO lines (traces) on printed circuit boards (PCBs) when the lines from source to load are long. A long line is defined as a line whose "electrical length" is equal to or greater than the rise time of its signal divided by the two-way propagation delay of the line per unit length. When the line is long, a voltage reflection might occur that the FIFO can interpret as a clock.

The termination matches the load impedance to the characteristic impedance of the PCB trace, which is typically 50 ohms or less for microstrip or stripline construction on G-10 glass epoxy material. For minimum voltage reflections, a slightly overdamped termination is preferred. Cypress recommends a series capacitor of 10 to 47pF and resistor of 47 ohms connected from the input pin (SI/SO) to ground (*Figure 11*). This termination network acts as a low-pass filter for short, high-frequency pulses and dissipates no DC power.

If you connect more than one FIFO in parallel to make a wider word, only one termination network is required. Put it at the input that is electrically the farthest from the source.

For the method of determining the values of R and C for the termination network, please refer to the low-pass filter analysis in the "Systems Design Considerations When Using Cypress CMOS Circuits" application note in this book. That application note also explains how to determine when a line is long. The line length at which a voltage reflection might occur is a function of the signal rise time, the unloaded (intrinsic) line propagation delay, the load, and the intrinsic line characteristic impedance.

Synchronous and Asynchronous Operation

When the SI and SO signals are derived from a common frequency source or clock, the FIFO is, by definition, operating in the synchronous mode. This approach establishes a precise, known relationship between the SI and SO signals. Conversely, when the SI and SO signals are derived from independent frequency sources, the FIFO is operating in an asynchronous mode.

In the synchronous mode, you can guarantee that the OR signal does not occur within the set-up-and-hold-time window that normally surrounds the output system clock edge or sampling signal. The same reasoning applies to the occurance of the IR signal, with respect to the input system clock.

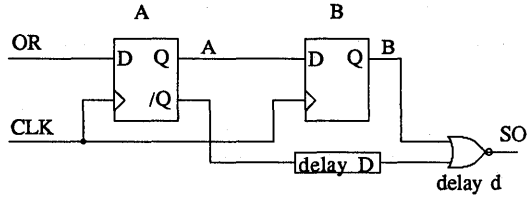


Figure 12. Pulse Synchronizer

In the asynchronous mode, you cannot assure a known relationship between the OR signal and the output system clock, with respect to either frequency or phase. It is the responsibility of the designer to ensure that, even though the output system clock edge might occur at the same time as OR, the FIFO still receives an SO clock wide enough for the FIFO to recognize reliably. The same reasoning applies to the SI signal generated in response to IR, under control of the input system clock.

Pulse Synchronizer

The circuit shown in *Figure 12* is recommended to generate the SO pulse as a function of OR, under control of the output system clock. Use an identical circuit to generate the SI pulse as a function of IR, under control of the input system clock. If you want to perform control functions on OR or IR, do so before they are clocked by the first D flip-flop.

Figure 13 shows a diagram of the two-stage shift register as a state machine. You can design more complex state machines for the task, but the idea is the same: reliably generate a single pulse of a known minimum width for every OR or IR Low-to-High signal transition.

Make the frequency of the clock to the pulse synchronizer at least twice the maximum rate at which you want to shift data into or out of the FIFO. For example, if you want to shift data into the FIFO at a 10-MHz SI rate, make the clock to the input pulse synchronizer 20 MHz. If you want to shift data out of the FIFO at a 15-MHz SO rate, make the clock to the output pulse synchronizer 30 MHz.

If a clock of this frequency is not available, you can easily double the frequency of the existing clock by delaying it and exclusive-ORing the delayed signal with the original signal. A circuit to do this appears in *Figure 14a*, with the timing shown in *Figure 14b*. If d1 is the propagation delay of the non-inverting buffer in this circuit and d2 is the XOR gate's delay, the width of the strobe is d1 + d2. This circuit does not generate a positive output strobe unless d1 > d2.

You can, of course, replace the non-inverting buffer with an even number of inverting buffers. Lumped delay elements such as gates act as glitch filters. A gate

whose propagation delay is d absorbs or filters-out short pulses whose width is less than, but almost equal to d.

When you use the pulse synchronizer shown in *Figure 12* under normal operating conditions, make SO's minimum pulse width one cycle of the output clock (CLK). However, when OR or IR changes within the forbidden window around the clock edge, the flip-flop might go into a metastable state (outputs between logic One and Zero). The amount of time the flip-flop stays in the metastable region is approximately 4X, where X is the flip-flop's clock-to-output propagation delay time.

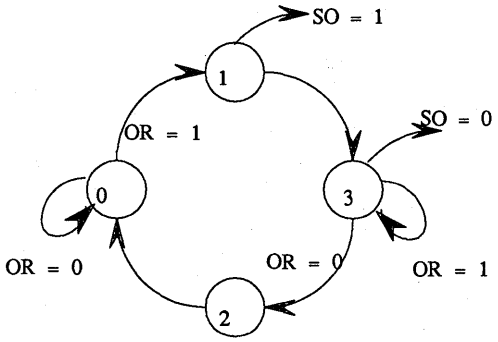
The minimum pulse width of the SO signal depends on the delay, d, through the NOR gate, plus any delay you might add (D, shown as a box) in the path from the A flip-flop's /Q output to the NOR gate's input. The NOR gate acts as a low-pass filter and does not pass pulses narrower than d. Adding an external delay, D, increases the minimum pulse width to d + D. Assuming equal gate turn-on and turn-off times, the maximum frequency at which the circuit can operate is

$$f_{(max.)} = \frac{1}{2(d + D)} \qquad \textit{Eq. 13}$$

Choose the total delay such that the FIFO can reliably detect the minimum pulse width. If only the NOR gate provides the delay, *Table 2* lists typical and maximum propagation delays under nominal V_{cc} and loading (20 pF) conditions. A 74LS02 NOR gate results in a minimum pulse width of 10 ns, which reliably operates a 25-MHz CY7C403 or CY7C404 FIFO.

If you want to operate a 10-MHz CY7C401/402, you can invert the A flip-flop's Q output through a 74LS04 and apply the result to the NOR gate's lower input. The minimum pulse width is then 10 + 10 = 20 ns. You can also use a delay line or RC network to delay the signal to the lower input of the NOR gate.

Use SO's rising edge to sample (clock) the FIFO data into a D-type flip-flop.



Transition Table

A	B	STATE	Description
0	0	0	idle at state 0
1	0	1	output SO=1
1	1	3	output SO=0
0	1	2	transition state

Figure 13. Pulse Synchronizer State Diagram

Operating FIFOs in Cascade Mode

When you connect two or more FIFOs together to make a deeper FIFO, they are said to be cascaded. There are two basic types of cascade mechanisms: serial and parallel, used by the small and large FIFOs, respectively. In the parallel method, data is steered between FIFOs using an internal token. In the serial method, data is passed serially from FIFO to FIFO using the handshaking signals.

The throughput of serially cascaded FIFOs is reduced in proportion to the reciprocal of the product of the fallthrough time and the number (N) of cascaded FIFOs. See Equation 2.

The throughput of FIFOs operating in the parallel cascade mode is a constant, independent of the number of FIFOs and equal to the throughput of a single FIFO operating alone.

Serial Cascade Analysis

A consideration in cascading FIFOs serially is to calculate the maximum SI and SO frequencies using the data sheet AC parameters. It is also useful to analyze the fallthrough (empty) and bubblethrough (full) conditions.

Another aspect of analyzing serially cascaded FIFOs is to understand burst mode. In this mode, you prevent the FIFOs from "thinking" they are empty, which avoids the devices' inherent cascaded frequency limitation.

Figure 15a shows the required interconnections between FIFOs for correct cascading. Data (DIA) is input to the A FIFO and then transferred to the B FIFO. The data flows from left to right, and it is standard practice to call FIFO A the upstream FIFO and FIFO B the downstream FIFO.

For the data to transfer reliably from FIFO A to FIFO B, the data must be valid at the inputs to FIFO B at least a set-up time before the Low-to-High transition of FIFO A's OR output. This is because the OR is ap-

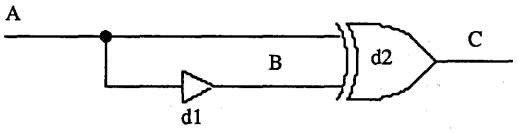


Figure 14a. Digital Frequency Doubler

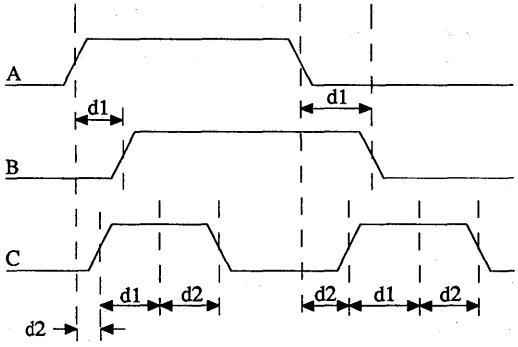


Figure 14b. Digital Frequency Doubler Timing

plied to FIFO B's SI input. As explained previously, data is sampled on SI's Low-to-High transition.

In the cascade configuration, the downstream FIFO's IR output connects to the upstream FIFO's SO input, and the upstream FIFO's OR output connects to the downstream FIFO's SI input. These two control connections are the only ones required to cascade the FIFOs. In theory, you can cascade any number of FIFOs in this manner.

The timing for serieally cascaded FIFOs appears in *Figure 15b*, which does not show the data. The signals begin in their quiescent states after a reset (/MR, not shown). Both FIFOs are initially empty.

There is one key difference between the quiescent state of a FIFO operating alone versus two or more FIFOs operating in cascade mode: In the stand-alone configuration, SO is Low, whereas in the cascade configuration, the SO inputs of the upstream N-1 of N cascaded FIFOs are High. This is true for all conditions, except when the downstream FIFO is full. As the downstream FIFOs fill, their IR outputs go Low, indicating that they are full.

When you cascade two FIFOs, the intrinsic handshaking frequency limitation goes away when the downstream FIFO becomes full.

Family	Typical (ns)	Maximum (ns)
LS	10	15
ALS	5	11
HCMOS	8	23
FACT	5	9.5

Table 2. Gate Propagation Delay Times

In operation, the producer samples the IRA line and, finding it High, presents the data to be written to FIFO A. A set-up time later, the producer causes a Low-to-High transition on FIFO A's SI input. FIFO A samples the data, and the IR output goes from High to Low.

Nothing further happens until the producer causes a High-to-Low transition on FIFO A's SI input. As a result, the write pointer is incremented, it settles, the IR output goes from Low to High, and the FIFO's internal state becomes empty + 1. Because FIFO A's SO input is High, the data just written is output on the DOA pins, and an internal one-shot is fired that causes a 15-ns pulse to appear on FIFO A's OR output. When the one-shot fires, the conditions at FIFO B's inputs are identical to those at FIFO A's inputs when the sequence began.

This cascade handshaking sequence repeats for every FIFO in the string. The first N-1 FIFOs must go from empty to empty + 1 and then back to empty to pass the data word to the last (Nth, or output) FIFO. This does not mean that the first data word must pass through all N or N - 1 FIFOs before the second (or subsequent) data words enter the first FIFO. However, the first FIFO must go from empty to empty + 1 and then back to empty before a second data word (rising edge on SI) can enter.

Serial Handshaking Calculations

Now consider how to calculate the intrinsic handshaking frequency for two or more FIFOs cascaded together. On the SIA signal's falling edge, fallthrough begins (t_{BT} on the data sheet). When FIFO A's OR output goes from Low to High, FIFO B samples the input data. In response to the Low-to-High transition on FIFO B's SI input, FIFO B's IR output goes from High to Low. This time is called t_{DLIR}. FIFO A is now empty, and FIFO B is empty + 1.

In equation form:

$$F_{(hs)} = \frac{1}{t_{BT} + t_{DLIR}} \qquad \textit{Eq. 14}$$

From the CY7C408/409 data sheet for the 35-MHz speed grade:

t_{BT} = 50 ns, t_{DLIR} = 15 ns

Substituting these values in Equation 14 yields:

$F_{(hs)}$ = 15.38 MHz

In practice, you will probably never observe this worst-case cascade-handshaking-frequency limitation, because the values given in the data sheet are "guardbanded." For typical Cypress FIFOs at room temperature and V_{cc} = 5V:

t_{BT} = 20 ns, t_{DLIR} = 10 ns

which yields a cascade handshaking frequency of 33.3 MHz. Note that this value applies to the entire string of cascaded FIFOs and is independent of the number of FIFOs cascaded together.

The same cascade-handshaking-frequency limitation occurs when both FIFOs are full and the downstream FIFO receives two SO pulses. In this case, the downstream FIFO goes from full to full - 1 and back to full. The empty location then bubbles through to the upstream FIFO, and downstream FIFO's IR output pulses. Internally, the one-shot is fired, and the upstream FIFO changes its OR output from High to Low and then back High (when the pulse ends).

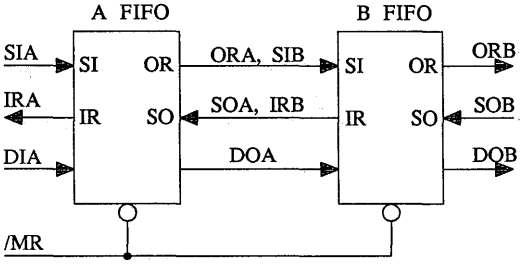


Figure 15a. Cascaded FIFOs: Intrisic Handshaking

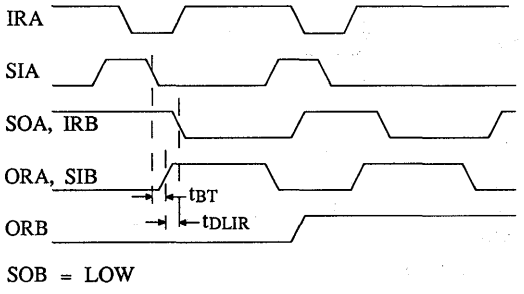


Figure 15b. Cascade Timing: Intrinsic Handshaking with FIFO

The cascade handshaking frequency is the reciprocal of the sum of the bubblethrough time and the propagation delay time from SO going Low to High to OR going High to Low (t_{DLOR}). By design, these parameters have the same values as the fallthrough time and t_{DLIR}, respectively. Therefore, the cascade handshaking frequency is the same for the full condition as it is for the empty condition.

Burst Input

It stands to reason that if the cascaded FIFOs can be made to think they are not empty, you can enter data at a higher rate than the cascade handshaking frequency. Also, if they can be made to think they are not full, you can remove data at a higher rate than the cascade handshaking frequency.

Figure 16a shows how to take advantage of these facts by adding an inverter between the downstream FIFO's IR output and the upstream FIFO's SO input. Note, however, that every FIFO whose SO input is the inverted IR output of a downstream FIFO has its capacity reduced by one word.

From the timing diagram in *Figure16b*, you can see that the composite FIFO never goes empty. The cascaded handshaking illustrated is essentially the same as that of *Figure 7*, which is the stand-alone output handshaking timing.

When the first (most upstream) FIFO is empty, there is a fallthrough time (t_{BT}) delay after the first word is shifted in. Or if the difference between the shift-in and cascade handshaking frequencies is great enough, the first FIFO goes empty. If the shift-in frequency is sufficently greater than the cascade handshaking frequency, the first FIFO goes full, and a fallthrough time (t_{BT}) delay occurs.

Except for the preceding conditions, adding the inverter enables the cascaded FIFOs to be either loaded at the stand-alone maximum shift-in frequency, or to be burst loaded using the AFE and HF flags of the CY7C408/409.

If you cascade N FIFOs together, inverters are required on the IR outputs of the N - 1 downstream FIFOs.

With the exception of the full and empty conditions of the first FIFO, the cascade handshaking frequency with the inverter is:

$$f_{(hsi)} = \frac{1}{t_{DLIR} + t_D} \qquad \textit{Eq. 15}$$

where $f_{(hsi)}$ is the handshaking frequency with the inverter and t_D is the inverter's Low-to-High propagation delay time.

Comparing Equations 13 and 14 reveals that if the inverter's delay is less than the fallthrough time, the cascade handshaking frequency with the inverter is less than the intrinsic cascaded handshaking frequency. If t_{DLIR} = 10 ns and t_D = 10 ns, then $f_{(hsi)}$ = 50 MHz. This means that because the handshaking frequency is

greater than the stand-alone SI/SO frequencies, the throughput is not limited by the handshaking frequency.

Care and Handling of Small FIFOs

The rest of this application note provides general guidelines for overcoming any problems you might have using the CY7C401 through 404, CY7C408/409, and CY3341 FIFOs.

One important factor to keep in mind involves the very high gain transistors used in the FIFOs to achieve the desired performance. These transistors' high speed can cause the FIFOs to respond to short pulses on the SI and SO inputs that bipolar, NMOS, and some CMOS FIFOs do not see. As a result, the small FIFOs might lock up or drop bits.

Avoiding Lock Up

The lock-up phenomenon occurs in the presence of excessive noise on the V_{CC} or ground lines; short pulses on SI or SO; or noise on the /MR line. Two distinct lock-up states have been observed: full and locked up.

When IR is Low and OR is High, the FIFO thinks it is full. However, in the full-lock-up state, no matter how many SO pulses are applied, the FIFO never goes empty (i.e., IR never stays High). You can get out the FIFO's contents, but those contents might not be the same as the data that went in.

The locked-up lock-up state should never occur. It is the quiescent state where both the IR and OR signals are Low. In other words, the FIFO thinks it is simultaneously full and empty.

The only method of recovery from the two lock-up states is to reset the FIFO by activating the /MR pin. All data is lost. If the FIFO is dropping bits, misaligning words, or occasionally just stopping, make sure that the /MR signal does not have noise on it. A small capacitor (47 to 100 pF) connected between the /MR pin and ground eliminates the noise.

Dropping Bits

The dropping of bits is annoying and unacceptable, but the data corruption does not cause the control logic to fail. Data corruption occurs because of either noise on the V_{CC} and ground pins or improper data sampling at the FIFO inputs or outputs. A significant amount of noise causes lock up; less noise can cause data corruption.

V_{CC} and Ground Noise

Reliable operation of the small FIFOs requires clean V_{CC} and grounds. Keep peak-to-peak V_{CC} noise to less than 200 mV. Additionally, keep the "quiet ground" (pin 7, DIP, CY7C408/409) separate from the "noisy ground" (pin 22, DIP, CY7C408/409), and connect both to system ground or a groundplane. Make the lead length from the pin to ground as short as possible.

Another noise-control procedure is to connect a 0.01-µF ceramic decoupling capacitor between each FIFO's V_{CC} and "noisy ground" pins. In addition, if either the SI or the SO frequency is over 5 MHz, connect a 100- to 400-pF mica capacitor or high-frequency-filtering ceramic capacitor between V_{CC} and "noisy ground," and connect a second 100-pF cpacitor between V_{CC} and "quiet ground." Keep lead lengths as short as possible.

For applications in which the SI and SO frequencies exceed 10 MHz, Cypress recommends a Pi filter on the V_{CC} line to the FIFOs. Because the filter is bidirectional, it keeps other ICs' noise from the FIFOs and the FIFOs' high-frequency noise from the other ICs. The inductor should be a subminature RF choke with a series DC resistance of 1Ω or less and an inductance of 100 µH. Make the capacitors 500-pF mica or ceramic types.

SI and SO Signal Considerations

To achieve the best results, make sure the SI and SO signals have rise times and fall times of 5 ns or less between 0.4 and 4V. At 5V V_{CC} and room temperature, the small FIFOs operate reliably with SI/SO pulses 5 ns wide, measured between the 1.5V levels.

Therefore, it is imperative that the signals be clean and slew rapidly between logic levels. If noise is superimposed on a slowly rising or falling signal, the FIFO might interpret the signal as multiple clocks.

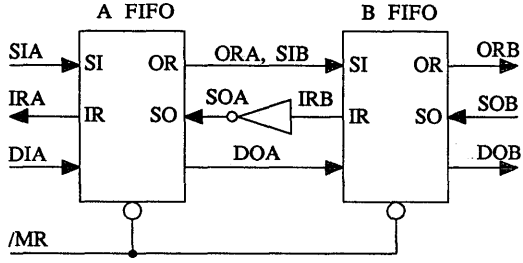


Figure 16a. Cascaded FIFOs: Burst Mode Operation

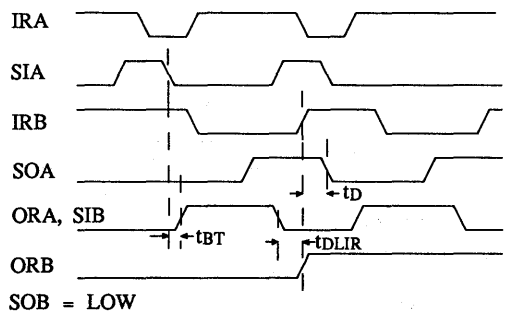


Figure 16b. Cascade Timing: Burst Mode with FIFO

The source that drives the SI/SO pin should have active devices pulling in each direction. That is, use totem-pole-output drivers instead of open-collector/drain outputs with a resistor to V_{cc}.

Beware of decoding glitches on the SI/SO signals. You can eliminate these glitches by using an AC termination network consisting of a series RC from the SI/SO pin to ground (*Figure11*). This network also acts as a filter and absorbs pulses that are shorter than four RC time constants. If the line is short and does not require termination, you can use a small capacitor (47 to 100 pF) to kill the glitches. When you connect FIFOs in parallel to make a wider word, one termination network is required for all SI pins and a second for all SO pins. Connect the network to the pin farthest from the source.

All the small FIFOs sample the input data for 10 ns after the SI pulse's Low-to-High transition. Therefore, the input data should be held stable at least 10 ns after SI's rising edge. Violation of the set-up and hold-time specifications can cause data corruption.

General Troubleshooting Guidelines

The switching speeds of CMOS devices are inversely proportional to temperature and directly proportional to supply voltage. Thus, the combination of low temperature and high V_{cc} is called the "fast-fast" corner, and high temperature and low V_{cc} is the "slow-slow" corner.

If increasing V_{cc} to the FIFO or PCB increases the number of failures, the problem is probably noise related. If increasing V_{cc} reduces the number of failures, the problem is probably due to marginal timing. If you reduce the temperature using a product such as Freezit while at low V_{cc}, and the failure rate increases, you have confirmed that the problem is marginal timing.

CY7C408/409 Only

If all else fails, you can increase the internal device thresholds by adding a diode (1N914, 1N4004) between "quiet ground" and power ground, cathode to power ground. This increases the threshold to V_t = 1.5V + 0.8V = 2.3V. A single diode suffices for many FIFOs. The number of FIFOs one diode can handle depends on the diode's forward current rating.

CYPRESS SEMICONDUCTOR

Understanding Large FIFOs

This application note explains the internal operation of the large FIFOs manufactured by Cypress and shows how to use the devices to accomplish depth and width expansion. Other topics covered here include FIFO interfacing, the writing and reading process, failure modes, and typical problem symptoms and solutions. This information applies to the following Cypress FIFOs: CY7C420, CY7C421, CY7C424, CY7C425, CY7C428, CY7C429, CY7C432, CY7C433, CY7C439, CYM4210, and CYM4220.

Timing parameters given in this application note are taken from the *Cypress Semiconductor BiCMOS/CMOS Data Book.*

Large FIFO Overview

The Cypress line of large FIFOs provide densities from 512 x 9 to 4K x 9 in monolithic devices; 8K and 16K x 9 in high-density modules; and a 2K x 9 bidirectional FIFO. Access times are as fast as 20 ns, and all the FIFOs feature identical, industry-standard pinouts. The monolithic devices are available in space-saving 300-mil-wide DIPs (odd-numbered devices) as well as industry-standard 600-mil-wide DIPs (even-numbered devices), and various surface-mount packages.

The CY7C420, CY7C421, CY7C424, CY7C425, CY7C428, CY7C429, CY7C432, and CY7C433 are fabricated using an advanced 0.8µ (drawn), n-well, CMOS technology. Input ESD protection is greater than 2000V, and careful layout, guard rings, and a substrate bias generator prevent latchup.

Although the first FIFOs utilized a shift-register type of architecture, today's large FIFOs employ an SRAM type of interface. Data is written into and read out of the devices, as with SRAM write and read operations. These operations can occur totally independently of one another and are made possible by a specially designed six-transistor, dual-ported SRAM cell. This cell makes use of separate read and write transistors to allow independent R/W operation.

Operating these FIFOs at their maximum throughput rates demands the generation of extremely narrow write and read pulses. To facilitate significantly higher throughput rates, Cypress has developed the CY7C440 and CY7C450 families of clocked, or self-timed FIFOs.

These FIFOs feature 70-MHz operation and are characterized by self-timed interfaces. You generate the read and write enables, which are combined internally with the appropriate clocks. Thus, you do not need to generate narrow read and write pulses. These FIFOs also feature totally independent, asynchronous, read and write operations.

The CY7C420/421, CY7C424/425, CY7C428/429, and CY7C432/433 are, respectively, 512, 1024, 2048, and 4098 words deep by 9 bits wide. Each FIFO is organized such that data is read out in the same sequential order in which it was written. Full, half-full and empty flags facilitate writing and reading. Additional pins are provided to facilitate unlimited expansion in width and depth, with no performance penalty.

Writing to and Reading From the FIFO

Figure 1 shows the large FIFOs' read and write timing. Reads and writes are asynchronous to each other. The read process begins with $\overline{R}$'s falling edge. The output data bus, Q0 - Q8, leaves the high-impedance state t_{LZR} ns after $\overline{R}$'s falling edge. The output data becomes valid t_A ns after that same falling edge. This t_A period is referred to as the FIFO's read access time. $\overline{R}$'s rising edge ends the read process.

The data on the Q0 - Q8 bus remains valid for t_{DVR} ns following the $\overline{R}$ rising edge. This is the output data hold time at the end of the read cycle. The internal circuitry then readies itself for the next read operation. This period is referred to as the t_{RR}, or read recovery time, and must be observed between consecutive read operations. The read signal's minimum pulse width is denoted by t_{PR} and is identical to the read access time, t_A. The maximum read frequency is the reciprocal of $t_{PR} + t_{RR}$.

The write process is similar to the read process. A write begins with the falling edge of the write line, $\overline{W}$, and terminates with $\overline{W}$'s rising edge. For a valid write to occur, the input data bus, D0 - D8, must be stable for t_{SD} ns prior to $\overline{W}$'s rising edge and for t_{HD} ns after this edge. These specifications are referred to as the data set-up and hold times, respectively. The write strobe also has a minimum negative pulse width, denoted as

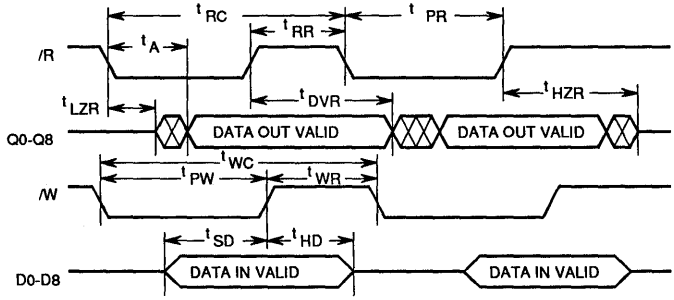


Figure 1. Asynchronous Read and Write Timing

t_{PW}. A minimum recovery time, t_{WR}, is required between write cycles.

The maximum write frequency is the reciprocal of $t_{PW} + t_{WR}$. As an example, a device with a 20-ns write strobe width and a 10-ns write recovery time yields a 30-ns write cycle time, or a 33.3-MHz maximum write cycle frequency.

You can determine the read cycle time (t_{RC}) by adding the access time (t_A) and the read recovery time (t_{RR}), which you can find in the FIFO data sheet. The maximum read frequency is the reciprocal of $t_A + t_{RR}$. For example, a Cypress FIFO with a 20-ns access time and a 10-ns read recovery time results in a 30-ns read cycle time, or 33.3-MHz maximum read cycle frequency.

The FIFOs include separate write and read counters (pointers). Each write or read operation increments the appropriate counter one position. When the FIFO is empty, both counters point to the same location. The relative position of these counters determines the device's status, which is indicated externally via empty, half-full, and full flags.

Applications

FIFOs are asynchronous devices that are ideal for interfacing between two asynchronous processes. A FIFO allows two systems running at different data rates to communicate by providing a temporary data or control buffer.

Typical FIFO applications include:

- Inter-processor communications, in which bidirectional devices are especially useful
- Communications systems, including local area networks
- Digital-signal-processing-based systems, for buffering real-time data
- Electronic data processing, CPU, and peripheral equipment, including high-performance disk controllers

Common FIFO Configurations

Every Cypress FIFO, from the 512 x 9 CY7C420/21 to the 4K x 9 CY7C432/3, are fully cascadeable. Width expansion allows you to create word widths of any multiple of nine bits. Cascading in depth creates FIFOs of various depths. Width and depth expansion modes are described here, along with design considerations.

Figure 2 illustrates stand-alone mode, and *Figure 3* shows width expansion mode. In both these modes, the $\overline{XI}$ (expansion in) pin is grounded and the FL (first load) pin is tied High.

The OR gates in the width-expansion design generate composite full, half-full, and empty flags (F, H-F, E). Composite flags are necessary because variations in propagation delays might prevent the individual FIFOs in the design from entering the F, H-F, or E states simultaneously. A composite flag properly reflects the instantaneous status of the entire word.

Figure 4 illustrates depth expansion. The $\overline{FL}$ (first load) pin on one device must be grounded to define that FIFO as the first FIFO to be written to. The FIFOs are then daisy-chained together by connecting one

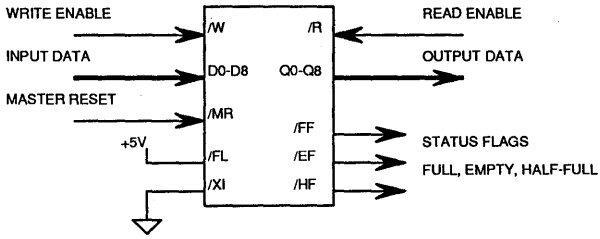


Figure 2. Stand-Alone Operation

device's $\overline{XO}$ (expansion out) output pin to the next device's $\overline{XI}$ (expansion in) input. The $\overline{XO}$ of the last device in the chain is connected to the $\overline{XI}$ of the first device, thus forming a token-passing ring.

Token passing allows the writing and reading processes to stay consistent. That is, the passing and holding of a read or write token tells an individual FIFO whether it is actively being read from or written to. In the token-passing procedure for write operations, the first FIFO is written to until it is filled. An internal write pointer determines the location written to, and after every write, the pointer is incremented. When the pointer reaches the last physical location, no more writes can occur to that device. At that point, the first FIFO passes the write token to the next FIFO in the chain via the $\overline{XO}$-$\overline{XI}$ interface. The second device, now in possession of the write token, receives all future written data until this device also fills up and passes the write token onto the next device in the chain.

If enough writes occur to fill up the FIFO chain, the last device fails in its attempt to pass the write token back to the first device. This is because the full FIFO cannot accept a write token. No further writes to the FIFO chain are allowed until a read operation occurs, which frees up an internal location. The relative positions of the internal write and read counters determine a device's status and whether it can accept data though a write operation. *Figure 5* shows the timing for write operations.

As with the procedure for writes, the first FIFO in the chain holds the read token. When the FIFO chain is read from, the device holding the read token supplies the data from the address specified by the device's read pointer. The read pointer is then incremented. The incrementing continues until the FIFO is empty, and the read token is passed to the next device in the chain. The passing of the read token is done via the $\overline{XO}$-$\overline{XI}$ interface. *Figure6* shows the timing for read operations.

A depth-expansion design must generate composite status flags to adequately reflect the instantaneous state of the FIFO chain, as is done for width expansion.

Retransmit

The retransmit feature is useful in communications for retransmitting packets of data and in disk drives for rewriting sectors. It is especially useful in applications where a single block of data in the FIFO must be sent out multiple times, as in a word or pattern generator.

Data can be retransmitted any number of times, and with Cypress FIFOs, the retransmit feature can be used at any time, no matter how much data the FIFO contains. This is in contrast to some competing FIFOs, such those from IDT, which do not allow use of the retransmit function when the FIFO is full.

In the retransmit operation, the read pointer is reset to its initial location and the $\overline{R}$ pin is pulsed until the read pointer advances to the same memory location addressed by the write pointer. The retransmit ($\overline{RT}$) pin is available in the single-device and width-expansion modes, but not in depth expansion because this pin designates the FIFO to be loaded first.

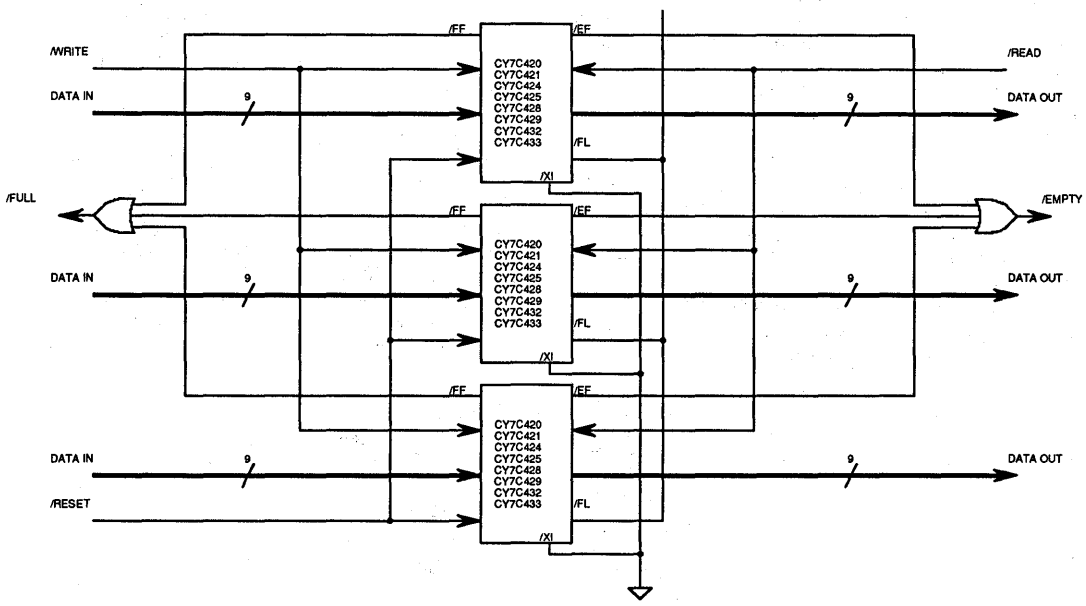


Figure 3. Width Expansion

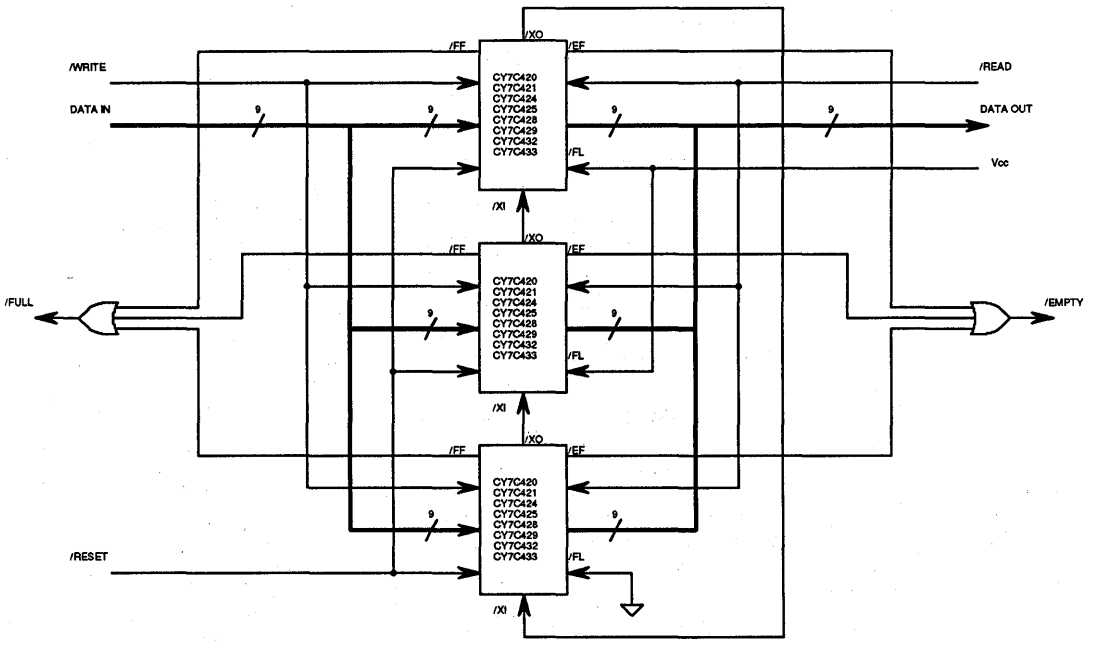
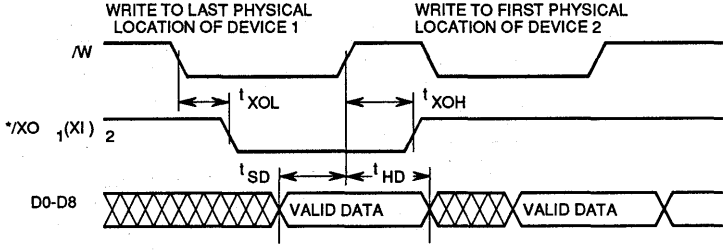


Figure 4. Depth Expansion

The retransmit function is initiated by asserting an active-Low pulse to the retransmit input, which resets the internal read counter to zero. Keep the $\overline{R}$ input inactive during this time; otherwise, the conflicting requirements on the read counter might cause it to become corrupted. The retransmit process does not affect the state of the write counter or the write process, though the retransmit timing constrains shown in *Figure 7* must not be violated.

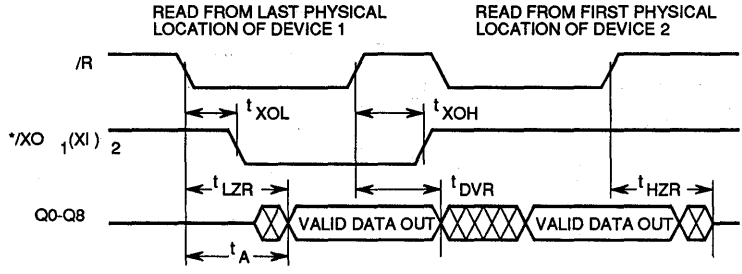
Note that the architectural description in the 1990 and previous Cypress data books incorrectly stated that the $\overline{W}$ input must be inactive during a retransmit cycle. No design or usage rules are violated if retransmit and write cycles overlap or occur simultaneously; the device does not lockup, and data is neither lost nor corrupted.

The reasons for the data book's retransmit/write restriction are more historical and application-oriented than functional. Specifically, the first large FIFOs did not permit writes during a retransmit cycle. This set a documentation precedent that all future devices had to match.



* Expansion Out of Device 1 (XO1) is connected to Expansion In of Device 2 (XI2)

Figure 5. Write Expansion Timing



* Expansion Out of Device 1 (XO1) is connected to Expansion In of Device 2 (XI2)

Figure 6. Read Expansion Timing

Additionally, keeping track of what data is currently in the FIFO and what data is being read out can become complicated. For example, if a FIFO is half full and the retransmit function is activated and writes continue, filling the FIFO to three quarters full before the read pointer catches up with the write pointer, the FIFO outputs all of the data.

Common Problems and Solutions

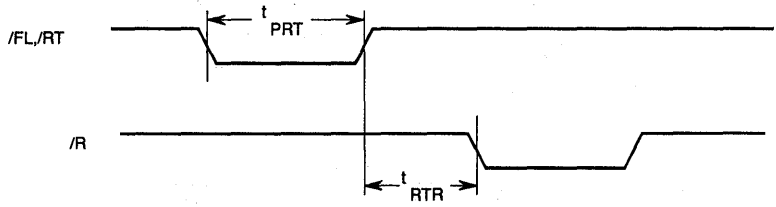
To help prevent problems and correct them when they occur, this section describes the causes and solutions to some common FIFO problems. The first problem to consider is corrupted or repetitive data in a FIFO.

Corrupted or Repetitive Data

The most common cause of corrupted and repetitive data being present in a FIFO is a spurious active signal (glitch) on the FIFO's $\overline{W}$ input. Because Cypress devices are extremely fast, a write pulse as short as 3 ns initiates a write. Write glitches cause whatever logic levels are present at the data inputs to be written into the FIFO, which can put false data into the device. If valid data is present at the data inputs, a write glitch causes this data to be written a second time, resulting in duplicated data.

Write glitches are often the result of voltage reflections due to impedance mismatches, which you can eliminate using impedance-matching termination networks. Termination networks are recommended on the $\overline{W}$ and $\overline{R}$ traces on printed circuit boards (PCBs) when the lines exceed approximately 4 inches from source to a single load. This line length assumes a 2-ns rise/fall time for the read and write strobes. For $\overline{R}$ and $\overline{W}$ signals with sub-2-ns rise/fall times, line lengths as short as 1 inch might require termination.

A termination network matches the load impedance to the PCB trace's characteristic impedance, which is typically 50Ω or less for microstrip or stripline construction on G-10 glass epoxy material. To minimize voltage reflections, a slightly overdamped termination is preferred. Cypress recommends a 47-pF (max) series capacitor and a 47-ohm resistor be connected from the read or write pin to ground (*Figure 8*). This termination network acts as a high-pass filter to short, high-frequency pulses and dissipates no DC power. Read or write lines that drive more than one FIFO require only one



t_{RTR} is the Retransmit Recovery time.
It is a timing window which must not be violated.

t_{PRT} is the minimum retransmit pulse width.

Figure 7. Retransmit Timing

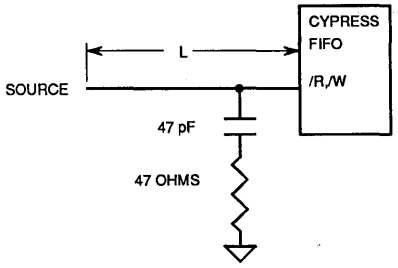


Figure 8. Recommended Termination Network

termination network. Put the network at the input that is electrically farthest from the source. For multiple loads, see the "Systems Design Considerations When Using Cypress CMOS Circuits" application note for help in determining the maximum line length.

FIFO data corruption can also be caused by violation of master-reset timing constraints. As shown in the timing diagram in *Figure 9*, the read and write signals must be inactive around the rising edge of $\overline{MR}$ (master reset) to satisfy the t_{RMR}, or master-reset recovery-time specification. This constraint is necessary because the FIFO goes through an internal initialization process during reset and requires a settling period after the reset terminates.

FIFO Locks-Up

Short noise pulses on the FIFO's master reset pin can cause the FIFO to not respond because it is "partially reset." If this problem occurs, you might need to terminate the master reset line.

Missing or Disappearing Data

Glitches on the $\overline{R}$ input can cause data to disappear because of an unintended read operation. The read increments the internal read counter, resulting in the loss of the current data word. Here again, a termination network eliminates the unwanted glitches.

Repetitive or Out-of-Sequence Data, False Full or Empty

A misaligned internal read or write pointer can cause a variety of symptoms, including repetitive or out-of-sequence data and false full and/or empty conditions. The two most common causes of misaligned pointers are master-reset violations and boundary-condition violations.

Boundary conditions are defined as the FIFO being either full or empty. When high-density FIFOs are connected in parallel to make a wider word, certain conditions can cause the FIFOs to choose individually to either ignore or act upon a read or write request. The system-level symptom of individual FIFOs making different decisions is word mis-alignment. The problem occurs in the empty condition when a read immediately follows a write and in the full condition when a write immediately follows a read.

Operation at the Empty Boundary

Consider a FIFO that has been reset and is empty. The empty flag is active (Low), and internal logic inhibits read operations. In the general case, the read and write signals are asynchronous. Upon completion of the write operation the internal state of the FIFO goes from empty to empty + 1. During this interval, a read operation might or might not be recognized. A read preceding the write is ignored; a read following the write is not. In between these conditions, the FIFO decides whether to recognize the read. During this aperture of uncertainty, you cannot determine whether the read will be ignored or not. With one FIFO, this uncertainty is acceptable. However, if two or more FIFOs are connected in parallel to make a wider word, some might ignore the read, and others might not.

Operation at the Full Boundary

A similar condition occurs when a single FIFO becomes full. The full flag is active (Low), and internal logic inhibits write operations. A read operation immediately followed by a write operation causes the FIFO to go from full to full - 1 and back to full. During the time the FIFO is going from full to full - 1, a write operation might or might not be recognized. The aperture of uncertainty applies here because the FIFO takes a finite amount of time to change states, and a write command arriving at this instant might be ignored.

Waiting at the Empty Boundary

Figure 10 shows the timing that prevents problems with reads at the empty boundary. Any device reading from the FIFO must wait an amount of time, t_{RAE}, after the termination of the write operation before causing a High-to-Low transition of the $\overline{R}$ signal. The $\overline{W}$ signal's rising edge indicates the termination of the write operation.

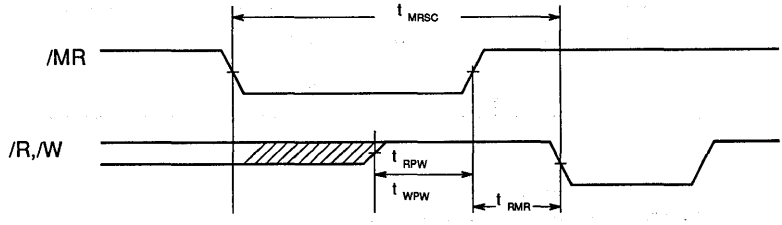
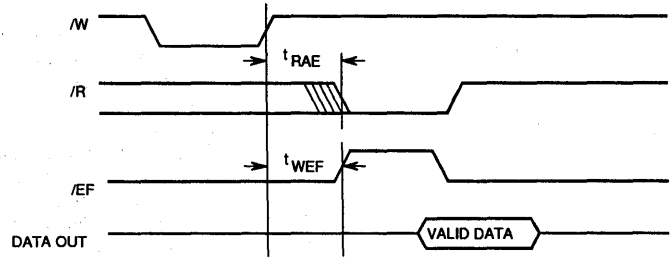


Figure 9. Master Reset Timing



t_{RAE} is an invalid read window.
A read operation should never be initiated inside this window.

Figure 10. Read Fall-Through Timing Violation

One way to satisfy this timing is to gate read operations with the composite empty flag ($\overline{EF}$) such that the read operation is prevented when the empty flag is active. Note, however, that the $\overline{R}$ signal can be Low either before or during the first write to the empty FIFO and the data still propagates to the outputs correctly.

Waiting at the Full Boundary

Figures11 shows the timing that prevents problems with writes at the full boundary. Any device writing to the FIFO must wait an amount of time, t_{WAF}, after the termination of the read operation before causing a High-to-Low transition of the $\overline{W}$ signal. The $\overline{R}$ signal's rising edge indicates the end of the read operation.

You can enforce this timing by gating write operations with the composite full flag ($\overline{FF}$) such that the write operation is prevented when the full flag is active. However, the $\overline{W}$ signal can be Low either before or during the first read from a full FIFO and the data is still properly written.

Empty Reads and Full Writes

When Cypress FIFOs are empty, their data outputs go to the high-impedance state. Therefore, attempting to read from an empty FIFO yields unpredictable data. Internal logic inhibits the read, and the read pointer is not incremented.

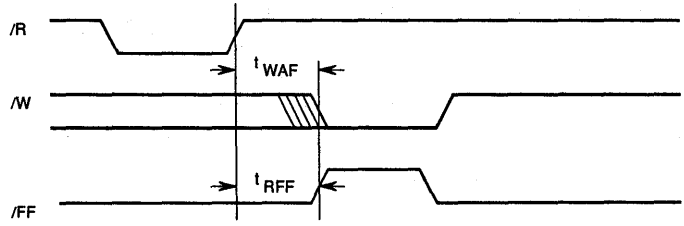
Internal logic also inhibits attempts to write to a full FIFO, and the write pointer is not incremented.

Intermittent Malfunctions

If all the timing requirements appear to be met and data in the FIFO is still corrupted, the cause is likely to be noise on the power supply. Random spikes on either the V_{CC} or ground pins of the FIFO are likely culprits when non-repeatable failures occur.

The cure for this problem is to add a high-pass filter capacitor between the device's power and ground pins. This practice is recommended whenever the read or write frequency exceeds 5 MHz. Use a very small (100 - 500 pF) ceramic or mica capacitor. Precision filtering capacitors of this type are available through suppliers such as Rogers Corporation, 2400 S. Roosevelt St., Tempe, AZ 85282.

The filter capacitor is in addition to the 0.1- or 0.01-μF decoupling capacitor that should always be present with any high-speed digital chip. Although decoupling capacitors are often referred to as bypass capacitor s — inferring filtering properties — their true function is to supply the instantaneous current required when many or all device outputs simultaneously switch from Low to High. This larger capacitor thus decouples or isolates the IC from the power distribution system.



t WAF is an invalid write window.
A write operation should never be initiated inside this window.

Figure 11. Write Bubble-Through Timing Violation

CYPRESS SEMICONDUCTOR

Designing with the CY7C439 Bidirectional FIFO (BIFO)

This application note describes the features of the CY7C439 bidirectional FIFO (BIFO) and shows how to use the BIFO in a multiprocessor communication design. The CY7C439 is a 2K x 9 FIFO memory that transfers data asynchronously at rates as high as 28.5 MHz.

BIFO Overview

Figure 1 shows a block diagram of the CY7C439 BIFO. The device has three internal data paths. The first path consists of a 2048-word-by-9-bit dual-ported RAM array, which allows half-duplex, bidirectional FIFO buffering. The second path, the registered bypass, allows registered message passing in the opposite direction from the FIFO-path operations. The last path, the transparent bypass, allows data to pass in either direction around the FIFO path.

The BIFO eliminates the need for other costly, space-intensive solutions that bidirectionally transfer data between two buses with disparate data rates. One alternative is a two-FIFO design (*Figure 2*), which requires a significant amount of board space and control circuitry. Although this solution achieves its objective, the two separate FIFOs are rarely needed because large amounts of data usually transfer in only one direction at a time. Another BIFO alternative utilizes one FIFO that can be switched from one direction to the other with bus-steering logic (*Figure 3*). Although this solution costs less than the previous one, it requires many MSI parts and thus requires more board area.

The CY7C439 solves all the problems associated with these alternative solutions. The CY7C439 utilizes significantly less board area, requires less power, and eliminates the need for complex FIFO control circuitry. Additionally, the CY7C439 is fully pin programmable,

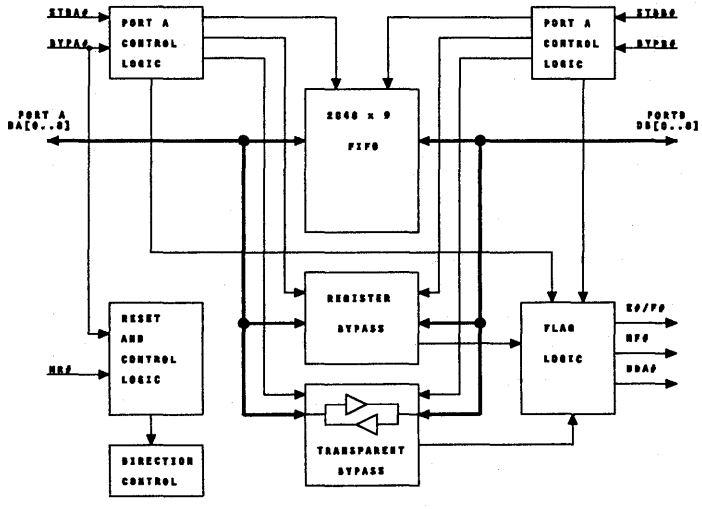


Figure 1. BIFO Block Diagram

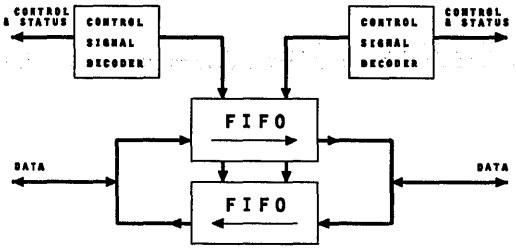


Figure 2. Two FIFO Design

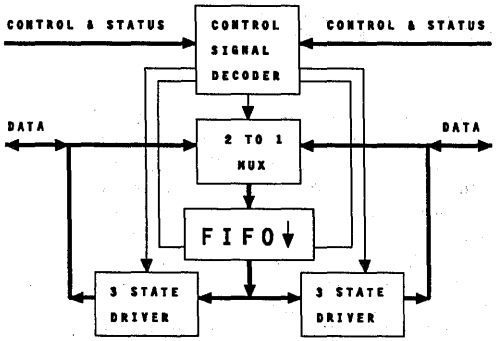


Figure 3. Switch FIFO Design

contains a hardware reset, allows message passing against the BIFO flow, and permits the initialization of dumb peripherals via the transparent bypass feature.

Half-Duplex BIFO Operation

When you reset the BIFO externally, you pulse the $\overline{MR}$ pin Low. During master reset, the BIFO's direction is set according to the state of the $\overline{BYPA}$ pin (*Table 1*). The $\overline{BYPA}$ state is latched internally on $\overline{MR}$'s rising edge. If the $\overline{BYPA}$ state is High on $\overline{MR}$'s rising edge, the BIFO direction is A to B, and the registered bypass direction is B to A. If, on the other hand, $\overline{BYPA}$ is Low on $\overline{MR}$'s rising edge, the FIFO direction is B to A, and the registered bypass direction is A to B. The master reset cycle is thus useful for setting the FIFO and

Table 1. Master Reset BIFO Direction Selection

$\overline{MR}$	$\overline{BYPA}$	$\overline{BYPB}$	$\overline{STBA}$	$\overline{STBB}$	Action
1	X	X	X	X	Normal Operation
⌐	1	1	1	1	FIFO (A-> B), Bypass (B->A)
⌐	0	1	1	1	FIFO (B-> A), Bypass (A->B)
0	X	X	X	X	Internal Reset

bypass directions as well as resetting the BIFO RAM array.

The bidirectional BIFO interface is similar to Cypress's CY7C42x family of FIFOs. In the CY7C42x FIFOs, data is written into the BIFO on the $\overline{W}$ line's rising edge and read out of the FIFO on the $\overline{R}$ line's falling edge. The CY7C439 works nearly the same. If the direction of the FIFO is from A to B, data is written into the FIFO on the rising edge of the $\overline{STBA}$ signal and read out of the FIFO on the falling edge of the $\overline{STBB}$ signal. The function of these two pins is reversed if the FIFO direction is set from B to A. *Table 2* shows these relationships.

The BIFO three-states its data lines on $\overline{STBB}$'s rising edge. BIFO circuitry does not allow additional Reads beyond empty or additional Writes beyond full. The "AC Timing" section describes the device's critical timing parameters.

Registered bypass

The CY7C439's registered bypass feature provides a way to send a word in the opposite direction to the FIFO data flow. The bypass feature is useful for message passing to indicate control and status information. In communication environments, for example, you can use the bypass register to indicate that a packet was not received correctly. This feature eliminates the additional circuitry required to allow a data consumer to communicate with a data producer.

The bypass operation does not affect the normal FIFO operation. The consumer writes bypass data into the register on the rising edge of $\overline{BYPx}$. The x in this pin name indicates that either the $\overline{BYPA}$ or $\overline{BYPB}$ pin is applicable, depending on the BIFO's direction. The assertion of the $\overline{BDA}$ flag signals the producer that it

Table 2. BIFO Operation Truth Table

Dir	$\overline{STBA}$	BYPA	$\overline{STBB}$	$\overline{BYPB}$	Action
A->B	⌐	1	⌐	1	FIFO Write at A, FIFO Read at B
A->B	1	⌐	⌐	1	FIFO Read at B, Reg Byp Read at A
A->B	⌐	1	1	⌐	FIFO Write at A, Reg Byp Write at B
B->A	⌐	1	⌐	1	FIFO Write at B, FIFO Read at A
B->A	1	⌐	⌐	1	FIFO Read at A, Reg Byp Read at B
B->A	⌐	1	1	⌐	FIFO Write at B, Reg Byp Write at A
ANY	1	1	0	0	No FIFO, Trans Data B to A

has a message waiting for it in the bypass register. The producer can then read the bypass register by pulsing the $\overline{BYPx}$ pin Low. The $\overline{BYPx}$ pins perform bypass register read and write functions with timing identical to that of the $\overline{STBx}$ signals.

Transparent Bypass

The CY7C439's transparent bypass capability allows the producer to transmit information through the BIFO without the consumer manipulating the BIFO to receive the information. This feature is useful for initializing dumb peripherals. Either side can initiate a transparent bypass by bringing both $\overline{STBx}$ and $\overline{BYPx}$ Low at the same time. The FIFO's contents are not affected by the transparent bypass operation. The port wishing to send data transparently to the other port must ensure that the other port will not attempt a FIFO read or write during the transparent bypass cycle.

Flag Operation

The BIFO provides two flag pins that can be decoded to represent one of four states (*Table 3*): empty; between empty and half full; between half full and full; and full. These flags indicate the FIFO's status and are useful for controlling the FIFO read and write operations.

AC Timing

Figure 4 shows the FIFO read and write timing diagram. As mentioned earlier, the timing looks very similar to that of the Cypress CY7C42x family of FIFOs. Assuming that the FIFO direction is from A to B, a read operation is performed by pulsing $\overline{STBB}$ Low while maintaining $\overline{BYPB}$ High. $\overline{STBB}$ must be held Low for a minimum time of t_{PR} (25 ns, for the CY7C439-25 part). The data lines remain three-stated for a minimum of t_{LZR} (3 ns) after $\overline{STBB}$'s falling edge, and data becomes available after t_A (25 ns). The $\overline{STBB}$ signal must recover for t_{RR} (10 ns) before another read operation is performed. The data lines remain valid for t_{DVR} (3 ns) and three-state after t_{HZR} (18 ns) from the rising edge of $\overline{STBB}$.

Table 3. BIFO Flag Operation

$\overline{E/F}$	$\overline{HF}$	Words in FIFO
0	1	0
1	1	1-1024
1	0	1025-2047
0	0	Full

The $\overline{STBA}$ signal is used to perform writes to the FIFO array. $\overline{STBA}$ must be Low for t_{PW} (25 ns) and recover for t_{WR} (10 ns). The data to be written into the FIFO must be set up for t_{SD} (15 ns) and held for t_{HD} (0 ns) from the rising edge of $\overline{STBA}$.

Figure5 shows the timing waveforms for the bypass register mode of operation. Reads from the bypass register look much the same as reads from the FIFO. During the bypass register reads or writes, the $\overline{STBx}$ signal must be High within t_{BSR} (10 ns) of the falling edge of $\overline{BYPx}$. The only differences between the timing for FIFO read and bypass register read are that the data lines remain three-stated for at least t_{BLZ} (10 ns) after $\overline{BYPA}$'s falling edge and that data will be available after t_{BA} (30 ns) on the -25 part. The bypass register write timing parameters are identical to those of the FIFO write parameters.

Figure 6 shows the timing waveforms for the transparent bypass mode of operation. This transceiver-like data path allows data to be driven from one port to another without the need for the consumer to control the BIFO to receive data. Either port can initiate a data transfer. $\overline{STBx}$ on the producing side must be High for t_{TSB} (10 ns) and must go Low no longer than t_{TBS} (10

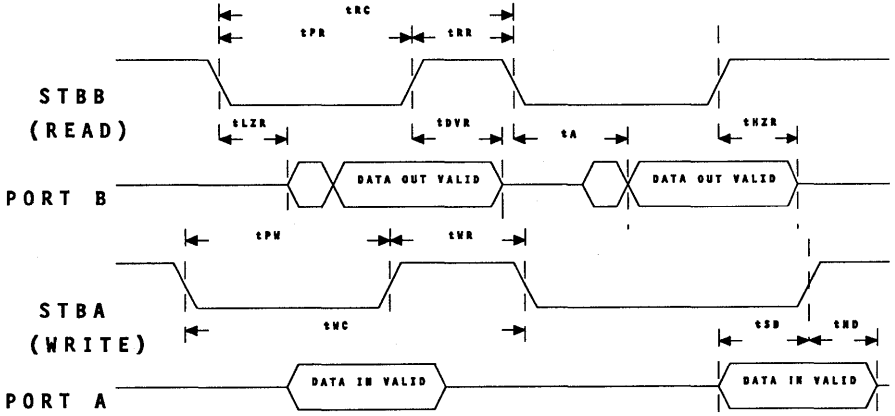


Figure 4. FIFO Read and Write Timing

ns) after the falling edge of $\overline{BYPx}$. Data is available t_{TPD} (20 ns) after the falling edge of $\overline{STBx}$, and the output data changes t_{DL} (20 ns) after the input data changes. The consumer bus three-states after t_{TSD} (18 ns) from the rising edge of $\overline{STBx}$.

Multiprocessor Communication Design

An excellent application for the CY7C439 BIFO is in interprocessor communication. Systems that process image, RADAR, and SONAR data and equipment that performs telecommunications bridging must all transfer significant amounts of data among system processors.

A design example shows the interface issues involved in communications between a 25-MHz Cypress CY7C601 SPARC processor and a 25-MHz 80386 used as an embedded processor. This high-speed processor bridge provides bidirectional data packet transfer, isolation of host processor from embedded processor, im-

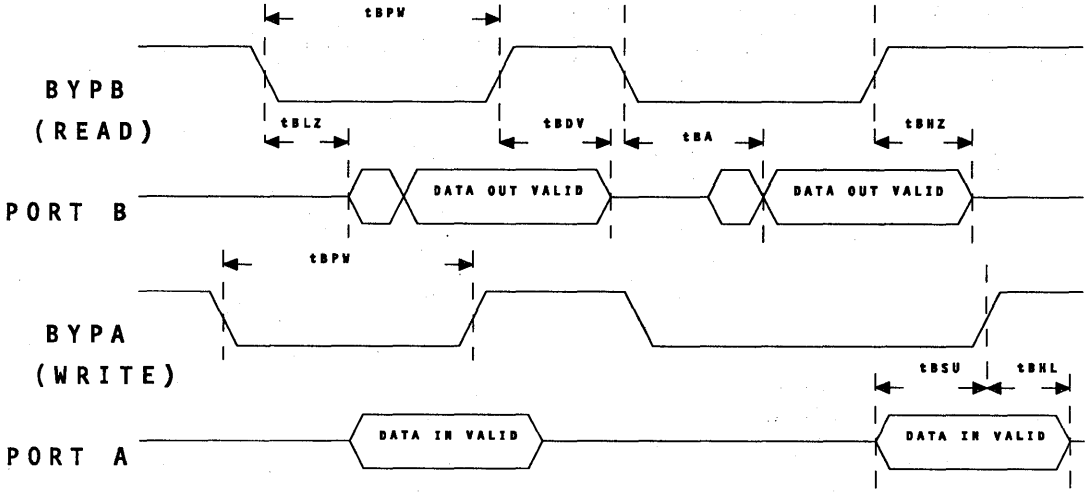


Figure 5. Bypass Register Timing

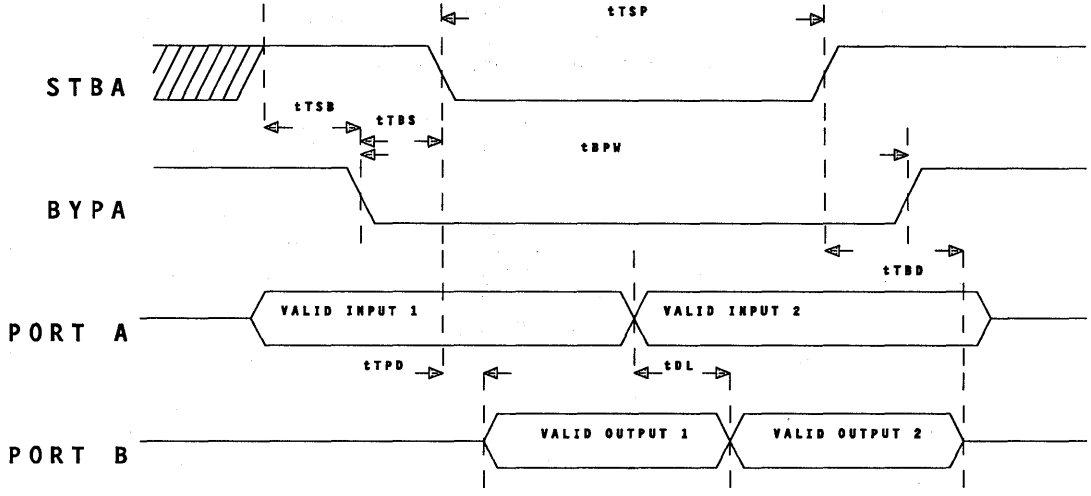


Figure 6. Transparent Bypass Timing

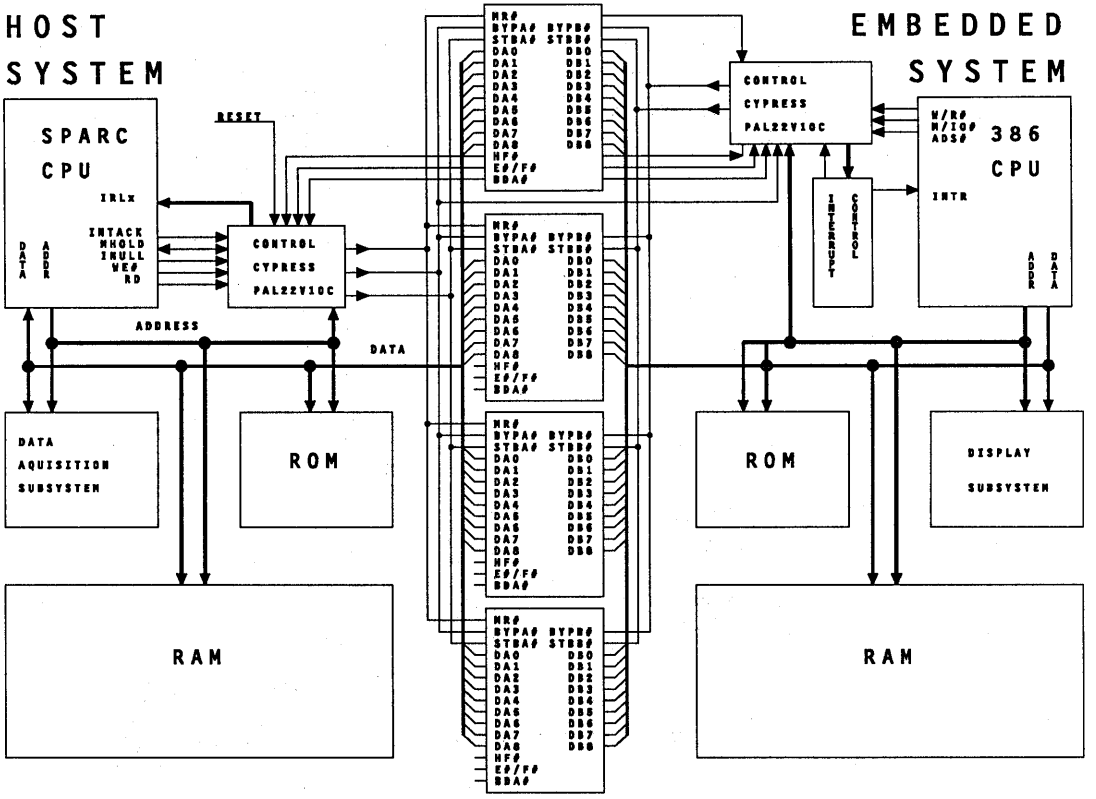


Figure 7. Design Example Block Diagram

plementation of message passing, and asynchronous communication by both processors.

Figure 7 shows a simplified block diagram of the design. The diagram shows the 32-bit bidirectional data path used to transfer data between the two processor systems. A 22V10 PAL implements control logic for the 386 embedded processor, which also employs an interrupt controller to prioritize the interrupt signals generated by the BIFO control logic.

CY7C601 BIFO Control Circuitry

The CY7C601 processor controls the direction of BIFO flow. A Cypress PAL22V10C-7 implements the control circuitry. This part has a 3-ns data set-up time, 0-ns data hold, and a 7-ns clock-to-output delay that makes it ideal in this high-speed application. The state machine implemented in the 22V10 uses a 3-ns delay of the CY7C601 system clock (required by the CY7C601 timing specifications).

The BIFO is implemented as memory-mapped I/O. The memory map for the CY7C601 appears in *Table 4*. The CY7C601 must reset the BIFO upon power up to ensure that the BIFO is in a known state. The BIFO is reset by selecting address $40000004. The CY7C601 initializes itself as a producer (BIFO direction from A to B) by writing to this address and initializes itself as a consumer (BIFO direction from B to A) by reading from this address.

Figure 8 shows a timing diagram for the CY7C601 configured as a producer. The diagram shows the relevant control signals and timing parameters for both

Table 4. Design Example Memory Map

Area	A31	A30	A29
PROM	0	0	X
BIFO	0	1	0
Other I/O	0	1	1
RAM	1	X	X

a read from the FIFO and a write to the bypass register. CLK is the host system clock. CLKD is the delayed system clock used in the BIFO control logic. ADDR represents the state of the A31, A30, A29, and A2 address lines. These signals, as well as the RD and $\overline{WE}$ control signals, are valid 7 ns before and 7 ns after CLK's rising edge.

The CY7C601 asserts RD at the beginning of a read cycle and $\overline{WE}$ in the second cycle of a write operation. Both RD and $\overline{WE}$ are used to determine if a valid read (load) or write (store) cycle has begun.

The INULL signal nullifies an active write cycle. Asserting INULL in the first cycle of a store operation cancels the operation. INULL appears in the second cycle of a all valid store operations (*Figure8*).

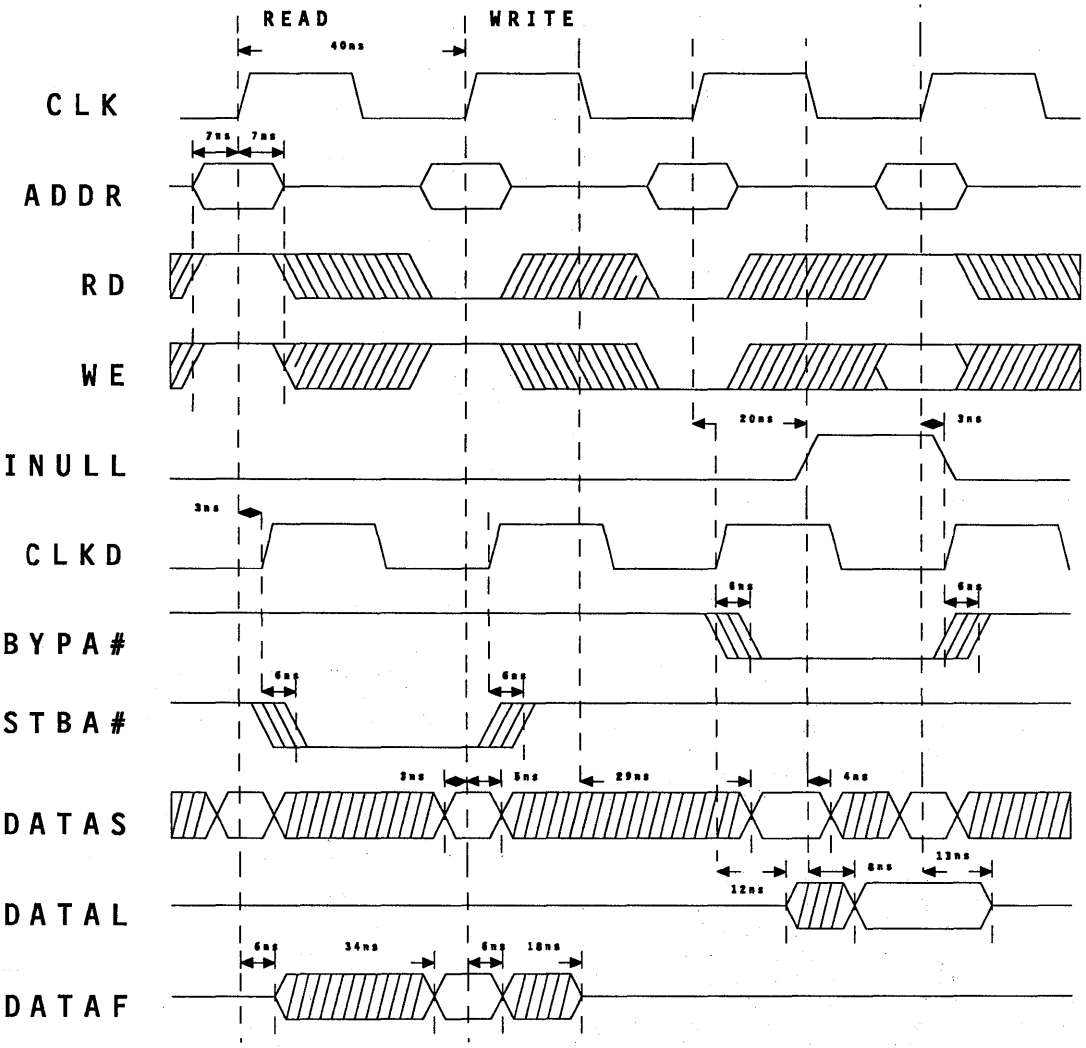


Figure 8. 7C601 Control Circuitry Timing Waveforms

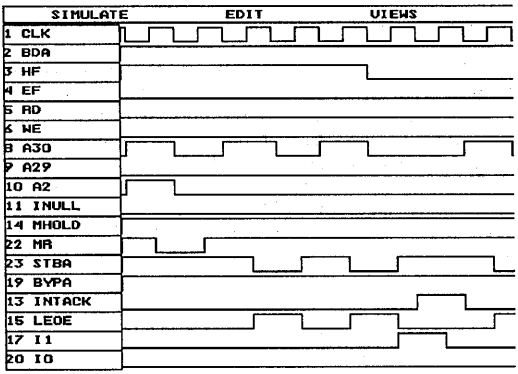


Figure 9. BIFO Initialization -- 7C601 Producer

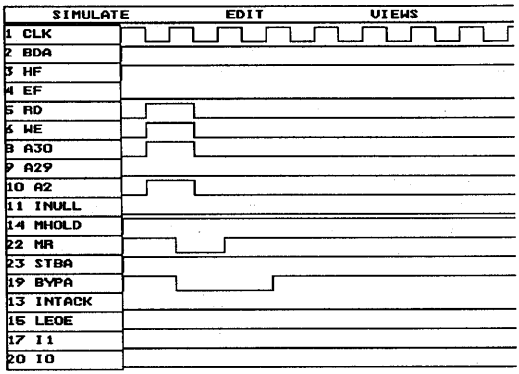


Figure 10. BIFO Initialization -- 7C601 Consumer

The DATAS signal shows the data timing requirements of the CY7C601. During a load operation (read from the BIFO), the CY7C601 requires valid data 3 ns before and 5 ns after the system clock's rising edge. During a store operation, the CY7C601 produces the data to write 29 ns after the falling edge of the first store cycle CLK; this data remains valid for 4 ns after the falling edge of the second store cycle CLK.

Figures 9 and *10* show the simulated waveforms for initializing the BIFO. To initialize the BIFO for FIFO read operations, $\overline{\text{BYPA}}$ must stay Low for two clock cycles; this ensures that $\overline{\text{BYPA}}$ is stable for the entire time $\overline{\text{MR}}$ is Low. At any time, the CY7C601 can reset the BIFO or switch the BIFO direction by writing to or reading from address $40000004. The control circuitry decodes this address by looking at address lines A31, A30, A29, and A2. A2 determines whether the BIFO is to be initialized or if a normal BIFO operation is to be performed. The resetting operation takes priority over all other operations.

The state diagram in *Figure 11* shows the BIFO control circuitry's behavior after the BIFO is initialized to receive data from the CY7C601. The boxes in this diagram represent the state of the control logic. The diamonds represent the conditions that produce the transition from one state to the next.

The BIFO is reset in state 11 and moves to WRITE_IDLE (State 10). The CY7C601 can begin writing to the BIFO at this time by writing data to memory location $40000000.

Because of the pipelined operation of the CY7C601, a latch must hold the write data to meet the BIFO's write set-up time. The data from the latch is shown as the DATAL signal in *Figure 8*. The latch not shown in *Figure 7* latches data from the CY7C601 and allows this data to remain on the bus after the CY7C601 removes the data.

The CY7C601 produces valid write data at the falling edge of the clock in the write's second cycle. A 12-ns-delayed LEOE signal from the 22V10 is asserted at this time to provide the latch enable and output enable of the latch; the delayed LEOE signal is deasserted at the end of the cycle. The LEOE signal ensures that the BIFO data set-up and hold times are met with respect to STBA. The simulated waveforms for a write cycle appear in *Figure 12*.

Each time the CY7C601 writes to the BIFO, the state machine checks to see if the BIFO has become half full ($\overline{\text{HF}}$ asserted). If $\overline{\text{HF}}$ is asserted, the state machine moves to state 13 and interrupts the CY7C601. *Figure 12* shows the timing of these activities.

The interrupt at $\overline{\text{HF}}$ is one of three possible interrupts the control circuitry can generate. A complete list of the interrupt values appears in *Table 5*.

The most efficient way to transfer data through the BIFO is based on fixed-length packets, ideally 1 Kbyte in size. This packet size makes efficient use of the CY7C439 because 1024-word packets can be transferred without interruption unless the $\overline{\text{HF}}$ flag is asserted.

If the $\overline{\text{HF}}$ flag is asserted, the remainder of the current packet can be transferred, then the CY7C601 dis-

Table 5. 7C601 BIFO Control Logic Interrupts

Name	Cause	I1/I0
Not_Empty	E/F goes High when not Reading	01
Empty	E/F goes Low when Reading	01
Not Half Full	HF goes High when Writing	10
Half Full	HF goes Low when Writing	10
Bypass Data	BDA goes Low when Writing	11

continues writing to the CY7C439 until the $\overline{HF}$ flag becomes deasserted (FIFO less than half full). The state machine interrupts the host processor at each of these two conditions — $\overline{HF}$ flag asserted when writing and $\overline{HF}$ flag deasserted when not writing.

If a design does not need to transfer fixed-length packets, the $\overline{E/F}$ flag should be monitored to determine when to stop writing, and the $\overline{HF}$ flag should be monitored to determine when to begin writing. This prevents continual CY7C601 interrupts. The CY7C601 can write to the BIFO until the BIFO is full, then do other work while the consumer reads the BIFO below the half full threshold.

The state machine allows the processor to continue writing to the BIFO and to acknowledge the interrupt that was generated when the FIFO became half full. This ability to continue writing without waiting for an interrupt acknowledge permits the CY7C601 to mask out the $\overline{HF}$ interrupt and continue transferring the current packet. If the CY7C601 acknowledges the interrupt, the state machine moves to state 14. There, the state machine enables the SPARC processor to con-

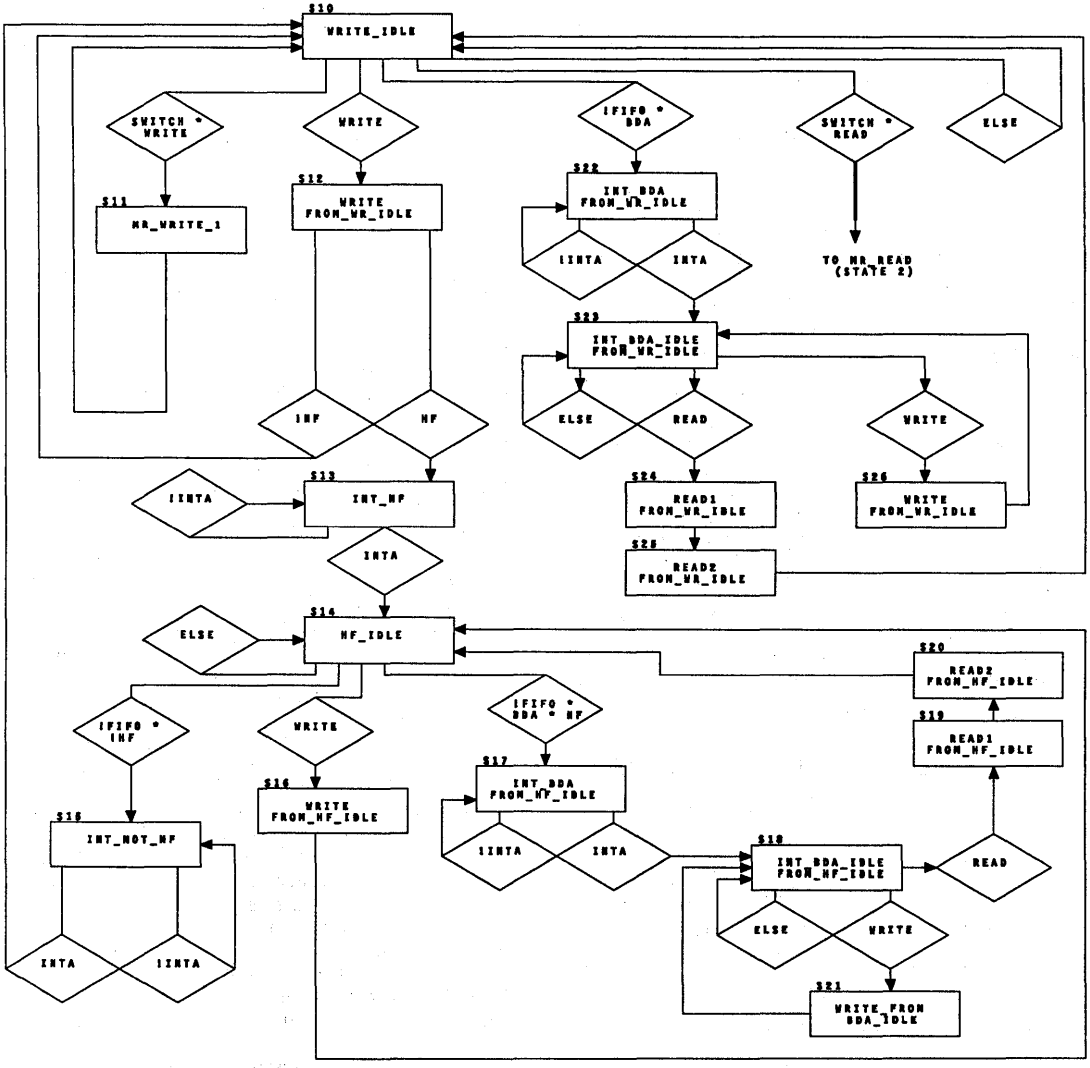


Figure 11. Writing State Diagram of the 7C601 Control Logic

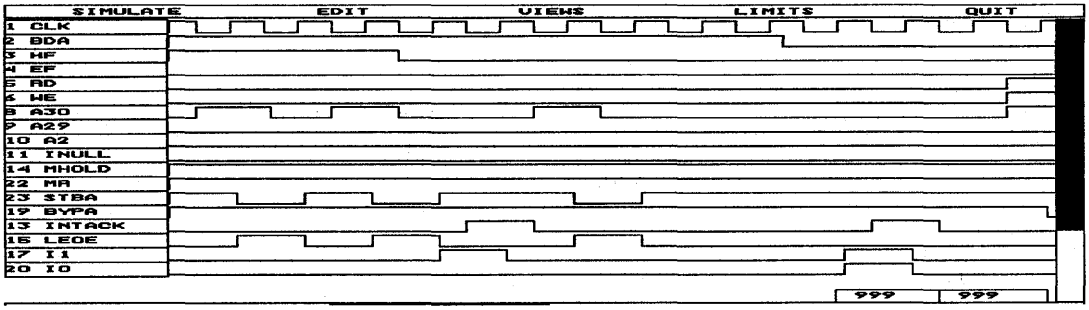


Figure 12. Write Cycle Simulation Waveforms

tinue writing to the BIFO. In this state, the state machine also continues monitoring the $\overline{HF}$ flag and causes an interrupt whenever the flag becomes unasserted (FIFO less than half full).

Another CY7C439 feature that this design utilizes is registered bypass. When the 386 has passed a message to the CY7C601, the $\overline{BDA}$ flag is asserted. If this happens during the HF_IDLE state, the control circuitry generates an interrupt (I1/I0 = 11). After the CY7C601 acknowledges the interrupt, the CY7C601 has the option of writing to the BIFO or reading the bypass register. If the bypass register is read, the state machine moves back to the HF_IDLE state. *Figure 13* shows a simulation of these activities.

It is especially important to note that during registered bypass read, the control circuitry does not monitor the $\overline{MHOLD}$ signal as in other states, but instead drives this signal Low, indicating that the CY7C601 should wait for the expected data. The $\overline{MHOLD}$ signal must be maintained while the missed data is strobed into the processor with the $\overline{MDS}$ signal. For this design, the $\overline{MHOLD}$ output from the control circuitry can also be used as the $\overline{MDS}$ signal.

During the HF_IDLE state, when the BIFO empties below half full ($\overline{HF}$ High), the state machine moves to the WRITE_IDLE state again (state 10). Writing can continue from this state, as before. If the $\overline{BDA}$ flag is asserted during the WRITE_IDLE state, the state machine operates the same as for a $\overline{BDA}$ assertion during the HF_IDLE state.

Figure 14 shows the state diagram of the BIFO configured as the consumer of BIFO data. *Figure 10* shows a simulation of the CY7C601 processor switching the direction of the FIFO from writing (A to B) to reading (B to A). From the READ_IDLE state, the control circuitry interrupts the CY7C601 processor (I1/I0 = 10) whenever the BIFO becomes not empty ($\overline{E/F}$ deasserted). After the CY7C601 has acknowledged the interrupt, it can begin reading from the BIFO. *Figure 15* shows a timing diagram of a read from the BIFO. The DATAS waveform shows the CY7C601's data timing requirements, and the DATAF waveform shows the timing of the data supplied by the BIFO (*Figure 8*). The BIFO holds data valid t_{DVR} after the rising edge of $\overline{STBx}$. The CY7C601, on the other hand, requires that valid data be maintained for at least 5 ns after the rising edge of the processor clock. The 22V10 control clock must therefore be delayed with a device such as a gate that has a delay of 5 ns or even a delay line (CLKD in *Figure 8*) to meet the CY7C601's data-hold requirements.

The CY7C601 can continue reading from the BIFO until the BIFO becomes empty ($\overline{E/F}$ asserted) or the CY7C601 has read all the information it needs. When the BIFO becomes empty, the CY7C601 control circuitry interrupts the processor (I1/I0 = 10); the

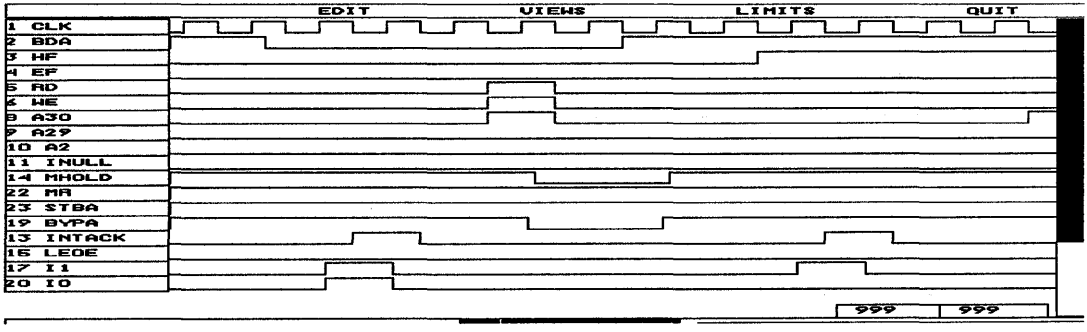


Figure 13. Bypass Register Read

CY7C601 cannot read from the BIFO any more until the BIFO offers the not empty flag.

If the processor reads until the BIFO is empty, the processor might continue reading after the BIFO empties due to the latency of the processor responding to the interrupt. This might cause the processor to read invalid data. This problem is avoided by one of two methods: Employ a special value as the last word written to the FIFO to indicate the transfer's end; or have the producer send the number of data words in the transmission at the transfer's beginning and have the consumer continue reading data until the specified number of words has been read.

From either the READ_IDLE state or from the NOT_EMPTY_IDLE state, the CY7C601 can write to the bypass register. This operation passes a message against the normal FIFO flow to the 386. *Figure 15* includes a bypass register write cycle. The control circuitry performs a bypass register write whenever the CY7C601 is consuming data from the BIFO and the processor performs a write to address location $40000000.

CY7C601 BIFO Control Design File

Appendix A lists the design file used to generate the equations for programming the 22V10. This design file was created using the LOG/iC software package from ISDATA. For a detailed description of this software package, refer to the application note, "Using LOG/iC to Program the CY7C330."

The *IDENTIFICATION section of the design file gives general information about the file. The *PAL section indicates that this file will be used to program a 22V10. The *X-NAMES section describes inputs to the 22V10, and the *Z-NAMES section describes registered outputs of the 22V10. The *Z-VALUES section assigns a unique value to all states in the state machine. These values indicate the signal level on each output while a current state is active, as well as the value of any additional state bits (Q[3..1]).

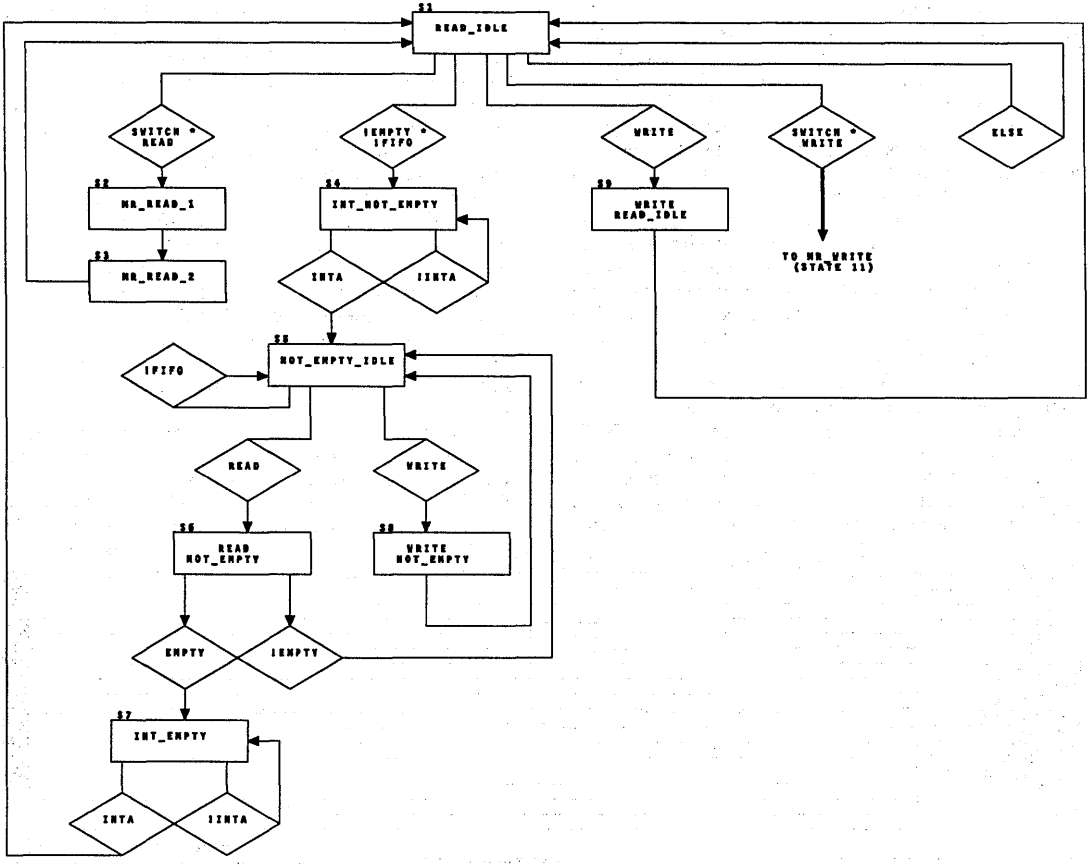


Figure 14. Reading State Diagram of the 7C601 Control Logic

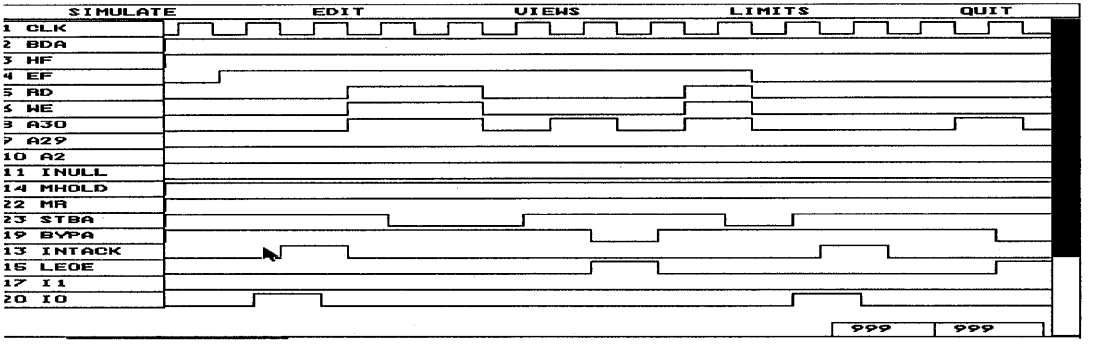


Figure 15. BIFO Read Simulation Waveforms

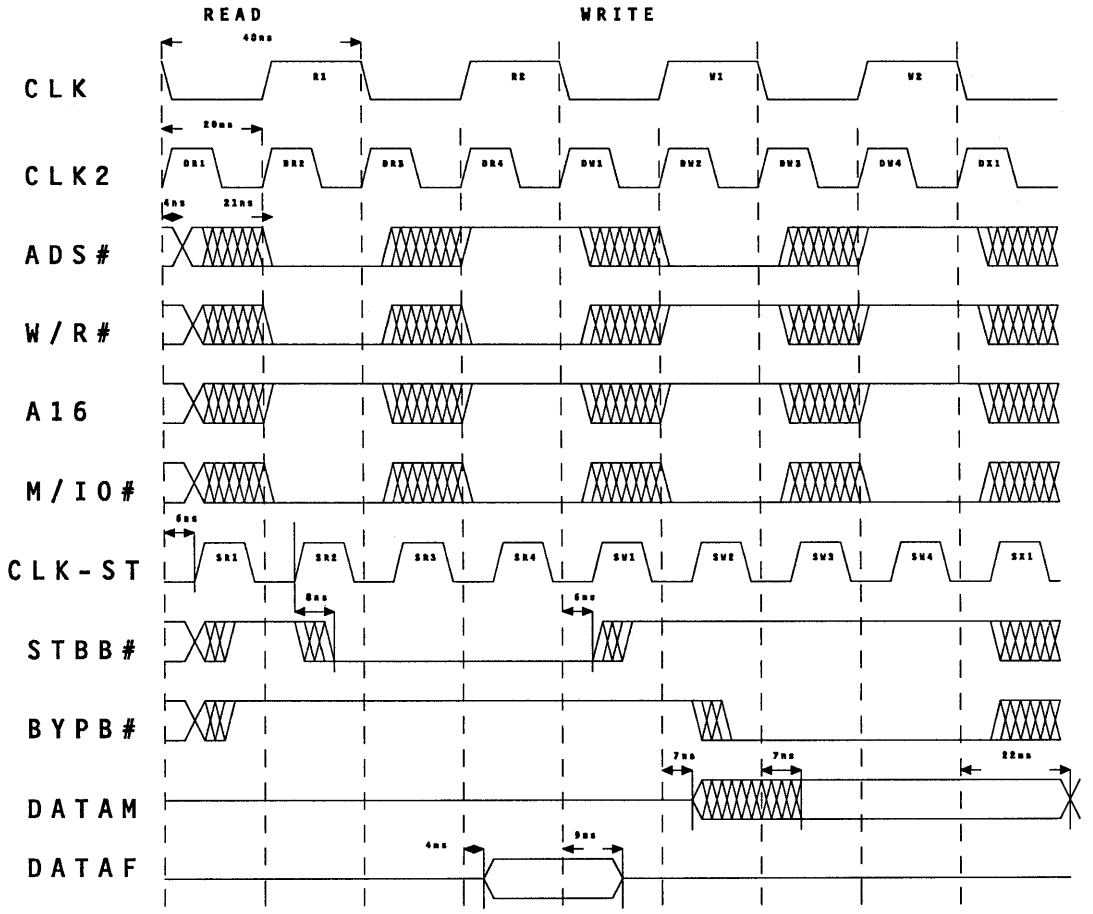


Figure 16. 80386 BIFO Control Logic Timing Diagram

The *FLOW-TABLE section describes the state transitions found in the state diagram. Each line of this section contains the current state, a possible state of the inputs, and the next state the state machine goes to if these inputs are true. For example, line one states that if the state machine is in State 1 (READ_IDLE) and several conditions are met — RD and $\overline{WE}$ are High (indicating a microprocessor read), A31 is Low, A30 is High, A29 is Low (FIFO is selected), A2 is High (FIFO reset address), and MHOLD is Low (7C601 has not been held) — then go to State 2 (MR_READ_1, a master reset cycle configuring the BIFO to transfer data to the CY7C601). Comparing the *Z-VALUES and *FLOW-TABLE sections to the state diagrams in *Figures 11* and *14* reveals that the state diagrams map easily into the LOG/iC design file.

The *STATE-ASSIGNMENT section indicates that the compiler should use the variables listed in the *Z-VALU ES section for state-assignment values. The *PIN section assigns the variable names to 22V10 pin numbers, and the *RUN-CONTROL section configures the compiler and requests various outputs.

The LOG/iC design file produces fully reduced equations that accurately describe the state diagram. The simulation waveforms shown in this application note are taken from the Cypress PLD ToolKit and reflect the function of the equations produced with the ISDATA software.

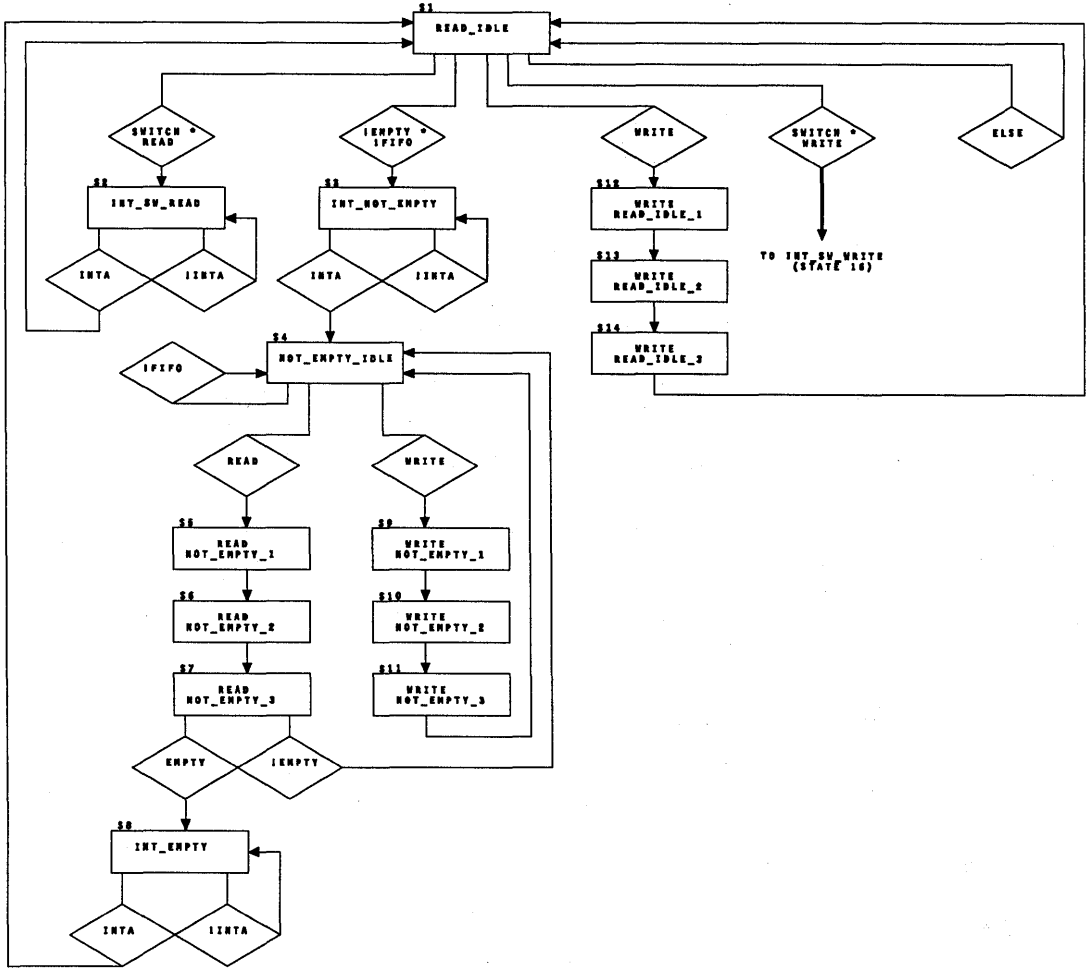


Figure 17. Reading State Diagram of the 80386 Control Logic

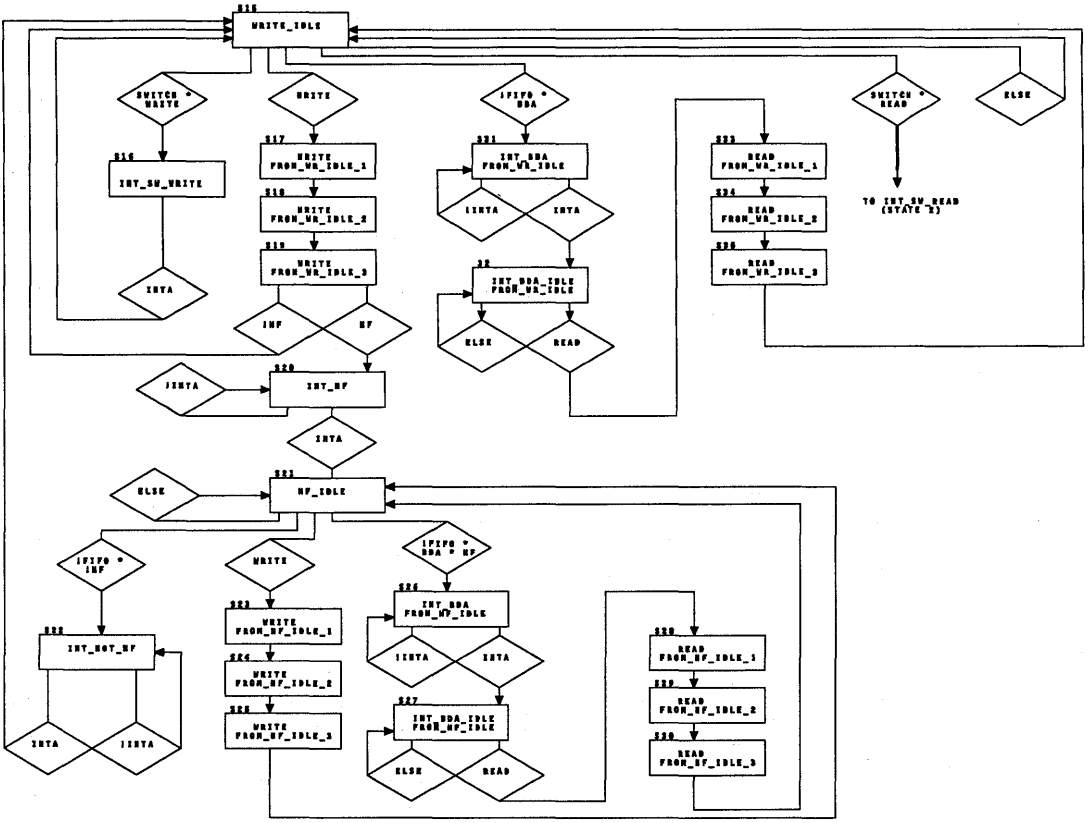


Figure 18. Writing State Diagram of the 80386 Control Logic

80386 BIFO Control Circuitry

A Cypress PAL22V10C-7 implements the 386 control circuitry. The state machine used to control the BIFO operation uses a 6-ns delay (CLK-ST in *Figure 16*) of the 386 system clock to capture the address and control information from the 386. The clock is delayed in the same manner as the delayed clock for the CY7C601

The 386 control logic closely resembles that of the CY7C601. Because the 386 uses a doubled system clock (CLK2 in *Figure16*), the $\overline{STBB}$ and $\overline{BYPB}$ signals must be strobed for three CLK2 cycles, in contrast to the one CLK cycle in the CY7C601 state machine. The other major difference between the two designs is that the 386 has a two-clock-cyle read, in contrast to the one-clock-cycle read for the CY7C601. This design easily meets the set-up and hold times of the BIFO. In fact, you can use the design for the 386 control logic at system speeds as high as 33 MHz.

Unlike the design on the CY7C601 side, the 386 side monitors the BIFO $\overline{MR}$ line and $\overline{BYPA}$ signal to determine the direction for which the BIFO is configured. At start up, the 601 resets the BIFO and sets the direction (usually configuring itself as producer). The embedded control circuitry notices the $\overline{MR}$ signal pulse Low and that the $\overline{BYPA}$ signal is High. These two signal states force the 386 control logic into the read portion of the state machine (*Figure 17*). If, at some later time, the CY7C601 switches the BIFO's direction so that the CY7C601 becomes the consumer of BIFO data, the CY7C601 pulses both $\overline{MR}$ and $\overline{BYPA}$ Low. This forces the 386 control circuitry into the write portion of the state machine (*Figure18*). *Figures19* and *20* show simulated waveforms for 386 control logic's master reset read and master reset write.

The 386 performs a read from the BIFO by driving W/$\overline{R}$ and M/$\overline{IO}$ Low and driving A16 High. As these states imply, the BIFO lies in the upper 32 Kbytes of memory-mapped I/O. The simulated waveforms shown in *Figure21* look similar to those of the CY7C601.

Figure 22 shows the simulated waveforms for a write to the BIFO. The 386 performs a write in the

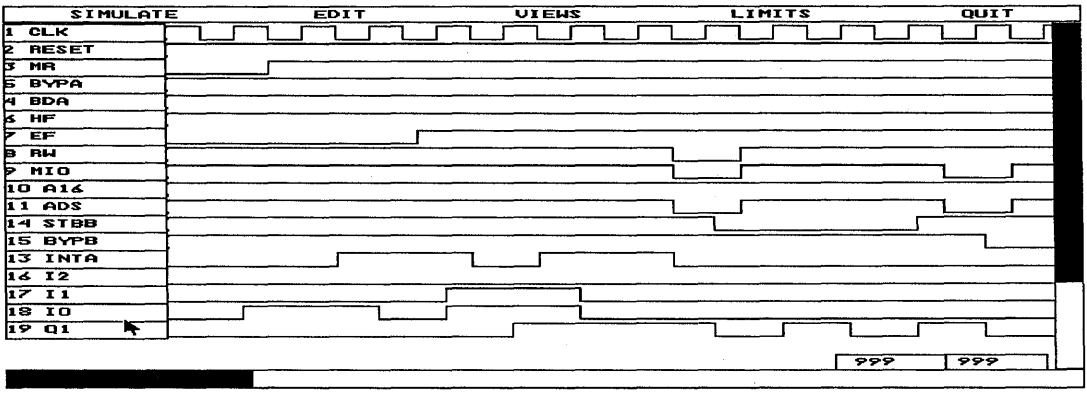


Figure 19. Master Reset Initialization -- 80386 Consumer

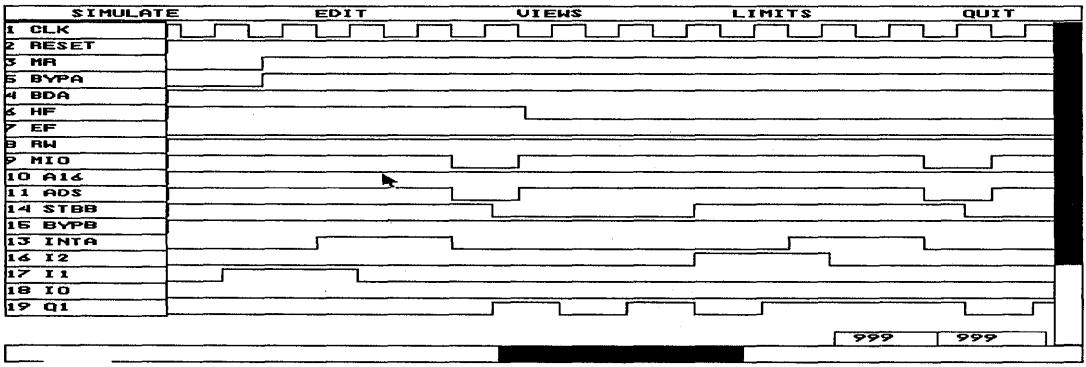


Figure 20. Master Reset Initialization -- 80386 Producer

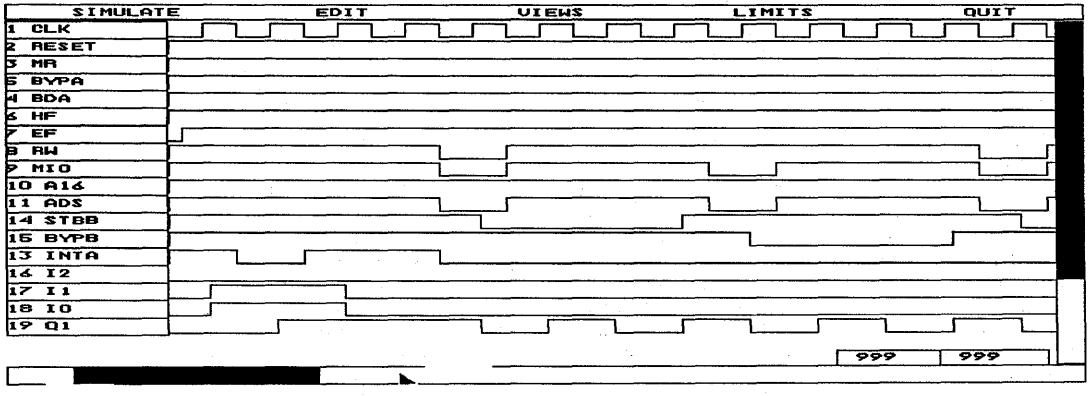


Figure 21. 80386 Control Logic Read Simulation Waveforms

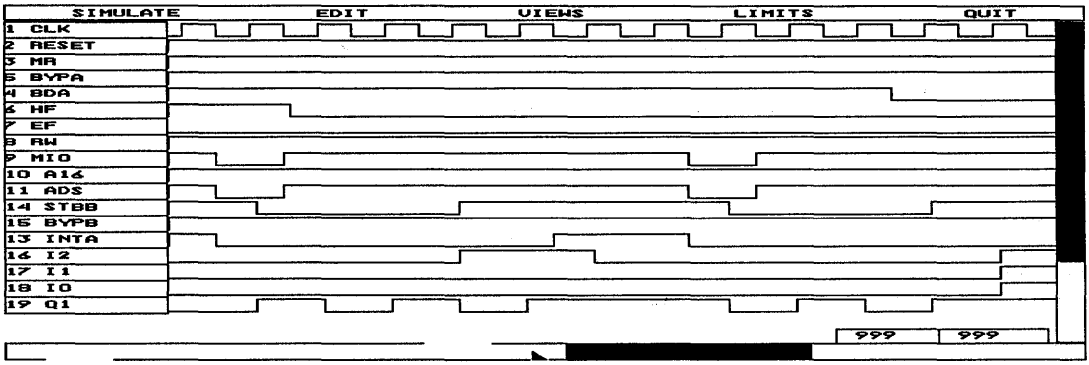


Figure 22. 80386 Control Logic Write Simulation Waveforms

same way as a read, with the exception of driving W/$\overline{R}$ High for the write.

The 386 control logic has seven separate interrupts that it sends to the interrupt controller (*Table 6*). In addition to generating the empty and half-full interrupts also generated by the CY7C601 control logic, the 386 state machine interrupts the microprocessor whenever the BIFO direction is switched. Two separate interrupts ensure that the microprocessor knows the direction in which the BIFO is switched.

The design file for the 386 control circuitry was created using LOG/iC and appears in *Appendix B*. The format of this file is the same as that of the CY7C601 22V10 control logic. Notice that the *FLOW-TABLE section contains fewer state transitions because the control logic does not have to decode address lines to determine if the BIFO direction has switched.

Table 6. 80386 BIFO Control Logic Interrupts

Name	Cause	I2/I1/I0
Switch Read	MR goes Low	001
Switch Write	MR and BYPA go Low	010
Not Empty	E/F goes High when not Reading	011
Empty	E/F goes Low when Reading	011
Not Half Full	HF goes High when Writing	100
Half Full	HF goes Low when Writing	100
Bypass Data	BDA goes low when Writing	111

Appendix A. 7C601 Control Logic Design File

```
*IDENTIFICATION
BIDIRECTIONAL FIFO CONTROL CIRCUITRY FOR THE CYPRESS 601
SEAN DINGMAN
CYPRESS SEMICONDUCTOR

*PAL
TYPE=PALC22V10;

*X-NAMES
CLK,BDA,HF,EF,RD,WE,A31,A30,A29,A2,INULL,MHOLD,INTACK;

*Y-NAMES
MHOLD;

*Z-NAMES
I1,I0,MR,STBA,BYPA,LEOE,Q3,Q2,Q1;

*Z-VALUES

S1  =  0 0 1 1 1  0   000     ; (A) READ_IDLE
S2  =  0 0 0 1 0  0   ---     ;      MR_READ_1
S3  =  0 0 1 1 0  0   0--     ; (B) MR_READ_2
S4  =  0 1 1 1 1  0   -0-     ; (E) INT_NOT_EMPTY
S5  =  0 0 1 1 1  0   001     ; (A) NOT_EMPTY_IDLE
S6  =  0 0 1 0 1  0   ---     ;      READ_FROM_NOT_EMPTY_IDLE
S7  =  0 1 1 1 1  0   -1-     ; (E) INT_EMPTY
S8  =  0 0 1 1 0  1   0-0     ; (C) WRITE_FROM_NOT_EMPTY_IDLE
S9  =  0 0 1 1 0  1   0-1     ; (C) WRITE_FROM_READ_IDLE

S10 =  0 0 1 1 1  0   010     ; (A) WRITE_IDLE
S11 =  0 0 0 1 1  0   ---     ;      MR_SW_WRITE
S12 =  0 0 1 0 1  1   -00     ; (D) WRITE_FROM_WRITE_IDLE
S13 =  1 0 1 1 1  0   -0-     ; (F) INT_HF
S14 =  0 0 1 1 1  0   011     ; (A) HF_IDLE
S15 =  1 0 1 1 1  0   -1-     ; (F) INT_NOT_HF
S16 =  0 0 1 0 1  1   -01     ; (D) WRITE_FROM_HF_IDLE
S17 =  1 1 1 1 1  0   0--     ; (G) INT_BDA_FROM_HF_IDLE
S18 =  0 0 1 1 1  0   1-0     ; (A) INT_BDA_IDLE_FROM_HF_IDLE
S19 =  0 0 1 1 0  0   100     ; (B) READ1_FROM_HF_IDLE
S20 =  0 0 1 1 0  0   101     ; (B) READ2_FROM_HF_IDLE
S21 =  0 0 1 0 1  1   -10     ; (D) WRITE_FROM_INT_BDA_IDLE_FROM_HF_IDLE
S22 =  1 1 1 1 1  0   1--     ; (G) INT_BDA_FROM_WRITE_IDLE
S23 =  0 0 1 1 1  0   1-1     ; (A) INT_BDA_IDLE_FROM_WRITE_IDLE
S24 =  0 0 1 1 0  0   110     ; (B) READ1_FROM_WRITE_IDLE
S25 =  0 0 1 1 0  0   111     ; (B) READ2_FROM_WRITE_IDLE
S26 =  0 0 1 0 1  1   -11     ; (D) WRITE_FROM_INT_BDA_IDLE_FROM_WRITE_IDLE


*BOOLEAN-EQUATIONS

MHOLD.OE = MR & STBA & /BYPA & Q3;
```

Appendix A. 7C601 Control Logic Design File (Continued)

```
*FLOW-TABLE


; READING
RELEVENT = EF,RD,WE,A31,A30,A29,A2,INULL,MHOLD,INTACK;

S 1 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 1 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 1 , X 1--1------, Y 1, F 4    ; READ_IDLE - INT_NOT_EMPTY -- MISC ADDR SPACE
S 1 , X 1--00-----, Y 1, F 4    ; READ_IDLE - INT_NOT_EMPTY
S 1 , X 1--011----, Y 1, F 4    ; READ_IDLE - INT_NOT_EMPTY
S 1 , X 1--0100---, Y 1, F 4    ; READ_IDLE - INT_NOT_EMPTY
S 1 , X 0-0010001-, Y 1, F 9    ; READ_IDLE - WRITE_FROM_READ_IDLE
S 1 , X 0--01001--, Y 1, F 1    ; READ_IDLE - READ_IDLE -- INULLED
S 1 , X 0--010000-, Y 1, F 1    ; READ_IDLE - READ_IDLE -- MHELD
S 1 , X 0--1------, Y 1, F 1    ; READ_IDLE - READ_IDLE -- MISC ADDR SPACE
S 1 , X 0--00-----, Y 1, F 1    ; READ_IDLE - READ_IDLE
S 1 , X 0--011----, Y 1, F 1    ; READ_IDLE - READ_IDLE
S 1 , X ---0101-0-, Y 1, F 1    ; READ_IDLE - READ_IDLE -- HELD SWITCH
S 1 , X -00010111-, Y 1, F 1    ; READ_IDLE - READ_IDLE -- INULLED SWITCH

S 2 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 2 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 2 , X ---1------, Y 1, F 3    ; MR_READ_1 - MR_READ_2
S 2 , X ---00-----, Y 1, F 3    ; MR_READ_1 - MR_READ_2
S 2 , X ---011----, Y 1, F 3    ; MR_READ_1 - MR_READ_2
S 2 , X ---0100---, Y 1, F 3    ; MR_READ_1 - MR_READ_2
S 2 , X ---0101-0-, Y 1, F 3    ; MR_READ_1 - MR_READ_2
S 2 , X -00010111-, Y 1, F 3    ; MR_READ_1 - MR_READ_2

S 3 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 3 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 3 , X ---1------, Y 1, F 1    ; MR_READ_2 - READ_IDLE -- IGNORE MHOLD
S 3 , X ---00-----, Y 1, F 1    ; MR_READ_2 - READ_IDLE -- AND INULL
S 3 , X ---011----, Y 1, F 1    ; MR_READ_2 - READ_IDLE -- FIFO ACTS
S 3 , X ---0100---, Y 1, F 1    ; MR_READ_2 - READ_IDLE -- INDEPENDENTLY
S 3 , X ---0101-0-, Y 1, F 1    ; MR_READ_2 - READ_IDLE
S 3 , X -00010111-, Y 1, F 1    ; MR_READ_2 - READ_IDLE

S 4 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 4 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 4 , X ---1-----0, Y 1, F 4    ; INT_NOT_EMPTY - INT_NOT_EMPTY
S 4 , X ---00----0, Y 1, F 4    ; INT_NOT_EMPTY - INT_NOT_EMPTY
S 4 , X ---011---0, Y 1, F 4    ; INT_NOT_EMPTY - INT_NOT_EMPTY
S 4 , X ---0100--0, Y 1, F 4    ; INT_NOT_EMPTY - INT_NOT_EMPTY
S 4 , X ---0101-00, Y 1, F 4    ; INT_NOT_EMPTY - INT_NOT_EMPTY
S 4 , X -000101110, Y 1, F 4    ; INT_NOT_EMPTY - INT_NOT_EMPTY
S 4 , X ---1-----1, Y 1, F 5    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
S 4 , X ---00----1, Y 1, F 5    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
S 4 , X ---011---1, Y 1, F 5    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
S 4 , X ---0100--1, Y 1, F 5    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
S 4 , X ---0101-01, Y 1, F 5    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
S 4 , X -000101111, Y 1, F 5    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
```

Appendix A. 7C601 Control Logic Design File (Continued)

```
S 5 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 5 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 5 , X -110100-1-, Y 1, F 6    ; NOT_EMPTY_IDLE - READ_NOT_EMPTY
S 5 , X -00010001-, Y 1, F 8    ; NOT_EMPTY_IDLE - WRITE_NOT_EMPTY
S 5 , X ---1------, Y 1, F 5    ; NOT_EMPTY_IDLE - NOT_EMPTY_IDLE -- MISC ADDR
S 5 , X ---00-----, Y 1, F 5    ; NOT_EMPTY_IDLE - NOT_EMPTY_IDLE
S 5 , X ---011----, Y 1, F 5    ; NOT_EMPTY_IDLE - NOT_EMPTY_IDLE
S 5 , X ---010--0-, Y 1, F 5    ; NOT_EMPTY_IDLE - NOT_EMPTY_IDLE -- MHELD
S 5 , X -00010-11-, Y 1, F 5    ; NOT_EMPTY_IDLE - NOT_EMPTY_IDLE -- INULLED

S 6 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 6 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 6 , X 1110100-1-, Y 1, F 6    ; READ_NOT_EMPTY - READ_NOT_EMPTY
S 6 , X 1000100---, Y 1, F 5    ; READ_NOT_EMPTY - NOT_EMPTY_IDLE
S 6 , X 0--1------, Y 1, F 7    ; READ_NOT_EMPTY - INT_EMPTY
S 6 , X 0--00-----, Y 1, F 7    ; READ_NOT_EMPTY - INT_EMPTY
S 6 , X 0--011----, Y 1, F 7    ; READ_NOT_EMPTY - INT_EMPTY
S 6 , X 0--0100---, Y 1, F 7    ; READ_NOT_EMPTY - INT_EMPTY
S 6 , X 0--0101-0-, Y 1, F 7    ; READ_NOT_EMPTY - INT_EMPTY
S 6 , X 000010111-, Y 1, F 7    ; READ_NOT_EMPTY - INT_EMPTY
S 6 , X 1--1------, Y 1, F 5    ; READ_NOT_EMPTY - NOT_EMPTY_IDLE
S 6 , X 1--00-----, Y 1, F 5    ; READ_NOT_EMPTY - NOT_EMPTY_IDLE
S 6 , X 1--011----, Y 1, F 5    ; READ_NOT_EMPTY - NOT_EMPTY_IDLE
S 6 , X 1--010--0-, Y 1, F 5    ; READ_NOT_EMPTY - NOT_EMPTY_IDLE
S 6 , X 100010-11-, Y 1, F 5    ; READ_NOT_EMPTY - NOT_EMPTY_IDLE

S 7 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 7 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 7 , X ---1-----0, Y 1, F 7    ; INT_EMPTY - INT_EMPTY
S 7 , X ---00----0, Y 1, F 7    ; INT_EMPTY - INT_EMPTY
S 7 , X ---011---0, Y 1, F 7    ; INT_EMPTY - INT_EMPTY
S 7 , X ---0100--0, Y 1, F 7    ; INT_EMPTY - INT_EMPTY
S 7 , X ---0101-00, Y 1, F 7    ; INT_EMPTY - READ_IDLE
S 7 , X -000101110, Y 1, F 7    ; INT_EMPTY - READ_IDLE
S 7 , X ---1-----1, Y 1, F 1    ; INT_EMPTY - READ_IDLE
S 7 , X ---00----1, Y 1, F 1    ; INT_EMPTY - READ_IDLE
S 7 , X ---011---1, Y 1, F 1    ; INT_EMPTY - READ_IDLE
S 7 , X ---0100--1, Y 1, F 1    ; INT_EMPTY - READ_IDLE
S 7 , X ---0101-01, Y 1, F 1    ; INT_EMPTY - READ_IDLE
S 7 , X -000101111, Y 1, F 1    ; INT_EMPTY - READ_IDLE

S 8 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 8 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 8 , X ---1------, Y 1, F 5    ; WRITE_NOT_EMPTY - NOT_EMPTY_IDLE
S 8 , X ---00-----, Y 1, F 5    ; WRITE_NOT_EMPTY - NOT_EMPTY_IDLE
S 8 , X ---011----, Y 1, F 5    ; WRITE_NOT_EMPTY - NOT_EMPTY_IDLE
S 8 , X ---0100---, Y 1, F 5    ; WRITE_NOT_EMPTY - NOT_EMPTY_IDLE
S 8 , X ---0101-0-, Y 1, F 5    ; WRITE_NOT_EMPTY - NOT_EMPTY_IDLE
S 8 , X -00010111-, Y 1, F 5    ; WRITE_NOT_EMPTY - NOT_EMPTY_IDLE
```

Appendix A. 7C601 Control Logic Design File (Continued)

```
S 9 , X -110101-1-, Y 1, F 2    ; MR_READ_1
S 9 , X -00010101-, Y 1, F 11   ; MR_WRITE
S 9 , X ---1------, Y 1, F 1    ; WRITE_READ_IDLE - READ_IDLE
S 9 , X ---00-----, Y 1, F 1    ; WRITE_READ_IDLE - READ_IDLE
S 9 , X ---011----, Y 1, F 1    ; WRITE_READ_IDLE - READ_IDLE
S 9 , X ---0100---, Y 1, F 1    ; WRITE_READ_IDLE - READ_IDLE
S 9 , X ---0101-0-, Y 1, F 1    ; WRITE_READ_IDLE - READ_IDLE
S 9 , X -00010111-, Y 1, F 1    ; WRITE_READ_IDLE - READ_IDLE


;WRITING
 RELEVENT=HF,BDA,RD,WE,A31,A30,A29,A2,INULL,MHOLD,INTACK;

S 10, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 10, X --00010101-, Y 1, F 11   ; MR_WRITE
S 10, X -0--1------, Y 1, F 22   ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 10, X -0--00-----, Y 1, F 22   ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 10, X -0--011----, Y 1, F 22   ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 10, X -0--0100---, Y 1, F 22   ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 10, X -0--0101-0-, Y 1, F 22   ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 10, X -000010111-, Y 1, F 22   ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 10, X -100010001-, Y 1, F 12   ; WRITE_IDLE - WRITE_FROM_WRITE_IDLE
S 10, X -1--1------, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -1--00-----, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -1--011----, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -1--0100-0-, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -100010011-, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -1110100---, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -1--0101-0-, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE
S 10, X -100010111-, Y 1, F 10   ; WRITE_IDLE - WRITE_IDLE

S 11, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 11, X --00010101-, Y 1, F 11   ; MR_WRITE
S 11, X ----1------, Y 1, F 10   ; MR_WRITE - WRITE_IDLE
S 11, X ----00-----, Y 1, F 10   ; MR_WRITE - WRITE_IDLE
S 11, X ----011----, Y 1, F 10   ; MR_WRITE - WRITE_IDLE
S 11, X ----0100---, Y 1, F 10   ; MR_WRITE - WRITE_IDLE
S 11, X ----0101-0-, Y 1, F 10   ; MR_WRITE - WRITE_IDLE
S 11, X --00010111-, Y 1, F 10   ; MR_WRITE - WRITE_IDLE

S 12, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 12, X --00010101-, Y 1, F 11   ; MR_WRITE
S 12, X 0---1------, Y 1, F 13   ; WRITE_FROM_WRITE_IDLE - INT_HF
S 12, X 0---00-----, Y 1, F 13   ; WRITE_FROM_WRITE_IDLE - INT_HF
S 12, X 0---011----, Y 1, F 13   ; WRITE_FROM_WRITE_IDLE - INT_HF
S 12, X 0---0100---, Y 1, F 13   ; WRITE_FROM_WRITE_IDLE - INT_HF
S 12, X 0---0101-0-, Y 1, F 13   ; WRITE_FROM_WRITE_IDLE - INT_HF
S 12, X 0-00010111-, Y 1, F 13   ; WRITE_FROM_WRITE_IDLE - INT_HF
S 12, X 1---1------, Y 1, F 10   ; WRITE_FROM_WRITE_IDLE - WRITE_IDLE
S 12, X 1---00-----, Y 1, F 10   ; WRITE_FROM_WRITE_IDLE - WRITE_IDLE
S 12, X 1---011----, Y 1, F 10   ; WRITE_FROM_WRITE_IDLE - WRITE_IDLE
S 12, X 1---0100---, Y 1, F 10   ; WRITE_FROM_WRITE_IDLE - WRITE_IDLE
S 12, X 1---0101-0-, Y 1, F 10   ; WRITE_FROM_WRITE_IDLE - WRITE_IDLE
S 12, X 1-00010111-, Y 1, F 10   ; WRITE_FROM_WRITE_IDLE - WRITE_IDLE
```

Appendix A. 7C601 Control Logic Design File (Continued)

```
S 13,  X --110101-1-, Y 1, F 2    ; MR_READ_1
S 13,  X --00010101-, Y 1, F 11   ; MR_WRITE
S 13,  X ----1-----0, Y 1, F 13   ; INT_HF - INT_HF
S 13,  X ----00----0, Y 1, F 13   ; INT_HF - INT_HF
S 13,  X ----011---0, Y 1, F 13   ; INT_HF - INT_HF
S 13,  X ----0100--0, Y 1, F 13   ; INT_HF - INT_HF
S 13,  X ----0101-00, Y 1, F 13   ; INT_HF - INT_HF
S 13,  X --000101110, Y 1, F 13   ; INT_HF - INT_HF
S 13,  X ----1-----1, Y 1, F 14   ; INT_HF - HF_IDLE
S 13,  X ----00----1, Y 1, F 14   ; INT_HF - HF_IDLE
S 13,  X ----011---1, Y 1, F 14   ; INT_HF - HF_IDLE
S 13,  X ----0100--1, Y 1, F 14   ; INT_HF - HF_IDLE
S 13,  X ----0101-01, Y 1, F 14   ; INT_HF - HF_IDLE
S 13,  X --000101111, Y 1, F 14   ; INT_HF - HF_IDLE

S 14,  X --110101-1-, Y 1, F 2    ; MR_READ_1
S 14,  X --00010101-, Y 1, F 11   ; MR_WRITE
S 14,  X 1---1------, Y 1, F 15   ; HF_IDLE - INT_NOT_HF
S 14,  X 1---00-----, Y 1, F 15   ; HF_IDLE - INT_NOT_HF
S 14,  X 1---011----, Y 1, F 15   ; HF_IDLE - INT_NOT_HF
S 14,  X 1---0100---, Y 1, F 15   ; HF_IDLE - INT_NOT_HF
S 14,  X 1---0101-0-, Y 1, F 15   ; HF_IDLE - INT_NOT_HF
S 14,  X 1-00010111-, Y 1, F 15   ; HF_IDLE - INT_NOT_HF
S 14,  X 00--1------, Y 1, F 17   ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 14,  X 00--00-----, Y 1, F 17   ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 14,  X 00--011----, Y 1, F 17   ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 14,  X 00--0100---, Y 1, F 17   ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 14,  X 00--0101-0-, Y 1, F 17   ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 14,  X 0000010111-, Y 1, F 17   ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 14,  X 0100010001-, Y 1, F 16   ; HF_IDLE - WRITE_FROM_HF_IDLE
S 14,  X 01--1------, Y 1, F 14   ; HF_IDLE - HF_IDLE
S 14,  X 01--00-----, Y 1, F 14   ; HF_IDLE - HF_IDLE
S 14,  X 01--011----, Y 1, F 14   ; HF_IDLE - HF_IDLE
S 14,  X 01110100---, Y 1, F 14   ; HF_IDLE - HF_IDLE
S 14,  X 01--010--0-, Y 1, F 14   ; HF_IDLE - HF_IDLE
S 14,  X 0100010-11-, Y 1, F 14   ; HF_IDLE - HF_IDLE

S 15,  X --110101-1-, Y 1, F 2    ; MR_READ_1
S 15,  X --00010101-, Y 1, F 11   ; MR_WRITE
S 15,  X ----1-----0, Y 1, F 15   ; INT_NOT_HF - INT_NOT_HF
S 15,  X ----00----0, Y 1, F 15   ; INT_NOT_HF - INT_NOT_HF
S 15,  X ----011---0, Y 1, F 15   ; INT_NOT_HF - INT_NOT_HF
S 15,  X ----0100--0, Y 1, F 15   ; INT_NOT_HF - INT_NOT_HF
S 15,  X ----0101-00, Y 1, F 15   ; INT_NOT_HF - INT_NOT_HF
S 15,  X --000101110, Y 1, F 15   ; INT_NOT_HF - INT_NOT_HF
S 15,  X ----1-----1, Y 1, F 10   ; INT_NOT_HF - WRITE_IDLE
S 15,  X ----00----1, Y 1, F 10   ; INT_NOT_HF - WRITE_IDLE
S 15,  X ----011---1, Y 1, F 10   ; INT_NOT_HF - WRITE_IDLE
S 15,  X ----0100--1, Y 1, F 10   ; INT_NOT_HF - WRITE_IDLE
S 15,  X ----0101-01, Y 1, F 10   ; INT_NOT_HF - WRITE_IDLE
S 15,  X --000101111, Y 1, F 10   ; INT_NOT_HF - WRITE_IDLE
```

Appendix A. 7C601 Control Logic Design File (Continued)

```
S 16, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 16, X --00010101-, Y 1, F 11   ; MR_WRITE
S 16, X ----1------, Y 1, F 14   ; WRITE_FROM_HF_IDLE - HF_IDLE
S 16, X ----00-----, Y 1, F 14   ; WRITE_FROM_HF_IDLE - HF_IDLE
S 16, X ----011----, Y 1, F 14   ; WRITE_FROM_HF_IDLE - HF_IDLE
S 16, X ----0100---, Y 1, F 14   ; WRITE_FROM_HF_IDLE - HF_IDLE
S 16, X ----0101-0-, Y 1, F 14   ; WRITE_FROM_HF_IDLE - HF_IDLE
S 16, X --00010111-, Y 1, F 14   ; WRITE_FROM_HF_IDLE - HF_IDLE

S 17, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 17, X --00010101-, Y 1, F 11   ; MR_WRITE
S 17, X ----1-----0, Y 1, F 17   ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 17, X ----00----0, Y 1, F 17   ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 17, X ----011---0, Y 1, F 17   ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 17, X ----0100--0, Y 1, F 17   ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 17, X ----0101-00, Y 1, F 17   ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 17, X --000101110, Y 1, F 17   ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 17, X ----1-----1, Y 1, F 18   ; INT_BDA_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 17, X ----00----1, Y 1, F 18   ; INT_BDA_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 17, X ----011---1, Y 1, F 18   ; INT_BDA_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 17, X ----0100--1, Y 1, F 18   ; INT_BDA_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 17, X ----0101-01, Y 1, F 18   ; INT_BDA_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 17, X --000101111, Y 1, F 18   ; INT_BDA_FROM_HF_IDLE -iNT_BDA_IDLE_FROM_HF_IDLE

S 18, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 18, X --00010101-, Y 1, F 11   ; MR_WRITE
S 18, X --110100-1-, Y 0, F 19   ; INT_BDA_IDLE_FROM_HF_IDLE - READ1_FROM_BDA_IDLE
S 18, X --00010001-, Y 1, F 21   ; INT_BDA_IDLE_FROM_HF_IDLE - WRITE_FROM_BDA_IDLE
S 18, X ----1------, Y 1, F 18   ; INT_BDA_IDLE_FROM_HF_IDLE-INT_BDA_IDLE_FROM_HF_IDLE
S 18, X ----00-----, Y 1, F 18   ; INT_BDA_IDLE_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 18, X ----011----, Y 1, F 18   ; INT_BDA_IDLE_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 18, X ----010--0-, Y 1, F 18   ; INT_BDA_IDLE_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 18, X --00010-11-, Y 1, F 18   ; INT_BDA_IDLE_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE


S 19, X ----------, Y 0, F 20    ; READ1_FROM_BDA_IDLE - HF_IDLE


S 20, X ----------, Y 0, F 14    ; READ2_FROM_BDA_IDLE - HF_IDLE


S 21, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 21, X --00010101-, Y 1, F 11   ; MR_WRITE
S 21, X ----1------, Y 1, F 18   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 21, X ----00-----, Y 1, F 18   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 21, X ----011----, Y 1, F 18   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 21, X ----0100---, Y 1, F 18   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 21, X ----0101-0-, Y 1, F 18   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 21, X --00010111-, Y 1, F 18   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
```

Appendix A. 7C601 Control Logic Design File (Continued)

```
S 22, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 22, X --00010101-, Y 1, F 11   ; MR_WRITE
S 22, X ----1-----0, Y 1, F 22   ; INT_BDA_FROM_WRITE_IDLE INT_BDA_FROM_WRITE_IDLE
S 22, X ----00----0, Y 1, F 22   ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 22, X ----011---0, Y 1, F 22   ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 22, X ----0100--0, Y 1, F 22   ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 22, X ----0101-00, Y 1, F 22   ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 22, X --000101110, Y 1, F 22   ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 22, X ----1-----1, Y 1, F 23   ; INT_BDA_FROM_WRITE_IDLE -INT_BDA_IDLE_FROM_WRITE_IDLE
S 22, X ----00----1, Y 1, F 23   ; INT_BDA_FROM_WRITE_IDLE -INT_BDA_IDLE_FROM_WRITE_IDLE
S 22, X ----011---1, Y 1, F 23   ; INT_BDA_FROM_WRITE_IDLE -INT_BDA_IDLE_FROM_WRITE_IDLE
S 22, X ----0100--1, Y 1, F 23   ; INT_BDA_FROM_WRITE_IDLE -INT_BDA_IDLE_FROM_WRITE_IDLE
S 22, X ----0101-01, Y 1, F 23   ; INT_BDA_FROM_WRITE_IDLE -INT_BDA_IDLE_FROM_WRITE_IDLE
S 22, X --000101111, Y 1, F 23   ; INT_BDA_FROM_WRITE_IDLE -INT_BDA_IDLE_FROM_WRITE_IDLE

S 23, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 23, X --00010101-, Y 1, F 11   ; MR_WRITE
S 23, X --110100-1-, Y 0, F 24   ; INT_BDA_IDLE_FROM_WRITE_IDLE - READ1_FROM_BDA_IDLE
S 23, X --00010001-, Y 1, F 26   ; INT_BDA_IDLE_FROM_WRITE_IDLE - WRITE_FROM_BDA_IDLE
S 23, X ----1------, Y 1, F 23   ;INT_BDA_IDLE_FROM_WRITE_IDLE INT_BDA_IDLE_FROM_WRITE_IDLE
S 23, X ----00-----, Y 1, F 23   ; INT_BDA_IDLE_FROM_WRITE_IDLE
S 23, X ----011----, Y 1, F 23   ; INT_BDA_IDLE_FROM_WRITE_IDLE
S 23, X ----010--0-, Y 1, F 23   ; INT_BDA_IDLE_FROM_WRITE_IDLE
S 23, X --00010-11-, Y 1, F 23   ; INT_BDA_IDLE_FROM_WRITE_IDLE

S 24, X ----------, Y 0, F 25    ; READ1_FROM_BDA_IDLE - WRITE_IDLE

S 25, X ----------, Y 0, F 10    ; READ2_FROM_BDA_IDLE - WRITE_IDLE

S 26, X --110101-1-, Y 1, F 2    ; MR_READ_1
S 26, X --00010101-, Y 1, F 11   ; MR_WRITE
S 26, X ----1------, Y 1, F 23   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 26, X ----00-----, Y 1, F 23   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 26, X ----011----, Y 1, F 23   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 26, X ----0100---, Y 1, F 23   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 26, X ----0101-0-, Y 1, F 23   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE
S 26, X --00010111-, Y 1, F 23   ; WRITE_FROM_BDA_IDLE - INT_BDA_IDLE_FROM_HF_IDLE


*STATE-ASSIGNMENT
Z-VALUES

*PIN
CLK = 1, BDA = 2, HF = 3, EF = 4, RD = 5, WE = 6, A31 = 7, A30 = 8,
A29 = 9, A2 = 10, INULL = 11, MHOLD = 14, INTACK = 13,
Q3 = 21, Q1 = 18, LEOE = 15, I1 = 17, BYPA = 19, I0 = 20,
Q2 = 16, MR = 22, STBA = 23;

*RUN-CONTROL
LISTING = LONG,SYMBOL-TABLE,EQUATIONS,PINOUT,PLOT,FUSEPLOT;
PROGFORMAT = L-EQUATIONS,JEDEC;
OPTIMIZATION = p-terms;
*END
```

Appendix B. 80386 Control Logic Design File

```
*IDENTIFICATION
BIDIRECTIONAL FIFO CONTROL CIRCUITRY FOR THE INTEL 386
SEAN DINGMAN
CYPRESS SEMICONDUCTOR

*PAL
TYPE=PALC22V10;

*X-NAMES
CLK,RESET,MR,BDA,BYPA,HF,EF,RW,MIO,A16,ADS,INTA;

*Z-NAMES
I2,I1,I0,STBB,BYPB,Q5,Q4,Q3,Q2,Q1;

*Z-VALUES

S1  =  0 0 0 1 1  --000      ; (A) READ_IDLE (6)
S2  =  0 0 1 1 1  -----      ;      INT_SW_READ
S3  =  0 1 1 1 1  0----      ; (D) INT_NOT_EMPTY
S4  =  0 0 0 1 1  --001      ; (A) NOT EMPTY IDLE
S5  =  0 0 0 0 1  -0000      ; (B) READ_FROM_NOT_EMPTY_IDLE_1 (9)
S6  =  0 0 0 0 1  -0001      ; (B) READ_FROM_NOT_EMPTY_IDLE_2
S7  =  0 0 0 0 1  -0010      ; (B) READ_FROM_NOT_EMPTY_IDLE_3
S8  =  0 1 1 1 1  1----      ; (D) INT_EMPTY
S9  =  0 0 0 1 0  -0000      ; (C) WRITE_FROM_NOT_EMPTY_IDLE_1 (12)
S10 =  0 0 0 1 0  -0001      ; (C) WRITE_FROM_NOT_EMPTY_IDLE_2
S11 =  0 0 0 1 0  -0010      ; (C) WRITE_FROM_NOT_EMPTY_IDLE_3
S12 =  0 0 0 1 0  -0011      ; (C) WRITE_FROM_READ_IDLE_1
S13 =  0 0 0 1 0  -0100      ; (C) WRITE_FROM_READ_IDLE_2
S14 =  0 0 0 1 0  -0101      ; (C) WRITE_FROM_READ_IDLE_3

S15 =  0 0 0 1 1  --010      ; (A) WRITE_IDLE
S16 =  0 1 0 1 1  -----      ;      INT_SW_WRITE
S17 =  0 0 0 0 1  -0011      ; (B) WRITE_FROM_WRITE_IDLE_1
S18 =  0 0 0 0 1  -0100      ; (B) WRITE_FROM_WRITE_IDLE_2
S19 =  0 0 0 0 1  -0101      ; (B) WRITE_FROM_WRITE_IDLE_3
S20 =  1 0 0 1 1  0----      ; (E) INT_HF
S21 =  0 0 0 1 1  --011      ; (A) HF_IDLE
S22 =  1 0 0 1 1  1----      ; (E) INT_NOT_HF
S23 =  0 0 0 0 1  -0110      ; (B) WRITE_FROM_HF_IDLE_1
S24 =  0 0 0 0 1  -0111      ; (B) WRITE_FROM_HF_IDLE_2
S25 =  0 0 0 0 1  -1000      ; (B) WRITE_FROM_HF_IDLE_3
S26 =  1 1 1 1 1  0----      ; (F) INT_BDA_FROM_HF_IDLE
S27 =  0 0 0 1 1  --100      ; (A) INT_BDA_IDLE_FROM_HF_IDLE
S28 =  0 0 0 1 0  -0110      ; (C) READ_FROM_HF_IDLE_1
S29 =  0 0 0 1 0  -0111      ; (C) READ_FROM_HF_IDLE_2
S30 =  0 0 0 1 0  -1000      ; (C) READ_FROM_HF_IDLE_3
S31 =  1 1 1 1 1  1----      ; (F) INT_BDA_FROM_WRITE_IDLE
S32 =  0 0 0 1 1  --101      ; (A) INT_BDA_IDLE_FROM_WRITE_IDLE
S33 =  0 0 0 1 0  -1001      ; (C) READ_FROM_WRITE_IDLE_1
S34 =  0 0 0 1 0  -1010      ; (C) READ_FROM_WRITE_IDLE_2
S35 =  0 0 0 1 0  -1011      ; (C) READ_FROM_WRITE_IDLE_3
```

Appendix B. 80386 Control Logic Design File (Continued)

```
*FLOW-TABLE

; RESETING STATES
RELEVENT = RESET,MR,BYPA ;
S[1..26],    X 0--, F 1        ; READ IDLE ON RESET
S[1..26],    X 100, F 16       ; SWITCH DIRECTIONS B-A
S[1..26],    X 101, F 2        ; SWITCH DIRECTIONS A-B

RELEVENT = MR = 1;
RELEVENT = RESET = 1;

; READING
RELEVENT = EF,RW,MIO,A16,ADS,INTA ;

S 1 , X 1-1---, F 3    ; READ_IDLE - INT_NOT_EMPTY
S 1 , X 1-00--, F 3    ; READ_IDLE - INT_NOT_EMPTY
S 1 , X -1010-, F 12   ; READ_IDLE - WRITE_BYPASS_FROM_READ_IDLE_1
S 1 , XREST    , F 1   ; READ_IDLE - READ_IDLE

S 2 , X -----1, F 1    ; INT_SW_READ - READ_IDLE
S 2 , X -----0, F 2    ; INT_SW_READ - INT_SW_READ

S 3 , X -----1, F 4    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE
S 3 , X -----0, F 3    ; INT_NOT_EMPTY - NOT_EMPTY_IDLE

S 4 , X -0010-, F 5    ; NOT_EMPTY_IDLE - READ_FROM_NOT_EMPTY_IDLE_1
S 4 , X -1010-, F 9    ; NOT_EMPTY_IDLE - WRITE_FROM_NOT_EMPTY_IDLE_1
S 4 , XREST    , F 4   ; NOT_EMPTY_IDLE - NOT_EMPTY_IDLE

S 5 , X ------, F 6    ; READ_FROM_NOT_EMPTY_IDLE_1 - READ_FROM_NOT_EMPTY_IDLE_2

S 6 , X ------, F 7    ; READ_FROM_NOT_EMPTY_IDLE_2 - READ_FROM_NOT_EMPTY_IDLE_3

S 7 , X 0-----, F 8    ; READ_FROM_NOT_EMPTY_IDLE_3 - INT_EMPTY
S 7 , X 1-----, F 4    ; READ_FROM_NOT_EMPTY_IDLE_3 - NOT_EMPTY_IDLE

S 8 , X -----1, F 1    ; INT_EMPTY - READ_IDLE
S 8 , X -----0, F 8    ; INT_EMPTY - INT_EMPTY

S 9 , X ------, F 10    ; WRITE_FROM_NOT_EMPTY_IDLE_1 -WRITE_FROM_NOT_EMPTY_IDLE_2

S 10, X ------, F 11    ; WRITE_FROM_NOT_EMPTY_IDLE_2 -WRITE_FROM_NOT_EMPTY_IDLE_3

S 11, X ------, F 4     ; WRITE_FROM_NOT_EMPTY_IDLE_3 -NOT_EMPTY_IDLE

S 12, X ------, F 13   ; WRITE_FROM_READ_IDLE_1 - WRITE_FROM_READ_IDLE_2

S 13, X ------, F 14   ; WRITE_FROM_READ_IDLE_2 - WRITE_FROM_READ_IDLE_3

S 14, X ------, F 1    ; WRITE_FROM_READ_IDLE_3 - READ_IDLE
```

Appendix B. 80386 Control Logic Design File (Continued)

```
;WRITING
RELEVENT = HF,BDA,RW,MIO,A16,ADS,INTA ;

S 15, X --1010-, F 17     ; WRITE_IDLE - WRITE_FROM_WRITE_IDLE_1
S 15, X -0-1---, F 31     ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 15, X -0-00--, F 31     ; WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 15, XREST     , F 15    ; WRITE_IDLE - WRITE_IDLE

S 16, X ------1, F 15     ; INT_SW_WRITE - WRITE_IDLE
S 16, X ------0, F 16     ; INT_SW_WRITE - INT_SW_WRITE

S 17, X -------, F 18     ; WRITE_FROM_WRITE_IDLE_1 - WRITE_FROM_WRITE_IDLE_2

S 18, X -------, F 19     ; WRITE_FROM_WRITE_IDLE_2 - WRITE_FROM_WRITE_IDLE_3

S 19, X 1------, F 15     ; WRITE_FROM_WRITE_IDLE_3 - WRITE_IDLE
S 19, X 0------, F 20     ; WRITE_FROM_WRITE_IDLE_3 - INT_HF

S 20, X ------1, F 21     ; INT_HF - HF_IDLE
S 20, X ------0, F 20     ; INT_HF - INT_HF

S 21, X 1--1---, F 22     ; HF_IDLE - INT_NOT_HF
S 21, X 1--00--, F 22     ; HF_IDLE - INT_NOT_HF
S 21, X --1010-, F 23     ; HF_IDLE - WRITE_FROM_HF_IDLE_1
S 21, X 00-1---, F 26     ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 21, X 00-00--, F 26     ; HF_IDLE - INT_BDA_FROM_HF_IDLE
S 21, X --0010-, F 27     ; HF_IDLE - READ_FROM_HF_IDLE_1
S 21, XREST     , F 21    ; HF_IDLE - HF_IDLE

S 22, X ------0, F 22     ; INT_NOT_HF - INT_NOT_HF
S 22, X ------1, F 15     ; INT_NOT_HF - WRITE_IDLE

S 23, X -------, F 24     ; WRITE_FROM_HF_IDLE_1 - WRITE_FROM_HF_IDLE_2

S 24, X -------, F 25     ; WRITE_FROM_HF_IDLE_2 - WRITE_FROM_HF_IDLE_3

S 25, X -------, F 21     ; WRITE_FROM_HF_IDLE_3 - HF_IDLE

S 26, X ------0, F 26     ; INT_BDA_FROM_HF_IDLE - INT_BDA_FROM_HF_IDLE
S 26, X ------1, F 27     ; INT_BDA_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE

S 27, X --0010-, F 28     ; INT_BDA_IDLE_FROM_HF_IDLE - READ_FROM_HF_IDLE_1
S 27, XREST     , F 27    ; INT_BDA_IDLE_FROM_HF_IDLE - INT_BDA_IDLE_FROM_HF_IDLE

S 28, X -------, F 29     ; READ_FROM_HF_IDLE_1 - READ_FROM_HF_IDLE_2

S 29, X -------, F 30     ; READ_FROM_HF_IDLE_2 - READ_FROM_HF_IDLE_3

S 30, X -------, F 21     ; READ_FROM_HF_IDLE_3 - HF_IDLE

S 31, X ------0, F 31     ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_FROM_WRITE_IDLE
S 31, X ------1, F 32     ; INT_BDA_FROM_WRITE_IDLE - INT_BDA_IDLE_FROM_WR_IDLE
```

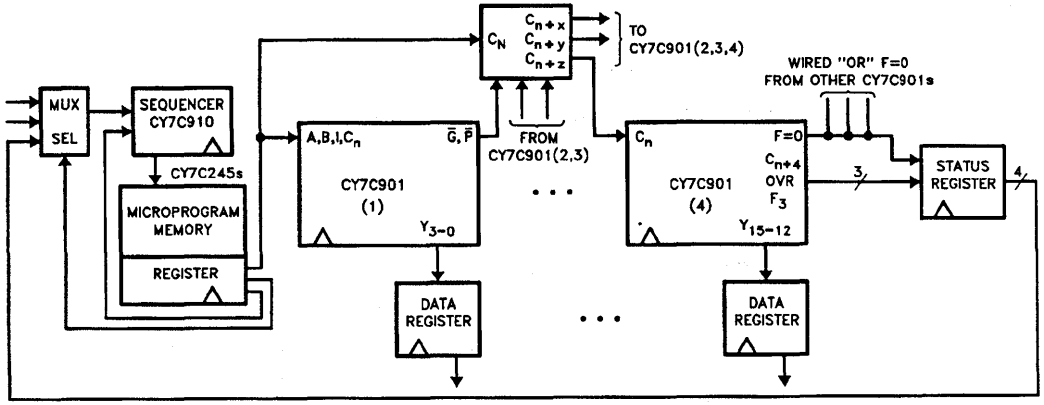
Appendix B. 80386 Control Logic Design File (Continued)

```
S 32, X --0010-, F 33     ; INT_BDA_IDLE_FROM_WR_IDLE - READ_FROM_WR_IDLE_1
S 32, XREST     , F 32     ; INT_BDA_IDLE_FROM_WR_IDLE - INT_BDA_IDLE_FROM_WR_IDLE

S 33, X -------, F 34     ; READ_FROM_WRITE_IDLE_1 - READ_FROM_WRITE_IDLE_2

S 34, X -------, F 35     ; READ_FROM_WRITE_IDLE_2 - READ_FROM_WRITE_IDLE_3

S 35, X -------, F 15     ; READ_FROM_WRITE_IDLE_3 - WRITE_IDLE

*STATE-ASSIGNMENT
Z-VALUES

*PIN
CLK = 1, RESET = 2, MR = 3, BDA = 4, BYPA = 5, HF = 6, EF = 7,
RW = 8, MIO = 9, A16 = 10, ADS = 11, INTA = 13, STBB = 14, BYPB = 15,
I2 = 16, I1 = 17, I0 = 18, Q1 = 19, Q2 = 20, Q3 = 21, Q4 = 22, Q5 = 23;

*RUN-CONTROL
LISTING = LONG,SYMBOL-TABLE,EQUATIONS,PINOUT,PLOT,FUSEPLOT;
PROGFORMAT = L-EQUATIONS,JEDEC;
OPTIMIZATION = p-terms;
*END
```

Microcoded System Performance

This application note describes the performance of Cypress's microcoded processor devices in 16- and 32-bit processors configurations. Included is a critical-path timing analysis of the data loop and control loop for generic 16- and 32-bit systems. A discussion of the speed and power advantages offered by CY7C9101 systems is also presented.

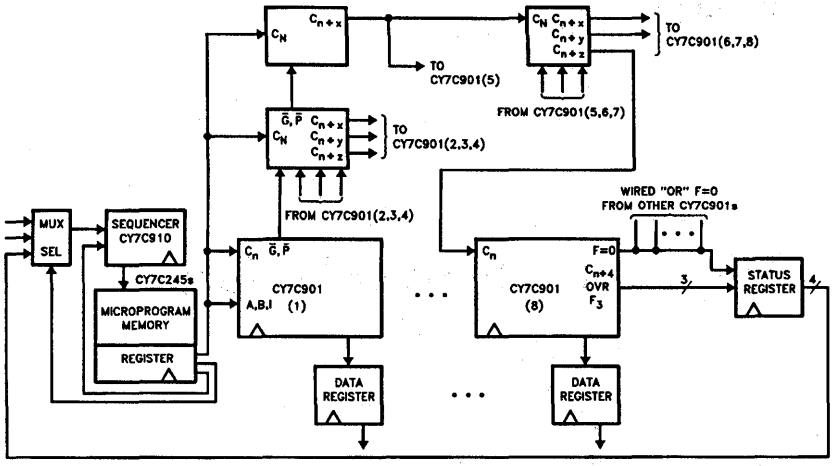
The Cypress microcoded processor family is the fastest available. Increasing functional integration is evident in the CY7C9101 16-bit slice, which is the equivalent of four CY7C901s (4-bit slices) and a 2902 carry lookahead generator. By placing these functions on a single chip, Cypress has reduced the interconnect delays between chips. Significant improvement in overall system



	Data Loop			Control Loop	
CY7C245	Clock to Output	12	CY7C245	Clock to Output	12
CY7C901	A, B to $\overline{G}$, $\overline{P}$	28	MUX	Select to Output	12
Carry Logic	$\overline{G}_0$, $\overline{P}_0$ to C_{n+z}	9	CY7C910	CC to Output	22
CY7C901	C_n to Worst Case	18	CY7C245	Access Time	20
Register	Setup	4			66 ns
		71 ns			

Minimum Clock Period = 71 ns

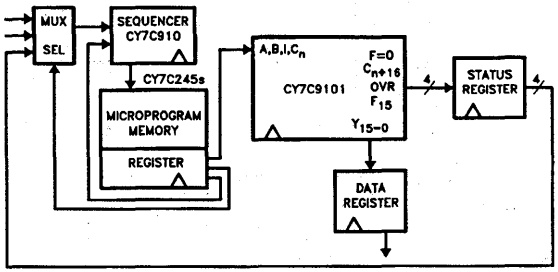
Figure 1. CY7C901-Based 16-Bit System (Pipelined System, Add without Simultaneous Shift)



	Data Loop			Control Loop	
CY7C245	Clock to Output	12	CY7C245	Clock to Output	12
CY7C901	A, B to $\overline{G}$, $\overline{P}$	28	MUX	Select to Output	12
Carry Logic	$\overline{G}_0$, $\overline{P}_0$ to $\overline{G}$, $\overline{P}$	12	CY7C910	CC to Output	22
	$\overline{G}_0$, $\overline{P}_0$ to $C_n + x$	9	CY7C245	Access Time	20
	C_n to $C_n + x, y, z$	14			66 ns
CY7C901	C_n to Worst Case	18			
Register	Setup	4			
		97 ns			

Minimum Clock Period = 97 ns

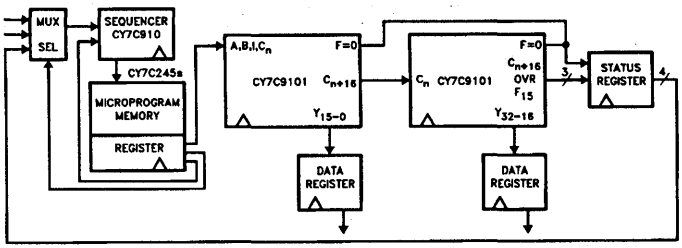
Figure 2. CY7C901-Based 32-Bit System (Pipelined System, Add without Simultaneous Shift)



	Data Loop			Control Loop	
CY7C245	Clock to Output	12	CY7C245	Clock to Output	12
CY7C9101	A, B to Y, $C_n + 16$, OVR	37	MUX	Select to Output	12
Register	Setup	4	CY7C910	CC to Output	22
		53 ns	CY7C245	Access Time	20
					66 ns

Minimum Clock Period = 66 ns

Figure 3. CY7C9101-Based 16-Bit System (Pipelined System, Add without Simultaneous Shift)



Data Loop			Control Loop		
CY7C245	Clock to Output	12	CY7C245	Clock to Output	12
CY7C9101	A, B to C_n + 16	35	MUX	Select to Output	12
CY7C9101	C_n to Worst Case	24	CY7C910	CC to Output	22
Register	Setup	4	CY7C245	Access Time	20
		75 ns			66 ns

Minimum Clock Period = 75 ns

Figure 4. CY7C9101-Based 32-Bit System (Pipelined System, Add without Simultaneous Shift)

throughput, reduced board space, and reduced power requirements are among the advantages of CY7C9101-based systems over CY7C901-based systems.

Minimum Cycle Time Calculations

Power is an important consideration in microcoded systems. For an equivalent system, the CY7C901 offers substantial savings in power over bipolar devices. Coupled with other low-power Cypress CMOS devices, the power savings over bipolar is clearly evident.

The functional integration of four CY7C901s with carry lookahead gives the CY7C9101 even greater advantages. The number of ALU elements is reduced by a factor of four, and there is a reduction in the carry logic needed. A comparison between bipolar, CY7C901-based, and CY7C9101-based systems appears in *Table 1*. Note that in this comparison the devices common to all 16- and 32-bit system configurations are included in the I_{CC} computations.

Cypress CMOS devices offer the fastest microcoded solutions, while keeping power consumption to reasonable levels. The CY7C901-based systems beat bipolar's fastest devices in a speed comparison, while consuming roughly one-third the power. Upgrading to the CY7C9101 results in even faster systems, at close to one-third the power of the CY7C901-based systems. This comparison is illustrated in *Table 2*.

Table 1. I_{CC} Calculations

I_{CC} Calculations for 16-Bit Systems (mA)			
	Cypress CMOS		
	CY7C901 Based	CY7C9101 Based	Bipolar
Sequencer	100	100	340
Registered PROM	90	90	185
Carry Logic	110	--	110
ALU Elments			
4x 4-Bit Slice	320		1060
16-Bit Slice		75	
Total	620	265	1695
I_{CC} Calculations for 32-Bit Systems (mA)			
Sequencer	100	100	340
Registered PROM	90	90	185
Carry Logic	330	110	330
ALU Elements			
8x 4-Bit Slice	640		2120
2x 16-Bit Slice		150	
Total	1160	450	2975

Table 2. Speed/Power Comparison of Bipolar, CY7C901, CY7C9101

	Minimum Clock Cycle (ns)			Maximum I_{CC} (mA)		
	Bipolar	CY7C901	CY7C9101	Bipolar	CY7C901	CY7C9101
16-Bit Systems	85	71	66	1695	620	265
32-Bit Systems	111	97	75	2975	1160	450

Systems with CMOS 16-bit Microprogrammed ALUs

This application note shows how to improve reliability, flexibility, and speed by diagramming timing for the CY7C9116 and CY7C9117 arithmetic and logic units (ALUs). Also highlighted are applications that benefit significantly from these devices' architecture and CMOS technology.

In the past, the dominant use of microprogrammed ALUs has been as general-purpose data processors in computers. Using microprogrammed machines in these applications improved performance because general-purpose microprocessors were too slow. In addition to allowing custom instruction sets, microprogrammed processors provided the only way to achieve the desired number of MIPS (millions of instructions per second).

With the advent of high-performance, 30-MIPS reduced instruction set computers (RISC), however, microprogrammed ALUs have relinquished their hold on general-purpose data-processor applications and found homes as custom processors or special-purpose controller s.

The CY7C9116/7

The CY7C9116 and CY7C9117 are extremely fast arithmetic and logic units implemented in a 1.2-micron, double-metal, CMOS process technology. As shown in *Figures 1* and *2*, the CY7C9117 differs from the CY7C9116 by incorporating separate buses for data input (D) and output (Y) and thus allows for the design

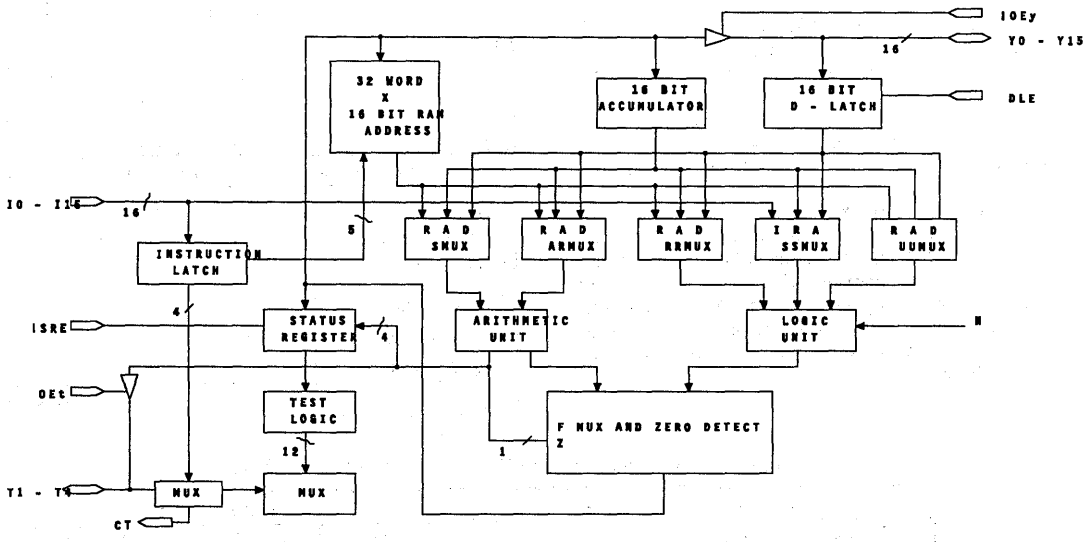


Figure 1. CY7C9116 Block Diagram

of faster microprogrammed systems. Otherwise the CY7C9116 and CY7C9117 are identical and will be described here as a single device. Both units are capable of 35-ns worst-case propagation delays from instruction in to data out.

The CY7C9116/7 contains a single-port, 32 x 16-bit-word register file; two operand arithmetic units; and three input logic units. Carry-look-ahead logic is also integrated with the logic and arithmetic units.

The CY7C9116/7's instructions can be divided into eleven types, as listed in *Table 1*. The on-chip barrel shifter attains single-clock operation on the extensive bit-manipulation and rotate instructions. In fact, all instructions in the ALU execute within one clock except for immediate instructions, where a second clock is needed to obtain the immediate operand.

The CY7C9116/7 is TTL compatible and fully interchangeable with its counterparts from Advanced Micro Devices and Texas Instruments. However, exercise caution when illegal instructions or undefined opcodes are used. Because the results are not predictable or guaranteed during these operations, they should not be used in any production system. *Table 2* shows an example of such a condition, when SOA is mistakenly encoded as an undefined operation.

Another feature of the Cypress CY7C9116/7 is that it allows the priority instruction to use both the source and destination as the accumulator. Be aware that older implementations of this architecture in bipolar technology do not allow such an operation. When you use older bipolar implementations or test devices, some machines might behave improperly, and undefined or illegal operations might produce different results for various device types, depending on vendor and technology.

Table 1. 7C9116/7 Instruction Types

Instruction Type	Example	
Single Operandinc:	src plus 1-> dest	
Two Operand add:	src plus src -> dest	
Single Bit Shift shup1:	src up 1	
Bit Oriented	setnr:	set RAM bit n
Rotate by n bits	rotrl:	rotate RAM n bits
Rotate & Merge	mdai:	rotate src and src' w/mask
Rotate & Compare	rotc:	rotate src cmp w/src' set cc
Prioritize	prtnr:	indicate highest priority bit
CRC crcf:	create crc fwd from qlink	
Status	rstst:	reset status register
No-Op	noop:	no effect

Faster Operation and Lower Power

Combining the CY7C9116/7's advanced microprogrammed architecture with Cypress's CMOS process technology provides many benefits. Specifically, custom computing units and controllers can operate at higher frequencies and consume less power — about

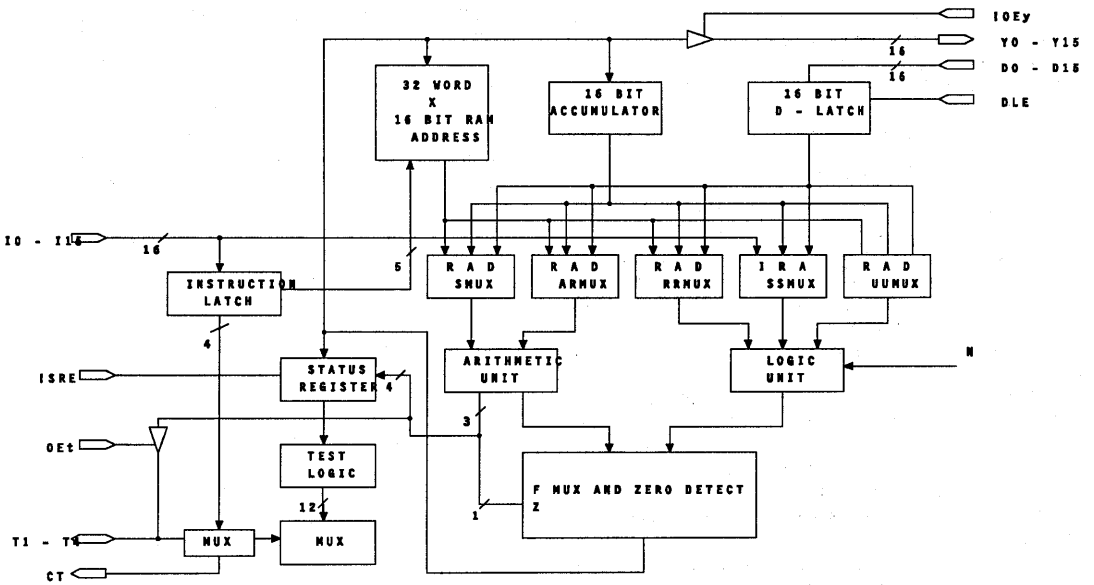


Figure 2. CY7C9117 Block Diagram

Table 2. Example Instruction Encoding Error

Vendor	SOA instruction: Instruction Code	ACC -> Y bus Result
	Correct encoding:	
All	1111 1000 1000 0000	0000 1110 0000 1101
	Coding Error	
AMD	1110 0110 1000 0000	1111 1111 1111 1110
Cypress	1110 0110 1000 0000	1111 0100 1000 1100
TI	1110 0110 1000 0000	0000 0000 0000 1001

Table 3. CMOS vs. Bipolar Performance and Power

	Cypress 7C9116/7	Generic 16-bit slice
Speed (ns)	35	53
Power (Icc, mA)		
Stactic	30	400
Max @10Mhz	150	600
Technology	CMOS	Bipolar

80% less — while offering higher reliability. *Table 3* compares the performance and power characteristics of a typical 16-bit microprogrammed ALU and the CY7C9116/7. The results show a significant power savings, which promote lower die temperatures and thus enhance the CY7C9116/7's reliability.

Other aspects of the CY7C9116/7's CMOS processing technology also contribute to increased system reliability. In the past, CMOS technologies experienced problems with destructive latch-up conditions. Cypress CMOS processes minimize this problem by employing guard rings and a substrate bias generator to achieve latch-up trigger currents in excess of 200 mA. Also contributing to reliability and performance are voltage supply tolerances of 10% and electrostatic discharge (ESD) protection circuitry, which allows the device to withstand voltages greater than 2001V.

System Timing

In microcoded systems, two loops determine system performance: the data and control loops. The control loop (*Figure 3*) is essentially the instruction stream for the CY7C9116/7. The current instruction combined with other status information generates a new address and instruction for the processor.

The data loop (*Figure 4*) moves information from an external source to a register; the CY7C9116/7 then uses the information to produce a result and status information for use by the external element. Because instructions and data are in separate domains, it should be apparent that this is a Harvard-style architecture. Thus, to achieve optimal performance, both the control and data loops should be as short as possible and equal in length.

Figure 5 shows an example of control loop timing for a typical CY7C9116/7 system. Four CY7C245A registered 2K x 8 PROMs implement the control store and current state register. The CY7C910 12-bit microsequencer allows for 4K words of addressing, i.e., instruction memory. In this example a 74F151 multiplexes status and condition-code information into the sequencer to complete the control loop. The components that make up this system are appropriate for embedded applications that have a fixed microcode control store.

You can improve system performance and flexibility by using Cypress static RAMs instead of PROMs, thus forming a writeable control store (WCS). (In this case, flexibility represents the ability to download or reprogram microcode at run time, which permits the system designer or user to load different applications or algorithms into the machine.) As diagrammed in *Figure 6*, four CY7C168 4K x 4 static RAMS can replace the ROMed microcode control store. However, you must add an external 74FCT374A register to replace the CY7C245A PROM's on-chip register. Thus, you pay a board space penalty for slightly improved performance and flexibility.

The data-loop timing for both the embedded and reprogrammable microcoded applications appears in *Figure 7*. Here, the CY7C9116/7 and its fast operation

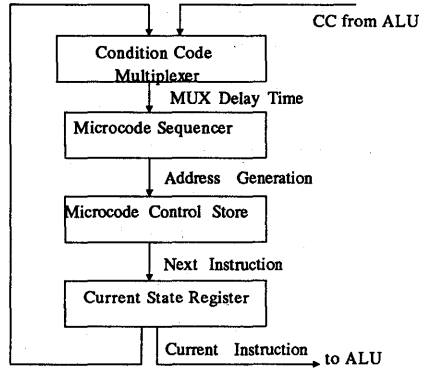


Figure 3. Microcoded System Control Loop

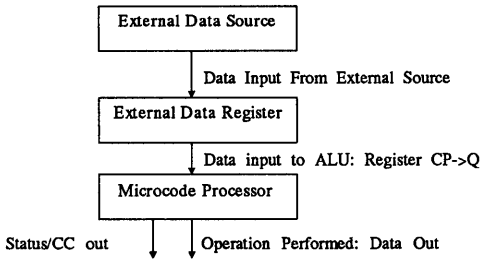


Figure 4. Microcoded System Data Loop

benefit the systems designer in two ways. First, because the data path is significantly faster than the control path, results are available early for the external data units, thereby allowing more time for external operations. Second, as faster memory technologies become available, you can design systems to operate at rates up to 25 MIPS.

Applications, Old and New

The applications for fast 16-bit microprogrammed CMOS ALUs fall into two categories. The first category resembles these devices' traditional use as a central processing unit for general-purpose computing. You might use a microprogrammed machine simply because instruction-set compatibility with previous machines is a design requirement. Here, the CY7C9116/7's speed and low power serve as powerful upgrades to existing hardware, with the possibility of lower cost from reduced power supply needs.

The more exciting applications for 16-bit microprogrammed ALUs are in loosely coupled

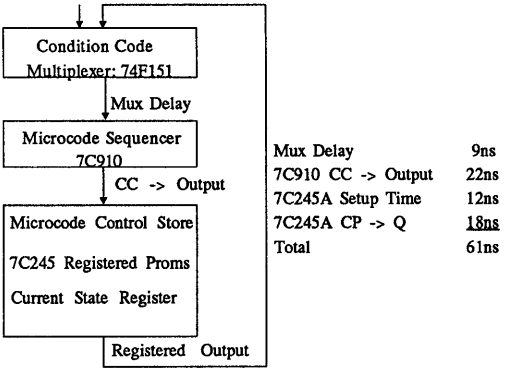


Figure 5. Embedded Application Control Loop Timing

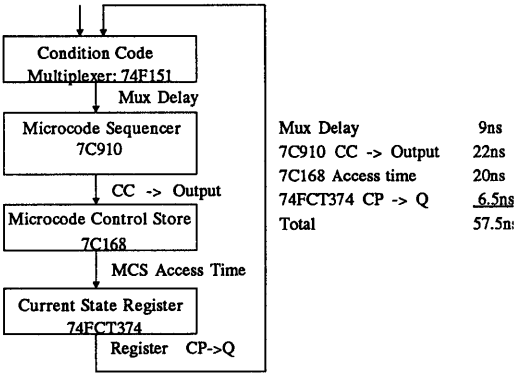


Figure 6. 7C9116/7 Reprogrammable Control Loop Timing

coprocessor or embedded controllers. Here, the CY7C9116/7's special bit, rotate, and CRC capabilities deliver significant performance advantages over "off-the-shelf" microprocessors. Graphics and imaging coprocessor s benefit from single-clock bit manipulation and rotation. The forward and reverse CRC instructions prove very helpful in communications and disk-controller applications, in terms of speed and code density. Graphics, communications, and disk controllers are just three examples that benefit from an application-specific instruction set, as provided by microprogrammed machines such as the CY7C9116/7.

There remain a myriad of custom control and embedded applications in military, industrial, and commercial systems that can exploit the performance and flexibility of the CY7C9116 and CY7C9117 CMOS 16-bit microprogrammed arithmetic and logic units.

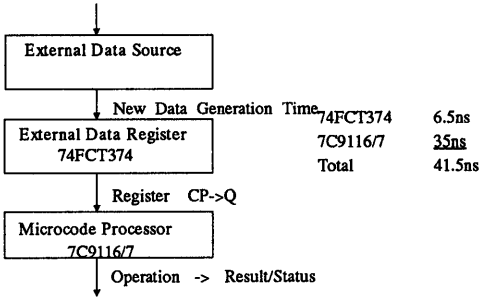


Figure 7. Microcoded System Data Loop Timing

Section Contents

CYPRESS SEMICONDUCTOR

SPARC Software Advantages over CISC

This application note explains the ways in which SPARC promotes more efficient software implementations of applications. Several attributes of the SPARC architecture make efficient high-level language (HLL) optimizing compilers possible. These attributes enable a compiler to map code from HLLs such as C, Fortran, and Pascal into SPARC native code without a significant loss in execution speed.

CISC Software Drawbacks

The efficiency of an optimizing compiler is critical. Before the development of RISC architecture, a compiler designer was faced with the near-impossible task of creating a compiler that mapped HLL code correctly into CISC native code and simultaneously generated an optimal CISC native code stream. There is no way of algorithmically resolving the twin objectives that compiled CISC native code be both correct and optimal — i.e., that the code does what the programmer defined in the HLL code and out of all the instruction streams possible, the one generated executes in the shortest time.

The following primary attributes of a CISC architecture cause this fundamental difficulty:

- A complex, non-orthogonal, overlapping instruction set
- Non-visible execution pipeline
- Destructive two-address architecture
- Mixed memory/register model of execution

A complex, non-orthogonal, overlapping instruction set allows you to substitute more than one native code instruction sequence for the same HLL instruction. For example, a typical CISC instruction set has more than one instruction for addressing memory, performing arithmetic instructions, testing and branching, etc. No one instruction in an instruction category is optimal for all situations, and there lies the problem.

A compiler only knows *how* to accomplish tasks, not why. The compiler does not understand what the programmer is trying to do with the HLL code. The compiler only knows how to parse the HLL code to produce native code for the target machine. Without knowing what the HLL program is trying to accomplish, the compiler cannot select the one optimal instruction out of several similar instructions that accomplish almost the same result. The compiler merely uses one that works in all situations. The generated code is optimal in terms of execution speed only by chance.

When the CPU's pipeline is not visible to the compiler, there is no way to schedule native code instructions to take advantage of unfilled pipeline slots. These pipeline "bubbles" exist in every computer architecture because of delays caused by the underlying system hardware. Because the compiler cannot schedule operations during these bubbles, the processor spends a significant portion of its time in an idle or No-op mode.

A destructive two-address architecture means that an instruction is of the form

A & B --> A

where A and B are registers, "&" is a logical or arithmetic operation, and "--" signifies that the results are moved to a destination (in this case, A). This instruction destroys the contents of A; hence the name, destructive two-address architecture. When A's contents are destroyed, the data that was stored in A cannot be used for further calculations.

This architecture imposes a stiff overhead penalty for an algorithm such as a recursive digital filter, in which the intermediate results of an input-value stream multiplied by some constant are reused to produce the final value. To overcome this limitation, the intermediate values must be saved somewhere, then constantly reloaded into the registers. A programmer might be able to save some of this overhead by loading multiple copies of the data into several registers and switching from one set of registers to the other. Although a programmer might have the craft and intelligence to do this, a compiler does not.

The mixed memory/register model of execution means that the instruction set allows the programmer to specify that values can be directly fetched from or stored

to memory. Because of the physical properties of electronic circuits, the data value does not appear in the CPU instantaneously. Some time is needed to assert the address lines, to let the read strobe reach a steady voltage level, etc. During this time, the processor is idle. Because of CISC's non-visible pipeline, another instruction cannot be scheduled to utilize the idle time. The pipeline bubble must be left unfilled, causing a decrease in processor efficiency.

RISC Software Advantages

In contrast, the following significant factors of a RISC machine make efficient optimizing compilers possible:

- A simplified, orthogonal instruction set
- Visible execution pipeline
- Load/store model of execution
- Non-destructive triadic address architecture

With a simplified, orthogonal instruction set, only a small set of native code instruction streams achieve the effect of an HLL instruction. This simplifies the compiler's task of selecting the correct native code stream to emulate an HLL instruction. Instead of spending effort to ensure that the native code does what the HLL program states, the compiler writer can concentrate on scheduling the generated native code so that it executes in the minimum amount of time.

SPARC's visible execution pipeline allows an optimizing compiler to see when idle periods occur. Using this knowledge, the compiler can re-schedule native code instructions to fill these empty slots in the pipeline.

In the load/store model of execution, data is first loaded from memory into registers or stored into registers before being sent to memory. Data load/stores are decoupled from ALU operations. This means that the ALU can be operating in parallel with the SPARC chip's load/store components, overlapping operations and increasing the processor's efficiency.

SPARC's non-destructive, triadic address architecture has instructions of the following form:

A & B --> C

Where A, B, and C are registers; "&" is a logical or arithmetic operation; and "--" signifies that the results are moved to a destination (in this case, C). The contents of the A and B registers are presereved during this operation — hence the name non-destructive. The compiler can reuse the data in both A and B in subsequent operations, saving the overhead of reloading intermediate data again and again.

In addition to allowing hardware speed to be increased by scaling the device geometry and/or retargeting to another semiconductor technology, the SPARC architecture allows the creation of efficient optimizing HLL compilers. This software advantage improves the productivity of application developers because they can write code in an HLL such as C and still achieve the performance they need.

The majority of applications for mainframes, minis, and PCs were first written in assembly code because that was the only way to attain the execution speed needed to run the application algorithm at a reasonable rate. Programmer productivity is measured in lines of debugged code per day. The number of lines produced is the same, whether they are lines of assembly or HLL code. Because one line of HLL code can be equivalent to ten or more lines of assembly code, the ability to write an application in C or another HLL can increase software productivity by a full order of magnitude.

CYPRESS SEMICONDUCTOR

Register Windows

This application note explains how the Cypress CY7C601 SPARC microprocessor uses register windows and shows how they decrease system execution time.

The CY7C601 is one of the few processors to use register windowing for context switching. When entering and returning from trap handlers and procedures, the CY7C601 thus enjoys a significant speed advantage over other processors with "flat" register files. Register windows are also the CY7C601's least understood architectural feature.

Register Windows

Most of today's microprocessors implement a register file as a contiguous piece of fast memory. If a processor has a flat register file, each register is addressed as an offset from the beginning of the register file. A register's effective address equals the register number times the register's size (usually 32 bits) plus the address base (0 for a flat register file).

The CY7C601's register windowing feature adds an entry to the processor state register (PSR) that provides the base address used to generate the effective register address. This entry in the PSR is called the Current Window Pointer (CWP). Changing the CWP by one offsets the register addressing by 16. Thus the effective register address is: (the CWP times 16 plus the register number) times the register size (32 bits). High-speed hardware ensures that the correct register can be selected and the data

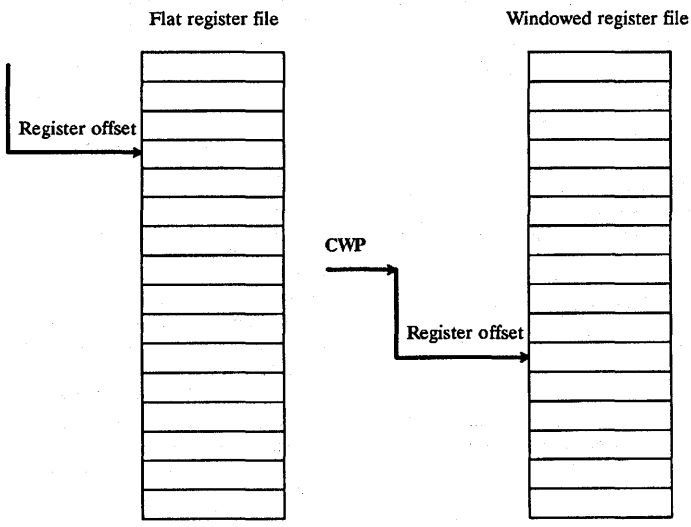


Figure 1. Addressing Mechanisms

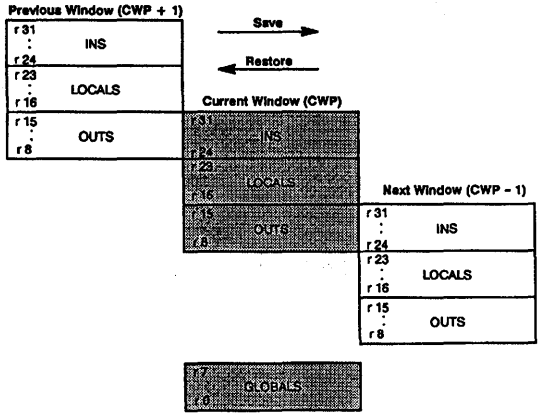


Figure 2. Overlapping Register Windows

loaded or extracted from the register in one clock. The diagram in *Figure 1* illustrates this register addressing mode.

The CWP allows you to partition the register file into separate sets, or windows. When a context switch is necessary after a trap is taken or a procedure called, the processor can save its old state information and get a new set of registers to use simply by incrementing the CWP. The CY7C601 performs this operation with one single-cycle instruction.

In most processors that use a flat register file, a process can get a new set of registers to use only after saving the current registers to memory to preserve state information. Depending on the number of registers to save and the clock time for a save instruction, the context switch can take quite a while.

Parts of the Register File

The CY7C601 employs four types of registers: outs, ins, locals, and globals (*Figure 2*). The ins registers contain values from the procedure that called the current procedure. The globals can be accessed by any procedure, no matter what the procedure's nesting level. The outs hold local information or pass information to a procedure that the current procedure calls. The locals are for the current procedure's exclusive use. The diagram on the next page gives a conceptual picture of what these overlapping register windows look like.

Registers are shared between procedures: the previous procedure's outs are the current procedure's ins. Parameters are passsed between procedures using the ins and outs. The ins contain the data and return address of the calling procedure.

Partitioning the register file into windows reduces the number of registers each process can use. This is why the CY7C601 has such a large register file. The CY7C601 has 136 registers for a maximum of eight windows — eight windows of 16 registers each plus eight global registers. Each procedure has 16 registers for its exclusive use — eight locals and eight outs. 32 registers can be addressed — eight locals, eight outs, eight globals, and eight ins (from the calling procedure).

97 percent of all procedures pass fewer than six parameters during a procedure call; the average is 2.1. Eight registers are more than sufficient for passing parameters between procedures. If more parameters need to be passed, one of the registers is used as a frame pointer, and the additional parameters are stored to memory. The eight outs can carry data and address from the current procedure to a procedure called by the current procedure or to data local to the procedure. As *Figure 2* shows, the CWP is decremented when a procedure is called and incremented when a procedure returns.

The Window Invalid Mask

The CWP is not the only unique CY7C601 hardware feature that supports register windows. The CY7C601 also includes a dedicated 32-bit register called the Window Invalid Mask (WIM). The WIM tells the CY7C601 how many windows it has and which ones are active. By comparing the WIM and the CWP, the CY7C601 can determine when it is attempting to utilize more windows than it has available. This would not cause a physical problem because the register file is implemented as a circular stack, but the data in the first window would be corrupted.

The processor's attempt to use more windows than it has causes a window overflow trap. During this trap, the processor saves the oldest window to memory. This is the only time when the CY7C601 microprocessor must save registers to memory during a context switch (unless more than eight parameters must be passed between procedures). On a window overflow, only 16 registers must be saved to memory (eight locals & eight outs).

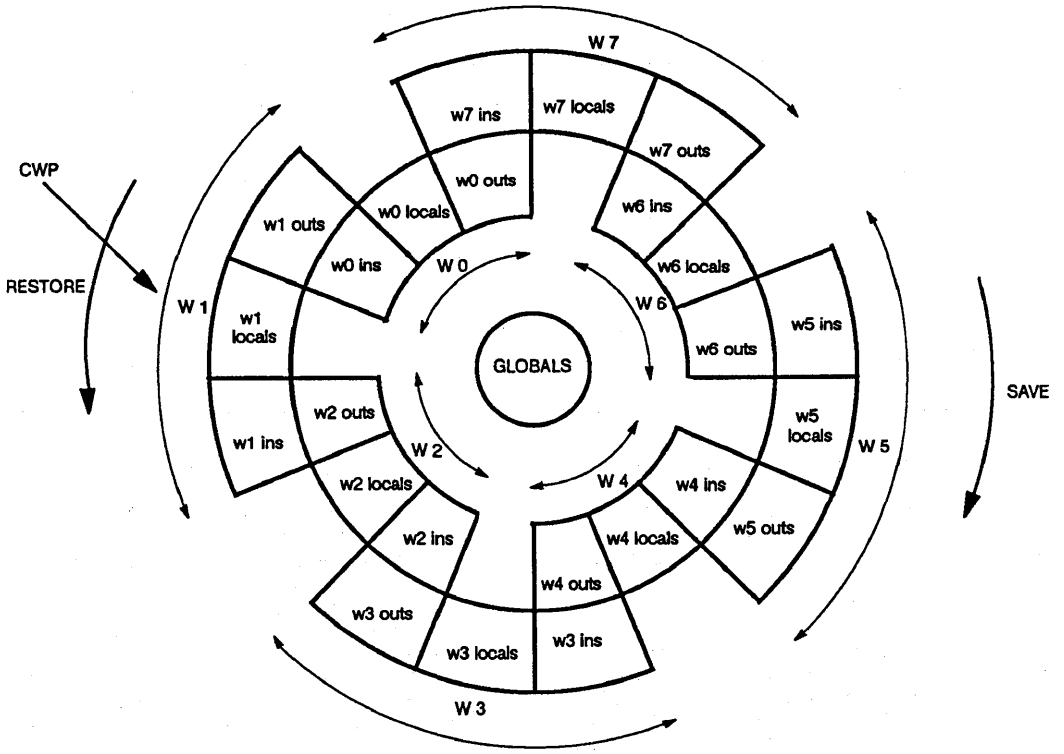


Figure 3. Register Window Concept with Eight Windows

An alternative way to conceptually view the CY7C601 register windows is to think of them as a ring of registers (*Figure 3*). As mentioned earlier, the CY7C601's register file is circular. If the CWP is pointing to window 7 and is incremented, the CWP now points to window 0. If the CWP points to window 0 and is decremented, the CWP points to window 7.

As an example, say that the CY7C601 makes a procedure call with the CWP pointing to window 1, as shown in *Figure 3*. Assume that the WIM has been set to reflect the fact that eight windows are physically implemented. Upon making the procedure call, the CY7C601 attempts a SAVE to provide the called procedure with a new set of registers. During a SAVE, the CWP is decremented by one (CWP = 0). The CY7C601 checks this value against the WIM. Because window 0 must be reserved for use by the trap handler, the CY7C601 has run out of windows for user procedures. Bit 0 of the WIM was set to reflect the fact that window 0 is reserved for system use. Upon checking the value of the CWP against the WIM, the CY7C601 detects a window overflow condition and causes a trap. During this window overflow trap, the CY7C601 increments the CWP to point back to window 1 and saves the calling procedure's registers (eight ins and eight locals) to a location in memory. Upon returning from the procedure, the registers are restored from memory and the values of the registers in window 1 are overwritten.

A Versatile Architecture

One of the advantages of register windowing is its versatility. The SPARC architecture has provisions for implementing up to 32 register windows, and the CY7C601 has eight. You can partition the available registers into different numbers of windows to increase the CY7C601's efficiency for specific applications. When you use a real-time operating system, for example, you can set the WIM to partition the register set into a small number of windows, say four. You can assign each real-time task to its own window. The total interrupt response time is now the interrupt latency (4 - 7 clocks) plus one clock to switch windows. Compare this response time to the response time for a CISC or RISC architecture with a flat register

Table 1. Register Windows vs a Flat Register File

Benchmark program	GCC	TeX
Percentage of CALL or RETURN instructions	1.8%	3.6%
Average registers stored per call	2.3	3.2
Loads (flat register file)	3,928,710	2,811,545
Loads SPARC (register windows)	3,313,317	2,736,979
Ratio loads windows/flat	0.84	0.97
Stores (flat register file)	2,037,226	1,974,078
Stores SPARC (register windows)	1,246,538	1,401,186
Ratio stores windows/flat	0.61	0.70

file, where saving register contents consumes most of the interrupt response time.

A register window architecture means that the CY7C601 must perform fewer load/stores. This is amply demonstrated by *Table 1*, which lists the loads and stores done by two microprocessor architectures: SPARC with eight register windows and another architecture with a flat register file.

The two benchmark programs used to obtain the data in the table are the Gnu C Compiler (GCC) and the text processing program TeX. Because of register windowing, the CY7C601 has to do up to 16 percent fewer loads and 39 percent fewer stores, compared to a microprocessor with a flat register file. Register windowing thus increases processing speed significantly.

Some of the load/store traffic generated by the use of a flat register file can be reduced by using interprocedural register allocation. This technique consolidates the use of registers to hold variables passed between procedures. By consolidating the number of registers used, less data needs to be saved to and restored from memory, reducing load/store traffic. This traffic reduction shrinks the edge that register windowing gives the CY7C601 over microprocessors with flat register files.

Interprocedural register allocation has a glaring weakness, however: It depends upon a complete knowledge of how many registers the called procedure uses and for what purpose. With an object-oriented language such as C++ or Smalltalk, this knowledge is not available at compile time. Interprocedural register allocation is therefore not possible when using an object-oriented language, and register windowing's performance edge comes to bear in full force.

The software world is shifting toward object-oriented languages such as C++ because of the need for increased productivity. Register windowing thus makes the CY7C601 the performance leader for today and promises to further soldify the CY7C601's lead in the future as the use of object-oriented languages increases.

CYPRESS SEMICONDUCTOR

CY7C600 System Design Footnotes

This application note covers several topics that have generated questions from SPARC systems designers. The intent here is to provide additional insight into the operation of the CY7C600 chip set through discussion of these short topics. Of course, a single paper cannot answer all questions regarding SPARC design. Please contact your local Cypress field applications engineer regarding any other questions you might have about SPARC.

Reset and Error Modes

The CY7C601/611 is reset by the assertion of the $\overline{\text{RESET}}$ signal for a minimum of eight clocks. The clock signal must be active for the CY7C601/611 to correctly synchronize upon receiving $\overline{\text{RESET}}$.

For systems using the CY7C604A/605A, the system reset signal is supplied to the CY7C604A/605A $\overline{\text{POR}}$ input for a minimum of eight clocks. Upon receiving the $\overline{\text{POR}}$ signal, the CY7C604A/605A asserts the $\overline{\text{IRST}}$ output, which drives the CY7C601 $\overline{\text{RESET}}$ input. $\overline{\text{IRST}}$ is released one clock after the $\overline{\text{POR}}$ input to the CY7C604A/605A is released.

The CY7C601/611 enters reset mode upon receiving the $\overline{\text{RESET}}$ signal at a rising clock edge. *Figure 1* illustrates CY7C601/611 reset timing. All processor operation halts. The CY7C601/611 asserts address 0x00000000, and the appropriate control signals for the first instruction access are asserted while $\overline{\text{RESET}}$ is asserted. The CY7C601/611 remains in reset mode until $\overline{\text{RESET}}$ is released, then the CY7C601/611 immediately enters execution mode. One clock after receiving the release of $\overline{\text{RESET}}$, the CY7C601/611 asserts the address for the next instruction access on the bus. On the clock after $\overline{\text{RESET}}$ is released, the CY7C601/611 latches the first instruction on the data bus. Note that the MAO and $\overline{\text{MHOLD}}$ signals must be de-asserted while $\overline{\text{RESET}}$ is asserted.

The CY7C601/611 initializes the enable traps (ET) and supervisor (S) bits of the processor state register (PSR), the program counter, and next program counter upon reset. All other registers in the CY7C601/611 are left unchanged. This feature provides easy error recovery in the case of an error-mode-generated reset (more on this later), because the registers are not changed after an error-causing condition.

Upon reset, the CY7C601/611 initializes the PSR's supervisor-mode bit to 1 (enabling supervisor mode) and sets the ET bit to 0 (traps disabled). The program counter (PC) and the next program counter (nPC) are initialized to 0 and 4, respectively. If the reset is a power-on (initial) reset, the state of all other registers are undefined. In addition, the state of all fields other than the PSR's ET and S bits are also undefined. A reset that occurs after the initial power-on reset (such as a reset to exit error mode) does not affect any registers other than the PSR, PC, and nPC.

Upon entering execution mode from a power-on reset, the software designer must ensure that the CY7C601/611 (and CY7C604A, if present) is properly initialized. Three registers in the CY7C601/611 *must* be initialized upon power-on reset: the processor state register (PSR), the trap base register (TBR), and the window invalid mask register (WIM).

One common mistake is to neglect to initialize the WIM register, which is undefined upon a power-on reset. If not initialized, this register can unexpectedly disable one or more windows. The processor state register does automatically initialize the register's S (supervisor) and ET (enable traps) bits upon reset, but all other fields must be initialized by software.

The TBR register must be initialized to point to the beginning of the trap vector table to handle traps. The register should be initialized before the PSR's ET bit is set. Note that three NOP instructions are generally inserted after writes to the PSR, WIM, and TBR registers to ensure that the CY7C601/611 correctly handles instructions immediately following these special register writes.

Error mode is a self-initiated halt mode that the CY7C601/611 enters upon encountering a synchronous trap when the PSR's ET bit is set to Zero (traps disabled). The processor also enters error mode if a return from trap

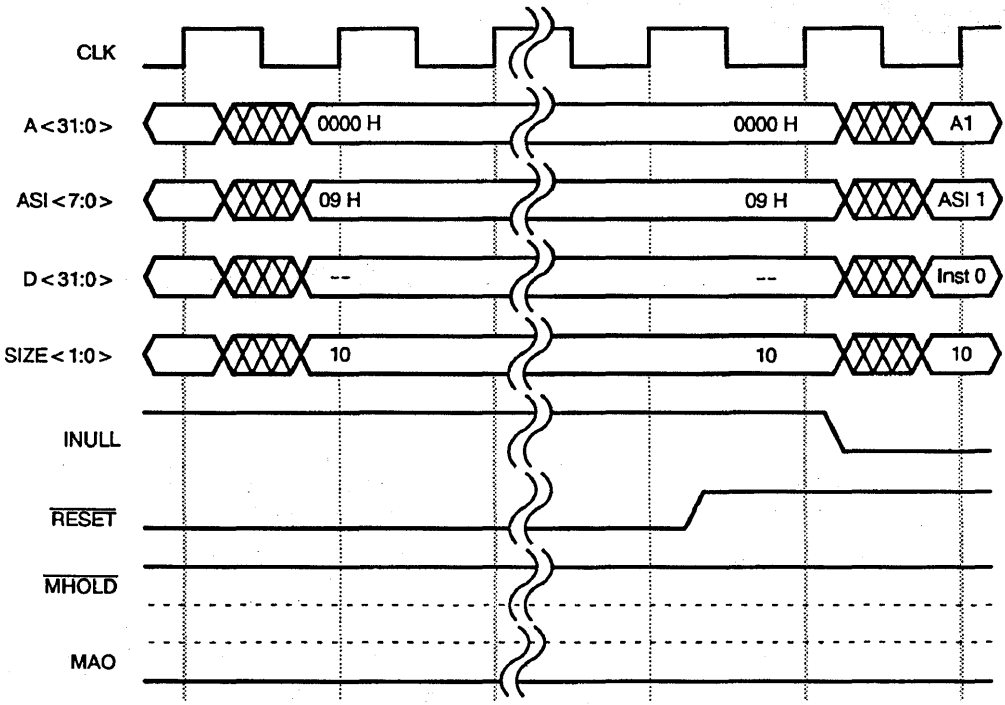


Figure 1. Power-On Reset Timing

(RETT) instruction is encountered with either the traps enabled (ET = 1) or the supervisor bit cleared. Upon encountering one of these conditions, the CY7C601/611 sets the TBR's tt (trap type) field to reflect the type of synchronous trap that caused the error state, after which the processor asserts the $\overline{\text{ERROR}}$ signal and halts (*Figure 2*). Error mode is exited when $\overline{\text{RESET}}$ is asserted.

A CY7C604A/605A responds to $\overline{\text{ERROR}}$ by executing a watchdog reset. During this reset, the CY7C604A/605A asserts the $\overline{\text{IRST}}$ output (used as the $\overline{\text{RESET}}$ input), sets the watchdog reset bit in the CY7C604A/605A reset register, and sets the boot-mode bit of the CY7C604A/605A system control register. All other registers in the CY7C604A/605A are left unchanged. Because the CY7C604A/605A enters boot mode, all instruction fetches made by the CY7C601 are fetched from physical memory on the Mbus, regardless of whether the cache is enabled. This action is appropriate because when the CY7C601 enters execution mode from reset, the processor executes the reset routine, which is generally stored in nonvolatile physical memory.

CY7C600 Pull-Up/Pull-Down Resistors

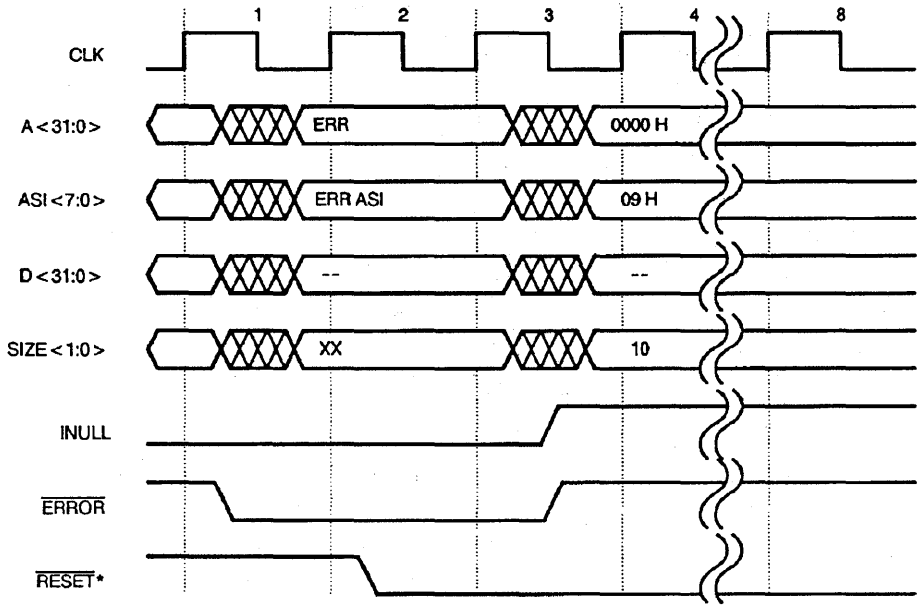
For proper operation, several signals for the CY7C600 chip set must be pulled either High or Low. This has often been overlooked in some SPARC designs, causing headaches for those debugging their hardware. *Table 1* lists the state to which you must tie CY7C600 signals for proper operation. The table assumes a CY7C604A/605A-based CPU.

Pay attention to the resistance value of passive pull ups and pull downs to ensure that they match the CY7C600 buffers' drive capabilities. CY7C600 buffers sink a minimum of 8 mA and source a minimum of -2 mA. 1 KW is a reasonable value for pull-up resistors on signals that must be driven by CY7C600 buffers. 10 KW is a reasonable pull-up or pull-down value on input signals such as $\overline{\text{BHOLD}}$, $\overline{\text{CP}}$, or CCCV.

Signal Termination

Design of high-speed CMOS systems requires close attention to board layout, PCB trace propagation delays, crosstalk, noise, clock skew, and signal terminations. This is extremely important to systems operating at 33 to 40 MHz and beyond. (Consult the application note, "High-speed CMOS SPARC System Design.") Cypress recommends close attention to clock distribution and termination for any high-speed CMOS design.

Termination is highly recommended on the clock signals for a CY7C600 system. Due to the minimum slew-rate requirement of 0.8 V/ns for CY7C600 parts, you must pay close attention to the clock drivers' drive and slew-



*RESET must be asserted for a minimum of 8 clocks

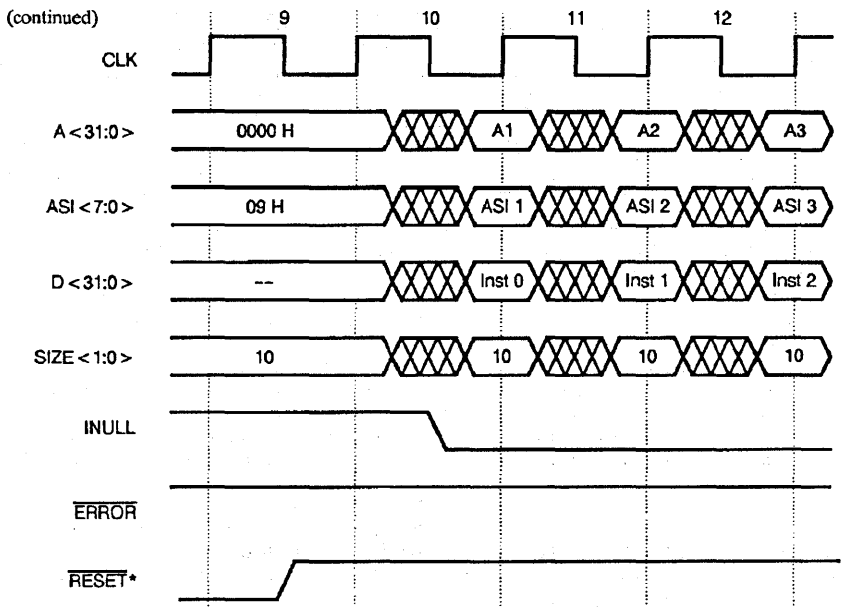


Figure 2. Error/Reset Timing

Table 1. CY7C600 Signal Pull Ups and Pull Downs

Signal	Part Affected	Pulled	Comments
$\overline{MBB}$	CY7C604/605 input/output	High	CY7C604/605 cannot acquire Mbus if not pulled up
$\overline{MRDY}$	CY7C604/605 input	High	
$\overline{MRTY}$	CY7C604/605 input	High	
$\overline{MAS}$	CY7C604/605 output	High	
$\overline{MERR}$	CY7C604/605 input	High	
$\overline{CMER}$	CY7C604/605 output	High	
$\overline{TOE}$	CY7C601, CY7C604/605 input	Low	Must be pulled Low for system to operate
$\overline{SNULL}$	CY7C604/605 input	High	Assuming a single-CY7C604/605 system
$\overline{MDS}$	CY7C601 input	High	CY7C604/605 allows this signal to three-state
$\overline{MEXC}$	CY7C601 input	High	CY7C604/605 allows this signal to three-state
$\overline{BHOLD}$	CY7C601, CY7C602 inputs	High	
$\overline{MHOLDB}$	CY7C601, CY7C602 inputs	High	
$\overline{CHOLD}$	CY7C601, CY7C602 inputs	High	Required if coprocessor is removable or not present
$\overline{CP}$	CY7C601 input	High	Required if coprocessor is removable or not present
CCC[1:0]	CY7C601 input	High	Required if coprocessor is removable or not present
CCCV	CY7C601 input	High	Required if coprocessor is removable or not present
MAO	CY7C601 input	Low	
FPSYN	CY7C601 input	Low	
$\overline{IFT}$	CY7C601 input	Low	
FNULL	CY7C604/605 input	Low	Required if CY7C602 is removable or not present
$\overline{FP}$	CY7C601 input	High	Required if CY7C602 is removable or not present
FCC[1:0]	CY7C601 input	High	Required if CY7C602 is removable or not present
FCCV	CY7C601 input	High	Required if CY7C602 is removable or not present
$\overline{FHOLD}$	CY7C601 input	High	Required if CY7C602 is removable or not present
$\overline{FEXC}$	CY7C601 input	High	Required if CY7C602 is removable or not present

rate capabilities. For many designs, you cannot use a simple parallel resistor termination because the buffer drive required to attain the minimum clock slew rate often exceeds the drive capabilities of available CMOS or TTL buffers.

One recommended method of clock signal termination is to use one or more diodes to clamp the clock signal voltages to within a single diode voltage drop of ground or V_{CC} (*Figure 3*). Unlike parallel resistor termination, diode termination does not require a high-drive clock buffer. And unlike AC termination, diode termination does not degrade the clock signal's slew rate.

INULL

The CY7C601/611 generates the INULL signal to indicate that the processor will ignore the current memory access. INULL is asserted before the rising clock edge on which the nullified memory access would have been latched (*Figure 4*). This event occurs when an instruction fetch is nullified in the pipeline, but no other valid address is yet available to assert on the address bus.

For cached systems, INULL prevents a cache miss on a nullified access. INULL is also used by the exception logic to prevent an exception that might be generated by a nullified access.

INULL is asserted when an address is generated in an interlock case, such as a load that produces a hardware interlock. INULL is also generated when a trap or interrupt is encountered. INULL is asserted in this case to nullify the address generated before the trapped instruction enters the pipeline's execute stage.

INULL is asserted:

- During the second address cycle of any store instruction (including atomic load/stores)
- For the third instruction fetch after a trapped instruction
- To nullify the error-causing address after a reset
- On a load that causes a hardware interlock
- On the execution of JMPL and RETT instructions

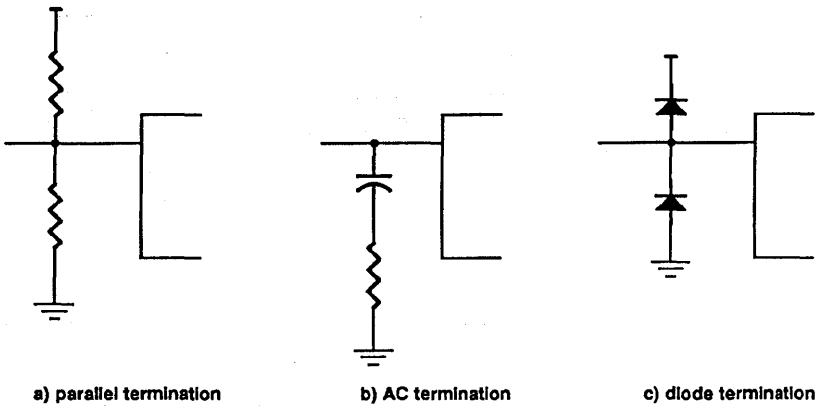


Figure 3. Signal Termination Examples

$\overline{\text{MHOLD}}$, $\overline{\text{MDS}}$, and $\overline{\text{MEXC}}$

The CY7C601/611 signals $\overline{\text{MHOLD}}$, $\overline{\text{MDS}}$, and $\overline{\text{MEXC}}$ are used by outside control logic to control CY7C601/611 memory accesses. Typically, this control logic takes the form of a cache controller or exception-generation controller.

$\overline{\text{MHOLD}}$ freezes the CY7C601/611's pipeline. External logic uses this signal to freeze the CY7C601/611's operation so that the supporting memory and exception logic can provide a response in synchronization with the CY7C601/611's pipeline. The CY7C601/611 samples $\overline{\text{MHOLD}}$ on the falling clock edge. Asserting $\overline{\text{MHOLD}}$ causes the CY7C601/611 to hold its outputs at the state that would be valid at the next rising clock edge. Note that this state is driven from the rising clock edge before $\overline{\text{MHOLD}}$ is asserted. From the perspective of the CY7C601/611, the pipeline is frozen before the rising clock edge following $\overline{\text{MHOLD}}$ assertion.

$\overline{\text{MDS}}$ is used during the assertion of $\overline{\text{MHOLD}}$ to cause the CY7C601/611 to latch the data present on the data bus. $\overline{\text{MDS}}$ also causes the CY7C601/611 to latch the state of the $\overline{\text{MEXC}}$ signal (described later) but does not cause the pipeline to advance. As *Figure 5* shows, $\overline{\text{MDS}}$ is sampled on the falling edge of the clock, and the information valid on the data bus at the next rising clock edge is latched into the CY7C601/611.

Because the pipeline does not advance until $\overline{\text{MHOLD}}$ is released, $\overline{\text{MDS}}$ can be asserted for more than one clock, although this is not necessary. The only qualification is that data must be valid for the rising clock edge after the last assertion of $\overline{\text{MDS}}$. The information on the data bus at this time is used by the CY7C601/611 when $\overline{\text{MHOLD}}$ is released.

$\overline{\text{MEXC}}$ indicates a memory exception to the CY7C601/611. Upon detecting an exception case, the exception control logic asserts $\overline{\text{MHOLD}}$ to halt the CY7C601/611 pipeline. $\overline{\text{MEXC}}$ and $\overline{\text{MDS}}$ are then asserted to signal the exception. $\overline{\text{MDS}}$ must be asserted with $\overline{\text{MEXC}}$ to cause the CY7C601/611 to latch the value of $\overline{\text{MEXC}}$ while $\overline{\text{MHOLD}}$ is asserted. Otherwise, the CY7C601/611 ignores the $\overline{\text{MEXC}}$ signal while $\overline{\text{MHOLD}}$ is asserted.

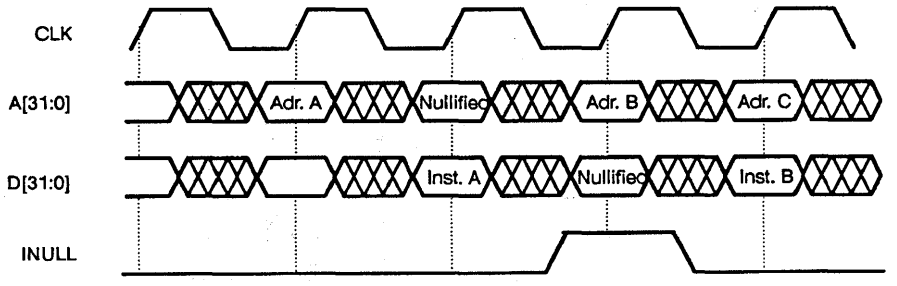


Figure 4. INULL Assertion

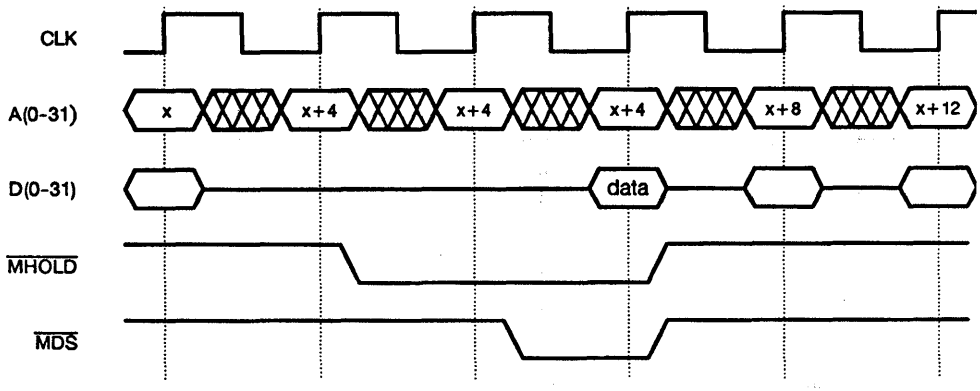


Figure 5. Wait-State Generation using $\overline{\text{MHOLD}}$ with $\overline{\text{MDS}}$

Wait-State Generation

Memory wait states can be generated using $\overline{\text{MHOLD}}$ or by stretching the CY7C601/611 clock. Because the CY7C601/611 is a fully static processor design, clock stretching is a simple method for generating memory wait states (*Figure 6*).

Another method is to use $\overline{\text{MHOLD}}$ to freeze the CY7C601/611 pipeline. You can do this in two ways. One way is to use $\overline{\text{MHOLD}}$ in the same manner as intended for a cache miss (*Figure 7*). $\overline{\text{MHOLD}}$ is asserted by the wait-state logic after the rising clock edge on which the CY7C601/611 would have latched the memory access. When the memory has responded to the access, the wait-state logic strobes $\overline{\text{MDS}}$ to make the CY7C601/611 latch the information, then releases $\overline{\text{MHOLD}}$.

You can also use $\overline{\text{MHOLD}}$ to halt the pipeline before the CY7C601/611 has missed the memory access. $\overline{\text{MHOLD}}$ must be asserted immediately after the rising clock edge of a memory access. This method requires fast logic even at 25 MHz and is probably not feasible for higher frequencies. The assertion of $\overline{\text{MHOLD}}$ must make the set-up time before the falling clock edge (*Figure 7*). With $\overline{\text{MHOLD}}$ asserted, the memory system can catch up with the CY7C601/611 and assert the data on the bus. The wait-state logic then releases $\overline{\text{MHOLD}}$, allowing the CY7C601/611 to latch the data.

Interrupts

Interrupts are signaled to the CY7C601/611 by asserting the interrupt request level inputs, IRL[3:0]. For the interrupt to be taken by the CY7C601/611, the value asserted on the IRL[3:0] signals must exceed the value stored in the processor interrupt level (PIL) field of the processor status register (PSR). An IRL level of 0 indicates no interrupt; a level of 15 indicates a non-maskable

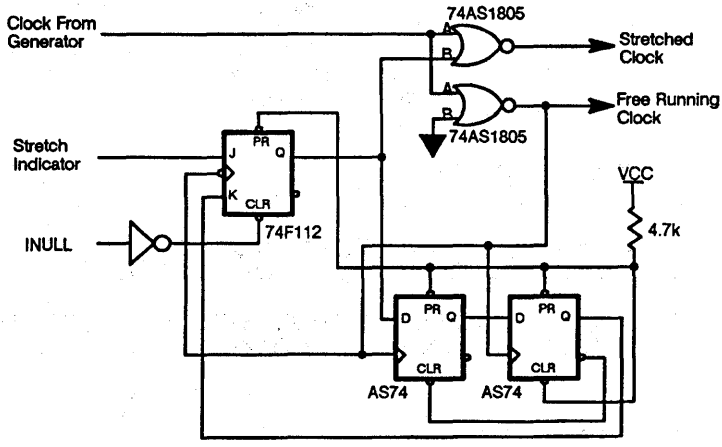


Figure 6. Simple Clock-Stretching Circuit

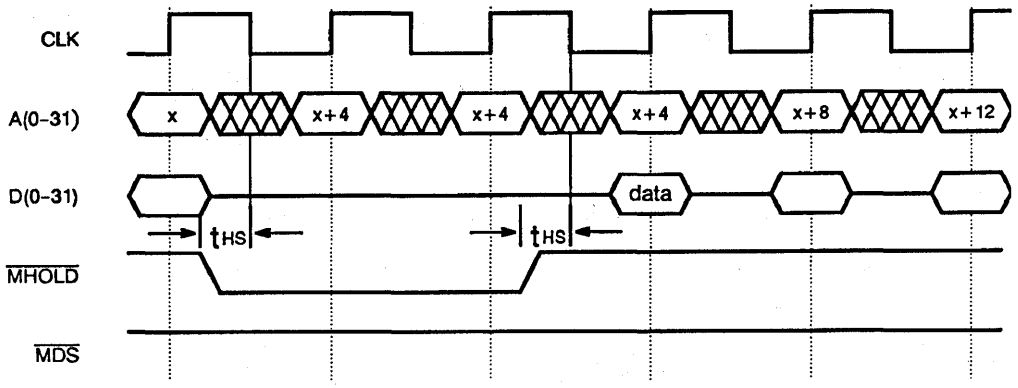


Figure 7. Wait-State Generation using $\overline{\text{MHOLD}}$ without $\overline{\text{MDS}}$

interrupt. In addition, the PSR's enable traps (ET) bit must also be set for the CY7C601/611 to respond to an interrupt input.

The IRL[3:0] inputs are sampled, then latched before the interrupt is allowed to be prioritized and taken by the CY7C601/611. This requires an IRL[3:0] value to be asserted two clocks before the CY7C601/611 accepts the interrupt input, thus helping to prevent extraneous interrupts. The CY7C601/611 uses interrupt acknowledge (INTACK) to acknowledge that an interrupt has been taken. The CY7C601/611 asserts INTACK after the rising clock edge upon which the address of the first trap instruction is asserted. Interrupts that are not taken, such as those masked by the PIL, are not acknowledged.

The prioritization stage of interrupt processing compares the interrupt's trap priority level against that of any other synchronous trap that might be occurring simultaneously. All other trap types take priority over interrupt traps, and in the case of contention, the other trap is serviced instead of the interrupt. In this case, INTACK is not asserted until the CY7C601/611 has returned from the trap handler and the interrupt level can again be sampled, latched, and prioritized.

The CY7C601/611 features extremely fast response time for interrupt inputs. Interrupt latency (interrupt reception to first trap address assertion) is from four to seven clocks. The three-clock variation in interrupt latency is due to the effect of multiple cycle instructions upon CY7C601/611 execution. Interrupt latencies greater than four cycles occur when a multiple-cycle instruction is fetched immediately before an instruction is interrupted. The worst-case, seven-clock interrupt latency occurs when a three-cycle instruction is fetched immediately before the interrupt.

Figure 8 shows the assertion of INTACK with respect to IRL[3:0] and the first interrupt address (T0). The CY7C601/611 generates T0 when the interrupted instruction reaches the pipeline's execute stage. The CY7C601/611 asserts INTACK during the clock cycle when the interrupted instruction reaches the pipeline's write stage. The memory system sees INTACK asserted on the rising clock edge on which the first trap instruction returns from memory.

Delays in interrupt acknowledgment are due to multicycle instructions in the pipeline that were fetched before the interrupted instruction. Therefore, if a multi-cycle instruction is in the pipeline's decode stage when the interrupt is sampled, that instruction's pipeline delays (designated by internal ops, or IOPs) affect the timing of the interrupt acknowledge. *Figure 9* shows these events; instruction 2 is the interrupted instruction. Interrupts are executed before the interrupted instruction, and thus instruction 2 is annulled (not executed).

If the CY7C604A asserts $\overline{\text{MHOLD}}$, the control signal outputs from the CY7C601 are frozen, and INTACK is asserted as long as $\overline{\text{MHOLD}}$ is asserted. $\overline{\text{MHOLD}}$ causes the pipeline for the CY7C601 to freeze, and all bus signals asserted by the CY7C601 during this freeze remain asserted on the bus.

An example of such a case appears in *Figure 10*, in which $\overline{\text{MHOLD}}$ is asserted due to the fetch of an interrupt instruction that has been declared non-cacheable in the MMU. This is a common case, as interrupt handlers are generally part of the kernel or monitor code. Declaring the interrupt routine to be in a non-cacheable segment of memory forces the CY7C604A to fetch the interrupt instruction from main memory. Thus, the CY7C604A must assert $\overline{\text{MHOLD}}$ until this instruction is fetched. The assertion of $\overline{\text{MHOLD}}$ causes the INTACK signal to remain asserted until $\overline{\text{MHOLD}}$ is released. Note that this case is likely to repeat, as the subsequent interrupt instructions probably reside in the same memory segment.

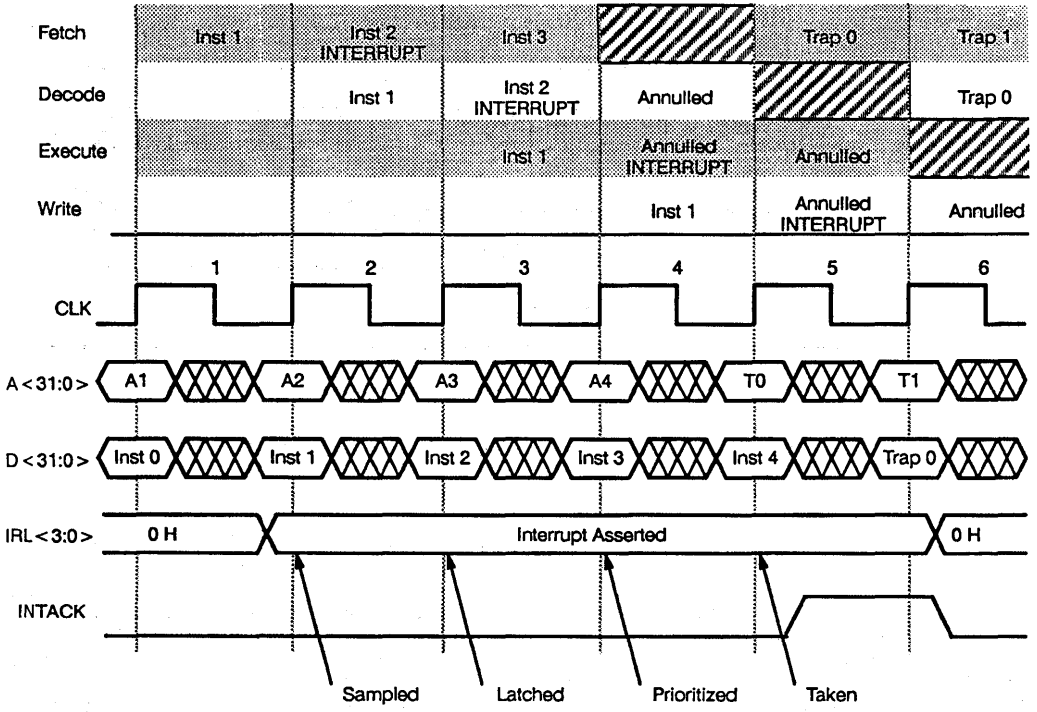


Figure 8. Best-Case Interrupt Latency

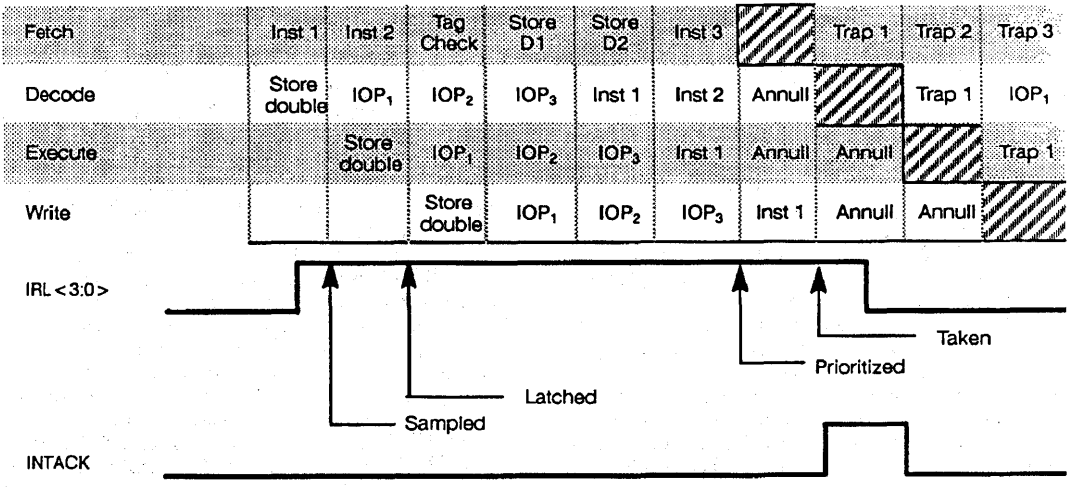


Figure 9. Worst-Case Interrupt Latency

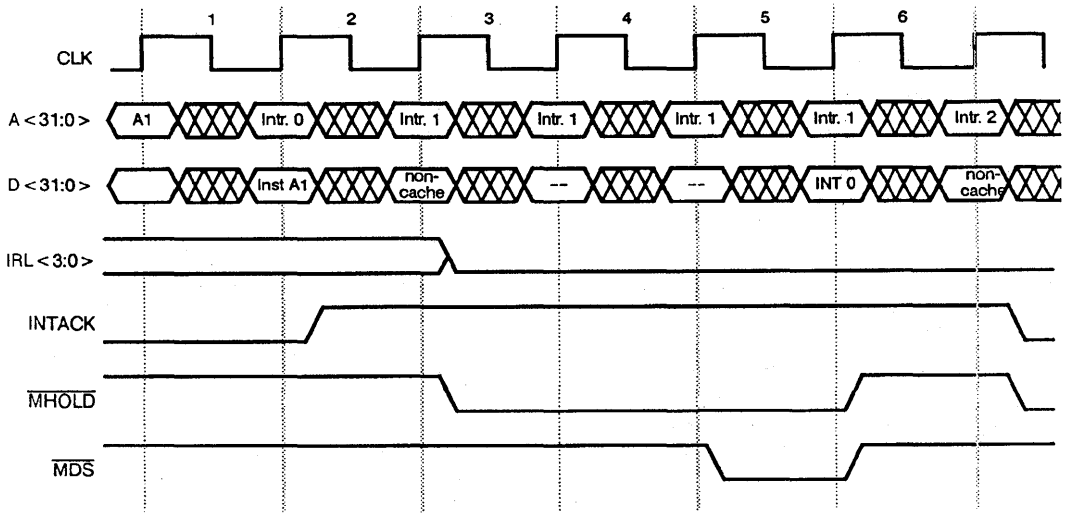


Figure 10. INTACK Frozen Due to $\overline{\text{MHOLD}}$

Nested Interrupts

Upon taking a non-reset trap, the CY7C601/611 executes the following operations:

- Sets the PSR's ET bit to Zero (traps disabled)
- Copies the PSR's S bit into the PS (previous supervisor) bit, then sets the S bit to One
- Decrements the CWP (current window pointer) by one (next window)
- Saves the PC and nPC into r[17] and r[18], respectively, of the trap window
- Sets the tt field of the TBR (trap base register) to the appropriate value (according to IRL[3:0])
- Writes the PC with the contents of the TBR, and writes the nPC with the value of the TBR + 4

Note that upon entering a trap, the CY7C601/611 immediately disables all other traps. Some systems require that the processor be able to respond to higher-priority interrupts or other traps while executing an interrupt handler. This capability is referred to as nested interrupts.

If the CY7C601/611 must support nested interrupts or traps, the software designer must re-enable traps after taking precautions to protect the previous state of the machine. Most software designers using SPARC systems use a stack to save windows and the state of the processor.

Note that for SPARC (and most RISC processors in general), the hardware does not implement the stack pointer and the process of saving the processor state upon entering a trap, which leaves the task to the software designer. This task includes saving the PSR, because the CY7C601/611 does not save the PSR upon entering a trap; instead, the CY7C601/611 saves the previous supervisor state in the PS bit. The CY7C601/611 does automatically save the return address for the trap in the trap-window registers r[17] and r[18].

Note that for nested interrupts, the PSR's PIL field must be updated to equal the current interrupt level *before* re-enabling traps. This protects the CY7C601/611 from further interruptions by the same level of interrupt.

In addition to saving the processor state, the software designer must determine how to handle potential window overflows, which can be caused by nested-trap handlers. This problem can occur because the CY7C601/611 does not check the WIM register for window overflow when the processor enters the next window to process a trap.

This aspect of the SPARC architecture is necessary to save at least one window for trap handlers. For instance, the CY7C601/611 checks the WIM register to detect potential window overflow when a SAVE instruction is executed. Upon detecting that a window save would push the processor state into a "WIMmed" window, the CY7C601/611 enters a window overflow trap. To process this trap without overwriting the current window registers, the CY7C601/611 jumps into the WIMmed window, ignoring the WIM register. Because the WIM register does not affect trap entry, the register must save a window for trap handlers. The register also prevents procedure calls from overwriting valid windows.

The use of nested interrupts adds another level of complexity to window management. If the entire set of non-WIMmed windows has been used, an interrupt jumps into the last (WIMmed) register window. If traps are enabled again without any other corrective actions, the next trap (or interrupt) overwrites the next window upon entering the trap. To prevent this problem, the software

Table 2. CY7C604A/605A Mbus Signals

Mbus Name	Signal Description	CY7C604/605 Name
AERR*	Asynchronous Error output	$\overline{\text{CMER}}$
RSTOUT*	Module reset output signal	$\overline{\text{MRST}}$
RSTIN*	Module reset input signal	$\overline{\text{POR}}$

designer must ensure that at least one additional window beyond the current trap window is available before re-enabling traps within a trap handler.

CY7C604A/605A Notes

Three CY7C604A/605A signals differ from the corresponding signal name used in the Mbus specification. *Table 2* lists these CY7C604A/605A Mbus signals and their corresponding Mbus names.

If you implement an Mbus arbiter, note that under certain conditions the CY7C604A/605A holds $\overline{\text{MBB}}$ active for multiple Mbus transactions. Those conditions are:

- When the CY7C604A is holding the bus during a table walk and has not received a relinquish-and-retry response
- When the CY7C604A is holding the bus for a retried write
- When the CY7C604A is holding the bus for a retried read
- When the CY7C604A is holding the bus for an atomic load/store that was not relinquished and retried
- When the CY7C604A is holding the bus to complete a burst access (normal operation)
- When the CY7C604A had the bus for the last transaction that was not relinquished and retried, and currently has a grant, and has an access pending

Accesses are considered to be pending for the CY7C604A only when one or more write accesses are queued in the write buffer. This can take the form of either multiple write accesses queued in the write buffer, or of one or more write accesses in the write buffer forcing a pending read access. In the latter case, the read access must remain pending until the write buffer is cleared. Read transactions must be delayed until the write buffer is cleared. This ensures data consistency in case one of the writes is to the same address as the read transaction.

CYPRESS SEMICONDUCTOR

The Impact of Memory Design on High-Performance RISC Microprocessors

Memory design has always been a crucial factor in the race for high-performance processing. Now the stakes are higher than ever before with the advent of RISC microprocessors, which require a memory access during every clock cycle and speeds exceeding 40 MHz.

To feed these high-performance engines, you are faced with building a CMOS or TTL-based memory system that must sustain a bandwidth on the order of 160 Mbytes per second (assuming 32-bit accesses, one access per clock, and 40 MHz). Because the processor can only run as fast as the memory system, a high-performance memory system is a crucial part of any RISC design.

Ideally, a memory system should be big, fast, and cheap. Unfortunately, these goals are often at odds with one another. A simple high-speed SRAM memory system large enough and fast enough to fulfill the RISC processor's needs would be ideal — if SRAMs were not expensive and power hungry and did not need much more board area than DRAMs. The latter cost much less and provide better memory density but also run several times slower and require a more complex addressing and control interface. Using only DRAM for a memory system implies multiple wait states for each memory access. This is disastrous to the performance of a high-speed RISC processing engine. The typical solution to these conflicting requirements of speed and density versus cost is to use a high-speed SRAM cache memory system backed by a DRAM main memory system.

Cache systems are a well-recognized and commonly used solution for high-speed processing systems. Cache memory systems were proposed early in the 1960s and have been used extensively in mainframes and minicomputers since the mid to late 1960s. Cache memory has become increasingly interesting to the designers of small computer systems as microprocessor speed and memory-bandwidth requirements have increased. Consequently, RISC processors make extensive use of cache systems to meet their memory-bandwidth needs.

Cache systems are not the only variable in the memory system performance equation, however. The memory system picture includes two factors: cache performance, which is often measured in terms of cache hit ratio, and cache miss penalty, which is a function of both the cache controller and the main memory system. The average memory access time gives a good perspective on the total memory solution:

$$t_{avg} = t_{ch}(c_{hr}) + t_{cm}(1 - c_{hr}) \qquad Eq.\ 1$$

where t_{avg} = average memory system access time
t_{ch} = cache-hit memory access time
t_{cm} = cache-miss memory access time
c_{hr} = cache hit ratio
$1\text{-}c_{hr}$ = cache miss ratio

For most cache systems, the cache-hit memory access time is one clock, which represents a zero-wait-state memory for RISC processors. A useful approximation for estimating performance for systems using RISC processors (assuming one clock per memory access), is that performance equals the product of the average memory system access time and the processor's average number of clocks per instruction (CPI). This product yields an adjusted system clocks per instruction value that is useful in estimating system performance:

$$CPI_{system} = CPI_{processor} \times t_{avg} \qquad Eq.\ 2$$

where CPI_{system} is the adjusted CPI for system performance

This rule-of-thumb equation illustrates the importance of memory system performance. As with any processor, only a memory system providing zero wait-state accesses permits a RISC processor to achieve maximum performance. Because a RISC system requires a memory access for every clock cycle, an average memory access time of

two clocks cuts the maximum attainable system performance in half!

As can be seen in Eq. 1, the cache hit ratio and the cache-miss memory access time are the two parameters that can be manipulated to achieve the maximum system performance within your constraints. Cache hit ratio is the ratio of cache hits to the number of total cache requests and is largely a function of the cache design. The cache-miss memory access time is the combination of the latency imposed by the cache controller as it fetches the missed cache line and the latency caused by the main memory system. Cache-miss memory access time is not directly a sum of these two latencies, though, because the cache line fetch timing and main memory timing overlap.

Cache hit ratio is important to memory system performance. Whether you are using a custom-designed cache controller or an off-the-shelf product, it is necessary to understand the factors that contribute to cache hit ratio. This understanding allows you to make a rough estimation of cache performance, which in turn helps you define the required main-memory-system performance required to meet the desired system performance. Along with processor performance, the cache and supporting memory system determine the achievable level of system performance.

Cache system performance

Cache performance and its contributing factors has been a topic of intense study in computer architecture circles. This application note is not intended to provide a detailed analysis of cache performance and design. However, a short discussion of cache performance serves as background for a discussion of memory system performance.

Cache hit ratio is the primary metric of cache performance and is strongly influenced by cache size. The larger a cache is, the more likely it is to hold the required datum. However, the tradeoff to cache size is cost. The reasons for avoiding a large SRAM memory system are system cost and memory density. If several megabytes of SRAM are an affordable option, why build a cache in the first place? The purpose of a cache is to provide enough high-speed memory to effectively increase processor performance, yet still stay within the system budget. Therefore the next question is: How big is big enough?

Unfortunately, the question of cache size is not easily answered. Caches often cannot be made arbitrarily big, but they need to be large enough so that their benefit offsets their cost. Assuming a system budget of some type (cost, power consumption, size, or a combination of these), a good approach to designing a cache is to choose a target cache hit ratio based on the system performance requirements. This target cache hit ratio is driven by system performance (as described by equations 1 and 2) and the system design constraints. If the system budget for the cache does not allow the cache hit ratio required to meet the system performance requirements, the supporting memory system features can be optimized to offset the cache's performance.

Size and Set Associativity

Set associativity also contributes to cache performance. Set associativity describes the number of memory locations to which a single address can be mapped. In other words, a cache with N-way set associativity can map any address to N number of cache locations.

A fully N-way-associative cache large enough to yield a high cache hit ratio is, in practice, extremely difficult to implement for a useful clock speed. Therefore, cache designs generally use four-way, two-way, and one-way (direct mapped) set-associative caching.

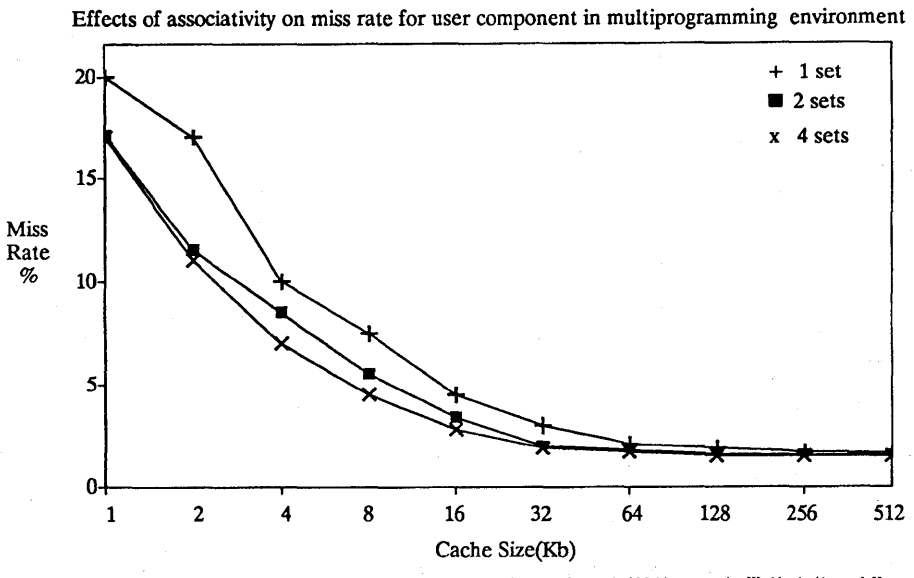
For smaller cache sizes, the greater the set associativity, the greater the cache hit ratio. However, studies have demonstrated that the benefits of set associativity decrease as cache size increases. *Figure 1* illustrates the cache hit ratios for 1-, 2-, and 4-way set-associative caches as a function of cache size. Note that as cache size increases, the cache hit ratio curves for the cache systems converge.

Multiple set associativity carries a penalty for cache system design. The greater the level of set associativity, the greater the number of cache tags that must be compared to determine a cache hit. This requirement directly affects the maximum clock speed at which a cache controller can operate. As *Figure 1* shows, a 2- or 4-way set-associative cache offers little performance advantage over the direct-mapped cache at cache sizes of 64 Kbytes and larger. Assuming that the cache can provide a sufficiently large memory size, reducing the level of set associativity carries the advantage of decreased cache controller complexity and increased maximum speed. The direct-mapped cache provides virtually the same cache hit ratio as the more complex 2- and 4-way set-associative caches, yet promotes greater overall system performance by allowing a faster system clock for the processor and cache system.

Block Size

Another contributing factor to cache hit ratio is cache block (or line) size, which is the number of bytes fetched by the cache upon a cache miss. Cache performance generally increases as the cache line size increases, because the cache fetches more data upon a miss and is more likely to contain the next segment of code. As the cache line size increases, however, processor delays caused by the cache line fetch and the likelihood of fetching unnecessary memory contents detract from performance. The net effect is that cache performance generally increases as a function of cache line size, but the small improvements in cache performance must be balanced against the disadvantage of processor stalls caused by the longer cache line fetches.

Many other factors contribute to a cache's overall performance. One such factor is the method by which the main memory is updated upon a write access to the cache. Because the cache contains a copy of data stored in main



*ACM Transactions On Computer Systems, 11/88 Vo. 6 No. 4, Cache Performance Of Operating System And Multiprogramming Workloads, (Agarwal, Hennesy, Horowitz)

Figure 1. Effects of Associativity

memory, writing to the cache changes that data, which outdates main memory's contents. This problem introduces the issue of maintaining data consistency between the cache and main memory.

Two caching modes are used to ensure data consistency: write through and copy back. Write-through caching avoids the data-consistency problem by writing to main memory with every cache-write access. The problem with the write-through approach is that write accesses incur main memory delays upon every write to the cache. You can avoid these delays by using write buffers to store the data from write accesses, but buffers solve the problem only to the extent that they can store the write-access data and unload it to main memory. Block store operations, such as those used in context switches, often cause processor stalls under write-through mode, when the write buffers become overwhelmed with data. The write-through method also has the disadvantage of increasing bus traffic, because each write access forces a bus transfer. For these reasons, designers of shared-bus multiprocessing systems have largely abandoned write-through caching.

The alternative, copy-back caching, allows the processor to write to cache memory without immediately updating main memory. The copy-back cache keeps a state bit in each cache tag entry to report the modified status of a cache line. If the processor writes to a cache line, the copy-back cache controller sets the modified bit for that cache line. When a cache line is no longer needed, the state of its modified bit is checked. If the cache line has not been modified, the cache line is overwritten with a new cache line. If the cache line has been modified, however, the modified cache line is written out to main memory before being replaced.

Copy-back caching has the advantage of allowing any number of write accesses to the cache without processor delays. It also conserves system bus bandwidth, because cache lines are only written to memory when the cache line is no longer needed.

Cache Speed

A design issue of growing importance is the difficulty of building a cache fast enough to meet the processor's needs. Designing a discrete CMOS or TTL cache controller that can achieve zero-wait-state performance is becoming prohibitively difficult at processor speeds of 25 MHz and beyond. Driven by the timing problems of high-speed cache design, many designers are using ASICs to implement custom cache controllers at speeds of 16 to 25 MHz. At speeds of 33 MHz and above, designers are relying on VLSI cache controllers.

The use of VLSI cache controllers as part of a processor chip set is becoming the preferred method of microprocessor CPU design. This approach minimizes design time, while offering superior performance with minimal cost. VLSI cache controllers also provide speed enhancements due to the integration of features such as cache tag memories and MMU controllers. By providing greater functional density than that achievable with ASICs, the VLSI custom controller offers greatly enhanced levels of integration and maximum system speed.

As semiconductor technology matures, increasing levels of CPU integration become possible. This has resulted in the recent emergence of integrated processors with on-chip cache systems. Integrated caches offer greater system integration and the opportunity for processor architectural improvements that are prohibitive to implement outside the chip.

However, one die currently cannot accommodate an entire CPU plus a cache that achieves a 96-percent cache hit ratio. The number of transistors that can be placed on a single chip is limited, which forces chip designers to reduce cache size to allow room for the processor. The cache size problem increases significantly as the processor becomes larger and more complex. The transistor budget for an integrated cache processor currently forces system tradeoffs that require the supporting memory system to compensate for the resulting low cache hit ratio.

Integrated cache processors can present a problem for the system designer attempting to assess system performance. Benchmarks for these processors are often carefully chosen or modified to maximize the cache hit ratio, providing performance numbers that you cannot achieve in the real world. This leads to a buyer-beware situation. In evaluating any cache system, whether it is on or off chip, weigh performance numbers against unbiased, authoritative research findings on similar cache designs.

Always keep in mind that processor performance depends on the entire memory system, not just the cache. Even a cache system with a very high hit ratio can result in mediocre system performance if a slow supporting memory system hinders the cache. You must therefore pay attention to the supporting memory system to achieve the desired system performance.

Getting the Speed You Need From Main Memory

As previously stated, minimum average memory access time represents maximum system performance. Equation 1 gives the average memory access time as the probability weighted sum of the average cache-hit memory access time and the average cache-miss memory access time. Although you always want to minimize the supporting memory latency, the importance of minimizing main-memory latency grows as the probability of a cache miss increases.

The obvious approach to minimizing the supporting memory's latency is to design a fast DRAM main memory. To provide maximum access speeds, DRAMs commonly provide fast sequential memory accesses via an addressing mechanism such as page mode, static column, or nibble mode. These features prove useful for cache line fetch, because a cache line is a fixed-length, sequential series of memory accesses. However, sequential accesses are often not enough.

Another method of increasing DRAM memory system speed is to employ interleaved banks of DRAM. This essentially involves supplying addresses to several banks of DRAM simultaneously and sequentially enabling the memory bank outputs to place each word of the cache line on the memory bus. This method does not necessarily reduce the latency associated with the initial cache-line access, but the latency for all subsequent accesses in the cache line is minimized.

Cache line prefetch is another method you can use to maximize main-memory performance. Because caches are designed around the concept of sequential memory accesses due to an effect known as spatial locality, a cache miss on any specific cache line increases the probability of a cache miss on the next cache line. Main memory can use this concept to anticipate the cache by prefetching the next cache line after servicing a cache-line fetch.

You can accomplish cache-line prefetch by designing a memory controller that can access the next cache line and store it into a prefetch buffer in the memory controller. You can also implement cache-line prefetch by asserting the next cache-line address to the memory in anticipation of the next cache line, thereby minimizing the initial memory access latency. Either method requires a more complex memory controller and an address competitor to prevent memory access errors. Note that by implementing cache-line prefetch in the main memory system as opposed to the CPU cache, you avoid unnecessary bus traffic for unused prefetched cache lines.

An extension of the cache-line prefetch approach is to employ secondary, or second-level, caching. The secondary cache is essentially a much larger cache used to support the smaller CPU cache. In general, the secondary cache is 2^k times larger than the primary CPU cache, and the secondary cache blocks contain 2^n primary cache blocks (where n and k are typically ≥ 2). The use of a secondary cache allows fast cache-line fetching by the primary-level cache, assuming a cache hit in the secondary cache. Upon a secondary-cache miss, data is supplied from main memory.

To minimize the secondary-cache-miss penalty, cache-line forwarding is generally used. This allows the cache line requested by the first-level cache to be fetched from main memory with essentially the same latency as main memory alone. The secondary cache updates itself with the missed cache line as the line is supplied to the primary cache. The secondary cache then fetches the remainder of the secondary cache line.

Note that the initial main memory access delay for a DRAM memory system is generally sufficient time for a secondary cache to determine if a cache hit has occurred. This delay can be used in designing a secondary cache that introduces no latency penalty over a main memory system alone. The cache-line address can be supplied simultaneously to both main memory and the secondary cache. The secondary cache uses the initial main memory access latency time to determine whether a cache hit occurred and to inhibit main memory, thus preventing bus contention.

Including a secondary cache in the system greatly reduces the latency associated with a primary cache miss. Assuming a primary-cache hit ratio of 90 percent, only 10

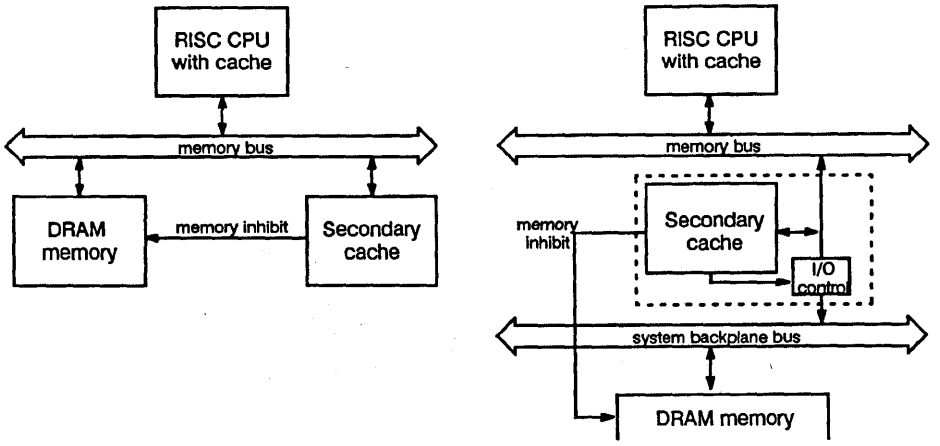


Figure 2. Secondary Cache Systems

percent of the memory accesses are made from the supporting memory system. With the inclusion of a secondary cache, 90 percent of those primary cache misses are fetched from the zero-wait-state secondary cache, again assuming a 90-percent cache hit rate. This leaves only 10 percent of the primary cache misses to be fetched from main memory. Therefore, the percentage of memory accesses that incur the full delay from main memory drops to 10 percent of 10 percent, or only 1 percent!

Secondary Cache System Applications

Secondary caching proves especially useful for supporting small, on-chip integrated processor caches and for supporting shared-bus, multiprocessing systems (*Figure 2*). For the latter, processing nodes with small- to medium-sized primary caches can share a large secondary cache. This approach allows equal or greater performance than nodes with large primary caches and also reduces the cost of each processor node.

The CY7C600 Chip Set

The Cypress CY7C600 SPARC RISC chip set is an example of an high-integration CPU with a VLSI cache subsystem (*Figure 3*). This chip set comprises the CY7C601 Integer Unit, the CY7C602 Floating-Point Unit, the CY7C604 Cache Controller and MMU, and two CY7C157 Cache RAMs. These five chips constitute a high-performance CPU that requires no glue logic and operates at speeds from 25 to 40 MHz, providing 29 MIPS of sustained integer performance.

As a part of this CPU, the CY7C604 provides a tightly coupled SPARC reference MMU and cache controller with cache tag RAM. The chip implements a 64-Kbyte, direct-mapped cache. The 64-Kbyte cache is estimated to provide an average cache hit ratio of 96 to 98 percent. You can expand the cache to a maximum of 256 Kbyte by using additional CY7C604 cache controller/MMUs and CY7C157 cache RAMs.

The CY7C604 provides a high degree of functional integration and includes features such as on-chip write and read buffers, cache-tag memory, and a SPARC reference MMU with 64 lockable TLB (table look-aside buffer) entries. The CY7C604 supports both copy-back and write-through cache modes, giving you the superior system performance of copy back or the simple cache coherency afforded by write through.

Under copy-back mode, the CY7C604's on-chip write and read buffers allow modified cache lines to be simultaneously flushed out of the cache while the missed cache line is fetched from main memory. Write buffers also boost performance in write-through mode and for non-cached memory accesses, allowing the CY7C604 to store up to four double-word memory writes without stalling the CY7C601 processor.

Incorporating cache-tag memory into the CY7C604 provides extremely fast recognition of cache hits or misses, thus allowing the cache to run faster than is possible with off-chip tag memory. Including the SPARC reference MMU with the cache controller on the CY7C604 allows tightly coupled operation between the cache controller and MMU functions. The MMU checks access-privilege status for all memory accesses, including those to cache, thereby protecting memory from unauthorized accesses. In addition, the SPARC reference MMU supports execute-only access protection for memory, providing an additional level of security for sensitive code and computing environments.

Cache miss latency for the CY7C604 cache is minimized by the use of high-speed, 0.8μ, dual-layer-metal, CMOS logic and support of the SPARC reference Mbus. Mbus is a 64-bit multiplexed address and data bus that supports burst-mode accesses and provides a peak bus

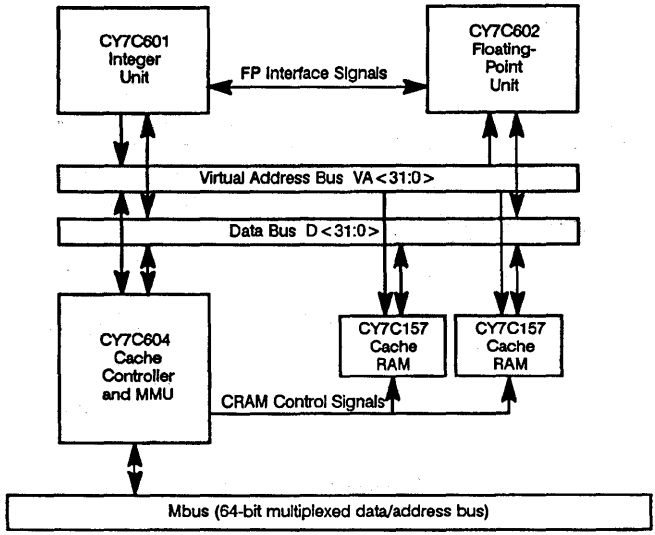


Figure 3. CY7C600 SPARC Chip Set

bandwidth of 320 Mbyte/s at 40 MHz. Mbus allows cache lines to be transferred in bursts, providing a fast interface to main memory. All CY7C604 burst accesses are in cache line lengths and on cache line boundaries, simplifying both the main memory design and the Mbus interface.

In addition to supporting the high-speed Mbus, the CY7C604 provides support signals for secondary-cache systems. The CY7C604 furnishes visibility into the cache operation from the memory bus by supplying a cache status signal. This function gives a secondary-cache controller greater flexibility in managing the status of its cache and can be used to increase secondary-cache efficiency.

The CY7C600 chip set is a high-performance RISC CPU, providing maximum system performance with minimal design effort. The chip set is available in speeds of 25 to 40 MHz, and its five-chip, no-glue-logic design offers a highly compact solution to state-of-the-art computing needs.

CYPRESS SEMICONDUCTOR

High-Speed CMOS SPARC System Design

This application note describes many of the effects caused by high clock speeds and rules of thumb for lessening the severity of the effects. Following these rules of thumb will help ensure a successful SPARC hardware design.

The SPARC (Scalable Processor ARChitecture) RISC processor is the only RISC processor architecture designed to be scalable, so that the processor's clock speed can increase as semiconductor process technology improves. The benefits of scalability appear most dramatically in the Cypress CY7C600 SPARC product family. In a little more than a year, Cypress has increased the clock speed on the CY7C601 integer unit from 25 to 40 Mhz.

As the CY7C600 SPARC family leaps upward from 25 to 40 Mhz, system designers must become more aware of the effects of fast clock speeds upon hardware design. High-speed hardware design is not a difficult art, but it does require careful and close attention to detail.

The effects that can lead to untraceable bugs in a high-speed system exist in a low-speed system; however, the magnitude of these effects in a slow system are small enough so that they can be safely ignored. This is not the case when clock speeds rise over 25 Mhz.

System Clock

At speeds above 25 Mhz, generating and distributing the system clock becomes a critical issue. The goal is to minimize the effects caused by duty-cycle imbalance, clock skew, and noise on clock lines.

Duty-Cycle Imbalance

Duty-cycle imbalance occurs when the clock signal's High and Low portions the are not symmetrical. Clock symmetry can vary from 40 to 60 percent, depending upon the hybrid crystal oscillator used. A simple way to ensure that the clock is symmetrical is to generate a signal at twice the frequency desired, then divide this frequency down to the system clock frequency using a D flip-flop (74AC110074). *Figure 1* depicts a simple clock generation circuit.

All physical devices exhibit an edge-dependent, propagation-delay asymmetry; i.e., the Low-to-High-going edge rises faster than the High-to-Low-going edge falls, or vice-versa. If a single driver buffers a clock line, the driver introduces asymmetry into the system clock signal. You can avoid this asymmetry by cascading two inverting

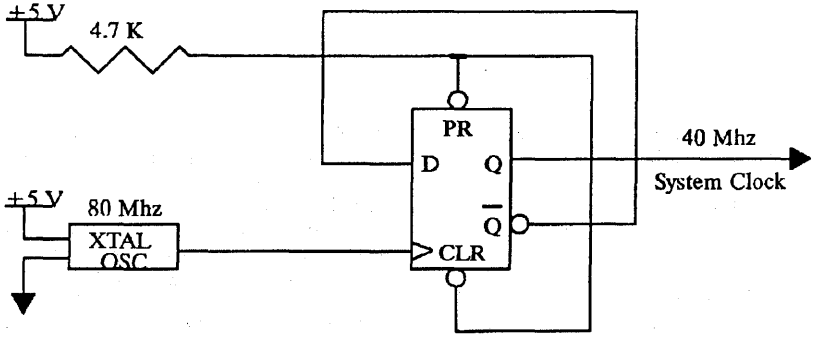


Figure 1. Symmetric-Duty-Cycle Clock Generation

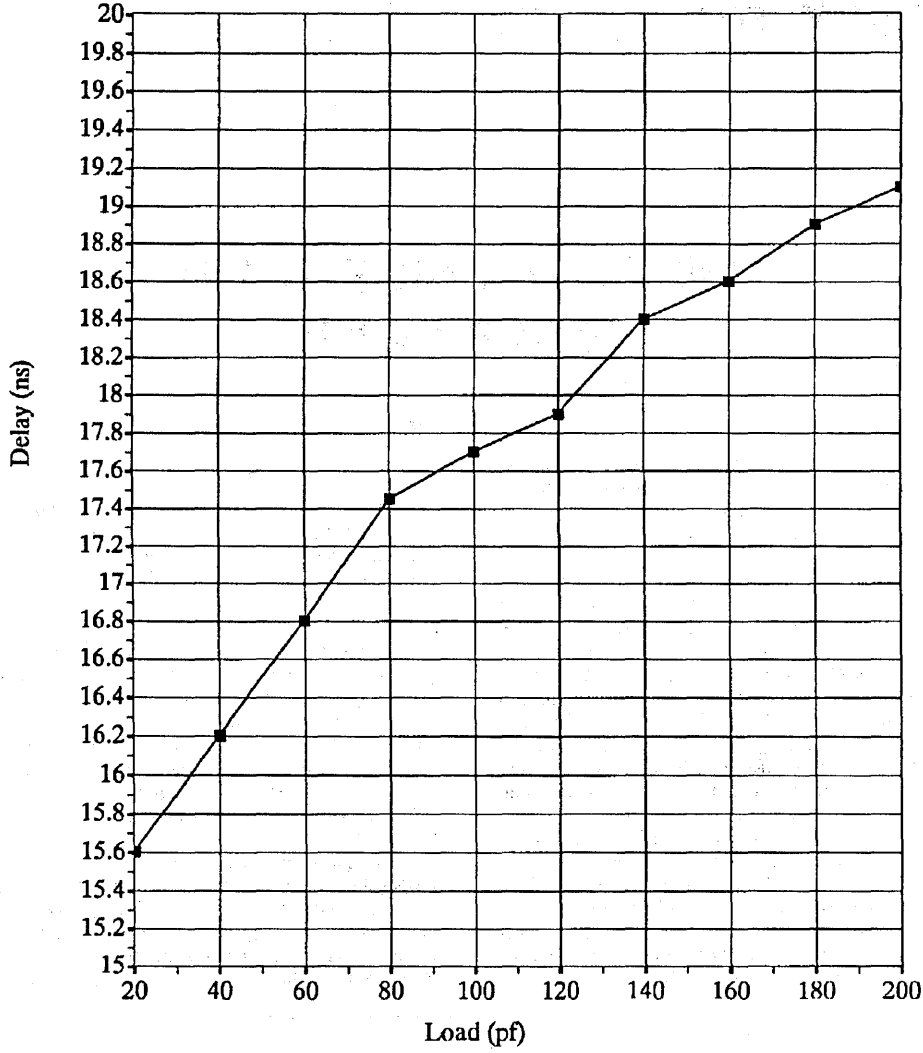


Figure 2. Delay (ns) vs Load (pF) for CY7C601

drivers in the same package. Because the drivers are in the same package, their delay characteristics are equivalent, and the differential between the Low-to-High transition and the High-to-Low transition is zero. A clock signal introduced into such a cascaded driver has the same symmetry going out as it had going in.

Clock Skew

Clock skew is caused by the need to distribute the system clock signal from a central point (the oscillator) to components that are dispersed on the printed circuit board (PCB). The only way to minimize clock skew is to design the PCB so that the fanout on all clock lines is equivalent. Use a chip with multiple on-board buffers to maximize line-driving capability.

The load on a clock line has three components: trace capacitance, socket capacitance, and input capacitance. Because the high integration of the CY7C600 SPARC family lends itself well to single-board designs, trace capacitance is not usually an issue. Socket and input capacitance dominate on PCBs.

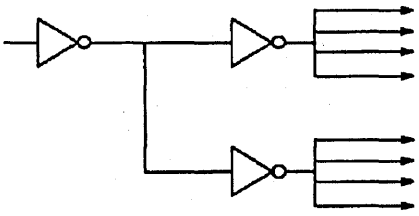


Figure 3. Parallel Clock Drivers

The pin grid array (PGA) package used for the CY7C600 family has extremely low capacitance. The maximum pin capacitances are 10 pF for input pins, 12 pF for output pins, and 15 pF for bus pins. You can limit other components' socket and input capacitances by using surface-mount design techniques.

As a rule of thumb, limit a clock buffer's fanout to eight to 14 devices. It is important to include both AC and DC loading in your fanout calculations. Data for the CY7C601 SPARC Integer Unit that relates delay to load appears in *Figure 2*.

AC characteristics for logic devices are usually calculated using a value of 50 pF. If more than 50 pF of capacitance is being driven, the driver's AC characteristics should be reduced for your calculations.

The input capacitance of a typical CMOS part is 5 pF. Bipolar logic is higher, with a typical input capacitance of 10 pF. Typical ECL parts are lower, with an input capacitance of about 3 pF. When you need a clock fanout greater a single buffer can supply, use the parallel driver scheme shown in *Figure 3*.

DC input current ratings are important when calculating total loading. The driving device must be able to sink the sum of the Low-level input currents to which it is connected. Low-level input current for bipolar logic ranges from -100 to -400 µA. The corresponding figure for CMOS is -1 to -5 µA, while ECL weighs in at 140 to 200 µA.

High-level input current for bipolar logic is from 20 to 50 µA, with CMOS at 1 to 5 µA and ECL at 265 to 350 µA. Because most bus drivers can sink up to -24 mA and source up to 48 mA, input current loading is seldom an issue. Input current loading might become significant when driving a parallel-resistor-terminated load. In such a case, use an AC termination scheme.

Clock-Line Noise

Noise on the clock distribution lines can have a ripple effect upon other logic. It is important to take steps to minimize self-generated noise on clock lines. Self-generated noise comes from two sources: reflectance from the end of the clock line and overshoot caused by line load capacitance. You can minimize reflectance by properly terminating the end of the clock line. (The "Noise Reduction" section covers line termination.) You can substantially reduce overshoot by using two parallel clock drive lines in place of one. This cuts the capacitance of the clock line in half.

Noise Generation

The CY7C600 family is fabricated using the Cypress CMOS process. Because of the fast edge rates (1 - 2 V/ns) and rail-to-rail voltage swings of high-speed Cypress CMOS logic, careful attention must be paid to signal noise. The primary sources of noise are ground bounce, power supply, crosstalk, and transmission-line reflections. You can combat noise effects by noise budgeting, good grounding, use of synchronous circuits, and proper line termination.

Ground Bounce

Ground-bounce noise arises when several outputs of a CMOS logic device switch from High to Low. This simultaneous switching causes a large sink current from the load capacitance to flow to ground through the device package inductance. This current develops a momentary potential whose magnitude equals the product of the package inductance and the sink current's rate of change:

$$V = L \times \frac{dI}{dt} \qquad \textit{Eq. 1}$$

where V is voltage, L is the package inductance, and dI/dt is the current's rate of change per unit time. This graph was computed using typical values of L and V.

Figure 4 illustrates typical ground bounce as seen at a device's output pin and the corresponding voltage induced across a ground pin. The voltage is normalized to 1V. If you apply 5V, for example, you see a ground bounce of approximately 0.75V 1.2 ns after the power is applied. Note the voltage undershoot at 0.5 ns caused by the inductance. Without damping or termination, you can expect the ground bounce to settle to zero in approximately 1.8 ns.

The fast edge rates of the CY7C600 devices can lead to a fairly large ground-bounce potential. This voltage spikes the Low state held on the quiescent outputs and can exceed the input Low-level maximum (0.8V), causing downstream logic to switch erroneously. Ground-bounce noise can also cause registers in the bounced device to lose their stored state. This is caused by the momentary disturbance in the device's ground and V_{CC} reference.

The switching of multiple outputs on a CMOS device also changes its propagation delay. The delay increases by approximately 200 ps per switched output. For a device with a large number of outputs, this additional delay should be included in worst-case timing analyses.

The magnitude of a given ground bounce is proportional to the package inductance and the number of outputs switched. By reducing parasitic inductance between the package, ground and V_{CC}, you can minimize the effect of ground bounce. The most effective way to reduce parasitic inductance is to use surface-mount technology (SMT). You can also reduce parasitic inductance by using packages with center V_{CC} and ground pins and by using low-inductance bypass and decoupling capacitors. For

parts in critical logic paths, use a standard decoupling capacitor (0.01 - 0.1 µF) along with a high-frequency decoupling capacitor (470 pF).

If you employ pin grid array (PGA) or through-hole technology, you can also reduce the effect of the ground bounce by using series damping resistors on the package outputs. The resistors lower the magnitude of the ground bounce before it reaches the downstream logic. As an added benefit, the magnitude of signal overshoot and undershoot is decreased. The tradeoff is slower switching rates, due to the increased RC time constant.

You can also reduce ground-bounce magnitude by using fewer outputs per package. *Figure 5* shows the relationship between ground-bounce magnitude and the number of outputs switched. Note that the relationship is roughly linear.

Further, ground-bounce magnitude is directly proportional to the power supply voltage. By reducing the magnitude of V_{CC}, you can reduce noise problems caused by ground bounce.

The use of only synchronous circuits provides a built-in resistance to false triggering caused by ground bounce. Synchronous circuits only trigger when inputs and the clock signal change. The ground-bounce noise produced by the upstream logic has one clock cycle minus the set-up time to settle before the next clock reaches and triggers the downstream logic.

If asynchronous logic is required, the use of an output pin close to the package ground pin reduces ground-bounce noise. The difference in noise magnitude between pins next to the ground pin and pins next to the V_{CC} pin can be as much as 50 percent.

To minimize the effect of ground-bounce noise upon the rest of the circuit, avoid running control signals through a device that drives data and/or address lines. The probability of multiple data or address lines simultaneously transitioning is high. If the device also contains control signals, they can be erroneously switched by the ensuing ground bounce.

Power Supply

Like the system clock, the power supply generates a global signal; its fluctuations have an effect upon every component in the system. Power-supply variations have a greater effect as clock speeds increase. High-frequency noise and ripple from the power supply can cause differences in voltage levels among different sections of the system. As a rule of thumb, high-frequency noise occurs whenever the mean wave length of the noise on the power lines is not several times greater than the length of the longest power line.

By causing the voltage levels to vary across the PCB, high-frequency noise and ripple from the power supply leads to a loss of noise immunity due to a reduction in the difference between the voltage value of input Low and input High. Bypass capacitors at the power supply input smooth out momentary current fluctuations. High-frequen-

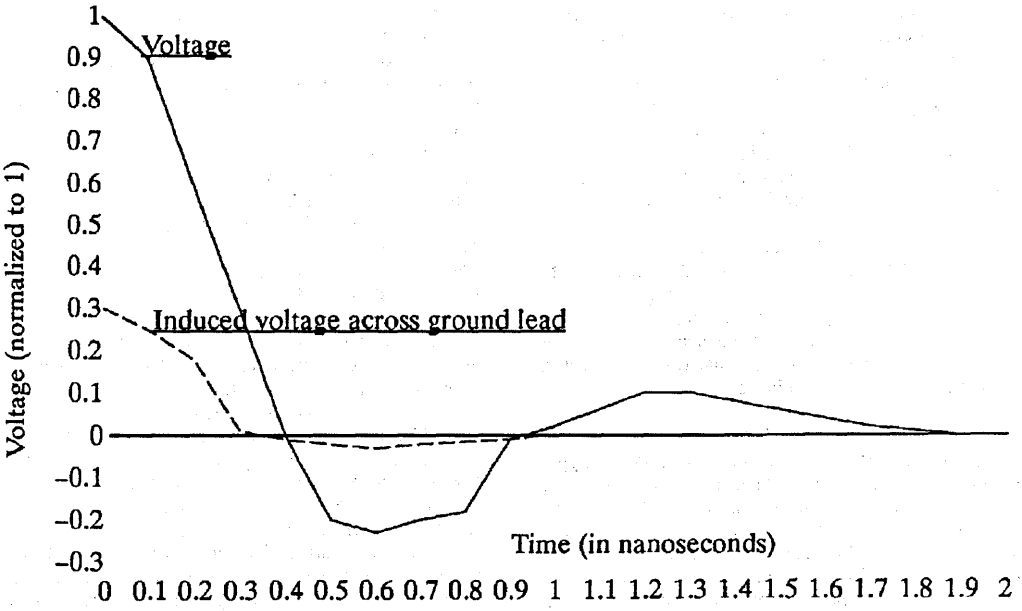


Figure 4. Ground Bounce and Voltage Induced on Ground Pin

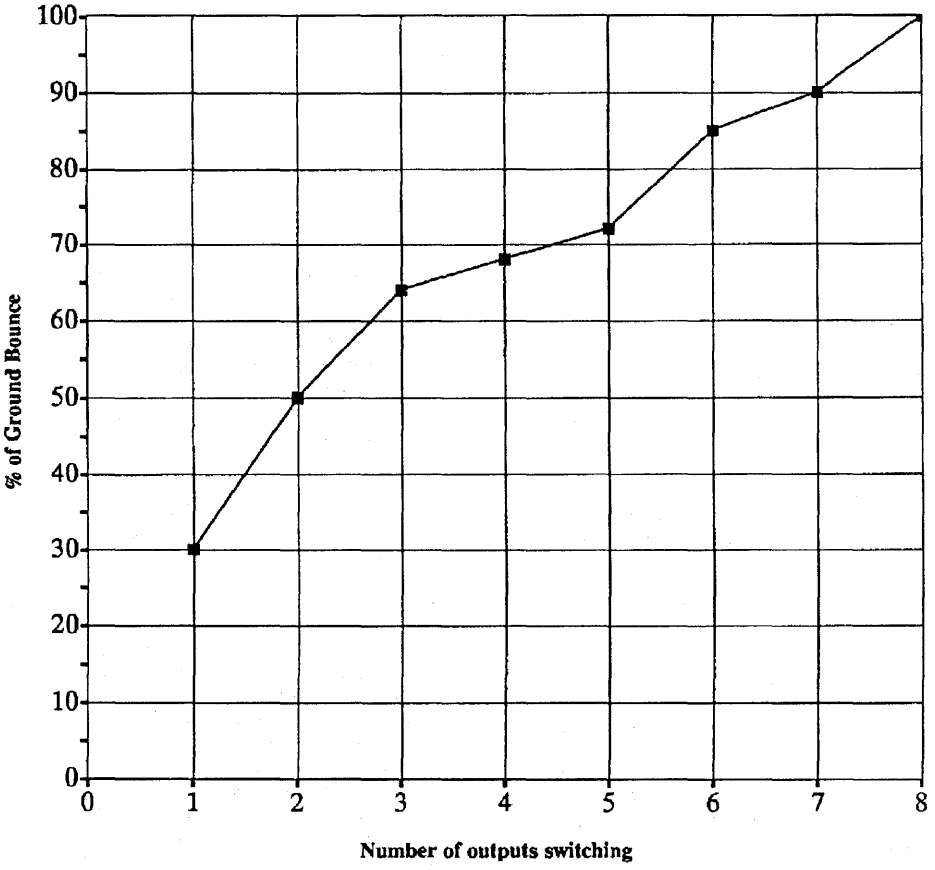


Figure 5. Ground Bounce Magnitude vs Number of Switched Outputs (from Reference 5)

cy power-supply noise should be specified to be under 50 mV peak to peak.

Crosstalk

Crosstalk occurs when a signal passing through a board trace or transmission line generates a corresponding signal in an adjacent quiescent trace or line. Crosstalk magnitude is proportional to three factors; edge rates, physical proximity of the lines, and the distance over which the two lines are adjacent. Because of the fast edge rates of CY7C600 devices (up to 2 V/ns) and other high-speed CMOS logic, crosstalk deserves careful consideration.

There are three ways to minimize crosstalk: grounding, shielding, and separation. During the initial design, maximize the distance between traces and minimize the length over which they are adjacent or parallel. Run ground strips alongside either the cross talker or the cross listener or between them.

To prevent crosstalk, critical signals such as the clock should always have a dedicated ground line. When possible, the signals on adjacent PCB layers should be perpendicular to each other. Use the power and ground layers as shields between signal layers. For backplane and wire-wrap applications, use twisted-pair for sensitive signals such as clocks, asynchronous set and clear signals, and asynchronous parallel loads. When using ribbon or flat cabling, make every other conductor a ground line.

If crosstalk occurs in an already-designed board or system, try these quick fixes to solve the problem: On PCBs, glue a grounded wire or copper strip alongside or between the affected traces. In a backplane or wire-wrap-

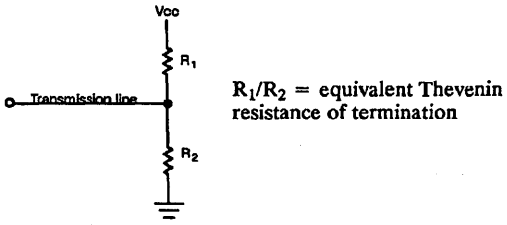


Figure 6. Split-Resistor Termination

ping situation, spiral a ground wire around the talker and/or listener to increase their shielding. Use a split-resistor termination on the offending line, where R1/R2 = the Thevenin resistance, which is the impedance of the line (*Figure 6*). (The "Parallel Termination" section explains how to determine the Thevenin resistance). You can use diode or active termination to reduce ringing (see the "Diode Termination" section). As a last resort, cut the offending crosstalk trace from the PCB and replace it with a wire. By re-routing the wire, you might reduce crosstalk, at the possible cost of greater propagation delay.

Transmission-Line Reflections

For long trace lengths or backplane connections, it is sometimes necessary to consider transmission-line effects. These effects are significant when the unloaded signal transition time is less than or equal to the round-trip substrate propagation delay. For ordinary PCB materials (G-10 epoxy), the round-trip propagation delay is approximately 0.295 ns/inch. Unloaded signal transition time for the CY7C600 devices varies from 3 to 2 ns, depending upon clock speed. Traces longer than 6 -10 inches should be treated as transmission lines for noise calculation purposes.

Transmission lines suffer from three types of noise effects: undershoot, overshoot, and ringing. Undershoot occurs when a signal's voltage level momentarily drops below the Low level (0V). Overshoot is the inverse — when a signal's voltage level momentarily rises above the High level (+5V). (Use of a 5V power supply is assumed.) Ringing is when a noise pulse keeps on reflecting back from the two ends of a trace or wire.

All of these effects result from reflectance at the end of the trace or wire. Depending on where the reflection appears in relation to the signal, a reflected noise pulse can manifest as undershoot or overshoot.

Because of the CMOS technology used to fabricate the CY7C600 devices, the parts resist damage caused by undershoot and overshoot on input lines. The devices are insensitive to -3V DC input levels (sustained) and -5V undershoot levels less than 10 ns long (measured at the 50 percent point). Input levels as high as +5.5V DC can be withstood without damage, as can momentary overshoot pulses of up to +6V DC.

Reflectance is caused by a mismatch between the line characteristic impedance and the load impedance. The following equation shows the relationships involved:

$$P_L = \frac{R_L - Z_0}{R_L + Z_0} \qquad Eq.\ 2$$

where R_L is the load impedance, Z_0 is the line impedance, and P_L is the coefficient of reflectance, which equals the reflected voltage over the incident voltage. The equation shows that the reflectance from the end of the line goes to zero as the term $R_L - Z_0$ goes to zero. Additionally, the magnitude of the reflectance decreases to zero as $R_L + Z_0$ goes to infinity. These relationships show two ways to decrease the reflection from the end of a transmission line: match the line's impedance to that of the load to minimize the voltage reflected or maximize the sum of both impedances to minimize the effect of the reflected voltage. The tradeoff is that maximizing impedance decreases the signal rise time. Both methods are discussed in the next section.

Reducing Noise

You can envision the effect of noise upon a system by using a design method called noise budgeting. In the simplest sense, noise budgeting is the allocation of noise to system noise sources (ground, power, crosstalk, etc.) in such a way that the noise immunity of individual components is not exceeded. Allocation is based on the calculated or expected values of the noise generated.

A noise budget table shows you the relative magnitude of noise generated by each source. This allows you to focus your noise-reduction efforts where they will have the most effect. A noise-budget table for a representative system appears in *Table 1*. The entries for DC and AC noise represent the peak noise values allocated to the specific noise source. For example, the expected peak noise due to EMI is 2 mV. Note that some noise sources, such as temperature, have only DC components; others, such as crosstalk, have only AC components.

The concept of a noise budget rests upon three points: effectivity measures, probability theory, and noise immunity. An effectivity measure relates a circuit parameter, such as temperature, to its effect upon a device's output level. For example, temperature has a DC effectivity measure of 1.0 in *Table 1*. This means that if the noise generated by temperature variations is 10 mV, then the noise output from the circuit equals 10 mV of noise times a 1.0 DC effectivity, or 10 mV. You can determine effectivity measures from vendor-supplied circuit characteristics and graphs.

Probability theory comes into play when you consider the chance that a noise event will falsely trigger a circuit. You can consider noise in a computer system as random for most practical purposes. This assumption might seem counter-intuitive, as noise is caused by transients, whether its source lies in temperature, ground levels, signal edges, etc. Each noise point source is deterministic; a reflected noise pulse is generated only when the incident signal reaches the end of a transmission line. However, the noise

Table 1. Noise Budget

Source	DC Noise (mV)	DC Effectivity	Equivalent DC (mV)	AC Noise (mV)	AC Effectivity	Equivalent AC (mV)	Total Effective Noise (mV)
Gnd to Gnd	11	1.0	11	105	0.45	47.25	58.25
PC card Crosstalk	--	--	--	75	1.0	75	75
Backpanel Crosstalk	--	--	--	71	1.0	71	71
V_{CC} Bus	230	0.10	23	80	0.29	23.2	46.2
Temperature	10	1.0	10	--	--	--	10
SIP Crosstalk	--	--	--	22	1.0	22	22
Termination	--	--	--	24	1.0	24	24
Wire Untwist	--	--	--	35	1.0	35	35
EMI	--	--	--	2	1.0	2	2

observed at any place in the system is the sum of all the point sources of noise. This overall noise is random, because it is the sum of time-diverse noise sources such as slow variations due to temperature, fast variations at clock edges, etc.

Note also that noise generated by an event might arrive at a component from different sources separated in time because of the different-length paths the noise took to arrive at the component. Therefore, noise magnitude assumes a Gaussian distribution — the familiar bell-shaped curve.

In a typical electronic system, no single point source can generate enough noise to falsely trigger a circuit. A circuit is only triggered when a group of random noise pulses sum to greater than the circuit's noise immunity. Because the resultant noise is the sum of random noise with a normal distribution, the probability that noise of a certain magnitude will be encountered equals the area under the normal curve. Peak noise voltage occurs within 3 sigma limits of the mean noise voltage on the normal curve. 99.7 percent of the area under the normal curve is within 3 sigma of the mean. Noise magnitude will be less than or equal to the peak noise 99.7 percent of the time. Thus, the peak noise voltage will be exceeded approximately 0.3 percent of the time. The peak equivalent noise for a system equals the root of the sum of the squares (RSS) of the individual sources:

$$\text{Noise}_{(\text{equivalent})} = \sqrt{S_1^2 + S_2^2 + \ldots}$$

where S1, S2... are the noise sources. The RSS of the system described in *Table 1* is 136.3 mV. This is the peak value of the total effective noise, and this value will not be exceeded 99.7 percent of the time. For a design to be immune to noise effects, the noise immunity of the component with the least amount of noise immunity must exceed this peak value by a wide margin. Conservatively, minimum noise immunity of 2 x peak is acceptable.

The last point upon which noise budgeting rests is noise immunity. Noise immunity is the amount of noise in volts that a component can absorb without changing state. The noise immunity for a component is the difference between input High voltage (I_{IH}) and input Low voltage (I_{IL}). For the CY7C600 family, I_{IH} = 2.1V and I_{IL} = 0.8V. This gives a noise immunity of 1.3V. For a CY7C600 component to be switched by a noise pulse, the noise must therefore have a magnitude of at least 1.3V.

Grounding Techniques

Like the clock and power, ground is a common signal for all components. For high-speed SPARC CMOS logic design, the use of proper grounding techniques is important to reduce crosstalk and increase switching rates.

The basic grounding technique for PCBs is to provide a ground comb on one side of the board. A ground comb is a series of parallel strips connected at one end by a perpendicular trace. The ground strips should only be connected at one end, to minimize noise coupling. The ground load caused by switching components on each strip should be dispersed in both time and space to decrease the amount of noise coupling between components.

Remember that the ground, while providing a common voltage reference, also provides an alternate path for noise signals. Ground bounce travels down signal lines as well as the ground plane to which the circuit is connected.

Take care to ensure that chips with many outputs that switch at the same time do not connect to the same ground plane. The ground plane should be connected to 10 percent of the edge connector pins spaced equally apart. This reduces the ground impedance, which minimizes crosstalk because multiple signals do not rely upon a single ground return path. Connect high-current circuits to a separate ground to minimize noise coupling to other circuits. For high-speed SPARC CMOS designs, use a

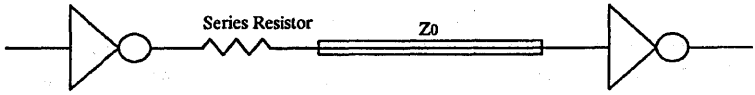


Figure 7. Series Termination

multi-layer PCB with separate ground and V_{CC} planes to decrease system-wide noise.

Using Synchronous Circuits

Because noise is random, large noise spikes can occur at any time. The only ones of interest are the ones that falsely trigger a circuit. In an asynchronous design, any noise pulse that exceeds a circuit's noise immunity triggers the circuit.

In a synchronous design, on the other hand, the circuit is triggered only when a clock edge occurs at the same time as a noise pulse that exceeds the circuit's noise immunity. A valid clock edge only occurs during 25 - 35 percent of the clock cycle, depending on whether one or both of the clock's edges can trigger an event. Thus, 65 - 75 percent of the noise pulses that randomly occur are unable to falsely trigger circuits. The inherent noise resistance of synchronous logic make it a must for robust high-speed system design.

Termination Methods

You can reduce reflectance from the end of a signal line by using proper termination. To do this, you use a resistance to either damp the reflected signal or match the impedance of a transmission line to the source or load, thus reducing the reflection's magnitude. Consider using termination techniques for signal lines longer than 6 1/2 inches. Termination is mandatory only for clock inputs, write and read strobes, and chip select and enable lines. Address and data lines usually have time to settle before they are sampled.

Series Termination

There are four methods of termination: series, parallel, AC, and diode. You can accomplish series termination by placing a resistor in series with the output of the device driving the signal trace (*Figure 7*). The intent is to match the trace's impedance, Z_0, to the circuit output impedance plus the resistor value. When these two quantities are equal, according to Equation 1, the reflectance from the driving device is zero. Thus, if a noise pulse is reflected back from the driven device's input pin, the pulse is absorbed when it meets the series resistor. Place the series resistor as close to the output pin as possible.

One of the advantages of series termination is that it causes no DC power dissipation and therefore does not add to the system's overall power requirements. Series termination does have several disadvantages, however. One is slower signal propagation, which is due to the larger RC time constant. Another disadvantage is that you cannot use distributed loading along the line. The series resistor's voltage-divider effect during the two-way propagation delay time causes any inputs attached along the line to see an input voltage halfway between the logic levels; the devices therefore fail to respond correctly.

Reflections at the receiving gate place no restriction on the number of lumped loads you can place at the end of the line because all reflections are absorbed at the source. However, the voltage drop across the series-terminating resistor limits the effective loading of the line.

A variation of series termination is series damping. In this technique, instead of matching the line's impedance to the driving circuit's output impedance plus the resistor value, you use a resistor of 10 to 75Ω. This resistor damps the noise pulses caused by reflection at impedance mismatches; the pulses are not completely absorbed as with series termination. Except for this difference, the advantages and disadvantages of both techniques are the same.

Parallel Termination

In parallel termination, you place a pull-up and a pull-down resistor at the end of the signal trace (*Figure 8*). The Thevenin equivalent value of the resistors equals the impedance of the signal trace. Two simple formulas for calculating the proper values of R_1 and R_2 are

$$R_2 = 2.6\,Z_0$$

$$R_1 = \frac{R_2}{1.6}$$

where Z_0 is the impedance of the signal trace.

The advantage of parallel termination is that the waveform along the full length of the line remains undistorted. Also, the rise time of the signal traveling down the terminated trace is unaffected. Additionally, you can use parallel termination when the signal line's characteristic impedance not completely defined. By approximating the impedance, the reflectivity coefficient is still relatively small; thus, overshoot and undershoot will probably remain within safe limits. On the negative side, the pull-down and pull-up resistors constantly dissipate power. For this reason, parallel termination should probably not be used in systems utilizing the high-speed CMOS CY7C600 family.

AC Termination

Figure 9 shows the AC termination method, which is the most common termination approach. It does not have the half-voltage disadvantage of series damping and causes no DC power dissipation. The latter feature is important when using low-power CY7C600 devices.

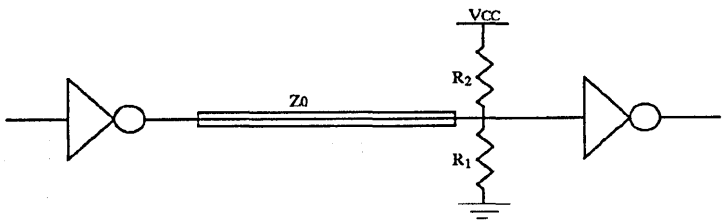


Figure 8. Parallel Termination

AC termination consumes no DC power because the capacitor blocks the path to ground. You can attach loads at any point along the trace, and they see a full voltage swing. AC termination also acts as a low-pass filter for short noise pulses. Any noise pulse less than 4(R x C) seconds wide is filtered out.

The value of the capacitor, C, must satisfy two conflicting requirements. It must be large enough to either absorb or supply the energy contained or removed when positive or negative noise pulses occur. Additionally, the capacitor must be small enough to avoid delaying the signal or slowing the signal's rise and fall times beyond the design limit. For CY7C600 applications, the minimum set-up times determine the maximum degradation of the signal rise time. The minimum set-up time can be as small as 4 ns for a 33-Mhz part. You can use the following formula to closely approximate the values for R and C:

$$C = \frac{T}{2.2R} \qquad \textit{Eq. 3}$$

where T is the maximum degradation of the signal rise time. Start with a value of R slightly less than that of Z_0, the characteristic line impedance. Then calculate the value of C. The combined impedance of the resistor and the capacitor approximate that of the line, reducing the reflectivity at the end of the trace to near zero. You can verify this by calculating the capacitive reactance, X_c, of the capacitor:

$$X_c = \frac{1}{2\pi f C} \qquad \textit{Eq. 4}$$

where f is the frequency of the signal passing through the signal trace. As an example, *Table 2* was calculated for a 4-ns maximum signal degradation. The calculated values are based upon a 50Ω PCB trace and a 120Ω wire-wrapping line. If you use a resistor value of 47Ω and a capacitor value of 38 pF, the termination closely matches the impedance of the PCB signal trace. Additionally, the termination acts as a low-pass filter, absorbing all noise pulses under 7.12 ns in length and over 140 Mhz in frequency.

The disadvantage of the AC termination method is that it requires two components, a capacitor and a resistor. Remember to keep the leads as short as possible to prevent ringing caused by lead inductance.

Diode Termination

Terminating a signal trace with a pair of Schottky diodes is called diode or active termination (*Figure 10*). The lower diode's low forward voltage, V_f, clamps the input signal to below ground, and upper diode does the same to V_{CC} + V_f. These effects significantly reduce signal overshoot and undershoot. If both undershoot and overshoot are not a problem, you might require only one diode.

The advantage of diode termination is that you do not have to match the line impedance exactly, as you do in series, parallel, and AC termination. Diodes are more expensive than resistors or capacitors but might reduce overall system cost because they eliminate the work of precisely determining line impedances. Additionally, if you discover that ringing is a problem during system checkout, you can easily add diodes. As with all termination methods, keep the leads as short as possible to avoid ringing caused by lead inductance.

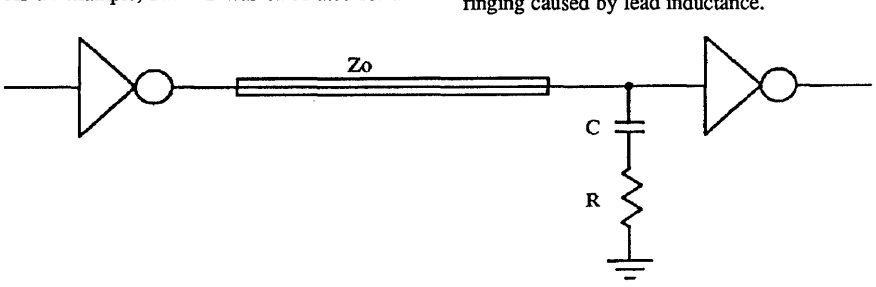


Figure 9. AC Termination

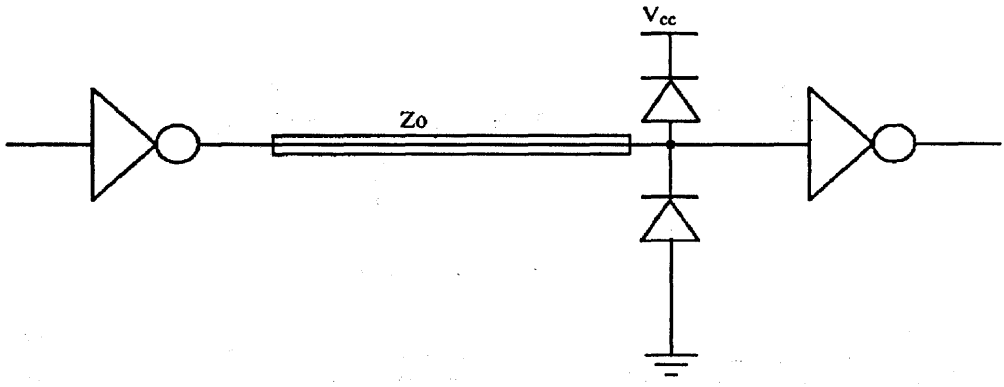


Figure 10. Diode Termination

The following Schottky diodes are suitable for termination purposes:

- 1N4148 (Switching)
- 1N5711
- MBD101 (Motorola)
- HP5042 (Hewlett-Packard)

References

1. *FAST Applications Handbook*, Fairchild, Inc., 1987.

2. Blood, Jr, William R. *MECL System Design Handbook*, Motorola Inc., 1988.

3. Hefner, Moore & Weinstein. *Advanced CMOS Logic Designer's Handbook*, Texas Instruments Inc., 1988.

4. *CY7C600 RISC Family Users Guide*, Cypress Semiconductor Corp., 1988

5. Tripp & Hall. "Good design methods quiet high-speed CMOS noise problems," *EDN*, October 29, 1987.

Table 2. AC Termination for a 4-ns Signal Degradation

Values	PCB	Wirewrapping
Z_0 (Ω)	50	120
R (Ω)	47	110
C (pF)	38	16
RC (ns)	1.78	1.76
4RC (ns) - passwidth	7.12 - (140 MHz)	7.04 - (142 MHz)

CYPRESS SEMICONDUCTOR

SPARC System Surface-Mount Design

This application note covers most of the pitfalls in SMT design and should help make your first SMT design successful. This is not a complete reference, however. For thorough coverage of SMT techniques, please refer to *References 1* and *2*.

Cypress's objective is to design and build the fastest, most capable SPARC chip sets in the world. As the operating frequency of Cypress's CY7C601, CY7C602, CY7C604, and CY7C157 SPARC chip set increases, concerns about factors such as package capacitance and inductance and PCB trace length become more important. You can reduce the impact of these factors by using surface-mount technology (SMT).

SMT differs from through-hole technology in that the component leads are placed directly onto the PCB rather than through the PCB. SMT permits greater component density, more reliable systems, and savings in labor and material costs. To gain these benefits, SMT demands care and precision in the placement and soldering of devices to the PCB.

Fine-pitch leads for SMT devices are not uncommon. Cypress and other advanced semiconductor vendors use 208-lead, 25-mil-pitch ceramic quad flat packs for many products. Fine-pitch packages such as these require precision in initial placement and alignment.

Ins and Outs of Surface-Mount Technology

Through-hole or leaded technology is a robust packaging technique that has served the electronics industry well in moving up the integration curve. In fact, if integrating more functions on chip was the only technology driver, through-hole technology would serve well for the foreseeable future. However, as the industry moves into the realm of system engineering, which involves multiple chips, other integration requirements reveal the flaws in through-hole technology.

The SPARC community's primary system-integration need is to reduce the physical and electrical distance between components to achieve higher clock frequencies. This is difficult to do because of the mechanical constraints imposed by through-hole technology.

Consider, for example, a typical through-hole package, the DIP. While small, a DIP is a physically imposing thing with its large package and stiff leads. These mechanical constraints are imposed by the need for the DIP leads to go through either a socket or the PCB. To achieve this penetration, the leads and package need stiffness and strength. This requirement causes the leads and package to have more mass and material, which, in turn, means greater capacitance, inductance, and package volume.

SMT packages do not have these mechanical constraints. Because surface-mount devices (SMDs) are placed onto, instead of inserted into the PCB, their strength and stiffness requirements are considerably lower. Thus, leads and package can be made as small as the number of signal leads and die bonds allow. Because the leads can be reduced to where they are just big enough to physically reach the PCB pads from the package, their capacitance and inductance are correspondingly reduced. This reduction decreases the capacitive and inductive mismatch between the leads, the PCB pads, and the PCB traces, which decreases the noise effects the component sees. The decreased noise effects allow the signal lines to run at higher frequencies without random problems caused by noise spikes.

An additional SMT benefit is the capacity to place more components in the same board area. Because SMT packages are smaller than through-hole packages, more components can reside in the same area. Because the components are closer together, the traces needed to connect them are shorter. This means less trace capacitance and impedance, which also makes higher operating frequencies possible.

As with everything in life, the advantages of SMT are not free. The primary difficulties encountered in using SMT involve placement and soldering. Placement of surface-mount devices is more difficult than for through-hole

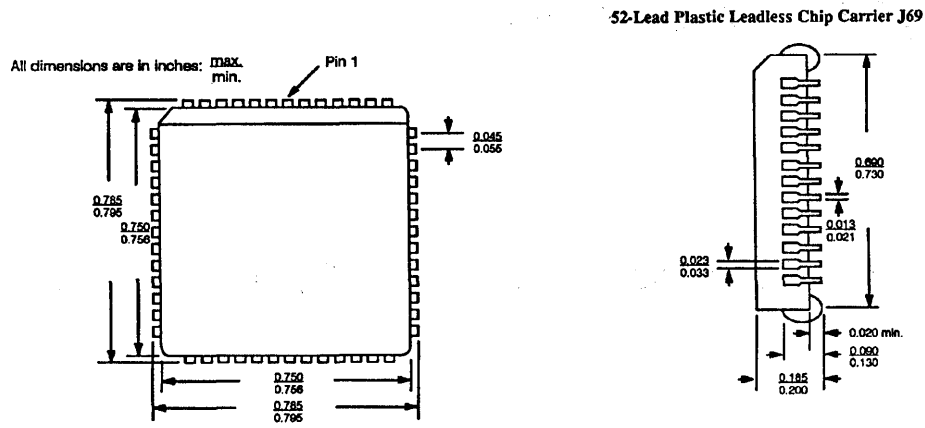


Figure 1. CY7C157 PLCC

devices because SMD placement is relative, not absolute. Because through-hole components are inserted into either the PCB or a socket, feedback on correct alignment is instantaneous: Either the component leads go into the holes, or they do not.

On the other hand, SMDs must be placed relative to the appropriate solder pads on the PCB. A misalignment of one or more leads does not become apparent until the placement is visually inspected. Additionally, the lead placement is not self-corrective. For a through-hole component, if one or more leads or PCB through-holes are slightly off, inserting the other leads tends to force the out-of-alignment leads into the correct orientation. This self correction does not exist for SMDs.

Soldering is the other area where differences between through-hole and surface-mount techniques become apparent. SMDs have a lower profile than through-hole devices, which puts SMDs closer to the PCB. If wave soldering is used, then a problem known as shadowing becomes a concern. Shadowing occurs when the solder wave must rise over the component instead of going under it, as in through-hole designs. The component body can shadow the component leads, preventing the solder from wetting them. As a result, some of the leads are not soldered to their PCB pads.

Another possible SMT problem caused by wave soldering is heat damage to components. While through-hole packages stand off from the PCB, SMDs sit on the board. The solder wave therefore washes over the SMD. If the solder temperature is not carefully controlled, the components can be damaged.

Fortunately, you can control all these difficulties peculiar to SMT by careful attention to the fine details of board stuffing and assembly.

Cypress Surface-Mount Packages

The Cypress product line includes three SMD package types: a 52-lead plastic leaded chip carrier (PLCC), a 160-pin plastic quad flat pack (PQFP), and a 208-lead ceramic quad flat pack (CQFP). The CY7C157 cache RAM comes in a 52-lead PLCC; the CY7C611 embedded controller is offered in a 160-lead PQFP; and the CY7C601 integer unit and CY7C604 and CY7C605 cache/memory management units can be packaged in a 208-lead CQFP. The drawings and form factors of these three packages appear in *Figures 1, 2,* and *3,* respectively.

Lead Handling for SMDs

The relative fragility of SMDs requires a change in handling procedures from that used for DIPs and other through-hole devices. These parts can be shipped and transported in carriers that allow flex and slight device movement. This type of packaging suits the JEDEC J69 52-lead PLCC used for the CY7C157 due to the robust nature of its leads.

However, this packaging is not suitable for the EIAJ standard 160-lead PQFP or 208-lead CQFPs that Cypress uses for non-memory devices. The leads in these packages are very fine and fragile and are susceptible to twisting or bending. These packages must be firmly fixed in place during transport. The best method is to use a waffle pak, in which the component leads are fixed by a small ridge of material that forms a box around the package. Sandwiching the package between two carriers holds it firmly in place. The ASAT 125C is a good example of this type of carrier.

Creating SMD Footprints on a PCB

Footprint or solder-pad design for PCBs is a critical part of good SMT design. This is because SMDs are not rigidly connected to the PCB during soldering, as are through-hole components. SMDs essentially float during the soldering process. This floating results from differences in surface tension due to uneven cooling after soldering.

The effects of floating can be reduced by carefully crafting the pad sizes. Reducing the pad width is the first step. A pad that is too long causes the SMD to float off the high point on the pad and over to one side. A pad that is too wide might allow the component to rotate. The ideal pad is almost exactly the same size as the SMD lead's contact surface. The pad width should equal 1.02 times the lead width, and the pad length should equal 1.02 times the lead contact length.

For PLCC devices (CY7C157), it is important that the lead footprints on the PCB not run too far under the package. Footprints should be extended out approximately 0.050 inch to the outside of the package. This helps reduce solder bridges under the PLCC, where they cannot be seen during visual inspection.

Fixing SMDs in Place

Through-hole devices are fixed in place in a socket or PCB either by lead bending or the mechanical tightness of the lead fit. SMDs are not. You must use adhesives to firmly fix SMDs in place before soldering. The only exception is when you use reflow soldering. In this technique, solder paste is applied to the PCB before the SMDs are place. Then, IR lamps or hot air cause the solder paste to reflow. The paste usually has sufficient adhesion to hold the devices in place until soldering is complete.

However, not all SMT PCBs can use solder reflow. If a board includes a mixture of through-hole and SMT devices, wave soldering or combination wave/reflow soldering is usually necessary. When you use these soldering techniques, you must apply an adhesive to the PCB to hold the SMDs in place until soldering.

The use of adhesives brings a new set of potential problems, especially relating to product reliability. The adhesive might absorb moisture and create a short on the PCB. The adhesive might also degrade in an unattractive way, causing marring or shorting of other system components. For these reasons, select adhesives for their lifetime properties. As a general guide to adhesive selection, follow this framework:

- *System*: Account for the intended application of the system in which the surface-mount PCB is used. An adhesive that has lifetime properties suitable for a workstation environment might not be optimum for industrial or military applications. Selecting the right type of adhesive at this stage prevents system failures during the product's lifetime.
- *Device*: Keep in mind the type of SMD used on the PCB. Most adhesives work with the plastic and ceramic SMT packages used by Cypress. However, other SMDs on your PCB might require a different type of adhesive.
- *Process*: How will the adhesive be applied to the PCB? The three available methods are pin transfer, Screen printing, and pressure syringe. Each method has its own advantages and disadvantages. The

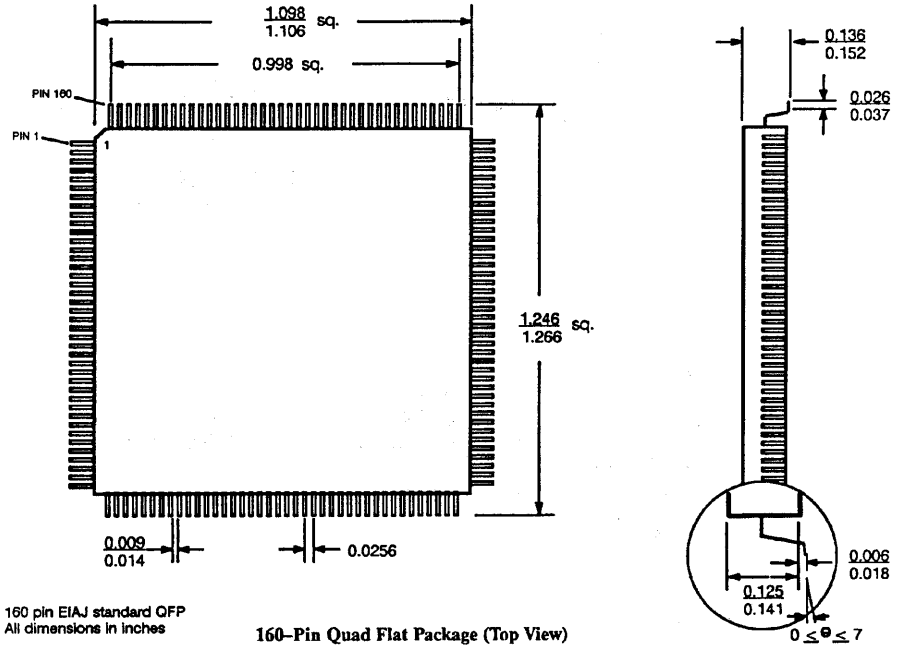


Figure 2. CY7C611 PQFP

dominant criteria for selection are the estimated production volume and the type of PCB substrate used. For example, screen printing demands a flat and distortion-free substrate. You cannot use this method for PCBs that already have components on them, such as pre-loaded mixed-print boards. Also limiting your choice is the fact that all adhesives are not compatible with all three methods.

- *Machine*: The application process you choose drives the selection of a machine for applying the adhesive. You can use either a stand-alone adhesive application machine or one integrated into a pick-and-place system.
- *Adhesive*: The final adhesive choice must be compatible with all the requirements you establish. This choice is often driven by the type of machine you choose, because the machine might have been designed with a specific adhesive type in mind. This makes the adhesive choice the responsibility of the machine manufacturer.

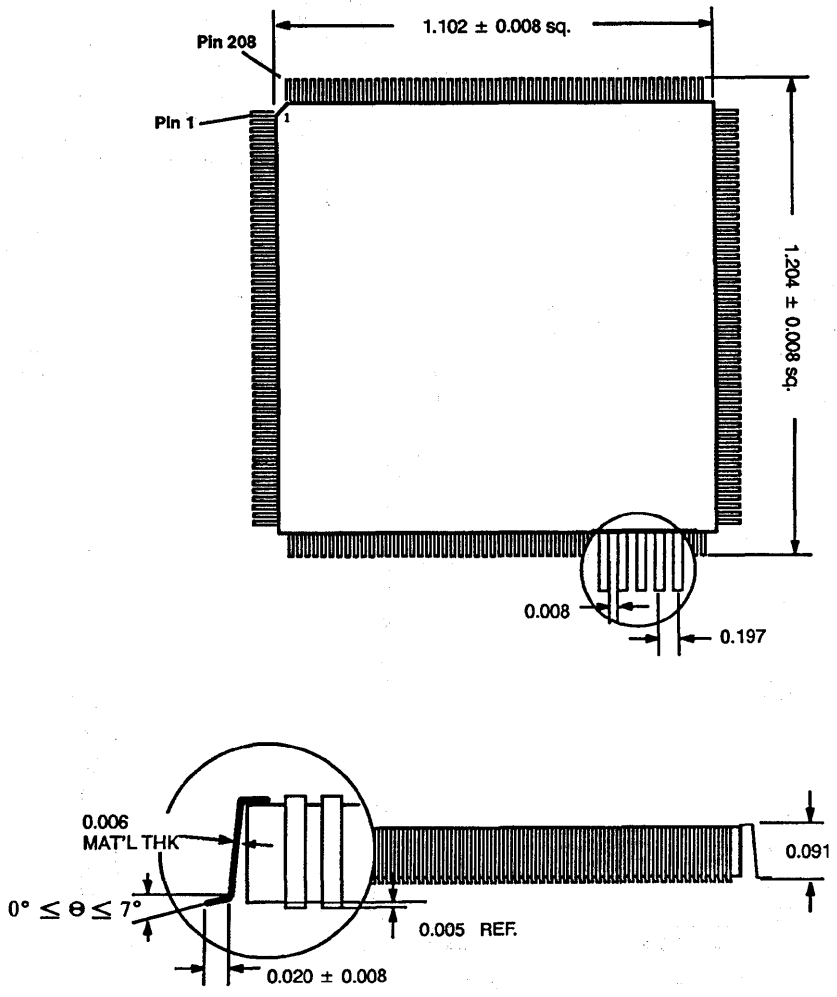


Figure 3. CY7C601/CY7C604/CY7C605 CQFP

SMD Alignment

The very fine pitch between leads on the 208-lead EIAJ CQFPs (0.0196 inch) places exacting requirements on pick-and-place machines. Cypress uses this package for some products, which usually require absolute placement to within ±0.002 inch or less in both X and Y coordinates, relative to the lead pads on the PCB. Additionally, angular error should be held to less than 1°. This requires that the pick-and-place machine have rotational correction capability. Vision capability is also needed.

It is important to realize that an interacting set of inaccuracies determine the required placement accuracy. The first inaccuracy is the location of the CQFP with respect to the vacuum pick-up nozzle. Usually, the pick-up nozzle only has a general idea of its position with respect to the true center of the device. This general idea is not sufficient for fine-pitch CQFP devices. Pick-up position needs to be controlled by accurately positioning the waffle pack (if used) in relation to the pick-and-place machine. The pick-up nozzle usually picks up the device, then repositions it by use of a centering system. Centering is done with reference to the leads' edge surfaces. Depending on the centering system's capability, it can achieve an accuracy of ±0.001 inch of the placement center to the device center. This is half the allowable error, and the device has not been placed on the PCB yet.

A vision system guides SMD placement on the PCB. The vision system first orients itself either by detecting fixed locating patterns on the PCB called fiducials or by looking for unique combinations of pads and vias that occur at fixed places on the PCB. Fiducials are the preferred method, because they take less processing capability for the vision system to recognize, and their location on the PCB can be more tightly controlled. One fiducial allows the vision system to locate itself with relation to the PCB. Two fiducials allows the vision system to establish a second-order level of correction, which encompasses X-Y offset, angular offset, and a linear expansion/contraction compensation for the medium. Adding a third fiducial improves the accuracy of these corrections through use of an interpolation algorithm in the vision machine software. It is a good idea to use several levels of fiducials to give several levels of position and angular correction: PCB to PCB, circuit to circuit, or component to component.

As mentioned earlier, inaccuracies have a compounding effect. The machine's location is determined by the inaccuracies of its placement on the shop floor. The PCB's location is determined by the inaccuracies of its placement in the PCB fixing jig. The location of the PCB features (vias, pads) is determined by the inaccuracies of the PCB fabrication and layer masking process.

Because of these inaccuracies, determining pad location by absolute methods — in terms of X-Y coordinates from the pick-and-place machine — does not work. The only way to achieve the required accuracy is to actively determine the location of the component on the end of the pick-up nozzle relative to the PCB. This is done by determining the location of the PCB by vision system inspection of the fiducials and extrapolating this location to determine the location of the SMD pads.

An accurate vision system can determine the location of a fiducial to within 0.0007 inch. In the worst case then, the starting inaccuracy of the pick-and-place machine is 0.0017 inch (0.001-inch pick-up nozzle inaccuracy plus 0.007-inch location inaccuracy). Because the leads must be placed within 0.002 inch of the actual pad location, this only leaves 0.003 inch for machine inaccuracies in arm location and in the PCB holding fixture.

The only way to reduce this inaccuracy is to use multiple fiducials, which permit an angular orientation accuracy of ±0.2°. The use of multiple fiducials means that you must use a computationally powerful vision system with interpolation algorithms, which implies high cost and slow fabrication.

Component Spacing

Because of the fine pitch of the EIAJ CQFPs used by Cypress, it is important to recognize the effects that the close tolerance of the PCB pads and vias can have upon the PCB's solderability. The prime objective here is to reduce solder bridging, which occurs when a pad, lead, or via connects to the wrong place, causing either shorting or a path for random circuit effects.

You can control solder bridging by ensuring that the clearance between PCB vias and pads is large enough to prevent solder migration. 0.01-inch air-gap distance between vias and pads is recommended. 0.012 inch is recommended where a 90° via point is adjacent to a pad. At least 0.025 inch should be available between pads.

Careful alignment of the solder mask is also helpful in reducing solder migration. Use a photo-imaged mask coating. Keep the maximum clearance of the solder mask in relation to the pads and PCB vias to 0.005 inch.

References

Traiser, John E. *Design Guidelines for Surface Mount Technology.* Academic Press, Inc., New York, 1990.

Prasad, *Surface Mount Technology Principles and Practice.*

CYPRESS SEMICONDUCTOR

Memory System Design for the CY7C601 SPARC Processor

This application note describes a simple 25-MHz CY7C601 memory design for non-cache-memory applications. The memory subsystem consists of 128 Kbytes of data RAM and 128 Kbytes of instruction RAM. (You can easily expand the instruction RAM to 256 Kbytes using this design.) The difference between data memory and instruction memory is that the CY7C601 integer unit (IU) is not allowed to write to instruction memory. This restriction implies that an external device loads instruction RAM at power-up.

The design utilizes the CY7C157 cache RAM, which is specifically intended for use with the CY7C601 and the CY7C604/605 cache/memory management unit (CMMU). When used in this environment, the CMMU provides all necessary control signals (byte writes and output enables). This article shows that the CY7C157 also adapts easily to non-cache applications.

First, this application note describes the CY7C157, followed by a brief description of the CY7C601 bus interface. Second, a design is presented that uses the CY7C330 EPLD to generate the byte-write signals and the CY7C332 EPLD to provide the output-enable signals. *Figure1* shows the design's block diagram.

CY7C157 Cache RAM

The CY7C157 cache RAM is a very high performance 16K x 16-bit static RAM. This device employs common I/O architecture and a self-timed byte-write mechanism. The self-timed write eliminates the difficult task of generating accurate write strobes in high-speed systems. Address and write-enable inputs load into input registers on the system clock's rising edge. The SRAM provides data-input and -output latches, along with an asynchronous output enable. The CY7C157 is available in 20-, 24-, and 33-ns speed grades. Because a 25-MHz IU requires the slowest device offered, 33 ns, this device is used for the memory system presented here.

CY7C601 Bus Interface

The IU has a 32-bit address bus and can directly address 4 Gbytes of memory. In the cycle prior to use, the IU sends the address bus, data bus, and all memory interface signals (except INULL) unlatched; they should be latched externally before being used (more on this later).

Memory Wait States and Exceptions

The memory design described here needs no wait states, but you can find information on this topic and on memory exceptions in the IU data sheet.

Bus Cycles

Assuming that the system does not contain a floating-point processor or a coprocessor, memory must deal with these bus cycles: instruction fetch, load single, load double, store single, store double, and atomic load/store.

Instruction Fetch

The IU sends out address and control bits at the beginning of the fetch cycle. Remember that you must latch these bits externally. At the end of the fetch cycle, the IU latches instruction data from the data bus into an on-chip instruction register.

The first cycle in *Figure2* illustrates an instruction fetch. Because all instruction fetches are single-cycle

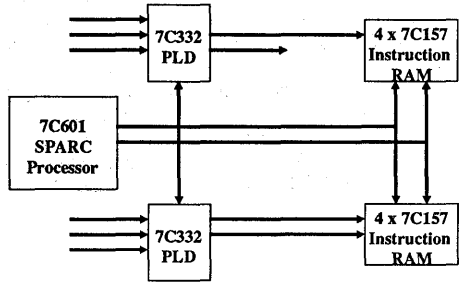


Figure 1. Block Diagram

operations, they incur no pipeline delays. Under some conditions, the processor is unable to fetch an instruction, usually because a prior multi-cycle instruction needs to use the bus. When this occurs, the processor asserts INULL to indicate that the current fetch cycle should be nullified.

Load Cycles

The first and second clock cycles in *Figure 2* show the timing for a load single integer instruction. Load single integer is a two-cycle operation: The first cycle fetches the load instruction, and the second cycle actually loads the required information from memory. A load double instruction is similar to the load single instruction except that a third cycle is added to fetch the second data word from memory. *Figure 2* also illustrates this event.

Store Cycles

Figure 3 illustrates store single and store double instructions. A store single requires three clocks: The store instruction is fetched during the first clock. During the second clock, the destination address of the store is driven onto the bus. Store data is driven onto the data bus at the middle of cycle two and removed at the middle of cycle three. Memory update occurs in cycle three. The store address's early arrival allows it to be checked for possible write-protect violations or memory exceptions in systems that implement these features.

The store double instruction closely resembles a store single instruction, except for an extra cycle needed to store the second data word. Note that the second store's address is set to the first address plus 4, and that the size bits are set to 11, indicating a double-bus access.

Atomic Load/Store Cycles

Atomic transactions consist of two or more transactions that are indivisible; once started, the sequence cannot be interrupted. To ensure bus access for the second transaction, the IU asserts the LOCK signal for the necessary length of time. *Figure 4* shows the timing of an atomic load/store instruction.

Design Considerations

Using the CY7C157s in a non-cache application requires generation of appropriate byte-write signals and output enables. Because the CY7C157 does not require a chip select when used with the CMMU, this design decodes separate sets of write enables for each 64 Kbytes (16 Kword deep) block of RAM. An output enable must also be generated on 16-Kword boundaries during reads. Because address and data set-up/hold requirements between the IU and the CY7C157 are guaranteed by design, you can concentrate on the write-enable and output-enable timing requirements of the CY7C157-33.

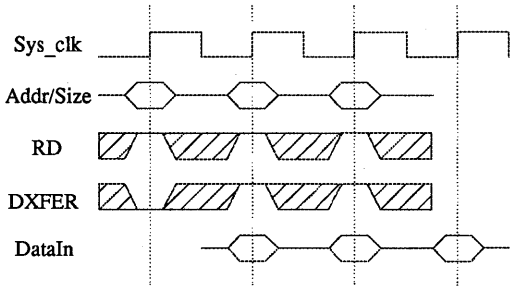
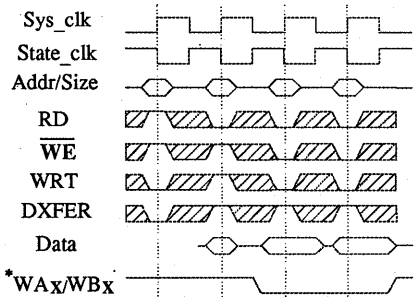


Figure 2. Load/Load Double Timing

The CY7C157 requires a 6-ns write-enable-set-up-to-clock-Low time and 3 ns write-enable hold from clock Low. From the store transaction timing diagrams, you can see that the store data valid times are referenced to the system clock's falling edge, while transaction information (address, size, etc.) is referenced to the same clock's rising edge. The desired PLD architecture for the write-enable generator must provide one clock for clocking in the transaction information and a separate clock for clocking out the write enables. The Cypress CY7C330 state machine can handle this task.

The next critical factor is: Can the CY7C330 meet the write enable set-up and hold times? Inspection of the CY7C330-50WC data sheet for t_{CO} and t_{OH} specs indicates that the device meets these conditions. *Figure 5* shows that any write enable is valid 15 ns after Sys_Ck's falling edge (thus providing a 25-ns set-up time) and is held for 3 ns after Sys_Ck's falling edge (matching the required hold time at the CY7C157).



* Signal from 7C330 PLD

Figure 3. Store/Store Double Timing

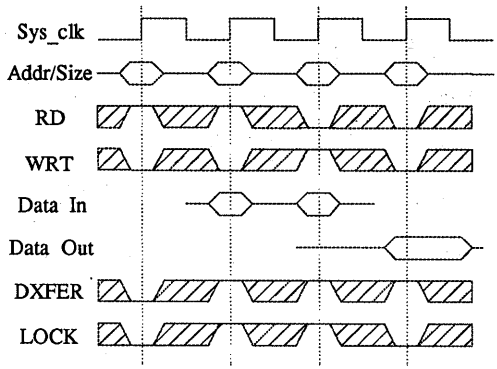


Figure 4. Atomic Load/Store Timing

For reads, *Figure5* shows that the CY7C332 output delay plus the CY7C157 output-enable time provides a 5-ns data set-up time, which easily meets the IU's 3-ns requirement. Data hold time requirements are determined by examining the CY7C332 output-enable hold time from Sys_Ck's falling edge. This hold time is 3 ns, which, when added to a 2-ns minimum turn-off time for the CY7C157, guarantees the required 5-ns data hold time at the IU.

CY7C330 Write-Enable Design

The signals required to generate the byte-write signals appear in *Table 1*. The signals are defined as follows:

State_Clock: the inverted version of System_Clock. State_Clock drives the state registers in the CY7C330 PLD.

System_Clock: the clock that drives the IU and CY7C330 input registers. All transaction information is valid on System_Clock's rising edge.

Advanced Write: The processor asserts (sets to 1) Advanced Write (WRT) during the first data cycle of single or double integer store instructions and during the second cycle of atomic load/store instructions. WRT is send out unlatched and must be latched externally before it is used.

Size(1:0): These two bits specify the data size associated with all transactions on the data bus. The IU sends out size bits unlatched. The value of these bits indicates the data size corresponding to the current cycle's memory address. The size bits are valid at the same time as the address bus. Because all instructions are 32 bits long, Size(1:0) is set to 10 during all instruction fetch cycles. Encoding of the size bits is shown in *Table 2*.

Address (1:0), Address 14: Address (1:0) decodes individual byte-write lines for writes within a 32-bit word boundary. The CY7C330 design described here also uses these lines to inhibit writes on unaligned boundaries; you can easily modify this feature to generate a memory exception. Address 14 selects between Bank A write enables (lower 16 Kwords) and Bank B write enables (upper 16 Kwords) for the data RAMs. The address is sent out unlatched and must be latched externally before use. If the address output enable (/AOE) or test output enable (/TOE) signals are deasserted, the address bus three-states.

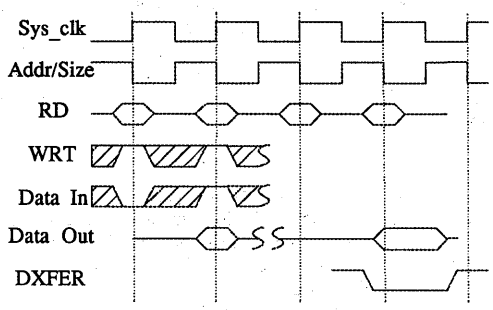


Figure 5. Actual Timing

INULL: occurs on two occasions. First, it always occurs during the second cycle of a store transaction to tell the memory subsystem that the current memory transaction has proceeded too far to be nullified; i.e., it is too late to initiate a wait state or memory exception. Second, INULL can occur during a transaction's first cycle to tell the memory subsystem to ignore the transaction entirely. This signal is of consequence only for store transactions that must be inhibited before the write occurs.

/Reset: an active-Low input to the CY7C330 PLD that forces all outputs to the inactive state. It is a clocked reset.

Table 1. Byte Write Signals

Name	Mnemonic
State_Clock	St_Ck
System_Clock	Sys_Ck
Advanced Write	WRT
Size(1:0)	Size1, Size0
Adr(1:0)	A1, A0
Adr14	A14
INULL	INULL
/Reset	!Rst
/Output Enable	!OE
/Write Enables - Bank A	!WA3 - !WA0
/Write Enables - Bank B	!WB3 - !WB0

Table 2. Size bit encoding

Size(1:0)	Transaction Type
00	Byte
01	Halfword
10	Word
11	Double Word

/OE: an active-Low input to the CY7C330 PLD that enables all the device's outputs. When High, all CY7C330 outputs are three-stated. The source file containing the PLD equations for the CY7C330 write-enable generator appears in *Appendix A.*

CY7C332 Output-Enable Design

Table 3 lists the signals used to generate the required output-enable signals. *Appendix B* shows the output-enable circuit's design file, implemented for the CY7C332 PLD using the Cypress PLD ToolKit. The PLD ToolKit is an assembler/simulator package for PLDs.

The design utilizes a CY7C332 to generate five instruction output enables and five data output enables for a Cypress SPARC-based, non-cached memory system. Each output enable is decoded on a 16-Kword boundary (word = 32 bits). The CY7C332 suits this application especially well, because this one PLD incorporates input latch/registers with output decoding. When combined with a CY7C330 programmed as a write-enable generator, complete memory control is achieved in just two PLDs.

Table 3. Output Enable Signals

Name	Mnemonic
System_Clock	Sys_Ck
Size(1:0)	Size1, Size0
Adr(16:14)	A16,A15,A14
INULL	INULL
/Reset	!Rst
/Output Enable-> 332	!OE
/Output Enables - Inst Bank	!IOE4 - !IOE0
/Output Enables - Data Bank	!DOE4 - !DOE0
Inst Fetch Mem Exception	!IFMEMx

Table 4. Memory Subsystem Characteristics

Component	Quantity	Power
CY7C157-33	8	1.375 W
CY7C330-50	1	0.99 W
CY7C332-20	1	0.99 W
TOTAL	**10**	**13.0 W**

Pin 1 of the CY7C332 is the system clock, active on the rising edge. Pins 2 - 4 are address bits 16 - 14, which are used in the output-enable decoding. Pins 5 and 6 are the IU size bits.

For instruction fetches, if SIZE does not equal 10B (see *Table 2*), then IFMEMx is made active. The SIZE bits are ignored for data fetches, because all alignment occurs in the IU.

RD = 1 signifies that the following cycle is a read cycle. DXFER = 1 signals that the following cycle is a data transfer. Conversely, if DXFER = 0, the next cycle is a non-data (instruction) cycle. The INULL signal is not needed here, because the CPU ignores instruction/data fetched in the next cycle anyway. DOEx and IOEx are the data output enables and instruction output enables, respectively. IFMEMx occurs when an instruction fetch is attempted with SIZE not equal to 10 (one word).

Conclusions

The design presented here provides 128 Kbytes of instruction memory and 128 Kbytes of data memory with just ten components (eight CY7C157's, one CY7C330, and one CY7C332). *Table 4* tabulates some of the memory subsystem's key characteristics.

You can easily expand the memory subsystem's capacity by using the CY7C330's four additional outputs as write enables. This change furnishes another 64 Kbytes of data memory. The CY7C332 design already provides output enables for 320 Kbytes of data memory and 320 Kbytes of instruction memory.

For systems requiring even larger memory spaces, you can make a tradeoff with the CY7C330. If the smallest write boundary is changed to half word (16 bits) instead of byte, the CY7C330 can provide byte writes for 384 Kbytes of data memory. Similarly, for systems requiring only 32-bit writes to data memory, a single CY7C330 can provide the required write enables for 768 Kbytes of memory. However, this configuration requires an additional CY7C332 to decode output enables for data memory reads.

Appendix A. ABEL CY7C330 Write Enable PLD Equations

```
Module SPARC_WRTENB flag '-r3'
title                                   'SPARC Write Enable Generator'
LIBRARY 'P330';                         "Enable various useful macros
IC device 'P330';

St_Ck,Sys_Ck,INULL, Rst                 Pin 1, 2, 10, 13;"Inputs
WRT,Size1,Size0,A1,A0,A14               Pin 3, 4, 5, 6, 7, 9;
Reset, Set                              node 29, 30; "Outputs and Internal Node declarations.
!WA3,!WA2,!WA1,!WA0                     Pin 28, 27, 26, 25;
,!WB3,!WB2,!WB1,!WB0                    Pin 24, 23, 20, 19;
!OE                                     Pin 14;

!WA3.OE                 istype 'Pin';   "Enable pin 14 as common OE for all outputs

SIZE = [Size1,Size0];
ADR = [A1, A0];                         "Definitions for readability and test vector generation
WA = [ WA3,WA2,WA1,WA0 ];
WB = [ WB3,WB2,WB1,WB0 ];

H,L,C,X,Z = 1,0,.C.,.X.,.Z.;            "Declarations

equations

WA3.OE = !OE;                           "Turn on outputs

WA3 :=      !Rst & !INULL & !A14 & WRT & (SIZE == 0) & (ADR == 3)
            # !Rst &!INULL &!A14 & WRT & (SIZE == 1) & (ADR == 2)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 2) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 3) & (ADR == 0)
            # !Rst & !A14 & (SIZE == 3) & (ADR == 0) & WA3;

WA2 :=      !Rst & !INULL & !A14 & WRT & (SIZE == 0) & (ADR == 2)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 1) & (ADR == 2)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 2) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 3) & (ADR == 0)
            # !Rst & !A14 & (SIZE == 3) & (ADR == 0)& WA2;

WA1 :=      !Rst & !INULL & !A14 & WRT & (SIZE == 0) & (ADR == 1)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 1) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 2) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 3) & (ADR == 0)
            # !Rst & !A14 & (SIZE == 3) & (ADR == 0) & WA1;

WA0 :=      !Rst & !INULL & !A14 & WRT & (SIZE == 0) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 1) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 2) & (ADR == 0)
            # !Rst & !INULL & !A14 & WRT & (SIZE == 3) & (ADR == 0)
            # !Rst & !A14 & (SIZE == 3) & (ADR == 0)& WA0;
```

Appendix A. ABEL CY7C330 Write Enable PLD Equations (Continued)

```
WB3 :=    !Rst & !INULL & A14 & WRT & (SIZE == 0) & (ADR == 3)
          # !Rst & !INULL & A14 & WRT & (SIZE == 1) & (ADR == 2)
          # !Rst & !INULL & A14 & WRT & (SIZE == 2) & (ADR == 0)
          # !Rst & !INULL & A14 & WRT & (SIZE == 3) & (ADR == 0)
          # !Rst & A14 & (SIZE == 3) & (ADR == 0)& WB3;

WB2 :=    !Rst & !INULL & A14 & WRT & (SIZE == 0) & (ADR == 2)
          # !Rst & !INULL & A14 & WRT & (SIZE == 1) & (ADR == 2)
          # !Rst & !INULL & A14 & WRT & (SIZE == 2) & (ADR == 0)
          # !Rst & !INULL & A14 & WRT & (SIZE == 3) & (ADR == 0)
          # !Rst & A14 & (SIZE == 3) & (ADR == 0)& WB2;

WB1 :=    !Rst & !INULL & A14 & WRT & (SIZE == 0) & (ADR == 1)
          # !Rst & !INULL & A14 & WRT & (SIZE == 1) & (ADR == 0)
          # !Rst & !INULL & A14 & WRT & (SIZE == 2) & (ADR == 0)
          # !Rst &!INULL & A14 & WRT & (SIZE == 3) & (ADR == 0)
          # !Rst & A14 & (SIZE == 3) & (ADR == 0)& WB1;

WB0 :=    !Rst & !INULL & A14 & WRT & (SIZE == 0) & (ADR == 0)
          # !Rst & !INULL & A14 & WRT & (SIZE == 1) & (ADR == 0)
          # !Rst & !INULL & A14 & WRT & (SIZE == 2) & (ADR == 0)
          # !Rst &!INULL & A14 & WRT & (SIZE == 3) & (ADR == 0)
          # !Rst & A14 & (SIZE == 3) & (ADR == 0)& WB0;
```

Appendix A. ABEL CY7C330 Write Enable PLD Equations (Continued)

```
"Test vectors for WA outputs, WB outputs are similar except for A14
"Note that the WA outputs are treated as active-high in the test vectors
"since they were declared as active-low in the pin declaration sections.

Test_vectors

([!OE,!Rst,St_Ck,Sys_Ck,WRT,INULL,SIZE,ADR,A14]  ->  [WA,WB]);
[ 0,0, 0, 0, X, X, X, X, X ] -> [ X,X ];
[ 0,0, 0, 1, X, X, X, X, X ] -> [ X,X ];
[ 0,0, 1, 0, X, X, X, X, X ] -> [ 0,0 ];                    "v1 Reset

"WRT = 0 = WAx inactive
[ 0,1, 0, 1, 0, 0, X, X, 0 ] -> [ 0,0 ];
[ 0,1, 1, 0, 0, 0, X, X, 0 ] -> [ 0,0 ];

"Halfword transactions to lower word (bytes 1:0)
[ 0,1, 0, 1, 1, 0, 1, 0, 0 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 1, 0, 0 ] -> [ 03,0 ];

"Halfword write on byte boundary results in IU generated alignment error.
[ 0,1, 0, 1, 1, 0, 1, 1, 0 ] -> [ 03,0 ];
[ 0,1, 1, 0, 1, 0, 1, 1, 0 ] -> [ 00,0 ];                    "v10

"Halfword write to upper word
[ 0,1, 0, 1, 1, 0, 1, 1, 0 ] -> [ 03,0 ];
[ 0,1, 1, 0, 1, 0, 1, 1, 0 ] -> [ 00,0 ];                    "v10

"Halfword write to upper word
[ 0,1, 0, 1, 1, 0, 1, 2, 0 ] -> [ 00,0 ];
[ 0,1, 1, 0, 1, 0, 1, 2, 0 ] -> [ 0c,0 ];
[ 0,1, 0, 1, 0, 0, 1, X, 0 ] -> [ 0c,0 ];
[ 0,1, 1, 0, 0, 0, 1, X, 0 ] -> [ 00,0 ];

"Word write on byte bndary results in IU generated alignment "error
[ 0,1, 0, 1, 1, 0, 1, 3, 0 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 1, 3, 0 ] -> [ 0,0 ];

"Verify WA follows byte writes correctly [!OE,!Rst,StCk,SyCk,W,I,S,ADR,A14]
[ 0,1, 0, 1, 1, 0, 0, 3, 0 ] -> [ 0,0 ];                     "v20
[ 0,1, 1, 0, 1, 0, 0, 3, 0 ] -> [ 08,0 ];
[ 0,1, 0, 1, 1, 0, 0, 2, 0 ] -> [ 08,0 ];                    "wrt byte 3
[ 0,1, 1, 0, 0, 0, 0, 2, 0 ] -> [ 04,0 ];
[ 0,1, 0, 1, 1, 0, 0, 1, 0 ] -> [ 04,0 ];                    "wrt byte 2
[ 0,1, 1, 0, 0, 0, 0, 1, 0 ] -> [ 02,0 ];
[ 0,1, 0, 1, 1, 0, 0, 0, 0 ] -> [ 02,0 ];                    "wrt byte 1
[ 0,1, 1, 0, 0, 0, 0, 0, 0 ] -> [ 01,0 ];
[ 0,1, 0, 1, 0, 0, 0, 0, 0 ] -> [ 01,0 ];                    "wrt byte 0
[ 0,1, 1, 0, 0, 0, 0, 0, 0 ] -> [ 00,0 ];                    "writes are inactive

"Verify single store works correctly [!OE,!Rst,StCk,SyCk,W,I,S,ADR,A14] for ease of programming only
[ 0,1, 0, 1, 1, 0, 2, 0, 0 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 2, 0, 0 ] -> [ 0f,0 ];
[ 0,1, 0, 1, 0, 0, 0, X, 0 ] -> [ 0f,0 ];
[ 0,1, 1, 0, 0, 0, 0, X, 0 ] -> [ 0,0 ];                     "v30
```

Appendix A. ABEL CY7C330 Write Enable PLD Equations (Continued)

```
"Verify WA responds correctly to double stores
[ 0,1, 0, 1, 1, 0, 3, X, 0 ] -> [ 0 ,0 ];
[ 0,1, 1, 0, 1, 0, 3, X, 0 ] -> [ 0f,0 ];
[ 0,1, 0, 1, 0, 0, 3, X, 0 ] -> [ 0f,0 ];
[ 0,1, 1, 0, 0, 0, 3, X, 0 ] -> [ 0f,0 ];
[ 0,1, 0, 1, 0, 0, 2, X, 0 ] -> [ 0f,0 ];
[ 0,1, 1, 0, 0, 0, 2, X, 0 ] -> [ 0 ,0 ];

"Do the same thing for the WB outputs (no comments)
[ 0,0, 0, 0, X, 0, X, X, X ] -> [ X,X ];
[ 0,0, 0, 1, X, 0, X, X, X ] -> [ X,X ];
[ 0,0, 1, 0, X, 0, X, X, X ] -> [ 0,0 ];                    "v1 Reset

"WRT = 0 = WA/B inactive
[ 0,1, 0, 1, 0, 0, X, X, 1 ] -> [ 0,0 ];
[ 0,1, 1, 0, 0, 0, X, X, 1 ] -> [ 0,0 ];

"Halfword transactions to lower word (bytes 1:0)
[ 0,1, 0, 1, 1, 0, 1, 0, 1 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 1, 0, 1 ] -> [ 0,03 ];

"Halfword write on byte boundary - occurence results in IU generated alignment error.
[ 0,1, 0, 1, 1, 0, 1, 1, 1 ] -> [ 0,03 ];                   "v10
[ 0,1, 1, 0, 1, 0, 1, 1, 1 ] -> [ 0,0 ];

"Halfword write to upper word
[ 0,1, 0, 1, 1, 0, 1, 2, 1 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 1, 2, 1 ] -> [ 0,0c ];
[ 0,1, 0, 1, 0, 0, 1, X, 1 ] -> [ 0,0c ];
[ 0,1, 1, 0, 0, 0, 1, X, 1 ] -> [ 0,0 ];

"Word write on byte boundary results in IU generated alignment "error
[ 0,1, 0, 1, 1, 0, 1, 3, 1 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 1, 3, 1 ] -> [ 0,0 ];

"Verify WB follows byte writes correctly
[!OE,!Rst,StCk,SyCk,W,I,S,ADR,A14]
[ 0,1, 0, 1, 1, 0, 0, 3, 1 ] -> [ 0,0 ];                    "v20
[ 0,1, 1, 0, 1, 0, 0, 3, 1 ] -> [ 0,08 ];
[ 0,1, 0, 1, 1, 0, 0, 2, 1 ] -> [ 0,08 ];                   "wrt byte 3
[ 0,1, 1, 0, 0, 0, 0, 2, 1 ] -> [ 0,04 ];
[ 0,1, 0, 1, 1, 0, 0, 1, 1 ] -> [ 0,04 ];                   "wrt byte 2
[ 0,1, 1, 0, 0, 0, 0, 1, 1 ] -> [ 0,02 ];
[ 0,1, 0, 1, 1, 0, 0, 0, 1 ] -> [ 0,02 ];                   "wrt byte 1
[ 0,1, 1, 0, 0, 0, 0, 0, 1 ] -> [ 0,01 ];
[ 0,1, 0, 1, 0, 0, 0, 0, 1 ] -> [ 0,01 ];                   "wrt byte 0
[ 0,1, 1, 0, 0, 0, 0, 0, 1 ] -> [ 0,0 ];                    "writes are inactive

"Verify single store works correctly [!OE,!Rst,StCk,SyCk,W, I, S,ADR,A14] for ease of programming only
[ 0,1, 0, 1, 1, 0, 2, 0, 1 ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 0, 2, 0, 1 ] -> [ 0,0f ];
[ 0,1, 0, 1, 0, 0, 0, X, 1 ] -> [ 0,0f ];
[ 0,1, 1, 0, 0, 0, 0, X, 1 ] -> [ 0,0 ];
```

Appendix A. ABEL CY7C330 Write Enable PLD Equations (Continued)

```
"Verify WB responds correctly to double stores
[ 0,1, 0, 1, 1, 0, 3, X, 1 ] -> [ 0 ,0 ];
[ 0,1, 1, 0, 1, 0, 3, X, 1 ] -> [ 0, 0f ];
[ 0,1, 0, 1, 0, 0, 3, X, 1 ] -> [ 0, 0f ];
[ 0,1, 1, 0, 0, 0, 3, X, 1 ] -> [ 0, 0f ];
[ 0,1, 0, 1, 0, 0, 2, X, 1 ] -> [ 0, 0f ];
[ 0,1, 1, 0, 0, 0, 2, X, 1 ] -> [ 0 ,0 ];

"Check that all WA's and WB's are inhibited when INULL occurs with WRT
[ 0,0, 0, 0, X, X, X, X, X ] -> [ X,X ];
[ 0,0, 0, 1, X, X, X, X, X ] -> [ X,X ];                    "v1 Reset
[ 0,0, 1, 0, X, X, X, X, X ] -> [ 0,0 ];
[ 0,1, 0, 1, 1, 1, 0, 3, X ] -> [ 0,0 ];
[ 0,1, 1, 0, 1, 1, 0, 3, X ] -> [ 0,0 ];
[ 0,1, 0, 1, 1, 1, 0, 2, X ] -> [ 0,0 ];                    "write byte 3
[ 0,1, 1, 0, 0, 1, 0, 2, X ] -> [ 0,0 ];
[ 0,1, 0, 1, 1, 1, 0, 1, X ] -> [ 0,0 ];                    "write byte 2
[ 0,1, 1, 0, 0, 1, 0, 1, X ] -> [ 0,0 ];
[ 0,1, 0, 1, 1, 1, 0, 0, X ] -> [ 0,0 ];                    "write byte 1
[ 0,1, 1, 0, 0, 1, 0, 0, X ] -> [ 0,0 ];
[ 0,1, 0, 1, 0, 1, 0, 0, X ] -> [ 0,0 ];                    "write byte 0
[ 0,1, 1, 0, 0, 1, 0, 0, X ] -> [ 0,0 ];                     "writes are inactive

"Double stores
[ 0,1, 0, 1, 1, 1, 3, X, X ] -> [ 0 ,0 ];
[ 0,1, 1, 0, 1, 1, 3, X, X ] -> [ 0, 0 ];                   "Inactive
[ 0,1, 0, 1, 0, 1, 3, X, X ] -> [ 0, 0 ];
[ 0,1, 1, 0, 0, 1, 3, X, X ] -> [ 0, 0 ];
[ 0,1, 0, 1, 0, 1, 2, X, X ] -> [ 0, 0 ];
[ 0,1, 1, 0, 0, 1, 2, X, X ] -> [ 0 ,0 ];

" MORE REALISTIC OCCURANCE OF DOUBLE STORE INULL
[ 0,1, 0, 1, 1, 0, 3, X, 0 ] -> [ 0 ,0 ];
[ 0,1, 1, 0, 1, 0, 3, X, 0 ] -> [ 0f, 0 ];
[ 0,1, 0, 1, 0, 1, 3, X, 0 ] -> [ 0f, 0 ];
[ 0,1, 1, 0, 0, 1, 3, X, 0 ] -> [ 0f, 0 ];
[ 0,1, 0, 1, 0, 0, 2, X, 0 ] -> [ 0f, 0 ];
[ 0,1, 1, 0, 0, 0, 2, X, 0 ] -> [ 0 ,0 ];

"Double stores
[ 0,1, 0, 1, 1, 0, 3, X, 1 ] -> [ 0 ,0 ];
[ 0,1, 1, 0, 1, 0, 3, X, 1 ] -> [ 0 ,0f ];
[ 0,1, 0, 1, 0, 1, 3, X, 1 ] -> [ 0 ,0f ];
[ 0,1, 1, 0, 0, 1, 3, X, 1 ] -> [ 0 ,0f ];
[ 0,1, 0, 1, 0, 0, 2, X, 1 ] -> [ 0 ,0f ];
[ 0,1, 1, 0, 0, 0, 2, X, 1 ] -> [ 0 ,0 ];

end  SPARC_WRTENB
```

Appendix B. PLD ToolKit Source Code for CY7C332 Write Enable

```
CY7C332;

CONFIGURE;

Sys_Ck,                                          {Pin 1 }
A16(ireg), A15(ireg), A14(ireg),                 {Pins 2 thru 4}
SIZE1(ireg), SIZE0(ireg),                        {Pins 5 and 6}
RD(ireg), DXFER(node = 9,ireg),                  {Pins 7 and 9}
                                                 {Pin 8 is GND}
!OE(node = 14),                                  {Pin 14 is out enb}
!DOE0(nenbpt),!DOE1(nenbpt),!DOE2(nenbpt),       {Pin 15 thru..}
!DOE3(nenbpt),!DOE4(nenbpt),                     {..19 }
!IFMEMx(node = 23,nenbpt),                       {Inst Fetch Mem Excp}
!IOE0(nenbpt),!IOE1(nenbpt),!IOE2(nenbpt),       {Pins 24 thru..}
!IOE3(nenbpt),!IOE4(nenbpt),                     {..28}

EQUATIONS;

IOE4 = RD & !DXFER & SIZE1 & !SIZE0 & !A16 & !A15 & !A14;
                                                 {A= 000}

IOE3 = RD & !DXFER & SIZE1 & !SIZE0 & !A16 & !A15 & A14;
                                                 {A = 001}

IOE2 = RD & !DXFER & SIZE1 & !SIZE0 & !A16 & A15 & !A14;
                                                 {A = 010}

IOE1 = RD & !DXFER & SIZE1 & !SIZE0 & !A16 & A15 & A14;
                                                 {A = 011}

IOE0 = RD & !DXFER & SIZE1 & !SIZE0 & A16 & !A15 & !A14;
                                                 {A = 100}

                                                 { Recall that for Inst Fetches only SIZE(1:0) = '10' is allowed}
IFMEMx =    RD & !DXFER & !SIZE1 & !SIZE0        {SZ = 00}
          & RD & !DXFER & !SIZE1 & SIZE0         {SZ = 01}
          & RD & !DXFER & SIZE1 & SIZE0;         {SZ = 11}

                                        {DOE's do not depend on SIZE bits, since IU does alignment internally}
DOE4 = RD & DXFER & !A16 & !A15 & !A14;          {A = 000}

DOE3 = RD & DXFER & !A16 & !A15 & A14;           {A = 001}

DOE2 = RD & DXFER & !A16 & A15 & !A14;           {A = 010}

DOE1 = RD & DXFER & !A16 & A15 & A14;            {A = 011}

DOE0 = RD & DXFER & A16 & !A15 & !A14;           {A = 100}
```

CYPRESS SEMICONDUCTOR

Cache Memory Design

The purpose of this application note is to provide a general understanding of the attributes and engineering tradeoffs of various cache designs. The first section discusses the cache design goal and methods of achieving that goal. Next, several major cache design factors are described, with an explanation of each factor's advantages and disadvantages. The application note then explores the conditions for and techniques used in design of multilevel cache for uniprocessor and multiprocessor environments.

The first commercial use of a cache memory was in 1969, the year IBM introduced the IBM 360/85. Since that time, cache memory has spread from mainframes to minicomputers to microcomputers, thus becoming an accepted design technique for a broad range of computing machines.

Cache memory is an engineering solution to unacceptably high main-memory access times relative to CPU cycle time — a difference so great that main-memory access time was severely limiting overall machine performance. The cache acts as a small, high-speed buffer between the CPU and main memory. This buffer is hidden from the outside world; thus the name cache.

If designed properly, the cache makes the machine appear to have a large amount of very fast main memory. As an example of the effectiveness of this approach, consider high-end machines such as the Amdahl 580 or IBM 3090. This caliber of machine has a main-memory access time of 200 - 500 ns and a cache access time of 20 - 50 ns, yielding an effective memory access time of 30 - 100 ns — a 5 to 7x increase in memory performance.

The use of cache memory has become very widespread, as evidenced by cache being directly supported or included on-chip in a variety of microprocessors: the National Semiconductor 32000 family, the Motorola 68000 family, the Intel 80386 and 80486, and all of the currently available RISC families, such as the Cypress CY7C600 SPARC family.

Cache Design

The objective of cache design is to reduce the effective (or average) memory access time to an acceptable level that is generally determined from cost/performance tradeoff analysis. You can achieve this goal by realizing that most processor reference streams are both highly sequential and highly loop oriented.

Therefore, a cache operates on the principle of spatial and temporal locality of reference. Spatial locality means that information the CPU will reference in the near future is likely to be logically close in main memory to information that is currently being referenced. Temporal locality means that information the CPU is currently referencing is likely to be referenced again in the near future. Through these mechanisms, you can design a cache to ensure a high probability that CPU references are located in the cache.

Spatial locality of reference is serviced in the following manner: If a cache miss occurs (the cache does not contain information requested by the CPU), the cache accesses main memory and retrieves the information currently being requested, as well as several additional locations that logically follow the current reference. This set of information is called a line or block. The next CPU reference now has a high statistical probability of being serviced by the cache, thus avoiding main memory's relatively long access time.

Temporal locality of reference is serviced by allowing information to remain in the cache for an extended period of time, only replacing the line in order to make room for a new one. You can use several algorithms to manage cache line replacement (more on this later). By allowing the information to remain in the cache and with a sufficient cache size, an entire loop of code can fit into the cache, allowing very high speed execution of instructions in the loop.

The Cache Design Goal

The goal of a cache design is to reduce the effective memory access time as seen by the CPU. Effective access time can be expressed as:

$$t_{eff} = t_{cache} + m \times t_{main}$$

where:

t_{cache} = Effective hit time of cache (i.e., cache access time)

m = Miss rate of cache

t_{main} = Main-memory access time (penalty beyond t_{cache} for main-memory accesses)

Thus, design of a cache revolves around:

- Minimizing the time for the cache to service a hit
- Maximizing the hit rate (hit rate = 1 - miss rate)
- Minimizing the delay due to a cache miss (included in t_{main})
- Minimizing the overhead delay associated with keeping main memory coherent — in agreement with the data in the cache — especially in multicache configurations (included in t_{main})

Generally, all these factors are affected in some way by the design parameters discussed below. To simplify the overall design process, you might find it useful to view cache design from the following macroarchitecture viewpoints, each of which can be broken into one or more microarchitectural parameters:

- Cache placement
  - physical vs. virtual cache
- Cache organization
  - cache mapping method
  - cache size
  - cache line size
  - split cache vs. combined cache
- Cache management
  - main-memory coherence schemes
  - line-replacement algorithms
  - fetching algorithms

The next several sections examine the microarchitectural aspects of these factors in detail, giving the performance tradeoffs relative to the four cache performance factors identified above. After describing these design parameters, this application note pulls together the critical parameters of cache design and presents a method of calculating estimates for effective cycle time.

Cache Placement

Along with the cache, an address translation unit — usually called a Memory Management Unit (MMU) — resides between the CPU and main memory. The MMU maps the virtual addresses generated by a program and used by the CPU to the physical addresses used to access main memory.

Figure 1 shows two ways of arranging the cache and MMU. You choose between these two approaches by answering the question: Where in the system should the MMU delay occur?. Traditionally, caches have been referenced with physical addresses, as in *Figure 1a*. A physical cache is easier to manage but slower than a virtual cache, which is referenced by virtual addresses as in *Figure 1b*.

A physical cache is slower because t_{cache} includes the address translation time; thus, the translation delay occurs on every memory reference. A virtual cache allows address translation to occur in parallel with cache access, thereby shifting the translation time penalty from t_{cache} to t_{main}. This significantly reduces the translation time's overall negative impact on t_{eff} if the hit rate is high.

The disadvantage of a virtual cache is that it is more difficult to manage, because support must be included to detect and correct aliases (or synonyms). Aliasing occurs when two virtual addresses translate to the same physical address. This situation can occur, for example, when two different programs in the CPU share pages placed in different locations in the two programs' respective address maps.

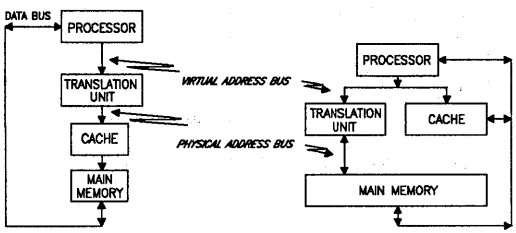
Aliasing can be detected and corrected in a number of ways. The most complete solution is to employ a set of virtual cache tags (cache tags are explained later) and a set of physical cache tags, which are used as cross-references to detect and prevent aliasing. The CY7C605 SPARC CMU-MP uses this methodology.

Another, less elegant, aliasing solution is to use an operating system detector that either forces shared data to the same cache line or marks shared data as non-cacheable. The CY7C604 SPARC CMU uses this technique.

Either of the two aliasing solutions described here allow you to take advantage of a virtual cache's faster response. As higher processor speeds place greater demands on cache systems, virtual caching schemes will become more popular.

On the other hand, system designers might not have to choose between physical and virtual cache much longer because, as integration levels increase, more and more microprocessors become available with on-board cache. In fact, several CISC (Complex Instruction Set Computer) chips already contain on-board cache (32000, 68030, 80486), and several RISC architectures have been proposed or introduced as a single chip with on-board cache. As a result, virtual cache vs. physical cache is likely to become a silicon design issue, with system-level designers focusing on methods of designing an efficient second-level cache to back up relatively small on-board cache.

In the event of a multilevel cache hierarchy, you might not have the option of choosing where to place the cache. If the cache is on the processor chip, it will probably force a physical level 2 cache. It is also probable that a



1a. Physical Cache System 1b. Virtual Cache System

Figure 1. Cache Placement

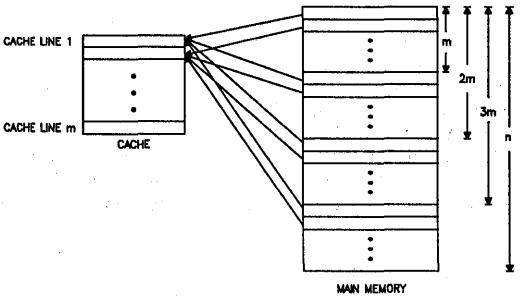


Figure 2. Direct Mapping

multiple-chip processor family will be partitioned such that it forces a physical level 2 cache.

Cache Organization

Cache organization has four basic parameters: cache mapping method, cache size, cache line size, and split vs. combined cache. Note that for a multilevel cache hierarchy, the organizational tradeoffs associated with cache size, cache mapping method, and cache line size are multidimensional. This is because choices made for the level 1 cache are likely to affect the performance of the level 2 cache and vice versa.

Because a cache can be viewed as a small moving window into portions of a larger main memory, main memory locations must be mapped to and from locations in the cache. The type of mapping you use affects both cache hit time and miss rate. Generally, an increase in hit rate exacts a penalty on cache hit time. However, recent research supports the idea that if a cache is sufficiently large, the relative difference in miss rate for various mapping methods becomes very small. This indicates that a sufficiently large cache should be mapped according to the scheme that exacts the least penalty on cache hit time.

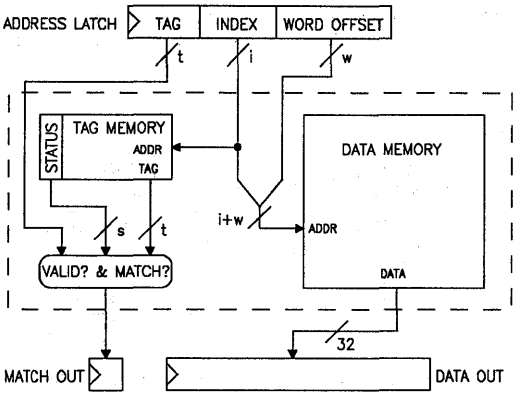


Figure 3. A Direct-Mapped Cache

The most widely used mapping schemes are based on the principle of associativity. A fully associative cache allows any location in main memory to be mapped to any location in the cache. An *n*-way set-associative cache (typically $n = 2, 4, 8$, etc.) allows any specific location to be mapped to n locations in the cache. A direct-mapped cache allows any specific location in main memory to be mapped to only one location in the cache; this scheme thus implements a 1-way set-associative cache. The following discussion details each technique, beginning with the least complex (direct-mapped) and finishing with the most complex (fully associative).

Direct Mapping

Figure 2 illustrates direct mapping. Each location in main memory maps to a unique location in the cache. For instance, location 1 in main memory maps to location 1 in the cache. Location 2 in main memory maps to location 2 in the cache. Location m in main memory maps to location m in the cache. Location $m+1$ in main memory maps to location 1 in the cache, etc.

A simplistic direct-mapped cache implementation appears in *Figure 3*. A direct-mapped cache consists of a data memory, a tag memory, and a comparator. The data memory contains the cached data and instructions; its size is defined as the cache size. The tag memory uses the comparator to determine whether the cache contains the line being addressed by the processor.

The memory address is split into three fields: a tag field, an index field, and a word-offset field. The tag field consists of the address's higher order bit. The index field addresses the tag memory to see if the line being accessed is the line the processor wants. This mechanism ensures that, for example, data from the desired cache location $2m+4$ is retrieved instead of data from cache location $4m+4$, which would reside in the same location in the cache. The line size is defined as the basic unit of transfer between the cache and main memory and is typically an even binary amount such as 16, 32, 64, or 128 bytes.

The number of bits in each field of the address can be deciphered as follows:

$i = \log_2$ (*# cache tag entries*)

$w = \log_2$ (*line size*)

$i + w = \log_2$ (*cache size*)

When an address is presented to the cache, the bits of the index field address the tag memory. The tag in the location addressed by the index field is presented at the tag memory's outputs. This tag is compared with the reference tag, while the cache subsystem also checks to see that its status bits (i.e., VALID, DIRTY, etc., explained later) are in appropriate states.

In parallel with the tag access and status check, $i+w$ bits are used to address the data memory; the accessed word is placed in the DATA OUT buffer. If the tags match and the status bits are all correct, the cache subsystem asserts the MATCH OUT signal, indicating that the information retrieved from the data memory is correct (a cache hit). If the tags do not match or the status bits are

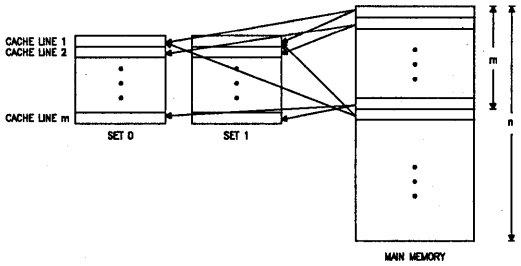


Figure 4. Set Associative Mapping

not correct, MATCH OUT is de-asserted (indicating that the data in DATA OUT is invalid and thus represents a cache miss), and the correct data is retrieved from main memory.

Consequently, a direct-mapped cache has two critical timing paths:

1. Read-data: accessing the data memory and passing the word to the DATA OUT register.
2. Asserting the MATCH OUT signal if the status bits are OK and the retrieved tag matches the reference tag.

Accordingly, the slower of paths 1 and 2 limit a direct-mapped cache's access time.

Set-Associative Cache Mapping

Figure 4 illustrates how set-associative mapping works for the two-way set-associative case. The cache consists of two sets, or banks, of memory cells, each containing *m* lines. Location 1 in main memory maps to cache line 1 of either set. Location 2 in main memory maps to cache line 2 of either set. Location *m* in main memory maps to cache line *m* of either set. Location *m+1* in main memory maps to cache line 1 of either set. Location *m+2* in main memory maps to cache line 2 of either set, and so on.

In this manner, each location in main memory has two chances of being in the cache. This scheme allows, for example, main-memory locations *m+z* and *5m+z* (where *z* is any integer) to coexist in the cache. This is an advantage because it supports the principle of temporal locality of reference very efficiently for small cache sizes.

As an example of how this type of cache works, consider a software loop that *m* cache lines cannot contain. As the loop executes, the cache begins to fill with instructions and data from the loop, eventually filling *m* lines of, say, set 1. At this point, rather than replacing cache lines of set 1 with new information from the loop, the cache can begin to fill the lines in set 2, thereby allowing the entire loop to reside in the cache. This results in a performance advantage. This advantage goes away, however, when the cache becomes sufficiently large.

Figure 5 shows an implementation of an *n*-way set associative cache, where $n = 2$. Each of the sets contains the same logic as the dashed block in the direct-mapped

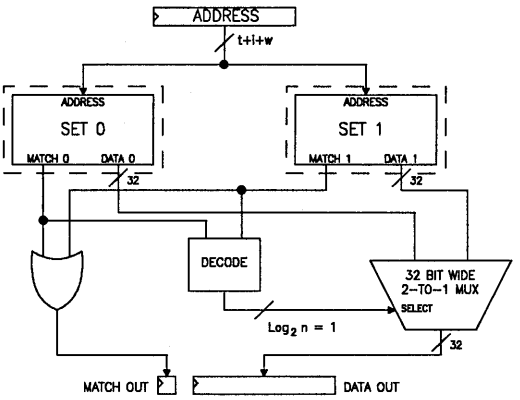


Figure 5. A 2-Way Set Associative Cache

cache diagram (*Figure 3*). Additionally, an OR function in the set-associative cache asserts MATCH OUT if either set contains a match. The decode function selects data from the bank containing the match and asserts a control line to the mux; this allows the matched data to propagate to DATA OUT.

This topology can be extended to *n*-way set associativity by having *n* sets of memory, an *n*-input OR function, an *n*-to-log2*n* decoder, and an *n*-to-1 mux.

Additionally, note that this is only one of several topologies. Another way of implementing the mux function is to assert RAM output enables based on the outcome of the matching function. Yet another way is to combine the OR and decode functions into one PLD.

Note as well that a multi-way set-associative cache has more logic levels than a direct-mapped cache. A multi-way set-associative cache contains three critical timing paths:

1. Read data: accessing the cache data memory in each of the sets.
2. Asserting the MATCH OUT signal in one of the sets, if the tag is matched and valid.
3. Select data: selecting the cached data from the set that matches, if there is a match.

Multi-way set-associative caches are slower than a direct-mapped cache because of the added logic delay associated with the select-data path. Therefore, a direct-mapped cache exhibits a faster cache hit time at a lower system cost.

Fully Associative Mapping

Figure 6 illustrates fully associative mapping. With a fully associative scheme, any location in main memory can be mapped to any location in the cache. This scheme theoretically produces the highest hit rate because there is no possibility of thrashing. Thrashing occurs when two or more data blocks that map to the same location in the cache start replacing each other frequently. The end result of thrashing is a drastic increase in t_{eff} due to increased

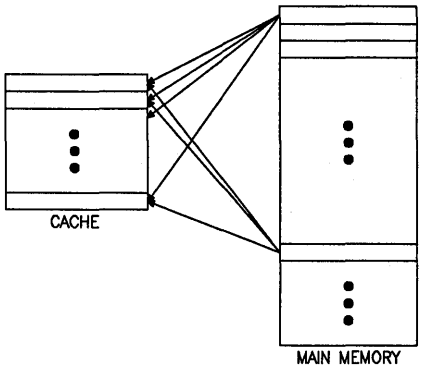


Figure 6. Fully Associative Mapping

miss rate. Thrashing becomes statistically unlikely, however, as cache size increases.

Figure 7 illustrates a simplistic fully associative cache. As shown, the address accesses a CAM (Content Addressable Memory) bank which simultaneously searches all locations for a match. If the CAM finds a valid match, the cache data RAM places the requested information in DATA OUT. If the CAM does not find a match, main memory must be accessed for the correct data.

Fully associative caches are very expensive to build due to the fact that CAM cells are not readily available. Consequently, most caches are designed with direct or set-associative mapping, which can be realized with SRAM technology.

Direct vs. Set-Associative Mapping

The trend in cache design is toward larger caches. In the past, cache sizes of 8 to 16 Kbytes were fairly common. Today, 64 Kbytes is probably the average, with many systems having much larger cache sizes.

As an example, consider the 80386 — a low-end processor — used in combination with the 82385 cache controller. The 82385 directly supports a 32-Kbyte cache and indirectly supports 64- and 128-Kbyte caches. The device supports both direct-mapped and two-way set-associative cache. By coupling the 82385 with the two Cypress CY7C184 Cache Data RAMs (designed specifically for this application), you can implement a 32-Kbyte cache with three chips.

As another example, the Cypress CY7C600 SPARC family — a high-end processor family — supports direct-mapped cache in 64-Kbyte clusters. Each cluster consists of one CY7C604 Cache Tag/Cache Controller/Memory Management Unit (CMU) and two CY7C157 16K x 16 Cache Data RAMs. Up to four clusters can be included per processor, implementing a direct-mapped cache as large as 256 Kbytes.

There are two basic reasons for this trend toward larger cache sizes: First, semiconductor technology can now easily support a 64-Kbyte cache size with reasonable chip count and speed. Second, the emergence of RISC (Reduced Instruction Set Computer) architectures has created a demand for higher cache hit rates and faster cache hit times — in other words, a large cache that is designed simply (i.e., fewer logic delays).

The trends toward larger cache sizes and faster hit times tend to favor the easier-to-design direct-mapped cache. The basic tradeoff involves associativity, defined as the number of cache lines in which a given block of data can reside. As associativity decreases, fewer lines are searched on a memory reference. This provides a potential implementation advantage because, as fewer lines are searched, logic delay paths disappear and the cache gets faster. A disadvantage to decreasing associativity is that the number of lines with identical tags that can simultaneously reside in the cache also decreases.

Valid arguments support the use of set associative-mapping over direct mapping — and vice versa. However, most researchers agree that the trend is toward direct mapping.

Two basic arguments are presented against direct mapping: First, a direct-mapped cache has a lower hit rate than a set-associative cache of the same size. This statement is true but is rapidly becoming a Don't Care. Consider *Figure 8*. For small cache size, direct mapping exhibits considerably higher miss rate than either two-way or four-way set-associative mapping. But for large cache size (64 Kbytes) the miss-ratio difference between direct mapping and set-associative mapping becomes a fraction of 1 percent.

Research presented in *Reference 4* shows that, for an 8-Kbyte unified instruction/data cache, the difference in miss rate for a two-way set-associative vs. a direct-mapped cache is around 1.3 percent. That figure drops to about 0.5 percent for a 32-Kbyte cache.

The end result is that for large cache size, the reduced logic delay inherent in direct mapping (specifically, elimination of the select-data path) produces a cache that is faster and displays essentially the same hit rate as a similarly sized set-associative cache. Thus, recent research supports the use of direct mapping.

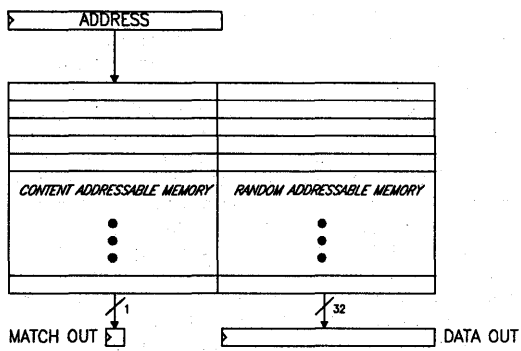


Figure 7. A Fully Associative Cache

The second argument against direct-mapped cache is that a direct-mapped cache is more prone to thrashing. On the surface, this makes a good deal of sense. But for larger cache size, the statistical likelihood of thrashing is so low that it becomes negligible. Additionally, for real-time applications in which deterministic response time to a memory reference is critical, the possibility of thrashing can be completely eliminated if cache entries can be locked — marked as non-replaceable so they are always in the cache.

Four sound arguments can be presented in support of direct mapping: First, direct-mapped cache is less expensive than set-associative cache due to elimination of the select-data logic. Second, the access time for a direct-mapped cache is faster than for a set-associative cache due to elimination of the select-data logic delays. Third, t_{eff} is generally lower for a direct-mapped cache than for a set-associative cache for a sufficiently large cache (generally 32 Kbytes), because t_{cache} is reduced and the (unfavorable) difference in m between the two cache types is negligible. Finally, you do not need to implement a cache line replacement policy for a direct-mapped cache because direct mapping has a one-to-one relationship between cache and main memory (more on cache replacement policies later).

Cache Size

Cache size has perhaps the single largest influence on miss ratio. In terms of miss-ratio impact, cache size is also the most difficult to quantify because it relates so closely to the principle of locality of reference — and therefore the software workload. In general, however, a larger cache has a lower miss ratio. On the other hand, large cache is also significantly more expensive to build given the relatively higher cost of fast SRAMs.

Additionally, mindlessly increasing the size of the cache can actually result in a performance decrease. This effect might result from an increase in output loading due to fan-in/fan-out limitations or the increase in cache-hit processing time due to the added logic delays necessary to manage a larger cache. Given the current state of semiconductor technology, cache sizes of 64 Kbytes are easy to achieve and generally large enough to allow a cache to obtain a 95-percent hit rate.

For multilevel cache hierarchies, a level 2 cache must generally be very much larger than the level 1 cache to be effective. Research results presented in *Reference 7* indicate that adding a level 2 cache can provide a worthwhile performance increase, given the proper combination of small level 1 cache and slow main memory.

Cache Line Size

Cache line size is defined as the basic unit of information transfer between the cache and main memory. Line size ranks second right behind cache size as the parameter that most affects cache performance. Proper choice of line size is important because it affects both miss rate and t_{main}.

Figure 9 presents data transposed from *Reference 10*. Note that for a given cache size, increasing the line size reduces the miss rate. Eventually, however, the miss rate begins to increase with larger line size, as shown by the 2-Kbyte curve in *Figure 9*.

Cache line size also affects t_{main}. Too-large line sizes have long transfer times (which increases t_{main}) and create difficulties in multiprocessing systems by generating excessive bus traffic. These problems especially affect primitive buses that do not support single-address, multiple-data-cycle burst transfers. The burst transfer capabilities of newer bus protocols, such as Futurebus and the SPARC reference standard Mbus (Module-bus), allow larger line sizes with less impact on t_{main}.

Additionally, larger line sizes tend to effect a degree of memory pollution. This problem occurs when information is loaded into the cache but never referenced by the processor.

For multicache organizations, having a level 2 cache line size greater than that of the level 1 cache has advantages that are not discussed in *Reference 7*:lower cache

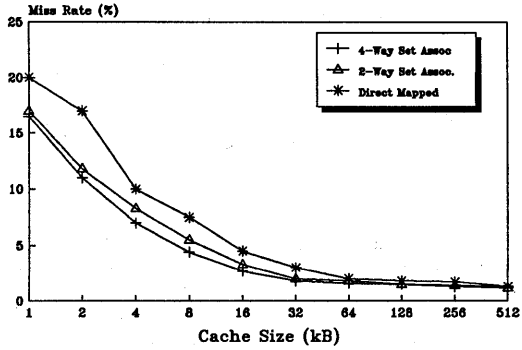


Figure 8. Cache Miss Rate as a Function of Associativity (Transcribed from *Reference 1*)

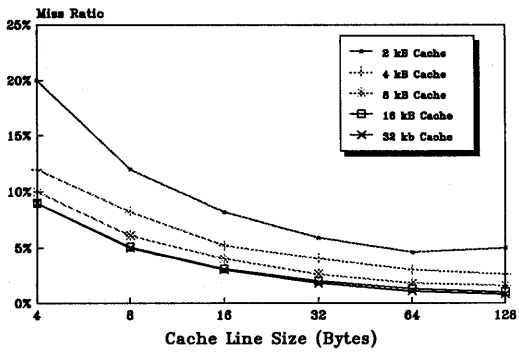


Figure 9. Cache Miss Rate as a Function of Line Size (Transposed from *Reference 10*)

tag cost and increased performance due to the pre-fetch nature of the line size difference.

If the level 2 cache line size is greater than that of the level 1 cache, you must consider some additional design elements. Generally, for example, the line-size ratio of level 2 to level 1 cache is set at a power of 2. Recall that line size is defined as the basic size of information transfer between the cache and main memory, or between the level 1 cache and the level 2 cache. If the line size of the level 2 cache is not equal to the line size of the level 1 cache, the level 2 cache controller must be able to communicate in two different sizes of data chunks.

A cache such as this level 2 cache is referred to as sector oriented. This type of cache maintains coherency in sizes equal to the level 1 cache line size, which is a sub-block of the level 2 cache line.

As a result, the level 2 cache tag entries must include bits to track the following status parameters for each sub-block:

- VALID means that the sub-block contains good data.
- DIRTY indicates that the cache line/sub-block has been written to and is no longer the same as main memory, i.e., main memory must be updated on replacement of a dirty line
- INCLUSION indicates that the sub-block is present in the level 1 cache.

To illustrate the operation of a sector-oriented cache, consider a 16-Kbyte, direct-mapped, level 1 cache with a 16-byte line size that is backed up by a 256-Kbyte, direct-mapped, level 2 cache. If the level 2 cache line size equals the level 1 cache line size (16 bytes), the level 2 cache has 256K/16, or 16K, cache tag entries. Assuming a 32-bit address, the tag size in bits is then 32 - $\log_2$(16K) - $\log_2$(16). This expression yields a tag size of 14 bits; adding 3 bits for VALID, DIRTY, and INCLUSION gives a total length of 17 bits. This length equates to a cache tag size of 16Kx17, or a 272-Kbit tag size.

If, on the other hand, the level 2 cache line size is set at 64 bytes, the level 2 cache has 4K tag entries. The tag size is then 14 bits plus the 3 status bits needed for each of the 4 sub-blocks in the level 2 cache line; this yields a total of 26 bits of tag. The total tag size is then 4Kx26, or 104 Kbits. This means that the tag for the sector-based level 2 cache costs 40 percent as much as the tag for the non-sector-based cache tag on a cost/bit basis. Therefore, in addition to the possible performance benefit associated with having a level 2 line size greater than the level 1 line size, the cache is less expensive as well.

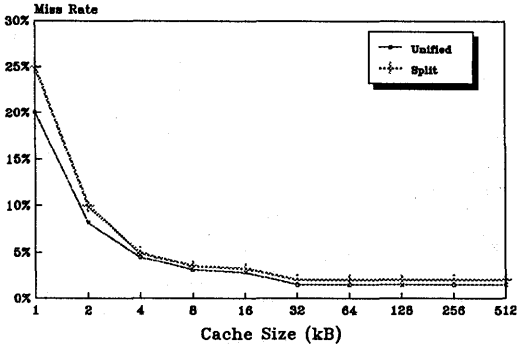


Figure 10. Miss Rate for Split cache vs. Combined Cache(Transcribed from *Reference 12*)

In summary, three factors influence cache line size choice:

1. The type of bus protocol used. A protocol that is capable of burst transfers, such as Futurebus or Mbus, permits a longer line size with a potential performance increase, due to reduced miss penalty for a given line size.
2. The structure of main memory. In other words, make sure that the line size does not create a bottleneck at the main memory interface.
3. Bus-bandwidth/data-contention considerations, especially in a multiprocessing environment.

The design task boils down to choosing a line size that is long enough to effect a good miss ratio, but short enough to minimize t_{main}. Typically, cache line size is 16, 32, 64, or 128 bytes.

Split vs. Combined Cache

In the past, computers have generally utilized a single cache for both instructions and data. It is possible, however, to design a system that has separate caches for instructions and data. Generally, as shown in *Figure 10*, a unified instruction/data cache results in slightly higher performance through a lower miss ratio. The advantages of splitting the cache are:

1. It makes design of the instruction cache easier because the cache's contents do not generally need to be modified.
2. It might eliminate conflict between data and instruction accesses in a pipelined architecture; this depends on the overall processor architecture.

There are also advantages to using a unified cache:

1. Cache design is simpler for a unified cache because both cache-to-main-memory and cache-to-processor communications are one to one.
2. A unified instruction/data cache tends to make more efficient use of the cache — which is a limited resource.

Cache Management

In this context, cache management refers to the policies governing the movement of information into and out of the cache. These policies do not relate directly to cache organization, but they do affect the cache controller's complexity. Specifically, cache management refers to the policies that:

1. Keep main memory coherent relative to cached information.

Table 1. Engineering Tradeoffs: Write-through vs. Copy-back

+/–	Write-through	Copy-back
+	Main memory always has the most up-to-date version of data – minimizing cache coherency problems for multicache configurations.	Produces a lower miss rate than write-through for some applications
+	Easy to implement in the cache controller.	Frees up bandwidth on the main memory bus due to less frequent memory updates.
–	Without buffering (e.g., poseted writes), CPU must wait for write to complete.	Difficult to realize in multiprocessing systems due to cache coherency issues.
–	If write buffers are present, extra logic must be included to ensure that data will not be referenced from main memory until it has been stored there.	Extra logic needed for DIRTY bit.
–	Generates increased bus traffic, which is especially bad for multiprocessing systems.	Results in a more complex controller design, because it caches writes in addition to reads.

2. Determine when new information should be loaded into the cache from main memory.
3. Choose the cache line that should be replaced with the new information being loaded into the cache, if a choice is available

Main-Memory Coherence Schemes

When the CPU modifies cached data, main memory needs to be notified of the change. Whether this notification happens sooner or later depends on the coherency scheme used. The two mainstream coherency schemes are called write through and copy back. Each policy has advantages and disadvantages, and each affects both the complexity of the cache controller and t_{eff}.

Using the write-through policy, all writes to cached locations are immediately written-through to main memory. This policy is the simpler of the two to implement, resulting in a less complex cache controller design. The write-through approach can result in a performance decrease, however, because the CPU usually must be held pending completion of the write. Write through can also cause problems due to increased bus traffic.

The copy-back policy only updates the cache on CPU store cycles, updating main memory only when it becomes necessary to replace a modified (or dirty) line in the cache. This policy requires an extra bit in the cache tag array to keep track of whether a line is clean or dirty. The main advantage of copy back is that it generates less memory bus traffic, resulting in higher performance. The main disadvantage of copy back is increased complexity of the cache controller. *Table 1* outlines the major advantages/disadvantages of both policies.

Additionally, a system can implement write allocation. This means that on a write miss, the data addressed by the write miss is loaded into the cache and then modified. With no write allocation, the data is written to main memory only, and the cache is not updated.

For multilevel cache systems, reducing the overhead required to maintain consistency between the level 1 cache, the level 2 cache, and main memory is a critical design factor. The tradeoff is one of cache controller complexity and the amount of bus bandwidth vs. cost. According to *Reference 7*, the level 1/level 2 cache coherency strategy can result in a 15-percent cache system performance differential. In a two-level cache, you can generally choose a write strategy independently of the level. From highest to lowest performance, the strategies are:

Level 1	Level 2
Copy Back	Copy Back
Copy Back	Write Through
Write Through	Copy Back
Write Through	Write Through

Line Replacement Algorithms

The line replacement algorithm decides which entry in the cache to replace when a new line must be loaded into the cache. For a direct-mapped cache, this task is straightforward, because each main-memory location maps to a unique line in the cache. A set-associative cache permits some latitude in choosing the set in which a line is replaced.

The most common methods of replacing cache lines are Least Recently Used (LRU) and First In/First Out (FIFO). The LRU algorithm keeps track of which line has gone the longest without being used, and replaces that line. The FIFO algorithm keeps track of the oldest line, and replaces that line. You can also use a random line-replacement algorithm, where the set containing the line to be replaced is chosen at random.

Curiously, research presented in *Reference 11* shows that random replacement generally produces higher hit ratios than either the LRU or FIFO algorithms. *Figure 11* uses data from *Reference 11* and shows relative hit ratios for four-way, set-associative cache using both the LRU

and random methods; two-way, set-associative cache using the same two methods; and direct replacement (for direct-mapped cache).

Two notes of caution: First, this data is fairly old (1983) and therefore results from use of an unreasonably small cache by today's standards. Second, the information was obtained by averaging trace data from three different C programs running under UNIX on a VAX-11. Thus, depending absolutely on this data would be inappropriate, especially for RISC machines.

On the other hand, relative comparisons of each policy and cache organization are most appropriate. Some interesting conclusions can be drawn from the data presented in *Figure 11*. First, the random replacement algorithm appears to provide nearly the same or better hit rates compared to LRU. This result is significant because a random replacement algorithm is very much easier to design into a cache controller and requires less hardware. The second conclusion is that for an 8-Kbyte (and presumably larger) cache size, direct-mapped cache offers nearly the same hit-ratio performance as two-way and four-way set-associative cache. This result supports the conclusions drawn in the section on cache mapping techniques.

Fetching Algorithms

Most caches use demand fetching, where a new line is requested from main memory only when a CPU reference results in a cache miss. This method minimizes the complexity of the cache controller.

An alternate method, called pre-fetching, can produce higher hit rates in some applications. Pre-fetching makes use of idle memory cycles to move data into the cache. Static pre-fetch is implemented at compile time, while dynamic pre-fetch occurs at run time.

Sequential dynamic pre-fetching can cut the miss rate in half, according to *Reference 11. Reference 5* estimates a reduction in miss rate of as much as 75 to 80 percent. This estimate points to a significant performance advantage, but sequential dynamic pre-fetching requires a large cache size to be effective. This is because dynamic pre-fetch can result in increased memory pollution; the statistical likelihood of this happening increases dramatically for decreasing cache size. Thus, if cache size is large and cache controller complexity is not a major issue, including a dynamic pre-fetch mechanism can result in a significant performance increase.

A pre-fetch mechanism also provides a way to improve the hit rate of a level 2 cache. Because the level 2 cache hit rate is usually fairly low anyway (generally 50 to 90 percent), memory pollution introduced by pre-fetch tends to be inconsequential. You can implement pre-fetch with minimal hardware overhead by making the line size of level 2 greater than the line size of level 1.

Pulling it all together

This section provides a simplistic method of calculating t_{eff} and the performance improvement of using a cache given various assumptions and design choices. Note that this methodology only provides "ballpark" figures. You can obtain more accurate figures by simulating an actual design — either directly or via a software model.

As presented earlier, the goal of cache design is to reduce the effective memory access time (t_{eff}) as seen by the CPU. Effective access time is defined as

$$t_{eff} = t_{cache} + m \times t_{main}$$

The following methodology does not take into account the effects of design choices on t_{cache} or t_{main} — i.e., these numbers are either already known or estimated. This methodology does, however, include the miss rate, which accounts for the following factors:

- Cache size
- Cache line size
- Cache mapping scheme
- Main memory coherency algorithm

These factors are included by modeling the miss rate as

$$m = M \times MRM + CF$$

where:

m	= Cache miss rate
M	= Raw miss rate
MRM	= Miss-rate multiplier
CF	= Coherency factor

The raw miss rate represents the miss rate strictly as a function of cache size and cache line size. *Table 2* provides the raw miss rate and assumes direct-mapped cache. The miss-rate multiplier is essentially a correction factor that accounts for variations in miss rate between direct-mapped and set-associative cache organizations; *Table 3* provides this value.

The coherency factor accounts for variations in miss rate due to the choice of main-memory coherency algorithm. Recall that the write-through policy does not cache CPU writes; instead writethrough forces all CPU writes to immediately pass through to main memory. Thus, CPU writes to a write-through cache can be regarded as cache

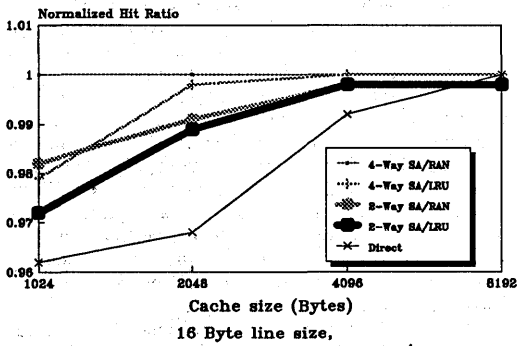


Figure 11. Cache Hit Rate as a Function of Replacement Algorithm

Table 2. Miss Rate as a Function of Cache Size and Cache Line Size						
Cache Size (Kbytes)	Cache Line Size (Bytes)					
	8	16	32	64	128	256
2	0.154	0.116	0.092	0.080	0.084	0.088
4	0.116	0.086	0.074	0.064	0.061	0.065
8	0.096	0.073	0.060	0.053	0.050	0.045
16	0.086	0.064	0.054	0.047	0.044	0.039
32	0.081	0.060	0.051	0.044	0.041	0.036
64	0.079	0.057	0.050	0.043	0.040	0.035
128	0.077	0.056	0.049	0.042	0.039	0.034
256	0.076	0.055	0.048	0.041	0.038	0.033

misses, meaning that CF > 0. If the cache uses write through with posted-write capability or uses the copy-back algorithm, CPU writes can be considered cache hits, meaning CF = 0. You obtain CF by determining or assuming the percentage of cache references that are writes and then derating the miss rate by that factor.

As an example, consider a 64-Kbyte, direct-mapped cache that can be accessed by the CPU in one cycle and that has a 32-byte line size; the cache uses write through, and 30 percent of cache references are writes. Assume a 15-cycle main-memory access time. From *Table 2*, M = 0.017. From *Table 3*, MRM = 1.000. CF = 0.300 (given). Then

$$m = M \times MRM + CF$$
$$= (0.050)\,(1.000) + 0.300$$
$$= 0.350$$

and

$$t_{eff} = t_{cache} + m \times t_{main}$$
$$= 1 + (0.350)\,(15)$$
$$= 6.25\ cycles$$

meaning that this system achieves a 2.4x performance increase using the cache described. Note that the same system with a copy-back cache achieves a t_{eff} of 1.75 cycles, resulting in an 8.57x performance improvement. Finally, consider a two-way set-associative cache using copy back. Now t_{eff} = 1.746 cycles, for a performance improvement of 8.59 — less than 0.2% better than a direct-mapped cache).

Multilevel Cache

Recent advances in silicon technology have allowed a new focus in cache design methodology. Increased gate densities in integrated circuits have helped make it possible to include a small- to-medium-sized cache on the CPU chip itself. Examples include the Motorola 68030/040, the Intel 80486, and Intel's i860 RISC chip. Additionally, because ICs available today support multiprocessing in a straightforward manner, multiprocessing systems will become more common. Examples of these ICs include the Intel 80486 and the Cypress CY7C600 RISC family.

Both of these developments tend to support a multilevel cache hierarchy. Four major factors support a move to a multilevel cache hierarchy:

1. The way the on-chip cache is implemented can force a cache partition. Specifically, a small on-chip cache with unacceptable or marginally acceptable hit rates might force you to add a second level of cache off chip to achieve a design's performance objectives. However, if the on-chip cache is designed improperly, a multilevel cache might be impossible or impractical. The on-chip cache must have the necessary hooks to permit communication between the first level and second level caches. If these hooks are not present, you will be forced to accept lower performance in return for higher integration. This problem occurs with the Intel i860, for example.

Table 3. Cache Miss Rate as a Function of Cache Size and Mapping Method									
Mapping Method	Cache Size (Kbytes)								
	2	4	8	16	32	64	128	256	512
Direct Mapped	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2-Way Set Assoc.	0.975	0.980	0.986	0.990	0.994	0.995	0.996	0.996	0.996
4-Way Set Assoc.	0.925	0.940	0.958	0.970	0.982	0.985	0.988	0.989	0.989

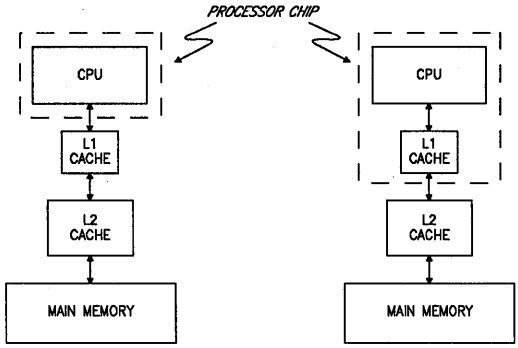


Figure 12. Multilevel Cache Hierarchy for Single-Processor Systems

2. Detailed study of t_{eff} reveals that a multilevel cache hierarchy can offer higher performance than a single-level cache hierarchy, especially if the difference between processor speed and memory speed is large. This speed difference might not result solely from increases in CPU speed, but can also result from the use of larger (and therefore slower) main memory.
3. Creating multiple cache levels opens the possibility of functionally tuning each cache level for highest performance. For example, you can optimize the first-level cache to minimize t_{eff} and the second-level cache for high hit ratio, reduced cost, or reduced interconnect traffic.
4. Increased usage of multiprocessing might force a multilevel cache hierarchy. Generally, each processor needs its own cache — especially if it is a RISC engine — to increase performance and decrease bus traffic. Bus bandwidth is an especially valuable resource in multiprocessing systems. Adding a second-level cache can reduce t_{eff}, especially if the level 1 cache does not meet performance objectives.

In considering these factors, you must resolve the cost vs. performance tradeoff of multilevel vs. single-level cache. This tradeoff depends on the processor architecture, the on-board cache (if any), the main-memory structure, and the type of connection between the cache and main memory.

Consequently, there are no set rules to justify inclusion of a multilevel cache hierarchy. However, recall that cache memory was created to solve performance problems stemming from extremely fast CPU speeds relative to main-memory access times. *Reference 5* states that these speeds must differ by a factor of 10 to justify use of a cache and a factor of 40 to justify use of a multilevel cache. The actual ratio that justifies inclusion of a multilevel cache hierarchy is a personal decision — generally as much a marketing decision as an engineering decision — and concrete statements regarding justification are not valid.

The balance of this application note focuses on multilevel cache hierarchies for uniprocessor and multiprocessor systems. In both cases, the hierarchy is limited to two levels.

Multilevel Cache in Single-Processor Systems

Figure 12 illustrates the cache hierarchy discussed in this section. In this hierarchy, the level 1 cache services processor references and obtains data on a miss from the level 2 cache. The level 2 cache services references from the level 1 cache and obtains data on a cache miss from main memory. The level 1 cache can be inside or outside the processor chip.

This hierarchy does not change the design goal and methods of achieving that goal, but it adds more variables to the equations. The effective memory access time can be expressed as:

$$t_{eff} = t_{L1} + m_{L1}\,(t_{L1} + m_{L2} \times t_{main})$$

where:

t_{L1}	= Level 1 cache access time
t_{L2}	= Level 2 cache access time (penalty beyond t_{L1})
m_{L1}	= Level 1 cache miss rate
m_{L2}	= Level 2 cache miss rate
t_{main}	= Main memory access time (penalty beyond t_{L2})

Minimizing the overhead delay associated with maintaining cache consistency is much more complex for multilevel cache hierarchies than for a single-level cache hierarchy. When beginning a multilevel cache design, you must carefully consider all the previously discussed design factors, but these design factors are now multidimensional problems.

The miss rate approximation presented earlier can be extended for two levels of cache. As an example, consider a system with a single-cycle, two-way set-associative, 8-Kbyte, level 1 cache that has a 32-byte line size and uses copy back; the system also has a single-cycle, direct-mapped, 128-Kbyte, level 2 cache that has a 128-byte line size and uses write through with 30-percent writes. Main-memory access requires 20 cycles. To evaluate this scheme's performance, compare the effective memory access time for both level 1 and level 2 cache with the access time for level 1 cache alone.

First, you can calculate m_{L1} from data in *Tables 2* and *3*:

$$m_{L1} = M_{L1} \times MRM_{L1} + CF_{L1}$$
$$= (0.060)(0.986) + 0$$
$$= 0.059$$

Then

$$t_{eff\,|L1\ only} = t_{L1} + m_{L1} \times t_{main}$$
$$= 1 + (0.059)\,(20)$$
$$= 2.18\ cycles$$

Next, you can calculate m_{L2}:

$$m_{L2} = M_{L2} \times MRM_{L2} + CF_{L2}$$
$$= (0.039)(1.000) + 0.300$$
$$= 0.339$$

Then

$$t_{eff}|_{L1\ \&\ L2} = t_{L1} + m_{L1}\ (t_{L1} + m_{L2} \times t_{main})$$
$$= 1 + 0.059\ [1 + 0.339\ (20)]$$
$$= 1.46\ cycles$$

Thus, the performance improvement over using only the level 1 cache is 33 percent. Note also that the evaluation model produces a t_{eff} of 1.459 cycles for a two-way set-associative, level 2 cache, which results in trading a more complex, more expensive cache controller design for essentially no performance improvement over a direct-mapped implementation. Additionally, if the level 2 cache is direct mapped and uses copy back, t_{eff} is 1.11 cycles, resulting in nearly a 50-percent improvement over using only the level 1 cache.

Multilevel Cache in Multiprocessing Systems

Multiprocessing systems are becoming increasingly prevalent in the industry because they allow the growth rate of computer system technology to be higher than the growth rate of processor technology. Programmers want these systems to have a global main memory. At the same time, the single most performance-limiting factor in multiprocessing systems is maintaining consistency between the global main memory and multiple processors, each having its own cache. Adding a second level of cache can aggravate this consistency problem, and in fact might cause a degradation in performance. However, a multilevel cache hierarchy can increase performance if implemented properly.

Multicache Consistency in Multiprocessing Systems

In multiprocessing systems, it is generally preferable for each processor to have a private cache, which minimizes bus traffic, and a common global main memory, which supports ease of programming. Because a multiprocessing environment generally includes multiple caches, which provide local windows on a large main memory, two or more caches can contain the same data. If this situation occurs, a change in the data in one cache renders the data in the other caches incoherent. Therefore, you need a set of rules — a multicache consistency protocol — to maintain consistency.

As described earlier, maintaining coherence in a uniprocessor system with a single level of cache is fairly simple because coherence only needs to be maintained between one cache and main memory. You can achieve this goal by implementing the copy-back or write-through protocols. The consistency problem is more complex in multiprocessing systems, where each processor has a private cache. This is because consistency must be maintained among a cache, it's "sibling" caches, and main memory.

The consistency problem in this case — while more complex — is well defined and has well-known solutions. Typically, for a multiprocessing system with a large number of processing elements, you employ a software consistency protocol. For systems with a small to medium number of processing elements, you usually implement a bus-based protocol.

Adding a second level of cache tends to aggravate the consistency problem by introducing another level at which consistency must be maintained. Because multicache, multiprocessor topologies have some combination of multiple level 1 caches interfacing to a single level 2 cache and/or multiple level 2 caches interfacing to a common global main memory, the effective memory access time equation must contain a component to account for time wasted while attempting to gain access to a "parent memory."

Therefore, the equation for multiprocessing systems with multilevel cache hierarchies has a contention delay term for consistency management traffic. The position at which this delay enters the equation depends on the topology used. Minimizing this and other delays caused by consistency management is critical to cache design in multiprocessing systems that have a multilevel cache hierarchy.

Now consider how this extra level of coherence management affects system performance. *Figure 13* presents three different multiprocessing topologies. Most authors agree that the level 2 caches should be supersets of their children caches. In this manner, the coherence management protocol can be moved as far from the processing element as possible. This allows the level 2 caches to shield the level 1 caches from unnecessary blind checks (snoops for data that is not in the cache) and invalidations that might propagate up from main memory.

Adhering to the Multilevel Inclusion (MLI) Principle — that all the data in the level 1 cache is in the level 2 cache — can minimize snoops, which halt the CPU. The MLI Principle is defined for set-associative caches in *Reference 2*: MLI can be achieved if the degree of set associativity of a parent (level 2) cache is greater than or equal to the product of the number of its children (level 1) caches, their degree of set associativity, and the ratio of their block sizes. Expressed mathematically:

$$Set\ Associativity_{L2} = \sum_{All\ L1's} \left[\ Set\ Associativity_{L1} \times \frac{Line\ Size_{L2}}{Line\ Size_{L1}}\ \right]$$

Note that MLI is not a requirement in multicache designs, and the scheme proposed in *Reference 2* is only one of several ways to achieve MLI. As shown later, MLI as stated in *Reference 2* is very restrictive and results in an extremely complex and expensive level 2 cache design. To enforce MLI according to the *Reference 2* scheme, for instance, the system described in the section Multilevel Cache in Single-Processor Systems becomes an eight-way set-associative, level 2 cache. This might be an unrealistic goal, because a 128-Kbyte cache of this complexity is expensive to implement.

For Topology A in *Figure 13*, however, you can implement MLI under the *Reference 2* scheme if, for example, you do it this way: The level 1 cache is a direct-

mapped, 16-Kbyte cache with 16-byte line size, and the level 2 cache is a four-way set-associative, 256-Kbyte, sector-based cache with 64-byte line size. Additionally, using Topology A, you can implement a simple cache-coherence protocol such as copy back or write through at the level 1 cache, which is generally small.

Cost effectiveness can dictate a fairly large sector-based level 2 cache. Consistency among the level 2 caches is maintained on the basis of the level 1 line size. A private level 2 cache services all level 1 cache misses. The level 1 cache is disturbed only when the need arises to replace a sub-block in a level 2 cache whose INCLUSION bit is set.

You can improve performance dramatically if the system meets two conditions: the bus can support direct data intervention (more on this later), and the level 2 cache controller has a bus-snooping mechanism that allows it to monitor bus activity and perform invalidations based on observed bus traffic. The effective memory access time for this topology is:

$$t_{eff} = t_{L1} + m_{L1}\,[t_{L1} + m_{L2}\,(t_{main} + t_{bus,\,L2\text{–}main})]$$

where $t_{bus,L2\text{-}main}$ is the time required for a given level 2 cache to acquire the bus. The advantage of this topology is that it is simple and fairly straightforward to implement. The main disadvantage is that the level 2 cache is not shared by several level 1 caches.

Topology B, which depicts a multiport level 2 cache connected to multiple other level 2 caches via a bus, is probably the least desirable of the 3 topologies shown for several reasons. For example, this topology contains two points at which contention might occur, resulting in an effective memory access time equation of:

$$t_{eff} = t_{L1} + m_{L1}\,[(t_{L1} + t_{contention,\,L1\text{–}L2}) + m_{L2}\,(t_{main} + t_{bus,\,L2\text{–}main})]$$

where $t_{contention,L1\text{-}L2}$ denotes the arbitration/contention penalty for a level 2 cache to service a level 1 cache. Thus, this design is slower than topology A.

Additionally, the logic required for arbitration at level 2 among the several level 1 caches is expensive. Finally, MLI is very difficult to obtain for this type of system. Consider a system with four 16-Kbyte, direct-mapped, level 1 caches that have a 16-byte line size connected to a 256-Kbyte, level 2 cache that has a 64-byte line size. The scheme proposed by *Reference 2* dictates that the level 2 cache be 16-way set associative.

Topology C, a bus-based hierarchy, is probably the most attractive topology for systems with a small to medium number of processing elements. Using this topology, MLI is guaranteed through the use of broadcast invalidations — notifications to all caches to invalidate shared lines that were written by another CPU into a private cache. The effective memory access time for this topology is given by:

$$t_{eff} = t_{L1} + m_{L1}\,[(t_{L1} + t_{bus,\,L1\text{–}L2}) + m_{L2}\,(t_{main} + t_{bus,\,L2\text{–}main})]$$

where $t_{bus,L1\text{-}L2}$ is the time required for a given level 1 cache to acquire the bus that connects the level 1 caches and the level 2 cache. Using well-designed buses, such as Futurebus or Mbus, reduces bus traffic in this topology to a minimum.

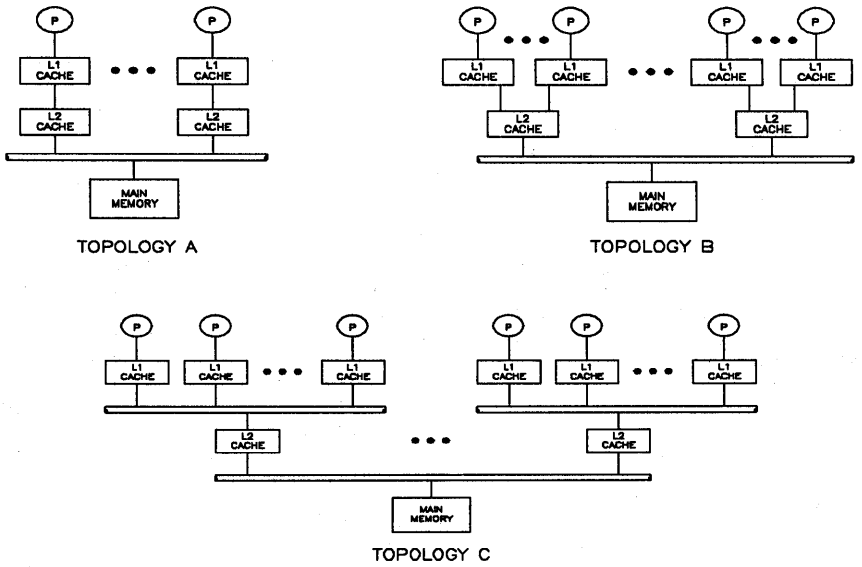


Figure 13. Multiprocessing Topologies with Multilevel Cache Hierarchies

Topology C's disadvantages are that it introduces greater hardware complexity, and that a manageable implementation requires the use of VLSI. (Such VLSI solutions are available in the Cypress CY7C600 family, however.) Additionally, even with a good bus protocol, the amount of bus traffic limits the number of resources that can share the bus. Despite these disadvantages, a bus-based multilevel cache hierarchy appears to be the most promising in terms of cost and performance.

Multilevel Cache in SPARC Multiprocessing

The Cypress CY7C600 RISC microprocessor family contains full support for multiprocessing, including an excellent bus-based, multicache consistency mechanism. This section covers the CY7C600 family members that comprise a multiprocessing (MP) cluster: the CY7C601 Integer Unit (IU), the CY7C602 Floating Point Unit (FPU), the CY7C605 Cache Tag-Cache Controller-Memory Management Unit for Multiprocessing (CMU-MP), and the CY7C157 16K x 16 Cache RAM. This part of the application note highlights the features of the CY7C605 CMU-MP that support multicache consistency. A section also covers Mbus. Finally, a SPARC multiprocessing system is extended to a multilevel cache hierarchy, which is demonstrated in two topologies. These topologies are then examined, with a focus on implementation and performance advantages/disadvantages.

The SPARC Multiprocessing Cluster

As presented in *Figure 14*, the basic SPARC multiprocessing cluster consists of a CY7C601 IU, a CY7C602 FPU, a CY7C605 CMU-MP, and two CY7C157 Cache RAMs. You can increase the cache size by adding up to three more CY7C605s and six more CY7C157s, as shown

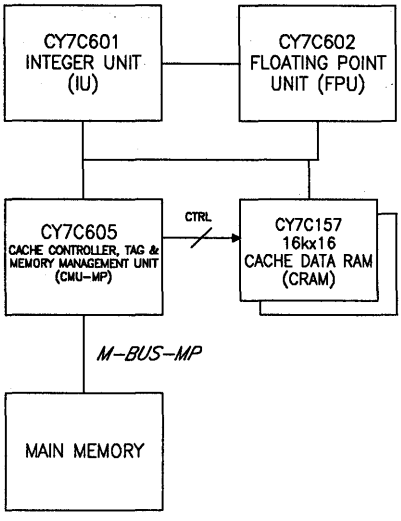


Figure 14. The SPARC Multiprocessing Cluster

in *Figure 15*. This change supports cache sizes from 64 to 256 Kbytes in 64-Kbyte increments. You can also connect several MP clusters via the Mbus to form a multiprocessing system, as shown in *Figure 16*.

The CY7C601 IU is fully compliant with the SPARC reference Instruction Set Architecture. The CY7C601 furnishes full support for eight register windows, a full IEEE floating point coprocessor interface, and a second generic (user-defined) coprocessor interface. The device is available at 25, 33, and 40 MHz (scalable to 50 MHz) and is implemented in a 0.8-micron, dual-layer-metal CMOS process.

The CY7C602 FPU is a single-chip, SPARC, floating-point processor. It provides full IEEE double-precision support, a dedicated register file, and 64-bit data paths. The CY7C602 is available at up to 40 MHZ (scalable to 50 MHz).

The CY7C157 Cache RAM is custom design for CY7C604 and CY7C605 cache systems. It is still a fairly generic cache RAM, however. The CY7C157 is a fully synchronous (self timed) device — much better suited to cache design than "industry standard" asynchronous RAMs. The CY7C157 scales in speed, matching the clock rate of the IU and CMU, and is implemented in 0.8-micron, dual-layer-metal CMOS technology.

The CY7C605 CMU-MP

The CY7C605 CMU-MP includes all the features of the CY7C604 uniprocessing CMU along with provisions for multiprocessing. Fully compliant with the SPARC Reference MMU Architecture Standard, the CY7C605 has a 32-bit (4 Gbyte) virtual address space and a 36-bit (64 Gbyte) physical address space. In addition to an on-board, 64-entry, fully associative translation lookaside buffer (TLB), the CY7C605 includes support for 4K multiple contexts, a 4-Kbyte page size, memory-address protection checking, hardware table walking, and sparse address spaces with a three-level page-table map.

For cache control, the CMU contains 2K, direct-mapped, virtual cache tag entries and support for a 32-byte line size; these features allow the device to manage a 64-Kbyte direct-mapped cache. The CY7C605 also supports either write through with no write allocate or copy back with write allocate. Copy back with write allocate does not degrade performance because the CMU has a full 32-byte cache read buffer.

The CMU can also perform posted writes via two on-chip, 32-bit write buffers, which support fully buffered Store Doubles. This capability improves the cache's performance when a write miss is encountered by allowing the main-memory update to occur in background. The CY7C605's cache lock mechanism allows entries to be locked in the cache, enabling deterministic responses for real-time applications. The device also provides for five levels of cache flushing. Its 64-bit multiplexed address/data bus provides the interface to Mbus.

The CY7C605 provides full alias detection and correction through use of both a virtual and physical cache tag array. The physical tags, which are not included in the

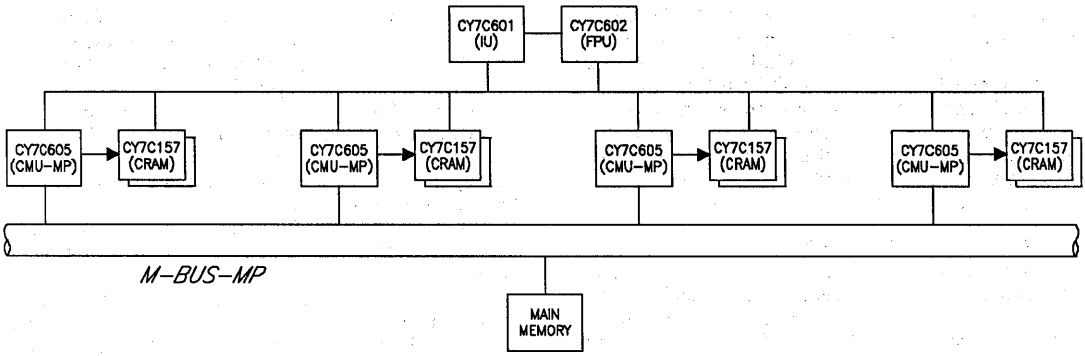


Figure 15. Fully-Extended SPARC Cache

CY7C604 CMU-UP, serve two purposes. First, this second bank of cache tags acts as a reverse translation unit, allowing on-chip detection and correction of aliasing. Second, the physical tag array permits bus snooping to occur completely independently of the processor, which interfaces to the virtual cache through the virtual cache tag array.

Bus snooping is an activity in which the CMU monitors all activity on the Mbus and responds to invalidation broadcasts or requests for data from other caches in the system. The key advantage of physical tag entries is that they enable the bus snooping logic to be decoupled from processor traffic, resulting in a substantial performance increase.

The CMU-MP contains full support for the MOESI (Modified, Owned, Exclusive, Shared, Invalid) cache consistency model. The MOESI model enables multiple caches to coexist on a single bus and share a global main memory, while guaranteeing multicache consistency. Using this methodology, each entry in a cache can be in one of five states: PRIVATE CLEAN, PRIVATE DIRTY, SHARED CLEAN, SHARED DIRTY, or INVALID. If an entry is located in only one cache in the system, it is either PRIVATE CLEAN or PRIVATE DIRTY.

If more than one cache shares unmodified data, they are all in the SHARED CLEAN state. Once a cache modifies shared data, it marks the data SHARED DIRTY, broadcasts an invalidation message informing other caches with that data to mark their entries INVALID, and immediately becomes responsible for responding to any further requests for that data. Note that any time a processor is in one of the DIRTY states, it becomes the "owner" of the data and is responsible for servicing any requests for that data.

Finally, the CMU-MP supports reflective main memory and direct data intervention. The latter provides a significant performance increase over indirect data intervention. To illustrate the difference, consider an MP system with a common main memory and, for simplicity, two caches. Cache A retrieves a line of information from main

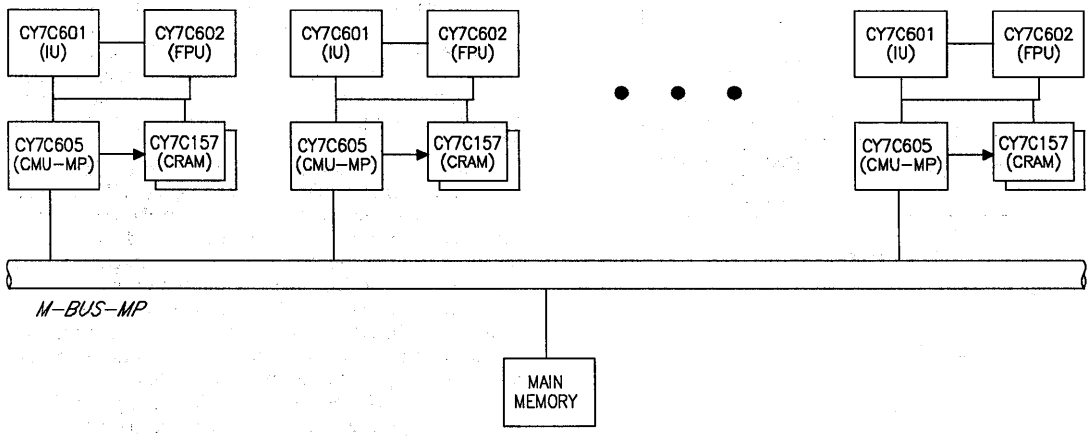
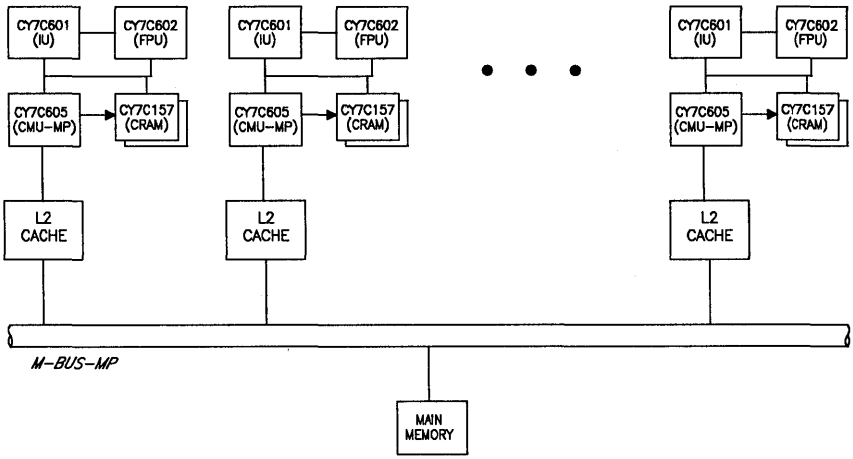


Figure 16. A SPARC Multiprocessing System



TOPOLOGY 1

Figure 17. SPARC Single-Level Cache Extension to Multilevel Cache Topology

memory and modifies it, thus becoming the owner of the data. At some point, Cache B requests the same piece of information.

In a system using indirect data intervention, Cache A informs Cache B that the miss occurred and that Cache B should attempt to gain access to the bus later. Cache A then seizes the bus and updates main memory. Meanwhile, Cache B tries to gain access to the bus, while its processor is on hold, awaiting the new data. When finished updating main memory, Cache A releases the bus. Cache B gains access, and begins to retrieve the data from main memory. Eventually, after a considerable number of cycles, processor B is released from hold and permitted to continue.

In a system using direct data intervention, Cache A supplies the data requested by Cache B directly, resulting in considerably fewer hold cycles for processor B. Additionally, with a reflective main memory system, main memory observes the information transfer and updates itself at the same time. With a non-reflective approach, main memory would contain stale data relative to the caches.

Mbus

Mbus is a fully synchronous, 64-bit, multiplexed address/data bus that supports multiple bus masters and has a peak transfer rate of 320 Mbyte/s at 40 MHz. All signals are sampled on rising clock edges and driven active and inactive. Mbus supports single-address/multiple-data-cycle bursts of 16, 32, 64, and 128 bytes, with full retry support. Finally, central arbitration is separate from the master and slave; the type of arbitration scheme used is completely up to you. The cache consistency model for the Mbus is based on the MOESI model.

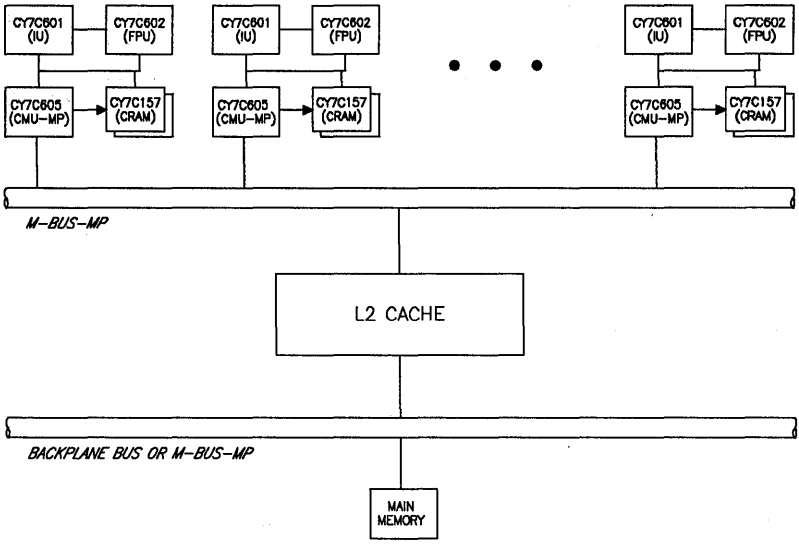
A Cache Hierarchy for SPARC MP Systems

This section presents two possible multilevel cache implementations for SPARC multiprocessing systems. For highest performance, both topologies require a level 2 cache controller that is as complex as the cache controller in the CY7C605. Specifically, the level 2 cache must support fully concurrent bus snooping and direct data intervention. In addition, it is generally preferable that the level 2 cache have a larger line size than the level 1 cache. The level 2 cache controller thus needs to be sector based, which increases the level 2 cache controller's complexity.

Figure 17 shows a single-level cache extension topology, which forces the level 2 cache to manage cache consistency. Consistency management is thus moved as far away from the processor as possible. This approach improves performance because it tends to cause fewer hold cycles for the processor. This topology also permits smaller level 2 caches — a definite advantage if the speed of the level 2 cache is critical, because small caches are easier to optimize for speed.

The main disadvantage of this topology is that the level 2 cache is not shared by several level 1 caches. This results in higher total system cost because each level 2 cache requires its own controller.

Topology 2 (*Figure 18*) is a multilayer bus-based hierarchy. This topology permits a common level 2 cache, whose single controller keeps costs lower. However, this topology probably requires a level 2 cache size of 2 Mbyte or more to achieve high system-level performance. This large cache size generally results in a slower (perhaps multicycle) level 2 cache. If cost of the level 2 cache is critical, however, this topology is probably the best choice.



TOPOLOGY 2

Figure 18. SPARC Bus-Based Multilevel Cache Topology

References

1. Agrawal, Hennesy, Horowitz, "Cache Performance of Operating Systems and Multiprogramming Workloads," *ACM Transactions on Computer Systems*, 11/88, Vol. 6, No. 4

2. Baer, J.L. and Wang, W.H., "On the Inclusion Properties for Multilevel Cache Hierarchies," *Proceedings of the 15th Annual Symposium on Computer Architectures*, February 1988, pp. 81 - 88

3. Gregory, Richard, "Caching Designs Eliminate Wait States to Relieve Bottlenecks," *Computer Design*, October 15, 1988, pp. 65 - 73

4. Hill, Mark D., "A Case for Direct-Mapped Caches," *IEEE Computer*, December 1988, pp. 25 - 40

5. Kabakibo, Aiman, et al, "A Survey of Cache Memory in Modern Microcomputer & Minicomputer Systems," *IEEE Micro*, March 1987, pp.210 - 227

6. Pohm, A.V. and Agrawal, O.P., *High Speed Memory Systems*, Reston Publishing, 1983

7. Short, R.T. and Levy, H.M., "A Simulation Study of Two-Level Caches," *Proceedings of the 15th Annual Symposium on Computer Architectures*, February 1988, pp. 81 - 88

8. Smith, A.J., "Cache Memories," *Computing Surveys*, Vol 14, No 3, September 1982, pp. 473 - 530

9. Smith, A.J., "Cache Memory Design: An Evolving Art," *IEEE Spectrum*, December, 1987, pp. 40 - 44

10. Smith, A.J., "Line (Block) Size Choice for CPU Memories," *IEEE Transactions on Computers*, Vol C-36, No 9, September 1987, pp. 1063 - 1075

11. Smith, J.E. and Goodman, J.R., "A Study of Cache Organizations and Replacement Policies," *ACM Computing Surveys*, 1983, pp. 132 - 137

12. Smith, *CPU Cache Memories*, University of California, Berkeley, 1984

CYPRESS SEMICONDUCTOR

Synchronous Trap Identification for CY7C600 Systems

This applications note discusses the decoding of the status bits in the CY7C601 SPARC processor's synchronous fault status register (SFSR). When a memory access fault occurs, these bits indicate the type of fault.

Due to the pipelined nature of the SPARC processor, multiple traps can occur before it leaves normal execution mode and vectors to a trap handler. If a multiple-trap situation occurs, the information in the SFSR and the synchronous fault address register (SFAR) might not reflect the status for the trap to which the CY7C601 first responds. Although the corrective course of action for the fault case depends on your system's characteristics, this application note explains how to interpret the fault so that it can be corrected.

Section 4.9 in the *SPARC RISC User's Guide* describes the operation of the SFSR and SFAR upon encountering a synchronous fault. Reviewing section 4.9 will help you understand the information given in this application note. A brief summary of the SFSR characteristics appears in the last section of this applications note.

Trap Handler Objectives

The objective for the trap handler is to resolve a memory access error, if possible. In the case of a double fault occurrence, the first of the two faults is generally, but not always, the desired fault to be corrected. In one group of cases, correcting the second fault is preferable, because the CY7C601 re-executes the instruction that caused the first fault upon leaving the trap handler.

Errors in address translation are generally non-recoverable, as they imply a mapping problem in the MMU virtual page-mapping tables. For these cases, the identification and recording of the error condition is the only purpose that the trap handler can serve.

Memory access errors are signaled when the CY7C604 or the CY7C605 cache and memory management units assert the MEXC signal. This event forces the CY7C601 to vector to either an instruction access exception or a data access exception. Instruction access exceptions are delayed until the fetched instruction reaches the execute stage in the CY7C601. Because data accesses are generated as a result of an instruction that has reached the execute stage, the exceptions associated with a data access are recognized immediately. This difference in the timing of exception recognition causes many of the double fault cases described in section 4.9.1 of the *SPARC RISC User's Guide*.

Upon detecting an instruction or data exception, the CY7C601 enters the corresponding trap. The two trap handlers share the task of identifying the synchronous fault case. The following sections describe the fault cases that each handler can identify using the contents of the SFSR and SFAR. *Figures 1* and *2* illustrate the decision tree seen by the data exception handler and the instruction exception handler, respectively.

Data Exception Fault Groups

Group D1

This group consists of case 14, as described in the *SPARC RISC User's Guide*. The CY7C601 traps for the data memory access fault. The information in the SFAR and SFSR reflects the instruction translation fault and is not useful for servicing the initial data access fault. The address of the data access instruction is not lost, however. The address is given by the PC stored in r[17], or local register 11, of the trap handler window.

Group D2

The members of this group are cases 12 and 13. Handling this group is straightforward in that the information in the SFSR and SFAR reflects the first occurring fault. However, translation faults in general are a non-recoverable type of error, as they imply a mapping problem within the page tables. Handling this type of fault consists of dropping the task altogether and recording the fault information for system debug.

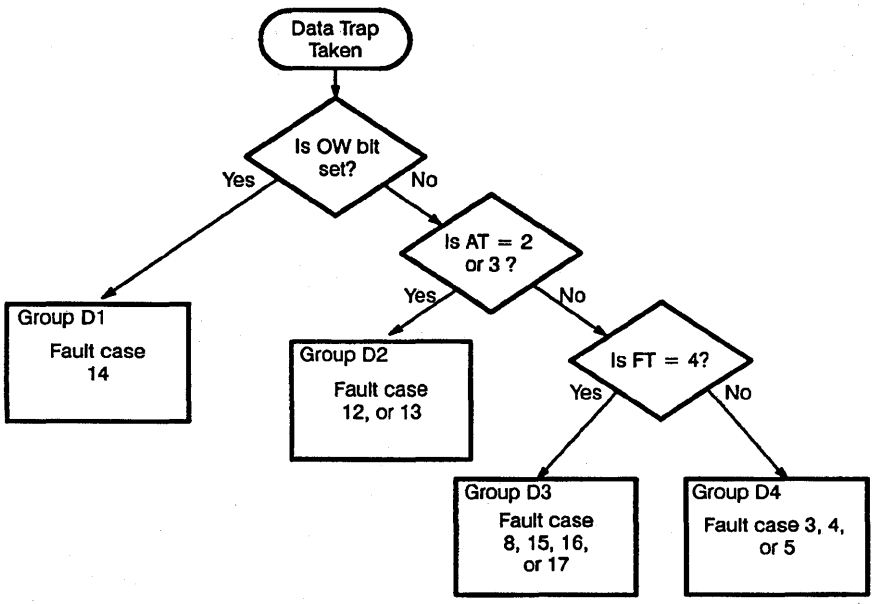


Figure 1. Data Access Fault Identification

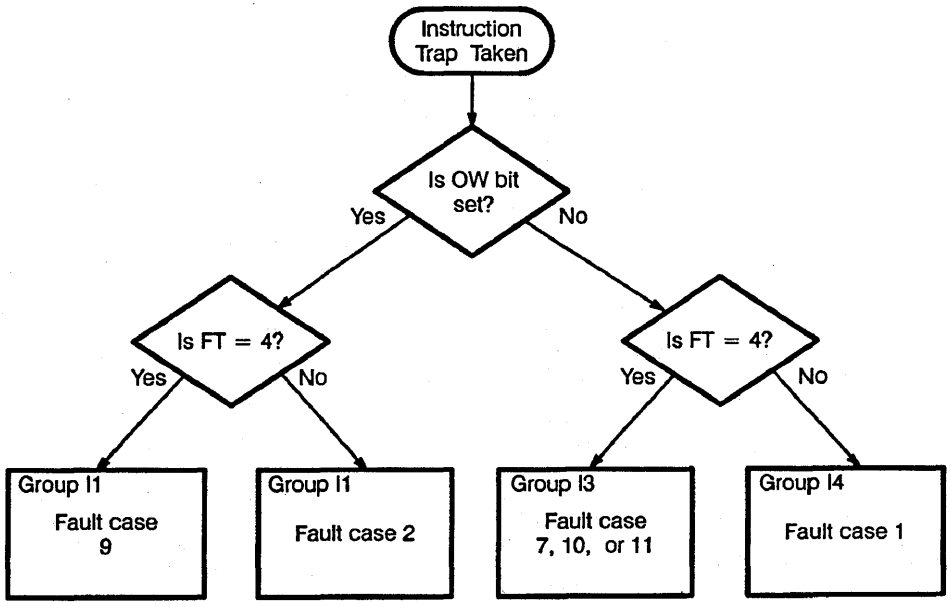


Figure 2. Instruction Access Fault Identification

RSV = Reserved
*CBT = Copy-back Translation Error
UC = Uncorrectable Error
TO = Time Out Error
BE = Bus Error
L = Level
AT = Access Type
FT = Fault Type
FAV = Fault Address Valid
OW = Over Write

(*CY7C604 only; reserved in CY7C605)

Figure 3. CY7C604/605 Synchronous Fault Status Register

Group D3

This group consists of cases 8, 15, 16, and 17. In cases 8, 16, and 17, the data access translation fault that the CY7C601 traps on is the either the only or the first of the two occurring faults. The information stored in the SFSR and SFAR is valid and can be stored in an error dump for debug purposes.

Case 15 is an instruction fault followed by a data access translation fault. The information for the instruction access fault is non-recoverable, as the CY7C601 traps on the data translation fault, which overwrites the instruction access fault information.

Group D4

This group consists of cases 3, 4, and 5. The CY7C601 traps on the data fault in all of these cases, and the information in the SFSR and SFAR reflects the information for the data fault. Case 3 is a single data fault, and is straightforward in its recovery. Case 4 is an instruction fault that is overwritten by a following data fault. The information for the initial instruction fault is lost, but can be recovered by correcting the data fault first. The CY7C601 reissues the address for the instruction that caused the initial instruction fault, allowing the fault to be handled. Case 5 is the occurrence of a data fault followed by an instruction fault. It should also be handled by correcting the data fault and allowing the CY7C601 to reissue the address for the fault-causing instruction.

Instruction Exception Fault Groups

Group I1

The single member of this group is case 9, an instruction access translation fault preceded by an instruction access fault. The CY7C601 traps for the first instruction access fault. The information in the SFAR and SFSR reflects the instruction translation fault and is not useful for servicing the initial instruction access fault. The address of the instruction is not lost, however. The instruction address is given by the PC stored in r[17], or local register 11, of the trap handler window. Recovery from this fault case involves using the old PC to re-execute the fault-causing instruction and attempting correction of the error condition.

Group I2

This group consists of case 2, which is a triple instruction fault. Note that the *SPARC RISC User's Guide* describes this case as a double instruction fault, but the *User's Guide* errata makes the correction to a triple fault.

The CY7C601 traps on the first occurring instruction trap, but the information in the SFSR and SFAR has been overwritten by the following instruction faults. The address of the first instruction fault can be recovered in the same manner as group I1.

Group I3

This group of faults includes cases 7, 10, and 11. All of these cases are translation faults on an instruction access. Case 7 is a singular occurrence of an instruction access translation fault. Cases 10 through 13 are occurrences of an instruction access translation fault followed by some other type of fault. The information in the SFSR and SFAR reflects the status stored from the first occurring fault. These faults involve translation errors, which are generally handled by dropping the task altogether and recording the fault information for system debug.

Group I4

This group consists of case 1, which is a single instruction fault. Information in the SFSR and SFAR is valid.

SFSR Description

The SFSR is described in sections 4.4.11 and 4.9 of the *SPARC RISC User's Guide*, but is briefly repeated here for reference.

Figure 3 gives the bit assignments for the SFSR. The SFSR's UC, TO, and BE bits are set according to the type of error signaled to the CY7C604 by an Mbus agent (such as memory or the Mbus arbiter) in response to an Mbus transaction. *Table 1* gives the encoding for the Mbus transaction response signals.

Mbus transactions that signal a bus error, time out, or uncorrectable error set the corresponding bit in the SFSR of the CY7C604/605, which then responds by asserting CMER to the interrupt logic. These bits describe Mbus error cases and do not apply to the synchronous fault

Table 1. Mbus Transaction Response Signal

MERR	MRDY	MRTY	Action
H	H	H	Nothing
H	H	L	Relinquish and Retry*
H	L	H	Data Strobe
H	L	L	Reserved
L	H	H	Bus Error
L	H	L	Time Out
L	L	H	Uncorrectable Error
L	L	L	Retry

Table 2. SFSR Fault Level

L	Level
0	Entry in Context Field
1	Entry in Level 1 Table
2	Entry in Level 2 Table
3	Entry in Level 3 Table

cases described in section 4.9 of the *SPARC RISC User's Guide*.

The SFSR's level (L) bits describe the level in which an incorrect page entry was found for translation faults. These bits are described in *Table 2*. Note that they are irrelevant for non-translation fault errors.

The access type (AT) bits are described in *Table 3*. They give the type of access that caused the currently reported memory access fault.

The fault type (FT) bits describe the type of error found by the CY7C604/605. *Table 4* gives the fault type for the case of a table walk that correctly finds a page table entry (PTE) but still causes a fault condition. The access type (AT) is compared against the access protection field of the PTE (ACC bits), and the fault type is set according to *Table 5*.

Table 3. SFSR Access Type

AT	Access Type
0	Load from User Data Space (ASI = 0xA)
1	Load from Supervisor Data Space (ASI = 0xB)
2	Load/Execute from User Instruction Space (ASI = 0x8)
3	Load/Execute from Supervisor Instruction Space (ASI = 0x9)
4	Store to User Data Space (ASI = 0xA)
5	Store to Supervisor Data Space (ASI = 0xB)
6	Store to User Instruction Space (ASI = 0x8)
7	Store to Supervisor Instruction Space (ASI = 0x9)

Table 4. SFSR Fault Type

FT	Fault Type
0	None
1	Invalid Address Error
2	Protection Error
3	Privilege Violation Error (user mode only)
4	Translation Error
5	Bus Access Error
6	Not Generated
7	Reserved

Table 5. Fault Type (FT) for PTE[ET] = 2 (valid PTE)

AT	ACC							
	0	1	2	3	4	5	6	7
0	0	0	0	0	2	0	3	3
1	0	0	0	0	2	0	0	0
2	2	2	0	0	0	2	3	3
3	2	2	0	0	0	2	0	0
4	2	0	2	0	2	2	3	3
5	2	0	2	0	2	0	2	0
6	2	2	2	0	2	2	3	3
7	2	2	2	0	2	2	2	0

CYPRESS SEMICONDUCTOR

An Introduction to Mbus

This application note provides an introduction to Mbus, a part of the SPARC architectural standard, which addresses the requirements for interfacing to a processor system's physical memory space.

In a system supporting virtual memory with cache, the physical memory and I/O interface are key components of the system architecture. Maintaining bandwidth and response time is critical to achieving adequate performance levels for the system.

Architectural Overview of Mbus

Mbus provides a high-performance interface to the physical address space in a SPARC system, with facilities to support the cache coherency requirements of symmetric multiprocessing. Mbus is intended to operate with SPARC processors that have local virtual caches, so that access to the physical address space only occurs in the event of a cache miss. With reasonable-sized local caches, the Mbus loading from an individual processor is in the range of 5 to 10 percent. This allows the Mbus to support other processors and I/O activities without degrading individual performance.

Bus overhead, which mostly consists of arbitration and transaction time, is a critical element in determining overall system performance. Many different bus-arbitration mechanisms are available, with a variety of cost/performance tradeoffs. For this reason, the SPARC architectural standard does not define a specific arbitration mechanism for Mbus. You thus have complete flexibility in system design.

Mbus does support bus arbitration that can operate concurrently with data transactions. When a system can use overlapped arbitration, bus arbitration incurs no bus overhead.

The second aspect of bus overhead, transaction time, is the bus time required to perform the actual data transfer. High bus bandwidth minimizes transaction time on Mbus, which is capable of peak data rates up to 320 Mbytes/s and 256 Mbytes/s sustained at 40 MHz.

Two Mbus compliance levels are defined to suit differing system requirements. Level 1 compliance is for uniprocessor applications, and level 2 for multiprocessing systems that incorporate shared memory with caching. This application note primarily focuses on level 1 compliant system design.

A complete set of Mbus communication protocols provide for access to physical memory and I/O channels.

Basic Structure of the Bus

Mbus is a 64-bit bus that can transfer up to 128 bytes in a single data burst, with support to transfer 8 bytes on each clock cycle. Elements on the bus operate in a master/slave relationship, where a master element initiates a transaction and a slave element responds by either accepting or providing data. A "ready" status line from the slave element controls data transfers. Data is not transferred until the ready line is asserted. This allows a slave element to operate even if it is not fast enough to handle data on each clock cycle.

Bus arbitration is supported on Mbus using a conventional request/grant mechanism, which assumes a centralized arbiter. The protocol enables arbitration to overlap data transfers. This feature allows the arbitration process to execute without using any bus cycles dedicated to arbitration. The algorithm for implementing the grant response to a bus request is user defined for maximum flexibility.

Several Mbus protocols support error conditions that can occur in a typical system implementation. These errors include: External Bus Error, Response Timeout, Uncorrectable Memory Error, and Transaction Retry. These protocols handle most of the error conditions encountered in a system interface to physical memory and I/O.

Multiprocessing Facilities

A significant trend in computer systems is toward multiprocessing. In a shared-memory multiprocessing system, maintaining local cache coherency without degrading system performance is a major architectural challenge.

Figure 1 shows the topology of a multiprocessing system. All processor nodes contain local caches and operate out of them most of the time. However, when one processor changes data that is shared, the other processors need to be made aware that the data has changed, so that

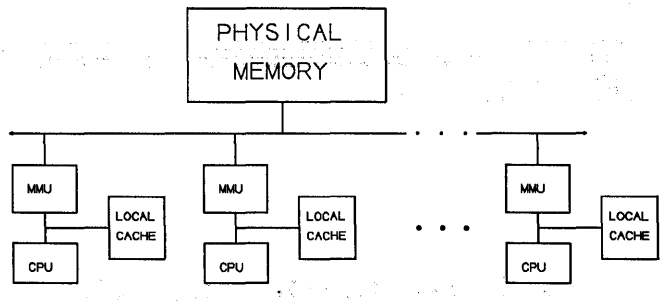


Figure 1. Multi-Processing Topology

subsequent references can be made without using stale data. Further, the multiprocessing system needs an efficient mechanism to allow a processor to access the modified data when it is required.

Mbus implements a bus snooping protocol that allows a processor node to communicate to the other nodes that a piece of shared data has been modified. Each processor node responds by marking that datum as invalid in the node's own cache tag. When a processor node references that datum, a cache miss occurs because the entry has been invalidated in the cache. The processor node then generates a normal Mbus read transaction to access the datum. Instead of the physical memory element supplying the datum, the Mbus protocol allows the datum to be supplied by the processor node whose local cache contains the current datum.

This approach has the advantage that no data transfer occurs on the bus until the shared data is needed, saving a considerable amount of bus bandwidth. Further, the performance penalty to access modified data is no worse than the a penalty of a normal cache miss.

When the processor node provides its modified data on the bus to the requesting node, the Mbus protocol allows the physical memory and other processor nodes to update their data. This reflective memory feature can save additional bus bandwidth by requiring that modified shared data be transferred only once, rather than each time a different node references the data.

Mbus Description

Mbus is a fully synchronous bus whose 64-bit data path (MAD) multiplexes address and data for each data transaction. All data is sampled on the rising edge of the system clock, MCLK, and a bus transaction can only be initiated by the bus master that currently owns the bus.

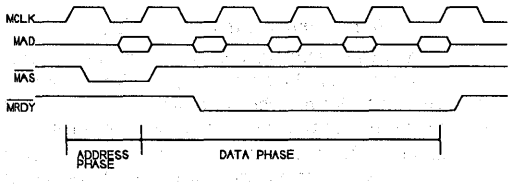


Figure 2. Basic M-Bus Transaction Timing

A transaction consists of an address phase followed by one or more data transfer cycles. The address phase provides a 36-bit physical address and a set of control fields that defines the transaction's nature and size. The data phase consists of multiple 64-bit transfers that are synchronous with the bus clock. A simple illustration of a 32-byte transaction appears in *Figure 2*.

Address Phase

Mbus's 64 data bits are defined as a 36-bit physical address space and a set of control fields that determine the type of bus cycle that is being initiated. The master signals the beginning of a cycle by placing the required address and control information on the data bus and asserting an address strobe (MAS) on the bus. The command fields are

MAD(36 - 39) Transaction Type (Type)
- 0 read
- 1 write
- 2 coherent invalidate*
- 3 read coherent*
- 4 coherent write and invalidate*
- 5 coherent read and invalidate*

*Level 2 only

MAD(40 - 42) Transaction Size (Size)
- 0 Byte
- 1 Halfword
- 2 Word
- 3 Doubleword
- 4 16 Bytes
- 5 32 Bytes
- 6 64 Bytes
- 7 128 Bytes

MAD(43) Memory Cacheable (MC)

This advisory bit indicates whether the address space for the transaction is cacheable.

MAD(44) Locked Transaction (MLOCK)

This bit signals that the transaction is part of a multi-transaction operation that must be indivisible; thus, the master will not relinquish the bus between transactions.

Table 1. Transaction Status Encoding

MERR*	MRDY*	MRTY*	Meaning
H	H	H	Idle cycle
H	H	L	Relinquish and Retry
H	L	H	Valid Data Transfer
H	L	L	undefined L1, reserved L2
L	H	H	ERROR1 => Bus Error
L	H	L	ERROR2 => Timeout
L	L	H	ERROR3 => Uncorrectable
L	L	L	Retry

MAD(45) Boot mode/Local (MBL)

This bit signals that the processor is in the boot mode or that the transaction is in the local space (Address Space Identifier (ASI) = 01). This is an advisory bit that the system can use, but is not required for compliance.

MAD(46 - 49) Virtual Address

This field contains bits 12 - 19 of the virtual address being accessed. These bits are used by virtually indexed secondary caches for synonym elimination, and they are only required in multiprocessing level 2 compliant systems.

MAD(50 - 59) Reserved

MAD(60 - 63) Module Identifier

These bits contain the ID(0 - 3) for the master initiating the transaction. Used by slave elements to keep track of which master to reconnect to when implementing Relinquish and Retry operations, these bits are used only for multiprocessing level 2 compliance.

Data Phase

The element that occupies the physical address defined during the address phase responds to the request by either accepting 64 bits of data for a write or providing data for a read. The slave signals the master its readiness to complete a data transfer by asserting a ready status on the bus. This provision allows a slave to operate at a data rate slower than that available on Mbus.

The Mbus command protocol supports up to 16 successive data transfers. This allows up to 128 bytes to be transferred in 17 system clock cycles.

Figure 3 illustrates a simple read and write transaction. For transactions that require multiple data phases (more than 8 bytes), Mbus supports an address wrap feature within the block being transferred. An address wrap is accomplished by specifying a burst starting address that is not on a block boundary. This feature can be useful for cache line transfers, where the CPU is waiting for a specific word. This word is transferred first, allowing the CPU to proceed while the balance of the cache line is transferred.

Block wrapping is implemented by not allowing the addresses accessed to cross a block boundary. When the

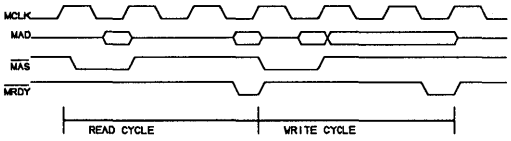


Figure 3. Mbus Read and Write Transaction Timing

starting address is not on a block boundary, the address sequence increments to the block boundary and then wraps to the block's start boundary. A simple example of a 32-byte transfer is illustrated below:

Block boundary is at 100000000000
100000010000 starts on the third 8-byte subblock
100000011000
100000000000 wraps to the start of the block
100000001000

Data Control Lines

Mbus provides two multiplexed data control lines:

MAS Address Strobe

The current bus master asserts this line for one clock cycle when a bus transaction's address phase is executed. The slave occupying the specified address in the physical memory space is expected to capture the address and command fields on the bus when MAS is asserted.

MRDY Ready

Slave elements use this line to signal to a master that requested data is ready for a read or data has been accepted for a write. A master monitors MRDY to know when a slave is ready for the next cycle in a data transaction.

Mbus Transaction Status and Encoding

MRDY combines with the MRTY and MERR control lines to encode the current status of a transaction's data phases. The slave element controls the status lines and thus determines how the current data phase cycle is terminated. The status encoding appears in *Table 1*.

The rest of this section describes the Mbus transaction activities.

The *Idle cycle* occurs when a slave element is not yet ready to transfer data to or from the master. The cycle occurs when the slave does not assert any of the status lines. The idle cycle thus effectively operates as a wait cycle on the bus. Note that this encoding also appears on the bus when there are no transactions currently being executed.

The *Data Transfer cycle*, executed by the slave element asserting MRDY for one bus cycle, indicates to the master that the slave is ready for the requested data transfer for the current data phase cycle.

The *Retry cycle* causes the master to restart the full bus transaction with the address and all data phases repeated. This cycle is often useful for memory modules

executing an ECC data correction that needs additional time.

The *Relinquish and Retry cycle* operates the same as a Retry cycle, except that the master must release the bus and re-arbitrate before starting the transaction cycle again. A Relinquish and Retry cycle typically is used for devices that have a long data latency or when the module is busy and cannot respond.

The *Bus Error* status is typically used to signal that an external bus error has occurred. This could be a bus parity error or invalid status. Note that this encoding is only a suggested definition. You can use the error encoding as a system-specific error if desired.

The *Bus Timeout Error* is generated by an external watchdog timer to signal that the time allotted for a full bus transaction has expired. It is important to note that this error applies to transactions requiring from one to 16 data phases, and the time limit chosen must accommodate the transaction requiring the greatest time. The suggested timeout interval for Mbus is 200 µs. This encoding to identify a timeout error is only a suggested definition. You can use the error encoding as a system-specific error if desired.

An *Uncorrectable Error* is typically generated by memory elements that encounter an uncorrectable error, such as parity or a multi-bit ECC error, in the data being accessed. This encoding to identify an uncorrectable error is only a suggested definition. You can use the encoding as a system-specific error if desired.

Interrupt Support

Mbus provides four dedicated lines, IRL[0:3], for feeding the current interrupt level to the processor. These lines typically connect directly to the CPU's interrupt inputs. An external interrupt controller is expected to drive the interrupt lines.

The four lines operate as an encoded, 16-level, priority interrupt request, ranging from no interrupt pending (0000) to non-maskable interrupt request (1111). The system is expected to include a separate interrupt request encoder to drive the IRL lines for Mbus.

Arbitration Mechanisms

Transfer of bus ownership on Mbus is accomplished using dedicated request and grant control lines from a central arbiter to the system's bus masters. The current master controls a busy status line (MBB) to signal that the bus is in use.

Arbitration between masters can occur concurrently with data transactions. This is accomplished in the following manner: When a master requires the bus, the master asserts its bus request (MBR) to the arbiter. The arbiter responds by asserting the bus grant (MBG) for the requesting master and deasserting the MBG for the current master. The new master deasserts its MBR on the next system clock cycle. When the requesting master detects the grant, that master does not take ownership of the bus until the bus busy (MBB) is inactive. This allows the current data transaction to complete before ownership is transferred.

This protocol places several requirements on the arbiter and the bus masters:

The current master must deassert MBB after the completion of a data transaction.

The current master must have its MBG active to initiate a new data transaction. The arbiter signals a master that it no longer has bus ownership by deasserting the MBG.

The arbiter is not allowed to re-arbitrate new requests after a new grant until MBB is deasserted.

Details of the control algorithm for bus arbitration appear in *Figure 4*, which is a state flowchart for a bus master arbitration state machine.

Module Identification and Configuration

An optional facility allows the CPU to identify and configure modules attached to the Mbus. This facility provides up to 16 logical positions on the bus, with the requirement that each logical position contain a small memory space dedicated to that logical position. An Mbus module supporting the configuration facility can incorporate any control or status registers required within its assigned memory space. Four dedicated lines are provided for each Mbus module (ID[0:3]) to identify the logical position the module occupies on the Mbus.

In a typical configuration, each slot on the Mbus has a unique value hard-wired on its ID control lines. A module decodes its configuration map space in a specific slot by using the ID value.

An Mbus Port Register (MPR) — a single 32-bit word at location FFFFFCh in the configuration space — is defined with a standard format to allow a uniform identification mechanism for a module. The format of the Port Register is defined in *Figure 5*, and the configuration address map for the 16 ID values is defined in *Table 2*.

The MPR fields are defined as follows:

MDEV - Mbus Device Identification Number

This field contains a unique vendor-defined identification number for the Mbus device being addressed.

MREV - Mbus Device Revision Number

This field contains the revision or configuration number for the Mbus device being addressed.

MVEND - Mbus Vendor Number

This field contains a unique vendor identification number for the Mbus device being addressed. The current vendor number assignments are:

0 Fujitsu
1 Cypress
2 (reserved)
3 LSI Logic
4 Texas Instruments

Note that on reset, a processor begins execution at location 0FF000000h. This is the same memory space as the first logical position in the configuration space. Thus, the ID = 0h logical position must be treated as predefined

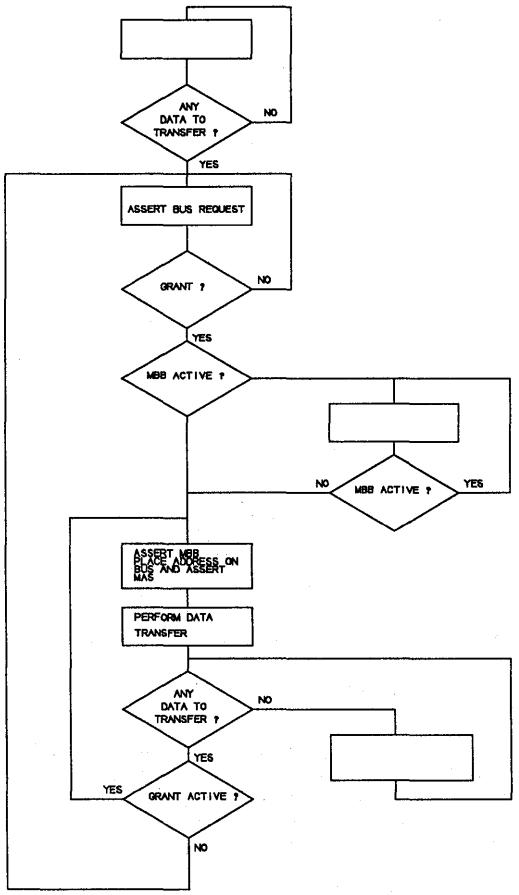


Figure 4. Bus Arbitration Flowchart

and considered the logical position for the Mbus boot PROM module.

AC Timing Parameters

Because the Mbus is fully synchronous, with all data sampled on MCLK's rising edge, all AC parametrics are specified as set-up and hold times with respect to this edge. The signals are grouped into two categories: data path (MAD) and control (CNTRL). The AC specifications are provided in *Table 3*. *Table 4* summarizes the DC characteristics and reflects the assumption that the maximum loading per module is a single CMOS load per line.

Processor Modules

Another part of the Mbus standard defined by SPARC International is a physical connector that allows you to take advantage of a wide variety of standard

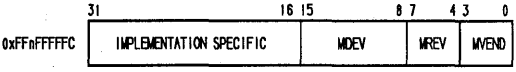


Figure 5. M-Bus Port Register Configuration

modules from multiple vendors. These modules are typically for memory, I/O devices, and bus adapters. Modules can be configured to be mounted either parallel or perpendicular to the mother board.

Cypress has developed a family of processor modules that incorporate the standard Mbus connector. These modules include a uniprocessor cluster module, a multiprocessor cluster module, and a dual multiprocessor cluster. A cluster is considered to be an integer unit, floating point unit, memory management unit, and a 64K cache.

The Mbus connector is available as a standard component from Amp Incorporated and has 100 signal pins in a dual row on 0.05-inch centers. The signal interconnects through the connector are a constant 50Ω impedance. Separate power and ground blades minimize the supply-rail impedance. *Table 5* defines the connector's pin assignments.

System Design Considerations

Virtually any Mbus-based system requires several support elements, including clock generator, watchdog timer, interrupt controller, and bus arbiter. The following sections examine the functional requirements and design considerations for each of these support elements. The support functions are relatively straightforward, and can be implemented with four PLDs, a TTL buffer and possibly a flip-flop.

Clock Generator

The system clock is derived from the clock generator, which should be crystal referenced. For many applications, a simple crystal-controlled oscillator module performs very well. On the other hand, operating with a 2x clock followed by a toggle flip-flop might be useful if the application requires true and complement clocks. Note that Mbus does support a true and complement clock distribution, although it is *not* required.

Clock distribution on the bus should be implemented using a single printed circuit trace with no stubs and characteristic impedance of 50 to 75Ω. The line must be

Table 2. Mbus Address Configuration Map

Configuration Spaces	Mbus Identifier
0xFF0000000 to 0xFF0FFFFFF	Range for ID=0x0
0xFF1000000 to 0xFF1FFFFFF	Range for ID=0x1
0xFF2000000 to 0xFF2FFFFFF	Range for ID=0x2
.	.
.	.
.	.
0xFFF000000 to 0xFFFFFFFFF	Range for ID=0xF

Table 3. Level 1 DC Characteristics

Level 1 DC Characteristics and Pin Capacitance (Ta = 0-70C)					
Symbol	Signal Description	Conditions	min	max	unit
Vih	Input High Voltage level		2.1	Vcc	V
Vil	Input low Voltage level		0.0	.8	V
Iil	Input Leakage			+- 1.0	uA
Iih	Input High Current			10	uA
Iilo	Input Low Current			- 10	uA
Voh	Output High Voltage	Ioh = -2mA	2.4	Vcc	V
Vol	Output Low Voltage	Ioi = 8mA	0.0	0.5	V
Cin	Input Capacitance			10	pF
Cout	Output Capacitance			12	pF
Ci/o	Input/Output Capacitance			15	pF

properly terminated. The requirements for clock distribution dictate the use of a low-propagation-delay buffer with the ability to drive the transmission line. If complementary clock distribution is required, the buffers must also have low delay skew.

Watchdog Timer

The watchdog timer provides the timing reference required for bus timeout error detection. The Mbus recommendation for the timeout interval in a 40-Mhz system is 200 µs. The actual value chosen for an application depends on the system clock rate and the worst-case transaction time of any element on the bus. Normally a value between 100 and 500 µs is adequate.

While a transaction is in process, the current master asserts the Bus Busy status line (MBB), which serves as the controlling status for the watchdog timer. Each time MBB is asserted, the timer is triggered. If the timer reaches terminal count before MBB is deasserted, a Bus Timeout Error is generated.

The watchdog timer can also be used to generate timing for the bus reset strobe (MRST). This is possible because the watchdog function does not have to operate during reset. The additional logic required to support both functions is minimal.

You can implement the watchdog timer in a pair of 22V10 PALs. *Figure 6* shows a block diagram of the function, with the flowchart for the reset state machine shown in *Figure 7*. The design incorporates a single counter, the watchdog timer, and the reset function.

Two counters make it possible to implement the function in two 22V10s. The modulo 40 counter uses a synchronous count enable connected to the terminal count of the modulo 250 counter. Thus, 250 x 40 = 10000 clock cycles for the timer to reach terminal count. At 40 Mhz, this value corresponds to a 200 µs timeout interval.

When RUN is asserted, the reset state machine is quiescent, MBB* is not asserted, and the counter chain is held reset. When MBB* asserts, the counter chain begins counting toward terminal count. In normal operation, MBB* is deasserted long before terminal count is reached, and the timer returns to the reset state. If MBB* remains asserted until terminal count is reached, however, MERR* and MRDY* are asserted for one clock cycle. The current master is expected to respond to this condition by terminating the transaction and deasserting MBB*.

In the case of a reset condition, it cannot be predicted if MBB* will be asserted at the start of the reset interval. It is therefore necessary to gate-out MBB* from the timing block during a reset interval. This is accomplished with GATE from the reset state machine.

Table 4. Level 1 AC Characteristics

Parameter	min	max	Unit
Tcp	25	25	ns
Tch	11	14	ns
Tci	11	14	ns
Tsi(MAD)	3	-	ns
Thi(MAD)	2	-	ns
Tdo(MAD)	-	18	ns
Tho(MAD)	4	-	ns
Tsi(CNTRL	3	-	ns
Thi(CNTRL)	2	-	ns
Tdo(CNTRL)	-	18	ns
Tho(CNTRL)	4	-	ns
* All times are for a Capacitive load of 100 pF			

Table 5. Mbus Connector Pin Assignments

Pin #	Signal Name	Pin #	Signal Name
1	SPARE1	51	SPARE7
2	SPARE2	52	MERR_
3	SPARE3	53	SPARE8
4	TOE_CMU	54	MAS_
5	SPARE4	55	MBR[1]_
6	MIRLO[1]	56	MBB_
7	MIRLO[0]	57	MBG[1]_
8	MIRLO[3]	58	SPARE9
9	MIRLO[2]	59	MAD[32]
10	SPARE5	60	MAD[33]
11	MAD[0]	61	MAD[34]
12	MAD[1]	62	MAD[35]
13	MAD[2]	63	MAD[36]
14	MAD[3]	64	MAD[37]
15	MAD[4]	65	MAD[38]
16	MAD[5]	66	MAD[39]
17	MAD[6]	67	MAD[40]
18	MAD[7]	68	MAD[41]
19	MAD[8]	69	MAD[42]
20	MAD[9]	70	MAD[43]
21	MAD[10]	71	MAD[44]
22	MAD[11]	72	MAD[45]
23	MAD[12]	73	MAD[46]
24	MAD[13]	74	MAD[47]
25	MAD[14]	75	MAD[48]
26	MAD[15]	76	MAD[49]
27	MAD[16]	77	MAD[50]
28	MAD[17]	78	MAD[51]
29	MAD[18]	79	MAD[52]
30	MAD[19]	80	MAD[53]
31	MAD[20]	81	MAD[54]
32	MAD[21]	82	MAD[55]
33	MAD[22]	83	MAD[56]
34	MAD[23]	84	MAD[57]
35	MAD[24]	85	MAD[58]
36	MAD[25]	86	MAD[59]
37	MAD[26]	87	MAD[60]
38	MAD[27]	88	MAD[61]
39	MAD[28]	89	MAD[62]
40	MAD[29]	90	MAD[63]
41	MAD[30]	91	SPARE10
42	MAD[31]	92	SPARE11
43	MBR[0]_	93	SPARE12
44	MSH_	94	SPARE13
45	MBG[0]_	95	SPARE14
46	MIH_	96	AERR_
47	MCLK0	97	RSTIN
48	MRTY_	98	SPARE15
49	SPARE6	99	SPARE16
50	MRDY_	100	SPARE17
Blade1	Graound	Blade2	+ 5V
Blade3	Ground	Blade4	+ 5V
Blade5	Ground		

When a reset occurs, RUN is deasserted. The state machine deasserts GATE to disable MBB* and hold the counter in the reset state. When RUN goes active, COUNT is asserted. This enables the counter, disables MERR* and MRTY*, and causes MRST* to assert. The state machine remains in this state until terminal count (TC) from the counter is detected. The state machine then asserts GATE and deasserts COUNT. The latter deasserts MRST* and enables MBB* to control the triggering of the timing chain. The state machine remains in this state until another reset occurs.

Interrupt Control

Interrupt processing for Mbus-based systems requires a simple priority encoder that uses individual interrupt requests to determine the priority level. The interrupt controller then drives the bus's four Interrupt Status Lines (ISL 0 - 3). System elements that generate interrupts are expected to assert their individual interrupt request line and hold it asserted until the processor takes action to clear the interrupt condition. You can easily implement this function in a single 22V10 with 16 inputs and four outputs.

Bus Arbitration

Most Mbus-based systems require some type of bus arbitration. In addition to the processor requiring access to the bus, I/O devices such as disk drives require access to the memory space for data transfer. Thus, the system needs at least a simple arbitration mechanism to allow the processor and the I/O device to share the bus.

You can implement many different arbitration strategies in an Mbus system. These strategies include fixed priority, round robin, dynamic assignment, or random priority. System performance requirements largely dictate the arbitration strategy for a specific application. The arbiter must, in any case, conform to the interface protocol defined by Mbus.

For a good example of Mbus arbitration, see the Cypress application note "Using the CY7C330 as a Multichannel Mbus Arbiter." This application note shows how to implement two different arbitration algorithms in a single CY7C330 PLD. Note that the design requires the availability of a 2X clock.

DRAM Memory Module Design

Several issues must be resolved in defining an Mbus memory module. The module's capacity, the required performance level, and cost are the basic constraints that dictate the module's design.

For reasonable performance, the memory must support Mbus's full 64-bit access per memory cycle. This implies a minimum capacity of 8 Mbytes for a 1M x 1 DRAM design. This is a reasonable minimum size and capacity increment. Alternatives include a 1M x 4, which reduces parts count but increases cost; a 256K x 4, which also reduces parts count and the minimum capacity to 2 Mbytes; and finally a 4M x 4, which increases the mini-

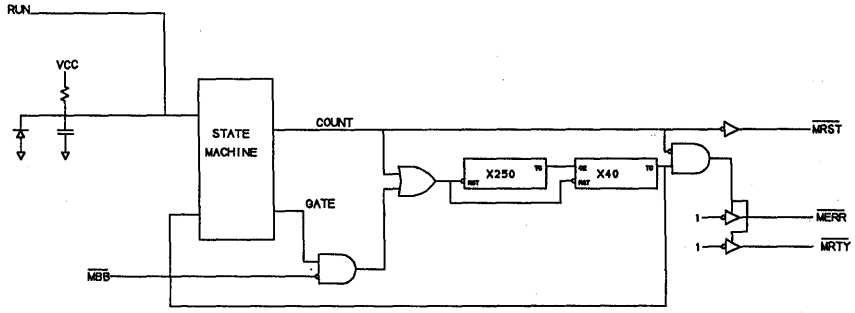


Figure 6. Watchdog Timer Block Diagram

mum capacity to 32 Mbytes with no increase in parts count.

Figure 8 shows a curve relating a system's relative performance to Mbus wait cycles. The curve is derived for 32-byte cache line replacements and a 95 percent cache hit ratio. Note that the relative performance does not strongly depend on the number of wait cycles. For example, doubling the transaction time for a cache line replacement to five wait cycles reduces performance by only 13.5 percent.

Thus, the curve indicates that the incremental improvement in performance for a reduction in the number of wait cycles is somewhat marginal, with only a 2- to 3-percent increase in performance for each wait cycle eliminated. You must therefore evaluate the relative cost for a given performance level to determine if an approach is cost effective.

Table 6 illustrates several performance-cost design points for 40-, 33-, and 25-Mhz systems. The small performance difference between the lowest-cost design for a specific clock frequency and the highest-performance design makes the low-cost implementation quite attractive from a cost-performance standpoint.

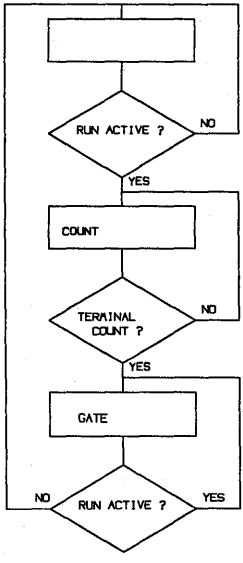


Figure 7. Reset State Machine Flowchart

An Example Memory Design

To better understand Mbus memory module design, consider an example of a design for a 25-Mhz system with 8-Mbyte capacity and 128K of boot PROM. The design supports the module identification facility. This example illustrates the design requirements without the additional issues involved in a full-speed, 40-Mhz module. A block diagram of the module appears in *Figure 9* and consists of three major functional blocks: interface decode, DRAM, and PROM/identification generation.

Interface Decode

The interface decode block decodes Mbus commands, and generates control signals, and supports the DRAM and PROM blocks for data transfer across the data bus. The decode block thus detects Mbus commands directed

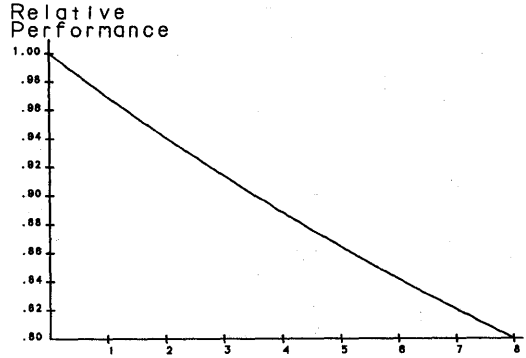


Figure 8. Relative Performance vs M-Bus Wait States

at the module; informs the DRAM, PROM or ID generation block of the request; and supports the transaction by providing control of the data transceiver, generating the MRDY as needed, and terminating the transaction when complete.

The decode block can locate the DRAM anywhere in the system's physical address space on a 1 Mbyte boundary. This is accomplished using the Mbus configuration facility to load the DRAM position in the module's port register.

A block diagram of the decode function appears in *Figure 10*. The block consists of a high-speed bus decode PLD, a pair of 22V10s for the address decode, the Mbus port register that contains the DRAM map position, and an auxiliary decode PLD for control decode within the module. The interface decode also provides transaction control for the data-bus buffer. The buffer is an FCT648 transceiver/register, which can be configured with a combinational or registered data path in either direction. The DRAM design requires a registered data path on a read operation.

The bus decode block provides the buffered clock for the module and generates MRDY out to the Mbus when the module is active. The ready signal is derived from the DRAM and PROM blocks' RDYSTB signals.

The block generates LDSTB to the decode registers when MAS is asserted on Mbus. The block detects that the module is active by monitoring the match signals from the address decode.

The decode block controls the bus buffer via BUSDIR and BUSSEL. BUSDIR normally causes data to go from Mbus into the module but reverses direction when the module is accessed and the transaction is a read operation. BUSSEL controls the type of output data path from the module for the transaction. When a DRAM read access occurs, the data path is registered; when a PROM or configuration port read access is executed, the datapath is combinational. These transactions are decoded using the match decodes and RD\WT. The bus decode function terminates the transaction by using CLR to clear the block's decode registers. This is initiated from either the DRAM or PROM block through the CLRSTB signals.

The address decode block decodes DRAM addresses by comparing the Mbus address to the map position for a match (more on the map position later). The PROM decode is a simple decode for address 000000000 to 00003FFFF. Two match signals for DRAM and PROM are implemented to avoid the additional delay that would occur from ANDing the decode outputs from the two PLDs. The two signals are ANDed in the DRAM and PROM control elements with no additional delay overhead.

The MPR block implements the write portion of the Mbus configuration facility. The decode PLD detects the configuration address space and matches the ID field to ID(0 - 3) for a module match. This condition is signaled with CMAT. The map position register is loaded from the data bus when CMAT is asserted and the transaction type is a write. The PROM block is responsible for the transaction termination via the CLRSTB.

The auxiliary decode block is a simple decode PLD that captures the transaction size and read/write status. Note that BOOT and LOCK are decoded in the auxiliary block, but are not used in this design example.

DRAM Block

The DRAM block is implemented using 70-ns RAMs operating in page mode with two wait states to initial access and zero wait states for up to 32 bytes. For transactions requiring more than 32 bytes, an additional wait state is required after every fourth transfer cycle. The additional wait state is necessary because the DRAM operates in page mode at a 50-ns cycle time. This causes the data access to skew out 10 ns per cycle, and an additional cycle allows the data access to resynchronize with MCK. *Figure 11* shows the basic timing for a 32-byte read transaction.

Among the numerous approaches to DRAM control design, the implementation required for an Mbus page-mode controller has no special peculiarities except for the

Table 6. Memory System Perfomance/Cost Anaylsis

Clock MHZ	Description	Wait Cycles	Absolute Perfomance	Relative Performance	Relative Complexity	Relative Cost
40	35ns 1Mx1 BiCMOS Non-multiplexed	4	0.88	0.88	0.8	2.0
40	70ns 256Kx4 Static Col 2 way interleave	5	0.86	0.86	1.5	1.5
40	60ns 1Mx1 DRAM Fast Page Mode	7	0.83	0.83	1.0	1.2
33	35ns 1Mx1 BiCMOS Non-multplexed	2	0.77	0.94	0.8	2.0
33	70ns 1Mx1 DRAM Fast Page Mode	4	0.72	0.88	1.0	1.0
25	45ns 1mx1 BiCMOS Non-multiplexed	1	0.60	0.97	0.8	1.7

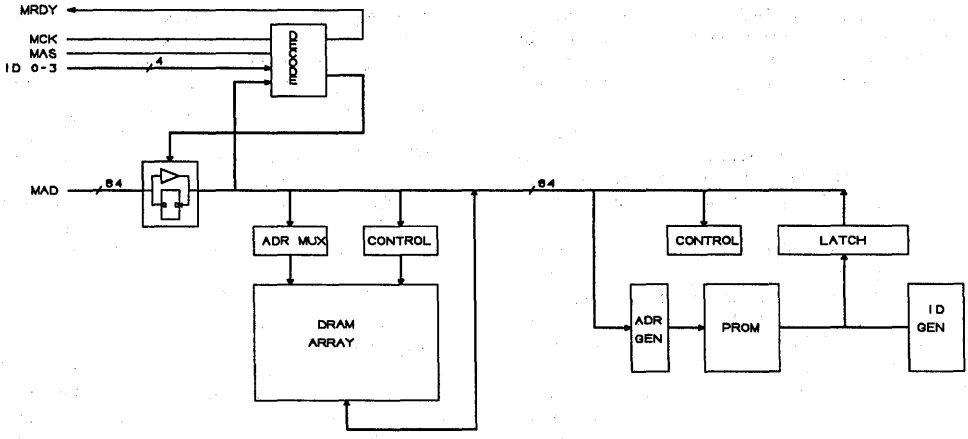


Figure 9. Mbus Memory Module Block Diagram

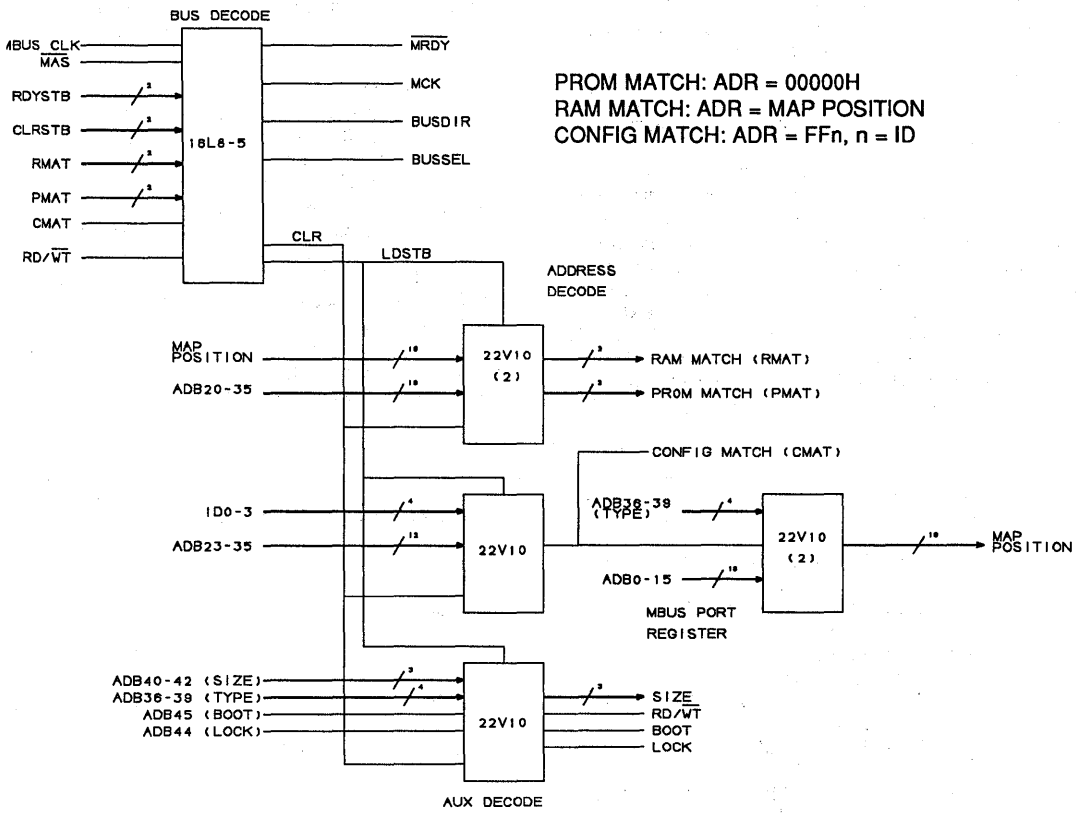


Figure 10. Interface Decode Block Diagram

address map requirement. This requirement affects only the page-address generation. The control block appears in *Figure 12* and consists of a 10-ns timing chain generator, a high-speed control state machine, an interval counter, and a refresh counter.

The timing chain generator derives a 10-ns interval from MCK using a 10-ns tapped delay line and a high-speed decode PLD. The output clock drives the control state machine and the interval counter.

The control state machine is a conventional finite state machine that generates the direct DRAM control, the data buffer strobe, and ready/clear timing to the interface decode block. The state machine uses the interval counter for the fixed idle intervals to reduce state-transition complexity. A DRAM cycle is initiated when RMAT is asserted, and the cycle type is determined by RD/WT. Refresh requests are also monitored from the refresh counter (RFREQ) and acknowledged (RACK).

The interval counter measures timing intervals with a resolution of 10 ns for the control state machine. The fixed intervals are the initial access delay (50 ns) and the RAS precharge (70 ns). In addition, the transaction size is decoded and used to count data transfers. Terminal count signals that the count interval is complete. A second terminal count is used during transfer counting to signal when the count equals 4 for the additional wait state.

Figure 13 shows the address generation block, which is implemented with three high-speed PLDs. The block generates the row address for the DRAM array on the initial access and provides the column addresses for page-mode operation. The column address sequence implements the address-wrap feature described earlier and decodes SIZE to determine the wrap point.

Figure 14 shows a functional block diagram of the PLD. It acts as a column-address latch — a presetable counter with a select mux for the preload input. This approach allows the counter to be loaded with the row or column address. The control decode uses the CMD input from the control state machine to control the counter and steer the data inputs. The SIZE input carry modifies the in and out operations for address wrapping.

PROM/ID Generation

The PROM and ID generation block combines the boot ROM function and the module identification block. Because the two elements are both read only, it is sensible to combine the functions under a single control element. The block diagram appears in *Figure 15*. A state machine controls the PROM and ID generation, and both elements share a set of output buffer registers.

A single 256K x 8 PROM implements the PROM block, which requires eight sequential accesses to assemble a full 64-bit word for transfer across the Mbus. The controller includes eight load strobes to sequentially load the bytes into the assembly registers. The PROM address generator drives the address input to the PROM and increments the address on command from the control state machine. The address generator also accommodates address wrapping by decoding SIZE from the interface decode block. This implementation reduces the component count to a minimum compared to a parallel-access approach, but is considerably lower in performance. When the PROM is used only for bootstrapping, the cost savings and higher density can be attractive.

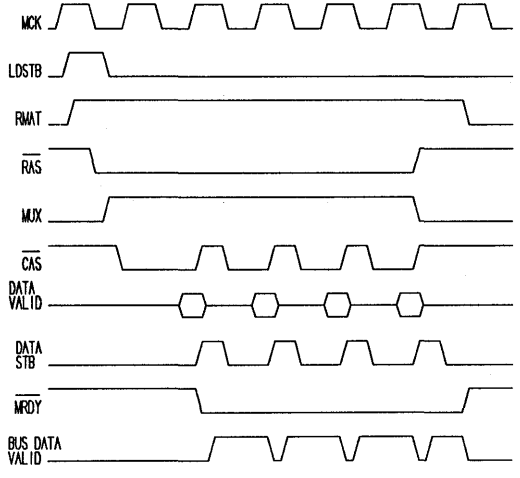
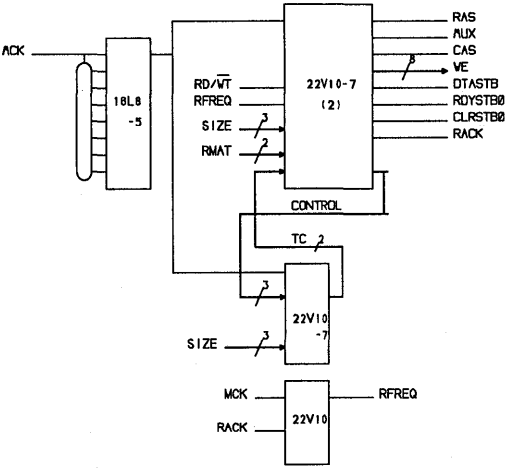


Figure 11. DRAM Control Timing



CONTROL	FUNCTION
000	LD ACCESS DELAY (5)
001	LD XFER COUNT (16 MAX)
010	CD RAS PRECHARGE (7)
011	COUNT
1XX	IDLE

Figure 12. DRAM Control Block Diagram

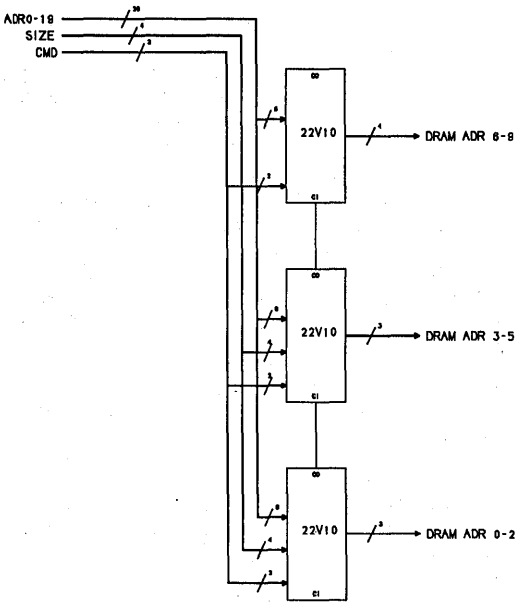
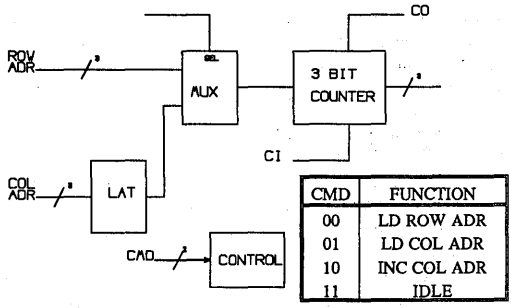


Figure 13. DRAM Address Generation Block Diagram



CMD	FUNCTION
00	LD ROW ADR
01	LD COL ADR
10	INC COL ADR
11	IDLE

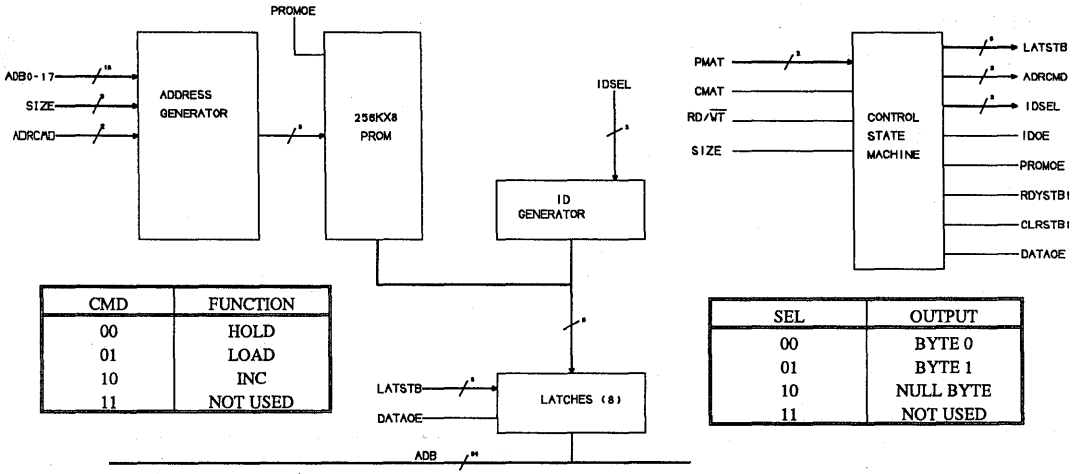
Figure 14. Individual Address Generator

The ID generator contains a 16-bit identification word to uniquely identify the module. The word is located in the lower 16 bits of the Mbus Port Register (FFFFFFFCh) and is implemented as a simple hard-wired block that outputs byte 0, byte 1, or a null byte, as directed by the controller. When a read access to the configuration space occurs, the controller loads the identification word into the lower 2 bytes and null bytes in the remaining byte locations.

References

1. *SPARC Mbus Specification* Rev 1.1, Published by Sun Microsystems.

2. Cypress *SPARC Users Guide*.



CMD	FUNCTION
00	HOLD
01	LOAD
10	INC
11	NOT USED

SEL	OUTPUT
00	BYTE 0
01	BYTE 1
10	NULL BYTE
11	NOT USED

Figure 15. PROM and ID Generation Block Diagram

CYPRESS SEMICONDUCTOR

Multiprocessing System Boot-Up

This application note describes a simple scheme to arbitrate granting of the Mbus among competing processing modules during system boot. This approach is not the only workable solution, but it is offered as a suggestion to SPARC multiprocessing system designers.

In a shared-bus multiprocessing system, all processing nodes simultaneously request the bus upon reset to execute their processor boot code. You must provide the mechanism to make the multiple processing nodes boot-up in their proper sequence. *Figure 1* illustrates the mechanism described here.

Boot-up Procedure

Upon release of the power-on reset signal, all processor modules (CY7C601s with CY7C605-based cache systems) wake up in boot mode and request the Mbus by asserting $\overline{\text{MBR}}$. Because there is one $\overline{\text{MBR}}$ signal for each CY7C605, the Mbus arbiter can identify the processors according to which $\overline{\text{MBR}}$ signal is asserted. By establishing a priority for the processor modules based upon their $\overline{\text{MBR}}$ signals, the arbiter can control which processor is allowed to boot first.

To allow a processor module to complete its boot procedure, the Mbus arbiter locks the grant for the processor module until the boot routine is completed. For example, processor module 0 asserts $\overline{\text{MBR}}$0 after reset, along with all other processor modules and their respective $\overline{\text{MBG}}$ signals. If processor module 0 is the highest-priority processor, the Mbus arbiter asserts $\overline{\text{MBG}}$0 to grant this module access to Mbus. During this boot procedure, the Mbus arbiter locks grant of the Mbus to processor module 0, ignoring all other Mbus requests. This allows the processor node to keep the Mbus until the node has finished booting.

After a processor module has booted-up, the Mbus grant must be unlocked. One way to do this is to use an Mbus-arbiter reset control register. This register is cleared upon power-on reset. After each processor finishes booting, that processor completes its boot routine by setting a bit in the reset control register. The Mbus arbiter uses the setting of a new bit to release the Mbus grant to that processor module. The Mbus arbiter then asserts grant to the next processor module, assuming that all the boot routines are not yet completed. Upon a processor module setting the last bit in the reset register, the Mbus arbiter leaves boot mode and assumes normal operation.

Note that a processor module can also determine its module identification number by reading the value stored in the reset control register. (Refer to the MID notes at the end of this application note.) Assuming that processing nodes might have unique portions of boot code, the value stored in the reset register can also be used to branch to different areas of boot code for each processor. The module identifier (MID) number can be read from the boot program for that processor, or determined from the reset register, and written into the CY7C605's SCR (system control register). Initializing the CY7C605 SCR's MID field is necessary if the CY7C605 is to supply the module identifier field of an Mbus address cycle. The module identifier field identifies which Mbus master has asserted an address on the Mbus and is highly useful to some multiprocessing systems.

Module Identifier (MID) Notes

Level-2 Mbus uses the MID field in the CY7C605's SCR to identify the module asserting an address onto the Mbus. *Figure 2* shows where the SCR's MID field is asserted in the Mbus address cycle. This information can be used by the Mbus arbiter or by a secondary cache to note which module asserted the current address on the Mbus. The MID(3:0) field of the CY7C605 system control register is writeable by asserting ASI = 4 H and the register address 0 H with the correct word to be written into the register.

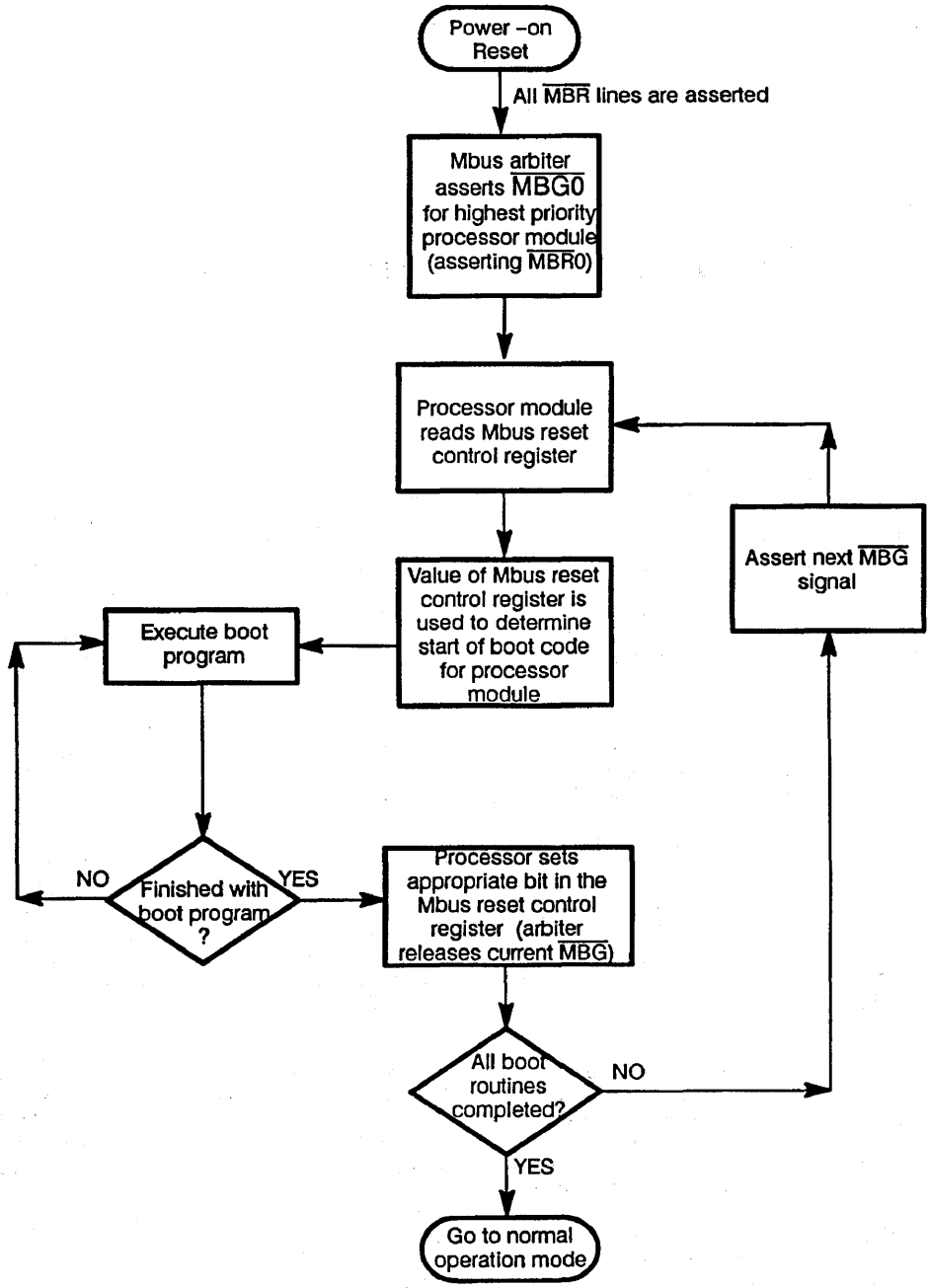


Figure 1. Boot-Up Mechanism

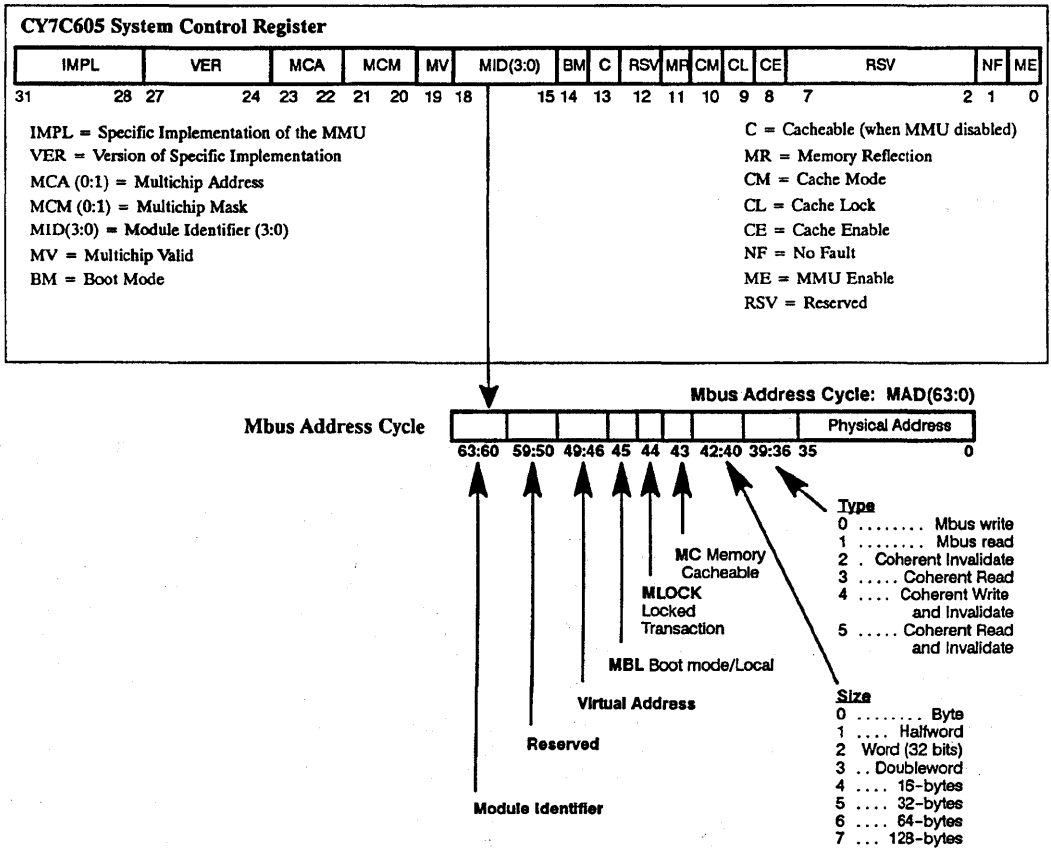


Figure 2. Module Identifier Field

CYPRESS SEMICONDUCTOR

Porting UNIX to the CY7C604 or CY7C605

This application note describes the issues involved in porting UNIX to a system that includes the CY7C601 SPARC microprocessor and either the CY7C604 or CY7C605 cache and memory management units. The assumption here is that the UNIX operating system has already been ported, leaving only the task of integrating the CY7C604/605 hardware into the virtual memory/cache sections of the operating system (O/S). This application note specifically addresses SunOS, which serves as an example for porting any UNIX variant.

The Cypress CY7C604 and CY7C605 are cache and memory management units that interface to the CY7C601 microprocessor without glue logic. Porting an operating system to either of these chips is not a difficult task, but it requires a complete understanding of what functions the CY7C604/605 provide for the operating system and its hardware translation layer. This application note should help provide that understanding.

The CY7C604 and CY7C605

The CY7C604 and CY7C605 share the following features:

- 4096 contexts for translation look-aside buffer (TLB) entries
- 64 fully associative TLB entries
- Page-level memory access protection
- Multi-level address mapping
- 4096 contexts for cache tags
- Virtual cache support
- Read-line and write-line buffer

Both devices conform to the SPARC reference MMU standard. The CY7C605 differs from the CY7C604 in its support for reflective memory and of multiprocessing by concurrent, transparent bus snooping (without a clock penalty).

Porting to the UNIX O/S

The UNIX operating system is divided into independent objects or layers that communicate with each other via fixed data structures. (For a detailed description of this O/S implementation philosophy and details of the various layers, see *Reference 2*.) This object orientation of UNIX simplifies the task of porting UNIX to different hardware architectures.

The UNIX layer that affects a port to the CY7C604/605 is the *hat*, or hardware address translation layer. The *hat* implementation is machine dependent: its functions rely upon the underlying machine architecture. The *hat* layer is used by the machine-independent address space (*as*) layer, which provides the operating system constructs necessary for virtual memory support. The *hat* layer provides the functions and structures needed to control and operate the underlying cache/memory management hardware in the system. The *hat* layer implements machine-dependent functions with machine-*independent* interfaces, as listed in *Table 1*.

Because the *hat* functions are machine dependent, their implementation depends upon the underlying cache-control/memory-management hardware. Their interfaces are machine independent to allow the *as* layer to utilize any implementation of the *hat* layer.

The *hat* layer also contains the machine-dependent control parameters and description and operation parameters. The control parameters and description tell the *as* layer what the cache controller/memory management unit (CC/MMU) hardware can do and how to control the hardware (i.e., whether the cache can be locked, and if so, what control bit accomplishes the locking). The operation parameters set limits on the CC/MMU hardware capabilities, including the number of contexts supported, the number of segments per context, etc. You can find a list of the SunOS parameters for the SPARCstation CC/MMU in /usr/share/sys/sun4c/mmu.h and in *Tables 2* through *5*.

Porting to the CY7C604

The information in *Tables 2* through *5* reflects the CC/MMU hardware implementation for the SPARCsta-

Table 1. Machine-Dependent Functions

Operation	Function
hat_init()	Initialize the structures and hardware
hat_alloc(as)	Allocate *hat* structure for *as*
hat_free(as)	Release all *hat* resources for *as*
hat_pageunload(pp)	Unload all translations to page *pp*
hat_pagesync(pp)	Sync ref and mod bits to page *pp*
hat_unlock(seg, addr)	Unlock translation at *addr*
hat_chgprot(seg, addr, len, prot)	Change protection values
hat_unload(seg, addr, len)	Unload translations
hat_memload(seg, addr, pp, prot, flags)	Load translations to page *pp*
hat_devload(seg, addr, pf, prot, flags)	Load translation to cookie *pF* (a cookie is a contiguous section of memory)

Table 2. SPARCstation Hardware Context and Segment Info

NPMENTPERPMGRP	64	Number of page map entries/page map group
NCTXS	8	Number of contexts
NPMBRPPERCTX	4096	Number of segments/context
NPMGRPS	128	Number of segments
MNCTXS	8	Maximum number of contexts
MNPMGRPS	128	Maximum number of segments

Table 3. SPARCstation Address Space Identifiers

ASI_UP	0x8	User program
ASI_SP	0x9	Supervisor program
ASI_UD	0xA	User data
ASI_SD	0xB	Supervisor data
ASI_FCS	0xC	Flush cache segment
ASI_FCP	0xD	Flush cache page
ASI_FCC	0xE	Flush cache context
ASI_CTL	0x2	Control space
ASI_SM	0x3	Segment map
ASI_PM	0x4	Page map

tion, which is implemented in gate arrays. The CY7C604 and CY7C605 are custom circuits that have a higher level of integration and capability. They differ from the SPARCstation CC/MMU in several vital respects.

Most importantly, the SPARCstation uses a two-level virtual memory space consisting of a segment and a page table; the CY7C604/605 CC/MMUs use a three-level virtual memory space composed of three page tables. The CY7C604/605 CC/MMUs determine the value of non-cached virtual addresses by performing a table walk through page tables in memory. The hardware performs this table walk without software intervention.

Modifying SunOS to utilize the CY7C604/605 is more complex than just changing the parameters. An additional parameter must be defined to represent the different nature of the virtual address space. Using as a model the mmu.h for the SUN 4_460 (which employs a three-level

Table 4. SPARCstation ASI Control Constants

CONTEXT_REG	0x30000000	Context register
SYTEM_ENABLE	0x40000000	System enable
SYNC_ERROR_REG	0x60000000	Synchronous error register
SYNC_VA_REG	0x60000004	Synchronous virtual address register
ASYNC_ERROR_REG	0x60000008	Asynchronous error register
ASYNC_VA_REG	0x6000000C	Asynchronous virtual address register
CACHE_TAGS	0x80000000	Cache tags
CACHE_DATA	0x90000000	Cache data
UART_BYPASS	0xF0000000	UART bypass

Table 5. SPARCstation Cache Constants

VAC_SIZE	0x10000	Virtual address cache size
VAC_LINESIZE	16	Virtual address cache line size
VAC_LINESHIFT	4	Line size in base 2
VAC_CTXFLUSH_COUNT	4096	Virtual address cache context flush count
VAC_SEGFLUSH_COUNT	4096	Virtual address cache segment flush count
VAC_PAGEFLUSH_COUNT	256	Virtual address cache page flush count

virtual address space), the mmu.h file for a system based on the CY7C604 has the parameters listed in *Tables 6* through *9* (new parameters are listed in boldface type).

The addition of NSMENTPERSMGRP, NSMGRPPERCTX, and NSMGRPS works directly with the SunOS because these constants are already defined and used by the SUN 4_460 O/S variant. However, the CY7C604 provides address space identifier (ASI) functions that are unavailable from the SPARCstation MMU. The additional functions are:

- Directly addressing the Mbus extended address space
- Flushing entries from the TLB
- Probing entries in the TLB
- Directly accessing the cache tags
- Flushing the cache lines for a memory region
- Flushing the cache lines for a specific user
- Operating in pass-thru mode

To use these additional functions, the O/S must be modified in two ways. First, the ASI values must be defined and made available by modifying the header files. Second, the *hat* functions in mmu.c must be modified to utilize these new capabilities when performing the *hat* functions. The calls to the *hat* layer from the *as* layer can remain the same; only the underlying implementation of those calls by the *hat* procedures must change.

Table 6. CY7C604 Hardware Context and Segment Information

NPMENTPERPMGRP	64	Number of page map entries/page map group
NSMENTPERSMGRP	64	Number of segment entries/segment group
NCTXS	4096	Number of contexts
NSMGRPPERCTX	256	Number of segment groups/context
NSMGRPS	64	Number of segment groups
NPMGRPS	64	Number of page map groups
MNCTXS	4096	Maximum number of contexts
MNPMGRPS	64	Maximum number of segments

Some of the CY7C604's ASI control constants are also new to the SunOS. Although the SPARCstation has a subset, the CY7C604 offers the O/S more control over the working of the cache and memory management. These capabilities are set by manipulating the bit fields in the SCR (System Control Register). The following additional capabilities are available when using the CY7C604:

- Enable/disable virtual cache
- Lock the entire cache
- Set the cache to write-through or copy-back operation
- Set instruction or data access to cacheable or non-cacheable when the MMU is disabled
- Set the CY7C601 for multichip mode
- Set the CY7C604 to signal/not-signal data access exceptions to the CY7C601
- Enable/disable the MMU
- Read/write the Context Table Pointer register
- Read/write the Instruction Access Page Table Pointer (IPTP) register
- Read/write the Data Access Page Table Pointer (DPTP) register
- Read/write the Root Pointer register

Table 7. CY7C604 Address Space Identifiers

ASI_EA	0x1	Mbus extended address space
ASI_MFP	0x3	MMU flush/probe
ASI_CTL	0x4	Control space (register access)
ASI_MDT	0x6	MMU diagnostics instruction/data TLB
ASI_UP	0x8	User program
ASI_SP	0x9	Supervisor program
ASI_UD	0xA	User data
ASI_SD	0xB	Supervisor data
ASI_CT	0xE	Cache tag access
ASI_FCP	0x10	Flush cache line (page)
ASI_FCS	0x11	Flush cache line (segment)
ASI_FCR	0x12	Flush cache line (region)
ASI_FCC	0x13	Flush cache line (context)
ASI_FCU	0x14	Flush cache line(user)
ASI_PTA	0x20-2F	MMU passthrough mode physical address

Table 8. CY7C604 ASI Control Constants

CONTROL_REG	0x0	System control register (SCR)
CONTEXT_PTR	0x100	Context table pointer register (CTPR)
CONTEXT_REG	0x200	Context register (CXR)
SYNC_ERROR_REG	0x300	Synchronous fault status register (SFSR)
SYNC_VA_REG	0x400	Synchronous fault address register (SFAR)
ASYNC_ERROR_REG	0x500	Asynchronous fault status register (AFSR)
ASYNC_VA_REG	0x600	Asynchronous fault address register (AFAR)
RESET_REG	0x700	Reset register (RR)
ROOT_PTR	0x1000	Root pointer register (RPR)
INS_PTP	0x1100	Instruction access PTP (IPTP)
DATA_PTP	0x1200	Data access PTP (DPTP)
INDT_REG	0x1300	Index tag Register (ITR)
TLBRC_REG	0x1400	TLB replacement control register (TRCR)

Table 9. CY7C604 Cache Constants

VAC_SIZE	0x10000	Virtual address cache size
VAC_LINESIZE	32	Virtual address cache line size
VAC_LINESHIFT	5	Line size in base 2
VAC_CTXFLUSH_COUNT	4096	Virtual address cache context flush count
VAC_SEGFLUSH_COUNT	256	Virtual address cache segment flush count
VAC_PAGEFLUSH_COUNT	64	Virtual address cache page flush count

- Read/write the Index Tag register
- Lock/unlock TLB entries by writing/reading the Replacement Counter (RC) and Initial Replacement Counter (IRC) fields in the TLB Replacement Control Register
- Read the Reset Register (RR) to ascertain whether a watch dog reset, software internal reset, or software external reset has occurred; writing to the RR is also possible

These additional functions do not have to be implemented for the O/S to use the CY7C604. Utilizing these optional functions allows the O/S to "customize" the CY7C604's capabilities to the task at hand, which increases system throughput and capability. Note that all of these functions are dynamic; they can be changed after system boot-up. Although parameters such as the multichip mode bit should not be altered after initialization, parameters such as the cache locking feature enable the O/S to fine-tune system operation.

The final modification that must be done to the O/S is in the area of trap handling. Like the SPARCstation MMU, the CY7C604 has both asynchronous and synchronous fault status and address registers. The interpretation of the bits set in these registers differs between the systems. Additionally, the methods differ in the way they handle the trap once it is correctly decoded. For example, the CY7C604 has level bits that determine the level where the fault occurred during a table walk (if applicable). The trap handler must use these bits to correctly recover from a fault.

Porting to a Multichip CY7C604 System

So far, this application note has focused on porting SunOS for the SPARCstation to a single-chip CY7C604 SPARC system. This section covers the issues involved in porting an O/S from a single-CY7C604 system to a multiple-CY7C604 system. The information given here assumes that all the modifications required to port the SunOS to a CY7C604 system have already been done.

Because the CY7C604 is cascadable, you can expand the cache from 64K with one CY7C604 up to 256K with four CY7C604s. In a multichip system, one CY7C604 responds to all addresses from the CY7C601 until all CY7C604s have been initialized. This one CY7C604 is designated as the boot-mode CY7C604, and it is the only CC/MMU to interface to the memory subsystem (via the Mbus). The boot-mode CY7C604 handles all the MMU functions; the other CY7C604s control their respective caches.

You designate the boot-mode CY7C604 by hardwiring the CSEL, MHOLD, and IOE signals. (See the CY7C604 data sheet or the User's Guide for details.) You configure all the CY7C604s in the system by setting the

multichip address field (MCA), multichip mask field (MCM), and the multichip valid (MV) bit in the devices' System Control Registers (SCRs). These values determine what address space each CY7C604 responds to. The initialization routine for SunOS (hat_init) must be re-written to set these fields to the correct value.

The only other modification that must be made to the O/S is to change the constant VAC_SIZE to its correct value. All other parameters and constants are unaltered.

Porting to a CY7C605

Although the CY7C605 is based upon the CY7C604, the devices differ in several respects. The most notable difference is the CY7C605's ability to support multiprocessing by transparent bus snooping. The CY7C605 also differs in its ability to support reflective memory and its inability to lock the cache. The only differences that affect the O/S are the latter two. These two capabilities are activated by accessing the System Control Register (SCR).

The CY7C605 SCR differs from the CY7C605 SCR in that the former has two additional fields:

- MID(3:0) — Module Identifier at SCR (18 - 15)
- MR — Reflective memory enable bit at SCR (11)

These fields use reserved (i.e., unimplemented) SCR bits in the CY7C604. This allows the fields in the CY7C605 SCR to be in the same bit position as the corresponding fields in the CY7C604 SCR. The only control field in the CY7C604 SCR that is not implemented in the CY7C605 SCR is the cache lock (CL) bit at SCR (9). This bit is reserved in the CY7C605.

The only changes needed to port a CY7C604-ready O/S to the CY7C605 are in the initialization routines in the *hat* layer. Trap handling and the implementation of the *hat* routines can remain the same because the CY7C605 has the same fault conditions and implements the same cache control and memory-management functions. Upon initialization, the module number for the CPU cluster incorporating the CY7C605 must be written into the SCR. If the system uses reflective memory, the MR bit must be set.

Porting to a Multichip CY7C605 System

The porting of the SunOS to a multiple-CY7C605 system is almost identical to porting to a multiple-CY7C604 system. The only change is a modification of the initialization code so that the SCR's multichip address and mask fields are set to the correct values.

References

1. *SPARC SunOS porting guide*, Sun Microsystems, June, 1988

2. *SunOS Virtual Memory Implementation*, J.P. Moran, Sun Microsystems

3. *SunOS on SPARC*, Kleiman & Williams, Sun Microsystems

4. *Virtual Address Cache in UNIX*, Ray Cheng, Sun Microsystems

5. *The SPARC reference MMU Rev 1.4*, Sun Microsystems, January, 1989

CYPRESS SEMICONDUCTOR

Getting Started With Real-Time Embedded System Development

This application note illustrates the use of a real-time operating environment from Mizar and Wind River Systems to develop, download, and test application code on Sun-based Ethernet systems. The application note has four sections:

- Mizar MZ7170 system description
- Wind River Systems VxWorks description
- Sample applications
- VxWorks

Mizar MZ7170 System Description

Integrating Wind River Systems' VxWorks real-time operating system with Mizar's MZ7170 real-time server produces an ideal environment for real-time applications. Mizar's application servers provide the foundation for SPARC-based applications prototyping or for specific application use. The VxWorks real-time operating system furnishes the platform for powerful application debugging or real-time system operation.

The Mizar system incorporates a VME-based system enclosure, which contains both a SPARC-based processor board and an Ethernet board. The VMEbus 32-bit master/slave interface provides seven interrupts; 16-, 24-, or 32-bit address generation; and 8-, 16-, or 32-bit data types. The 20-MHz CPU comes with 1 Mbyte of zero-wait-state memory and 256 Kbytes of socketed PROM, which contains the VxWorks operating system and debugger. You can expand the PROM space by utilizing the full 4-Mbyte capability of the board's 32-pin JEDEC PROM sockets. These PROMS could also contain your embedded applications linked with the VxWorks operating system.

A Dallas Semiconductor DS1287 real-time clock provides battery-backed-up, time-keeping information. An SCN68681 DUART provides two full-duplex asynchronous RS-232C serial ports, which can transmit data at rates as high as 38.4K baud. The initial configuration of serial port 0 provides 9600 baud with 8 data bits, 1 stop bit, and no parity. This channel can connect the Mizar system directly to a monitor. The SCN68681 also provides a programmable timer, along with the ability to generate a level 3, autovectored, SPARC CPU interrupt request under three conditions: (1) upon receiving a character, (2) when the transmitter buffer becomes empty, or (3) when the 16-bit timer reaches its desired limit.

In addition, a bank of 10 processor-readable dip switches provides selectable features such as fixed addresses within a system. LEDs indicate SYSFAIL, processor-to-RAM access, processor-to-VMEbus access, and VMEbus-to-RAM access. You can also define the operation of four additional LEDs. *Table 1* shows the memory map of the Mizar system.

The on-board 32-bit read/write control register controls several functions on the MZ7170 CPU board. The register is located at address FE000400. The CPU can only read to or write from this register in 32-bit-word

Table 1. Mizar System Memory Map

Address	Description
00000000 - 003FFFFF	PROM
00400000 - 007FFFFF	reserved
00800000 - 008FFFFF	RAM
00900000 - 00BFFFFF	Reserved for RAM expansion
00C00000 - 00FFFFFF	reserved
01000000 - FDFFFFFF	VMEbus A32 master
FE000000 - FE0001FF	68681 DUART (D0 - D7 only)
FE000200 - FE0003FF	DS1287 Real-Time clock (D0-D7 only)
FE000400 - FE0005FF	Control register (D0-D31 only)
FE000600 - FE0007FF	Status register (D0-D15 only)
FE000800 - FE0009FF	Mailbox interrupt 1 clear
FE000A00 - FE000BFF	Mailbox interrupt 2 clear
FE000C00 - FE000DFF	VMEbus interrupter (D0-D7 only)
FE000E00 - FE000EFF	VMEbus slave base addr (D0-D31)
FE001000 - FE00FFFF	reserved for extra peripherals
FE010000 - FE7FFFFF	reserved
FE800000 - FEFFFFFF	Memory mapped VMEbus IACK
FF000000 - FFFEFFFF	VMEbus A24 master
FFFF0000 - FFFFFFFF	VMEbus A16 master

Table 2. Power-up Condition of Control Register

Bit	Value	Description
0	0	bus request inhibit
1	1	bus request level 0
2	1	bus request level 1
3	1	system bus error enable
4	0	system bus controller; priority (1)/round robin (0)
5	1	local arbiter bus clear ignore
6	1	local arbiter PREL inhibit
7	0	local arbiter ROR inhibit
8	0	interrupter level 0
9	0	interrupter level 1
10	0	interrupter level 2
11	0	slave A16 decoding inhibit
12	0	slave A24 decoding inhibit
13	0	slave A32 decoding inhibit
14	0	interrupter inhibit
15	0	FPCHAIN
16	0	fail and abort inhibit
17	1	VME IRQ1 inhibit
18	1	VME IRQ2 inhibit
19	1	VME IRQ3 inhibit
20	0	VME IRQ4 inhibit
21	1	VME IRQ5 inhibit
22	1	VME IRQ6 inhibit
23	1	VME IRQ7 inhibit
24	0	PROM wait state 0
25	1	PROM wait state 1
26	1	local bus error enbable
27	0	SYSFAIL
28	0	LED1
29	1	LED2
30	1	LED3
31	0	LED4

operations. *Table 2* shows the descriptions and values of the fields within this register at initialization.

When the SPARC CPU receives an interrupt, it halts execution of the current task and jumps to the interrupt handler. The SPARC processor supplies its own vector during the interrupt. This vector which points to a location containing four instructions. *Table 3* shows the defined interrupts.

Any of three signals from the backplane — ABORT active, SYSFAIL true, or ACFAIL true — sends a non-maskable level 15 interrupt to the CPU. The mailbox interrupt allows any VMEbus master to interrupt the CPU by performing a write operation to the mailbox's VME address location.

The SCN68681 has two separate interrupt locations. The first sends a level 3 interrupt to the CPU when, (1) an Rx holding register receives a character, (2) a Tx holding register becomes empty, or (3) the timer reaches its terminal count. The second SCN68681 interrupt location sends a level 11 interrupt based on the DUART's independent timer. In addition to the interrupts mentioned so far, the MZ7170 can handle all of the seven VME request levels.

Table 3. System Level Interrupts

Level	Description
1	spare
2	VME IRQ1
3	DUART serial
4	VME IRQ2
5	Mailbox 2
6	VME IRQ3
7	Mailbox 1
8	VME IRQ4
9	Real-Time clock
10	VME IRQ5
11	DUART timer
12	VME IRQ6
13	spare
14	VME IRQ7
15	Miscellaneous (abort, sysfail, or acfail)

Table 4 shows the MZ7170 memory map as seen from the VMEbus.

Wind River Systems' VxWorks

Software development begins on the Sun system using the SunOS enhanced UNIX operating system. System developers use UNIX for software development and non-real-time facets of an application. VxWorks, on the other hand, is used for testing, debugging, and running real-time applications. After the development phase, the application programmer can integrate the real-time application into other machines running VxWorks or UNIX. Applications can also operate on a stand-alone basis.

A development system usually contains one or more multi-user UNIX machines connected over an Ethernet network to one or more VxWorks target systems. The UNIX systems can contain large main memory, extensive disk space, and printers, while the VxWorks systems usually has only the resources required for the real-time applications.

Developers create and compile source code in the UNIX environment in the usual way. The code does not have to be linked with the VxWorks system library. Instead, the VxWorks loader loads the object modules, while dynamically resolving external symbol references.

Table 4. VMEbus MZ7170 Memory Map

Address	Description
01000000 - 0010FFFF	DPRAM - 1 MByte
FFFF8000 - FFFF80FE	Mailbox 2 in VMEbus A16 space (Even bytes)
FFFF8001 - FFFF80FF	Mailbox 1 in VMEbus A16 space (Odd bytes)

Application developers can load modules over the Ethernet network and begin debugging. The VxWorks debugging system examines data variables, calls subroutines, spawns tasks, disassembles code in memory, sets breakpoints, and obtains subroutine call tracebacks using the original symbol names. The operating kernel safely traps and reports hardware interrupts. VxWorks can link fully developed applications with the real-time operating kernel and produce an executable file that you can load into a PROM.

The VxWorks system utilizes task-blocking semaphores for intertask communication and multi-tasking to achieve task synchronization and coordination. The multi-tasking kernel uses interrupt-driven, priority-based task scheduling to realize very fast context-switch times and low interrupt latency. Inter-task communication mechanisms include these semaphores, shared memory, ring buffers, linked lists, pipes, sockets, remote procedure calls, and signals. The higher-level structures, such as sockets and pipes, use semaphores as their basic building blocks.

The scheduling is pre-emptive. If a higher-priority task becomes ready to run, the kernel interrupts the current task and switches to the higher-priority task. A task's context consists of the task's program counter; the CPU registers; the dynamic variable stack; the function call stack; I/O assignments for standard input, output, and error; the delay timer; signal handlers; code debuggers; and performance-monitoring values.

Table 5. SPARC-Specific MZ7170 Routines

Routine	Description
char *sysModel()	Return model name of the system CPU
VOID sysHwInit()	Initialize hardware
char *sysMemTop()	Get top of memory address
STATUS sysToMonitor(startType)	Transfer to ROM monitor
STATUS sysClkConnect(routine,arg)	Connect a routine to the system clock interrupt
VOID sysClkDisable()	Turn off system clock interrupts
VOID sysClkEnable()	Turn system clock interrupts on
int sysClkRateGet()	Get rate of the system clock
VOID sysClkRateSet(ticksPerSecond)	Set rate of the system clock
STATUS sysAuxClkConnect(routine,arg)	Connect a routine to the auxiliary clock interrupt
VOID sysAuxClkDisconnet()	Clear the auxiliary clock routine
VOID sysAuxClkDisable()	Turn off auxiliary clock interrupts
VOID sysAuxClkEnable()	Turn auxiliary clock interrupts on
int sysAuxClkRateGet()	Get rate of auxiliary clock
VOID sysAuxClkRateSet(ticksPerSecond)	Set rate of auxiliary clock
STATUS sysLocalToBusAdrs(adrsSpace,LocalAdrs,pBusAdrs)	Convert local address to bus address
STATUS sysBusToLocalAdrs(adrsSpace,busAdrs,pLocalAdrs)	Convert bus address to local address
STATUS sysIntDisable(intLevel)	Disable VMEbus interrupt level
STATUS sysIntEnable(intLevel)	Enable VMEbus interrupt level
STATUS sysBusIntAck(intLevel)	Acknowledge VMEbus interrupt
STATUS sysBusIntGen(intLevel,intVector)	Generate VMEbus interrupt
STATUS sysMailboxConnect(routine,arg)	Connect a routine to mailbox interrupt #1
STATUS sysMailboxEnable(mailboxAdrs)	Enable mailbox interrupt #1
int sysProcNumGet()	Get processor number
VOID sysProcNumSet(procNum)	Set processor number
BOOL sysBusTas(addr)	Test and set across VMEbus
VOID sysImrSet(setBits,clearBits)	Set and clear bits in the M68681 DUART int register
STATUS sysDuartConnect(recvRoutine,xmitRoutine)	Connect interrupt routines for the MZ7170 DUART
int sysMailboxAddressGet()	Get currently defined mailbox addresses
int sysMailboxAddressSet()	Set mailbox addresses as determined by sysProcNum
ULONG sysBCRGet()	Return the value of the board control register
VOID sysBCRSet(mask,value)	Set bits in the board control register
ULONG sysSARGet()	Return the value of the slave address register
VOID sysSARSet(mask,value)	Set bits in the slave address regsiter
ULONG sysStatusGet()	Return the value of the board status register
int sysFrontPanelSwitches()	Read DIP switches
STATUS sysMailbox2Connect(routine,arg)	Connect a routine to mailbox interrupt #2
STATUS sysMailbox2Enable(mailboxAdrs)	Enable mailbox interrupt #2

Table 6. Help Commands

help	Print this list
dbgHelp	Print debugger help info
nfsHelp	Print nfs help info
netHelp	Print network help info
spyHelp	Print task histogrammer help info
timexHelp	Print execution timer help info

VxWorks' extensive networking facility uses the TCP/IP protocol to implement all network communications. VxWorks' network facilities supports process-to-process sockets, remote command execution, remote login, remote procedure calls, remote file access, and remote source level debugging.

VxWorks contains an interactive command line shell. This shell provides the ability to interpret and execute the C language, including calls to functions and references to variables. The VxWorks real-time operating system extends the non-real-time aspects of UNIX C. The VxWorks loader has the ability to load object modules anywhere in memory. The loader uses the object module symbol table to create a system-wide symbol table. The run-time linker ensures that every task can use a single copy of a set of subroutines instead of requiring that each task have its own copy of each routine.

The VxWorks debugging facility has routines to display system and task status. This facility also has routines that give a symbolic disassembly of any loaded module, a trace-back facility for nested C routines, safe trapping of hardware exceptions, and breakpoint and single step facilities. In addition, the dbxWorks facility from Sun Microsystems allows remote source-level debugging.

The VxWorks operating system also provides many other facilities that real-time application developers need. For example, VxWorks provides a timer library to obtain the execution times of various functions and subroutines. Uniform device access, buffered I/O, and serial communication drivers provide C-like real-time access to all the standard devices.

Table 7. Useful Commands

Command	Parameters	Description
h	[n]	Print (or set) shell history
i	[task]	Summary of tasks' TCBs
ti	task	Complete info on TCB for task
sp	adr,args	Spawn a task, pri=100, opt=0,stk=20000
taskSpawn	name,pri,opt,stk,adr,args	Spawn task
td	task	Delete a task
ts	task	Suspend a task
tr	task	Resume a task
d	[adr[,nwords]]	Display memory
m	adr	Modify memory
mRegs	[task]	Modify a task's registers interactively
0-i7,l0-l7, o0-o7,g1-g7, pc,npc,psr,wim,y	[task]	Display a register of a task
version		Print VxWorks version info, and boot line
iam "	user"["passwd"]	Set user name and passwd
whoami		Print user name
devs		List devices
cd	"path"	Set current working path
pwd		Print working path
ls	["path"]	List contents of directory
rename	"old","new"	Change name of file
copy	["in"]["out"]	Copy in file to out file (0 = std in/out)
ld	[syms[,noAbort]]	Load std in into memory (syms = add symbols to table: -1 = none, 0 = globals, 1 = all)
lkup	["substr"]	List symbols in system symbol table
lkAddr	adr	List symbol table entries near address
checkStack	[task]	List task stack sizes and usage
printErrno	value	Print the name of a status value
period	secs,adr,args...	Spawn task to call function periodically
repeat	n,adr,args...	Spawn task to call function n times (0=forever)
diskinit	"device"	Format and initialize RT-11 device
squeeze	"device"	Squeeze free space on RT-11 device

A library contains a number of SPARC-specific routines that allow you to build system-independent code. These routines manipulate the CPU board's primary functions and are included with #define DEBUG. *Table 5* shows the routines' usage and description.

Sample Applications

The development system can debug, test, run, and benchmark real-time applications. You begin the development process by creating an application in C using the normal UNIX environment. Compile the application code by typing:

cc -c -O -I/usr/vw/h <filename>

The -c flag suppresses linking with the UNIX C libraries and leaves the undefined externals unresolved. The VxWorks linking loader resolves these unresolved externals. The optional -O flag optimizes the code. The -I flag tells the compiler where to find the VxWorks header files.

The MIZAR/VxWorks system is linked to the Cypress Ethernet Network. To access the system, type

rlogin mizar

The VxWorks user shell displays a ->. The shell contains the last 20 commands issued, and you can access the shell by issuing vi-like commands. This interactive shell evaluates and executes virtually any C command. For example, the command

-> printf("hello world")

produces the response

hello world

You can get help by typing any of the help commands shown in *Table 6*. Refer to *Table 7* for a list of other useful commands and their explanations.

System developers can easily benchmark code and time context switches within the VxWorks operating system. You create an application first in the host UNIX environment. The sample program used for illustration purposes is a C program containing little more than a loop that iterates 100 times. This program measures its own execution time — a necessary feature for benchmarking user applications. *Figure 1* shows the source code for this application.

```
#include "vxWorks.h"

timeLoop()
{
        int loops,i;

        loops = 100;

        for(i=0;i<loops;++i);
}

timeMain()
{
        timexN(timeLoop);
}
```

Figure 1. Sample C Test Code

```
-> l timeLoop, 31
_timeLoop:
008f9c30  033fffff  sethi   %hi (0xfffffffb8), %g1
008f9c34  820063b8 add %g1, %lo (0xfffffffb8), %g1
008f9c38  9de38001  save%sp, %g1, %sp
008f9c3c  90102064  mov 0x64, %o0
008f9c40  d027bffc  st  %o0, [%fp - 0x00000004]
008f9c44  c027bff8  clr [%fp - 0x00000008]
008f9c48  d207bff8  ld  [%fp - 0x00000008], %o1
008f9c4c  d407bffc  ld  [%fp - 0x00000004], %o2
008f9c50  80a2400a  cmp %o1, %o2
008f9c54  16800007  bge 0x008f9c70
008f9c58  01000000  nop
008f9c5c  d607bff8  ld  [%fp - 0x00000008], %o3
008f9c60  9602e001  add %o3, 1, %o3
008f9c64  d627bff8  st  %o3, [%fp - 0x00000008]
008f9c68  10bffff8  b   0x008f9c48
008f9c6c  01000000  nop
008f9c70  90102000  clr %o0
008f9c74  b0100008  mov %o0, %i0
008f9c78  81c7e008  ret
008f9c7c  81e80000  restore
_timeMain:
008f9c80  033fffff  sethi   %hi (0xfffffffa0), %g1
008f9c84  820063a0 add %g1, %lo (0xfffffffa0), %g1
008f9c88  9de38001  save%sp, %g1, %sp
008f9c8c  110023e7  sethi   %hi (_timeLoop), %o0
008f9c90  90122030  or  %o0, %lo (_timeLoop), %o0
008f9c94  7ffc8cab  call_timexN
008f9c98  01000000  nop
008f9c9c  90102000  clr %o0
008f9ca0  b0100008  mov %o0, %i0
008f9ca4  81c7e008  ret
008f9ca8  81e80000  restore
```

Figure 2. Disassembled Un-Optimized Object Code

The timexN() subroutine in this program continues executing the subroutine or function passed to it until the subroutine's execution time is known to within ±2 percent. VxWorks supplies this routine along with an extensive library of other routines, which perform tasks ranging from network communication, to device drivers and linked-list manipulation. These UNIX C-compatible routines are optimized for speed and real-time operation.

The following user login session assumes that joe has logged into the UNIX environment with a home directory called /home/joe and a benchmark working directory called bench. All UNIX operating system prompts, therefore, begin with Cypress/home/joe/bench, while all MIZAR/VxWorks operating system prompts begin with ->. To compile the source code (timetest), type the following at the UNIX prompt:

Cypress/home/joe/bench: cc -c -I/usr/vw/h timetest.c

Table 8. Useful Debugger Commands

```
->dbgHelp
dbgHelp                          Print this list
dbgInit                          Install debug facilities
b                                Display breakpoints
b       addr[,task[,count]]      Set breakpoint
bd      addr[,task]              Delete breakpoint
bdall   [task]                   Delete all breakpoints
c       [task[,addr]]            Continue from breakpoint
cret    [task]                   Continue to subroutine return
s       [task[,addr]]            Single step
so      [task]                   Single step/step over subroutine
l       [adr[,nInst]]            List disassembled memory
tt      [task]                   Do stack trace on task
```

```
-> l timeLoop, 13
_timeLoop:
008f9bd8  9a102000  clr %o5
008f9bdc  9a036001  add %o5, 1, %o5
008f9be0  80a36064  cmp %o5, 0x64
008f9be4  26bfffff  bl,a0x008f9be0
008f9be8  9a036001  add %o5, 1, %o5
008f9bec  81c3e008  retl
008f9bf0  90002000  add 0, %o0
_timeMain:
008f9bf4  9de3bfa0  save%sp, 0xfffffa0, %sp
008f9bf8  110023e6  sethi  %hi (0x008f9800), %o0
008f9bfc  7ffc8cd1  call_timexN
008f9c00  901223d8  or  %o0, 0x3d8, %o0
008f9c04  81c7e008  ret
```

Figure 3. Disassembled Optimized Object Code

This command produces unoptimized, unlinked code that has headers located in the directory /usr/vw/h. To log into the Mizar system type

Cypress/home/joe/bench: rlogin mizar

To switch the Mizar operating system root directory to that of the user (joe), type

-> iam "joe"

Next, switch to the working directory:

-> cd "~/bench"

To link and load the test program, type

-> ld < timetest.o

To initialize the debugging facility, which allows you to disassemble code, set breakpoints, step through code, and perform other tasks (*Table 8*), type

-> dbgInit

For example, *Figure 2* shows the disassembled code for this test program. This code was produced using the debugger command "l" (list disassembled memory). The listing shows the hex memory address, the hex instruction code, and the instruction itself. To execute the program, type

-> timeMain

The operating system responds with

timex: 7500 reps, time per rep = 90 +/- 2 (2%) microsecs

This response indicates that after 7500 iterations of the routine timeMain, the routine found that it took 90 ± 2 µs to execute. TimeMain is the name of the subroutine that represents the main part of the program. You can also get the timing of this code by typing

-> timexN(timeLoop)

The system responds with

timex: 7725 reps, time per rep = 91 +/- 2 (2%) microsecs

To exit the Mizar system type

-> ~.

Real-time programs can also take advantage of the C compiler's optimization features. To create a fully optimized version of the code, type

Cypress/home/joe/bench: cc -c -O4 -I/usr/vw/h timetest.c

The significantly optimized code appears in *Figure 3*. This code also has a significantly better execution time, as shown by typing

-> timeMain

timex: 46250 reps, time per rep = 14 +/- 0 (0%) microsecs

CYPRESS SEMICONDUCTOR

SPARC as a Real-Time Controller

In addition to giving an overview of real-time system characteristics, this application note shows how the Cypress SPARC chip set supports real-time operations. Special attention is given to operating models that either reduce procedure call overhead or minimize the time needed for a context switch.

A real-time system must react to external events as they happen. These systems are, by nature, event driven as they respond to external, asynchronous stimuli and must do so in a timely manner. If both logical correctness and timing correctness are not satisfied, severe consequences can result. Although the need for logical correctness is obvious, the need for timing correctness arises due to the possible physical impact of the controlling system's activities. If a computer controlling a satellite does not respond to an external event in time, for example, the satellite might collide with a foreign object and be knocked out of orbit.

At the highest level, you can view a real-time system as one that acquires data and detects the occurrence of events by means of hardware inputs. These inputs are then processed and the results transmitted to hardware outputs. An embedded computer can be used to process the data, with a real-time operating system controlling the computer.

When defining a real-time system, it is essential to partition the functions to be performed into individual units called tasks. Each task is implemented as a software module that can be invoked to perform a specific function. Although many tasks are usually associated with a real-time system, only a limited number of processors is generally available to execute these tasks. This application note concentrates on the simplest case, where a single processor is involved.

Because multiple tasks compete for use of a limited resource, the processor, it is crucial that tasks be prioritized. The highest-priority task that is ready to run at any given time must actually be running. This requirement often leads to a case where a higher-priority task becomes ready while a lower-priority task is executing. In this case, the lower-priority task must immediately be pre-empted, and the higher-priority task must take control of the processor. This concept of pre-emptive scheduling is essential in all real-time systems.

The real-time systems design considerations described so far deal with the general behavior of a real-time system. To put these generalities into perspective, consider the following example.

Dealing with Overhead

In this example, several tasks are defined in prioritized order (task 1 through task 6). Included in this system is a real-time clock that generates an interrupt to the processor every 500 µs. *Table 1* lists the CPU requirements for this example.

Tasks 1 through 5 all have specific jobs that require a fixed amount of time. Task 6 checks for user commands, and thus the amount of time it needs varies depending on whether a user command is present.

Based on the data in *Table 1*, the CPU time requirements for a second of processing time for each task appear in *Table 2*.

Tasks 1 through 5 use 743 ms, which leaves 257 ms for the background task to execute. This means that the background task executes at a worst-case rate of 1.3 times per second. In the best case, the background task's frequency is 25 times per second. This rate allows display updating 25 times per second, while user commands can only be processed at the rate of 1.3 per second.

So far, this example has not accounted for the overhead associated with switching processor contexts be-

Table 1. CPU Requirements

Task	Duration	Operating Speed
1	35 us	2000 Hz
2	100 us	1000 Hz
3	1 ms	333 Hz
4	200 us	200 Hz

Table 2. CPU Time Per Second

Task	Time/ Invocation	Invocations	Total Time
1	35 us	2000	70 ms
2	100 us	1000	100 ms
3	1 ms	333	333 ms
4	200 us	200	40 ms

tween tasks. This overhead includes several operations. Specifically, the state of the processor at the time of pre-emption is saved with each context switch. Then the scheduler determines the next task to run. Finally, the state of the new task is loaded into the processor. In commercially available real-time operating systems, the time required for a task switch generally ranges from 25 µs to over 100 ms for some processors.

Including a 25-µs task switch overhead, the CPU usage during 1 second breaks down as shown in *Table 3*. More than 14 percent of the total CPU time is spent on overhead; no useful work was done.

In this case, the background task only runs at a best-case frequency of 11 times per second, while the worst-case frequency is only once every other second.

Increasing the context switch overhead to 35 µs produces an interesting effect associated with real-time systems, as shown in *Table 4*. Although it seems as if the system works, critical timing parameters have been violated. For example, task 5 is scheduled to run a second time when it has not received enough CPU time to complete its first run. To help compute context switch overhead, you can use the example C program that appears in *Appendix A*.

Interrupt Latency

The need to meet externally imposed deadlines lies at the heart of a real-time system. In real-time computing, the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced. A system must be fast as well as predictable.

The parameter used to specify a system's predictability is its worst-case interrupt latency. This parameter is defined as the maximum amount of time a system takes before responding to an external event. Interrupt latency usually indicates a specific processor's worthiness as a real-time controller.

Interrupt latency directly affects two key system performance factors: the guaranteed response time to an event and the guaranteed response time of any individual task. The latter is the maximum amount of time it takes to pass control from a lower-priority task to a pre-empting higher-priority task.

Table 3. 25-us Context Switch Overhead

Tasks 1-5	743 ms
Switch Overhead	25 us
Number of Switches	5733
Overhead	143 ms

You can think of the response time to an event as the maximum amount of time that elapses before the system can identify that an event has occurred and respond with the necessary action. In the case of detecting meltdown in a nuclear power plant, the processor could use the instructions directly from the interrupt handler to perform the critical actions necessary to shut the reactor down. This avoids the time penalty of a context switch.

Table 5 shows the effect of interrupt latency in a real-time system. Many factors contribute to this effect. The processor itself has a worst-case interrupt response time, and the memory subsystem might also contribute to interrupt latency. The operating system might be required to disable interrupts during critical sections of code, thus adding to interrupt latency.

Interrupt response time varies among processors. Some processors are designed such that they save the entire state of the machine when an interrupt occurs. In this case, the interrupt handler starts executing without regard to the context of the interrupted task. Although this practice might be convenient for the person writing the interrupt handler, it adds to the system's overhead and slows interrupt response time.

Other processors vector to the interrupt handler and make the interrupt routine responsible for saving any part of the interrupt task state that the handler might use. The state of the interrupt task must then be restored upon exit from the interrupt handler. This is a good approach because it does not introduce any unnecessary overhead.

The best approach in minimizing interrupt latency at the processor level is to employ a dedicated set of registers reserved for interrupt handlers. With this approach, the interrupt handler need not be concerned

Table 4. 35-us Context Switch Overhead

Tasks 1-5	743 ms
Switch Overhead	35 us
Number of Switches	5733
Overhead	200 ms

with saving and restoring the interrupted task's working registers.

Another factor you must account for is memory system latency. In a design using dynamic memory, the interrupt latency includes the worst-case memory-cycle timing for fetching interrupt handler instructions. In a cache system, the worst-case timing includes the time penalties of a cache miss. With processors running in the 25- to 40-Mhz range, failure to consider these latencies can have drastic effects.

Just as important as the time taken to switch tasks or respond to interrupts is the time window during which the operating system is unable to do these things. An operating system's ability to do a context switch in 10 µs is not useful if the operating system disables context switching for 50 ms or more while doing something else.

An operating system might disable interrupts to place a task in a ready queue or to access a critical region while doing inter-task communication, resource allocation, or task synchronization. When accessing a critical region, a real-time system must provide a way to get uninterrupted access to a shared variable. Some processors support this requirement in hardware; however, the following example shows the overhead involved when hardware does not support uninterruptable access to shared variables.

Access to Shared Variables

This example defines two tasks *Table 6*. Task 1 counts the number of input pulses from an input stream. Task 2 reads the total number of pulses every second, clears the count variable, and performs a series of operations based on the total number of pulses. If special care is not taken in accessing the shared count variable, the following might occur:

1. Task 1 has control /* count is at 200 */
   count->register
   register+1->register
   interrupt occurs
2. Task 2 gets control (One second has elapsed)
   count->register
   0->count
   execute based on count
3. Task 1 resumes
   register->count

Table 5. Effect of Interrupt Latency

Event	Worst-Case Time
Task Switch	35 us
Interrupt latency	25 us
Response to event	25 us

A serious problem has occurred: The variable count contains a value of 201 when the count should be 1. This is a common problem that must be overcome in a multitasking environment. The key to eliminating the problem is uninterruptable updating of shared variables. In processors without hardware support for this capability, the only way to update a shared variable without the possibility of pre-emption is to disable interrupts. *Table 7* shows modifications. (Note that this solution is valid only for single-processor systems. In a multiprocessor system, some form of hardware lockout is essential.)

Although this solution works, and the maximum amount of time in which interrupts are disabled is minimal, everything is not as it seems. The main problem is that interrupts can only be disabled in supervisor mode. This means that a software trap must be executed, the processor must branch to a trap vector, change into supervisor mode, execute the few uninterruptable instructions, then go back to the original point. You must consider the time during which the processor is uninterruptable when calculating worst-case interrupt latency.

SPARC as a Real-Time Controller

As real-time systems vary widely in requirements, it is important that a specific processor chip set provide the flexibility to meet the needs of specific applications. It does not make sense to pay for a processor that has a built-in floating point unit to do strictly integer operations. The same holds true for a processor with a built-in MMU when you use only a physical memory system. The Cypress SPARC chip set is specifically designed to meet the needs of individual applications without forcing you to buy something you do not need. *Table 8*shows the SPARC family of chips. You can use these parts in any combination to create a system that fits your application. family of chips. You can use these parts in any combination to create a system that fits your application. family of chips. You can use these parts in any combination to create a system that fits your application.

Processor Interrupt Response Time

The CY7C601 SPARC integer unit minimizes interrupt latency at the processor level. The processor dedicates eight of its 136 registers strictly for use by interrupt handlers. When an interrupt occurs, the interrupt routine automatically gets a new set of eight registers with which to work. On an interrupt, the processor switches to supervisor mode, gets the new set of

Table 6. Format of Tasks

Task 1	Task 2
count->register	count->register
register+1->register	0->count

Table 7. Modified Format of Tasks

Task 1	Task 2
disable interrupts	disable interrupts
count->register	count->register
register+1->register	0->count
register->count	enable interrupts

registers, and completes execution of the first instruction in the interrupt routine in a worst-case time of 14 clock cycles. At 40 MHz that time equals 350 ns.

Two of the CY7C601's interrupt-handling registers automatically save the program counter and next program counter of the interrupted task, with the remaining six registers at the disposal of the interrupt routine. Upon return from the interrupt, the processor automatically restores the state of the interrupted task; this is done in two clock cycles, or 50 ns at 40 MHz.

Achieving Deterministic Response Time

The CY7C604 CMU has two special features that help guarantee deterministic response for systems using either virtual or physical addressing, with or without cache memory. The MMU allows selected pages to be locked into the Translation Lookahead Buffer (TLB). This capability ensures that critical memory pages are always in main memory, avoiding the delay associated with a table walk.

In systems using cache memory, the CY7C604 allows the cache to be locked. You can load the cache with time-critical code, such as interrupt handlers and time-critical tasks, and be sure that these routines will always be present in the cache. With these features, memory latency is no longer a problem, and predictability is guaranteed.

Semaphore Support in Hardware

Included in the CY7C601's instruction set are two instructions that provide uninterruptable access to an external memory location. The SWAP instruction exchanges the contents of a selected register with the contents of the addressed memory location. The atomic load-store instruction moves a byte from memory into the selected register and then rewrites the same byte in memory to all Ones. The CY7C601 executes both instructions without allowing intervening asynchronous traps.

Table 8. RISC 600 Family of SPARC Chips

Device	Description
CY7C601	Integer Unit
CY7C602	Floating Point Processor
CY7C604	CacheTag-Controller/ MMU

You can use either of these instructions to create a semaphore for accessing a critical region without the need to enter supervisor mode and disable interrupts. The SWAP instruction can be used for counting semaphores, and the atomic load-store is appropriate for a simple semaphore for critical regions.

Alternate Register Models For SPARC

The Cypress CY7C601 has a total of 136 32-bit registers, which are divided into a set of 128 local registers and eight globals. The use of these registers is configurable by accessing a processor register called the Current Window Pointer (CWP). Two common operating models are supported by commercially available compilers and operating systems: The standard register windowing model is optimized to minimize procedure call overhead, and an alternate model significantly reduces the time required for a context switch.

Register Windowing Model

For the register windowing model, the register file is divided into a set of eight overlapping register windows. Each window contains a set of 24 local registers. The registers in each window are divided into three sets of eight registers referred to as INS, LOCALS, and OUTS. At any given time, the processor can access only one window and the eight globals. The windows are joined together in a circular stack, with each window sharing its INS and OUTS with adjacent windows. Two instructions provide for rotating the windows among procedures.

A save instruction is used with a procedure call to allocate the next window for the called procedure. Before executing the save instruction, the calling procedure stores the parameters to be passed in its OUT registers. Upon execution of the save instruction, the register set is rotated such that the called procedure has access to the passed parameters in its IN registers.

A restore instruction is used with a return from procedure to restore the register set of the calling procedure. Before executing the restore instruction, the called procedure stores in its IN registers the parameters to be returned to the calling procedure. Upon execution of the restore instruction, the register set is rotated back to its previous position with the returned parameters in the caller's OUT registers.

Because the processor logically provides new LOCALS and OUTS with each procedure call, local register values need not be saved and restored across calls. The overlapping registers also minimize the overhead of passing and returning procedure parameters because the parameters are passed in registers instead of the main memory stack.

Fast Task Switch Register Model

For the fast task switch register model, the register set is divided into four non-overlapping sets of 24 registers. Three of the four register sets are dedicated to the three highest-priority or time-dependent tasks. All the remaining tasks share the other set of registers. Associated with each register set are a set of eight independent registers for use by interrupt handlers. These registers also store the state of the processor on a task switch.

Using this register model, the processor can do a task switch to any of the three highest priority tasks in under a microsecond. A task switch to one of the other tasks can be done in less than 3 μs.

When an interrupt occurs, the processor automatically switches register sets to access the interrupt registers corresponding to the new task. If the interrupt initiates a task switch, the state of the processor is saved in the interrupt registers. If the new task is one of the three high-priority tasks, the task's state is loaded from its dedicated interrupt registers, and execution begins immediately. In this case, the state of the machine is merely the PSR, PC, NPC and possibly a few other control registers. The general-purpose registers are not affected, as they are dedicated to general-purpose tasks.

If the new task shares a set of registers, the state of the task previously using that register set is saved to memory and the new task's state is loaded into the processor. This state includes the minimal processor state as well as the 24 general-purpose registers.

To understand this model's task switching behavior, consider two examples:

Example 1—Switching to a higher-priority task

1) Interrupt occurs
   Automatically switches to interrupt registers
   PC and NPC saved in interrupt registers
2) Save PSR and any other control register to interrupt registers
3) Load the pointer to the new task's interrupt registers into the CWP
4) Restore new task's PSR and any other control registers
5) Execute RETT (return from trap)

Example 2—Switching to a lower-priority task

1) Interrupt occurs
   Automatically switches to interrupt registers
   PC and NPC saved in interrupt registers
2) Save PSR and any other control registers to interrupt registers
3) Load pointer to the shared set of working registers into the CWP
4) Save the registers to memory
   (these are the registers of the previous task using the window)
5) Restore the working registers of the new task from memory
6) Update the CWP to point to the shared task's interrupt registers
7) Save to memory the eight interrupt registers containing the state of the previous task running out of these registers
8) Restore the state of the new task
9) RETT (return from trap)

Each register model has certain advantages. Using register windowing significantly reduces both procedure-call overhead and data-bus traffic as parameters are passed in registers. This approach also has the affect of caching local variables because each procedure gets a new set of local registers. The price paid for this advantage lies in the context switch overhead. On a context switch, the processor must save and restore all the used registers—up to 120, as determined by the Window Invalid Mask (WIM), a processor status register.

When using the fast context switch register model, on the other hand, you do not get the ultra-fast procedure calls that result from register windowing. You do get the benefit of four separate register files and very fast context switching, however. In this model, parameters are passed on the stack as is done on most other architectures. Each task's allocation of 24 general-purpose local registers and eight global registers is the same as the total number of registers in most other architectures.

Because register usage in the CY7C601 is configurable by software, you can mix these models to achieve the benefits of both. The SPARC register set and the entire Cypress chip set has been designed to cover a wide range of applications efficiently.

Appendix A. Sample C Program to Compute Context Switch Overhead

```
/*****************************************************************************************  ***********/
/*                                                                                                      */
/* This program is used for determining the overhead of context switching in a real-time system. This simulation does not take into */
/* account interrupt latency, memory latency, or any of the other many possible forms of overhead associated with a realtime      */
/* system, but these can easily be added. The current version should be sufficient to give a good idea of how much time the kernel  */
/* is spending on context switching.                                                                    */
/*****************************************************************************************  ***********/
#include   \c\ms\include\math.h
#include   \c\ms\include\stdio.h
#define BCKGRND 100
FILE *fp; int openfile; char fname[35]; int numtasks;
main (argc, argv)
int argc;
char *argv[];
{
int             i,j;
int             iterations;
int             curr_task;
int             time[100];
int             duration[100];
int             frequency[100];
int             total;
float           background;
int             swtime;
int             switchh;
int             temp;
int             temp1;
int             sampfreq;
float           tempflt;
float           cs_time;

create_file();
temp = 0;

/* get number of simulation points per second */
while  (temp==0)
     {
     place (7,4,"Enter the sampling rate in Hz (100 - 10000) : ");
     locate (7,58);
     ceol();
     iterations = 0;
     temp = getchar();
     while  (temp<>0xa)
               {
               if ((temp > = 0x30) && (temp < = 0x39))
                         {
                         temp = temp - 0x30;
                         iterations = iterations * 10;
                         iterations = iterations + temp;
                         }
               temp = getchar();
               },
     temp = 1;
     locate (22,5);
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
        ceol();
        if (((iterations % 100) != 0) || (iterations = 100) || (iterations = 10000))
                {
                temp = 0;
                place (22,5,"Error must be = 100 or 10,000 and a mult of 100");
                }
        },
/* time in microseconds of one clock tick */
sampfreq = 10000 / (iterations / 100);

place (8,4,"Enter the context switch overhead in microseconds : ");
locate (8,58);
swtime = 0;
temp = getchar();
while  (temp<>0xa)
        {
        if ((temp > = 0x30) && (temp< = 0x39))
                {
                temp = temp - 0x30;
                swtime = swtime * 10;
                swtime = swtime + temp;
                }
        temp = getchar();
        },

temp = 0;
while  (temp==0)
        {
        place (9,4," Enter the number of tasks (100 max) : ");
        locate (9,58);
        ceol();
        numtasks = 0;
        temp = getchar();
        while (temp < > 0xa)
                {
                if ((temp > = 0x30) && (temp < = 0x39));
                        {
                        temp = temp - 0x30;
                        numtasks = numtasks * 10;
                        numtasks = numtasks + temp;
                        }
                temp = getchar();
                }
                temp = 1;
                locate (20,5);
                ceol();     if (numtasks > 100)
                        {
                        temp = 0;
                        place (20,5 ,"Maximum number of tasks is 100");
                        }
                 },
/* tasks numbered 0 to n */
numtasks = numtasks - 1;
for  (i=0;  i=numtasks;  i++)
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
    {
    temp = 0;
    while (temp==0)
            {
            locate (i+11,4);
            printf("Enter the frequency of task %d in Hz",i);
            locate (i+11,60);
            ceol();
            frequency[i] = 0;
            temp = getchar();
            while (tem<>0xa)
                    {
                    if ((temp > = 0x30) && ( temp < = 0x39))
                            {
                            temp = temp - 0x30;
                            frequency[i] = frequency[i] * 10;
                            frequency[i] = frequency[i] + temp;
                            }
                    temp = getchar();
                    }
            locate (20,5);
            ceol();
            locate (21,5);
            ceol();
            if (frequency[i] 0)
                    {
                    if ((iterations % frequency[i]) != 0)
                            {
                            locate (20,5);
                            printf (" Warning : %d and the simulator frequency : %d are not multiples",frequency[i],iterations);
                            place (21,5," Would you like to re-enter the value (not mandatory) (y/n) : ");
                            locate (21,70);
                            temp = getchar();
                            temp1 = getchar();
                            /* CR */
                            if ((temp=='Y') || (temp=='y'))
                                    temp = 0;
                            else
                                    temp = 1;
                            }
                    }
            else
                    {
                    place (20,5," Frequency must be greater than zero ");
                    temp = 0;
                    }
            }
    /* frequency[i] will be used with modulo operator to see when task ready */
    frequency[i] = iterations / frequency[i];
    /* integer divide */
    }
locate (20,5);
ceol();
locate (21,5);
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
ceol();
for  (i=0;  i=numtasks;  i++)
     {
     locate    (i+numtasks+14,4);
     printf("Enter the duration of task %d in microseconds",i);
     locate  (i+numtasks+14,  60);
     duration[i] = 0;
     temp = getchar();
     while  (temp<>0xa)
               {
               if ((temp > = 0x30) && (temp < = 0x39))
                         {
                         temp = temp - 0x30;
                         duration[i] = duration[i] * 10;
                         duration[i] = duration[i] + temp;
                         }
               temp = getchar();
               }
     }
/* init ialize current task */
curr_task = BCKGRND;
/* init current task, task switch needed for 1st task background task time of execution */
background = 0;
/* number of context switches */
switchh = 0;
/* init total time left in this time s lice */
total = 0;
/* check to see whether a disk file is to be opened */
     if    (openfile==1)
     init_file();
cls();
/* init time spent in individual tasks */
for  (i=0;  i=numtasks;  i++)
     time[i] = 0;
/* iterations start at 0 */
iterations = iterations - 1;
/* main simulation loop */
for  (j=0;  j=iterations;  j++)
     /* number of samples */
     {
     /* screen oup ut to show system didn't die */
               if ((j % 100)==0)
               {
               locate (10,6);
               printf (" Doing simulation loop %d of %d ", j,iterations+1);
               },
     total = total + sampfreq;
     /* increment clock time for each time slice scheduling of tasks */
     for  (i=0;  i=numtasks;  i++)
               {
               /* check if task is scheduled to execute */
               if  ((j  %  frequency[i])==0)
                         /* modu lo operator */
                         if (time[i]= = 0)
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
        /* has it completed from last time */
        {
        time[i] = duration[i];
        /* init time slice required */
        if (openfile == 1)
                fprintf(fp,"task%d is ready \n",i);
        }
else
/* hasn't completed previous scheduled time */
        {
        cls();
        locate (10,6);
        printf (" Need a faster processor, check simulation file");
        if (openfile==1)
                fprintf (fp," task %d has been scheduled again but has not completed",i);
        goto p1; /* abort simulation */
        }
}
/* print which clock tick in file */
if (openfile==1)
        fprintf(fp,"%d:",j);
/* executing of tasks */
for (i= 0; i= numtasks; i+ + )
        /* check for tasks 0 to n being ready */
        {
        if (total > 0)
                /* check if there is time to run the task */
                {
                if (time[i]>0)
                        /* is this particular task ready to run */
                        {
                        /* does a context switch actually take place or was */
                        if (i != curr_task)
                                {
                                total = total - swtime;
                                /* context switch time */
                                switchh = switchh + 1;
                                /* # of context switches */
                                }
                        /* can task time slice be completed */
                        if (total = time[i])
                                {
                                if (openfile= = 1)
                                        fprintf(fp,"%d",time[i]);
                                total = total - time[i];
                                /* time left in slice */
                                time[i] = 0;
                                /* update ready list */
                                curr_task = i;
                                /* mark as last task to run */
                                }
                        /* can run portion of task */
                        else
                                {
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
                                        /* use remaining time available in simulation slice */
                                        if (total 0)                                        {
                                                /* time still required by the task */
                                                time[i] = time[i] - total;
                                        total = 0;
                                        /* time slice has expired */
                                        }
                                /* mark in sim file that a context switch has started for */
                                /* one task but a higher priority task has become ready */
                                /* and will has pre-empted the scheduled task  */
                                else
                                        {
                                        if (openfile==1)
                                                fprintf(fp,"X");
                                        }
                                /* mark state of processor */
                                curr_t ask = i;
                                }
                        }
                else
                        {
                        if (openfile == 1)
                                fprintf(fp,"-");
                        }
                }
        else
                {
                if (openfile= =1)
                        fprintf(fp,"-");
                }
        }
        /* background */
        /* if time left after all scheduled tasks have run, let bac kground task run */
        if (total 0)
                {
                /* check to see if background was last to use the processor */
                if (curr_task != BCKGRND)
                        {
                        switchh = switchh + 1;
                        total = total - swtime;
                        }
                curr_task = BCKGRND;
                /* set curr_task to background */
                if (total 0)
                        {
                        if (openfile==1)
                                fprintf(fp,"%d",total);
                        /* add to background task exec ution time */
                        background = background + total;
                        total = 0;
                        /* background takes all remaining time */
                        }
                 else
                        {
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
                        if (openfile==1)
                                fprintf(fp,"X");
                        }
                }
        else
                {
                if (openfile==1)
                        fprintf(fp,"-");
                }
        if (openfile==1)
                fprintf(fp,"\n");
         }
    /* screen output */
     cls();
    for (i = 0; i=numtasks; i++)
            {
            tempflt = (((float)iterations + 1) / (float)frequency[i]) * (float)duration[i];
            tempflt = tempflt / 1000;
            locate (i+4,6);
            printf (" Total execution time for task %d : %6.2f ms",i,tempflt);
            }
    locate (numtasks + 6,6);
    printf (" There were %d context switc hes",switchh);
    cs_time = ((float) swtime * (float) switchh) / 1000;
    locate (numtasks + 8,6);
    printf (" Context switch ove rhead : %6.2f ",cs_time);
    locate (numtasks + 10,6);
    printf ("Time available for background tasks : %6.2f ",background / 1 000);
    p1: locate (numtasks + 15,6);
    printf ("For more info look at simulation file ");
    locate (22,1);
    if (openfile==1)
            fclose(fp);
    }
}
/* screen utilities supported with ansi.sys clear screen utility */
cls ()
{
printf ("%c[ 2J",27);
}
ceol ()
{
printf ( "%c[K",27);
}
locate (row,col)
int row,col;
{
printf("%c[%d;%dH",27,  row,col);
}
place  (row,col,text)
int row,col; char text[];
{
locate (row,col);
puts (text);
```

Appendix A. Sample C Program to Compute Context Switch Overhead (continued)

```
}
create_file()
{
int temp,i;
openfile = 0;
for   (i=0;i#;i++)
      fname[i] = 0;
cls();
place (5,4,"Enter file to be created (Retu rn for no file): ");
locate (5,51);
i = 0;
temp = getchar();
if (temp=0xa)
      {
      temp = getchar();
      while  (temp<>0xa)
                  {
                  fname[ i] = temp;
                  i++;
                  temp = getchar();
                  }
      fp = fopen(fname,"w");
      openfile = 1;
      }
}
init_file()
{
int i;
fp rintf(fp,"\n\n\n\n  Simulation  Results  \n\n\n\n\n");
fprintf(fp,"      Tick      ");
for (i=0; i=numtasks; i++ )
      fprintf(fp,"task%d     ",i);
fprintf(fp,"background      \n\n");
},
```

CYPRESS SEMICONDUCTOR

Memory Protection and Address Exception Logic for the CY7C611 SPARC Controller

This application note describes an address validity check circuit for the Cypress CY7C611 SPARC-compatible RISC controller. The design provides validity checks on 32-bit word boundaries for the entire 24-bit CY7C611 address space. If an address falls outside a valid boundary, the check circuit generates a memory exception.

The absence of a memory management unit (MMU) often distinguishes an embedded microprocessor from a central processing unit. This does not mean that some MMU functions are not desired for embedded applications, but these applications usually do not need the full range of such functions (mapping, access protection, validity check, etc.). However, a circuit that performs an address validity check definitely has applications in embedded systems, provided that the circuit can be implemented with a reasonable number of components. The circuit described here is implemented in two Cypress EPLDs: a CY7C332 and a CY7C361.

The circuit contains two functional blocks: the address-checking circuit and the memory-exception generator (*Figure1*). The address-checking circuit checks the SPARC processor's most significant 22 address bits against an arbitrary memory map. The memory map used for this design appears in *Table 1*. If an address

Table 1. System Memory Map

ADDRESS	DESCRIPTION	EXCEPTION
000000 – 07FFFF	Boot PROM	N
080000 – 0FFFFF	Unused	Y
100000	I/O Status Reg.	N
100004	I/O Control Reg.	N
100008 – 1FFFFF	Unused	Y
200000 – 5FFFFF	4Mb RAM	N
600000 – BFFFFF	Unused	Y
C00000 – DFFFFF	I/O Interface	N
E00000 – FFFFFF	Reserved for expansion	Y

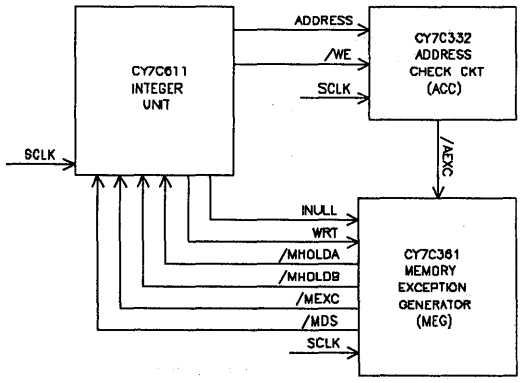


Figure 1. Block Diagram

exception occurs, the memory-exception generator sends a memory exception to the processor.

These circuits offer an example of how the logic functions built into the CY7C332 and CY7C361 can implement designs that would otherwise be very difficult to implement. The CY7C332's transparent latch mode permits the design to make the most of the 10-ns address setup time provided by the CY7C611 SPARC controller. The CY7C361 combinatorial input configuration acts upon the exception information without incurring any extra clock delay, while the single-registered configuration holds CY7C611 bus transaction information. The CY7C361 Mealy input inhibits memory exceptions that might occur because of a nullified address from the CY7C611, and the termination macrocell configuration inhibits further exceptions before the current exception completes.

CY7C611 Memory Interface

The CY7C611 sends most of its memory interface signals out unlatched. Thus, these signals are only valid a short time before and after the system clock's rising

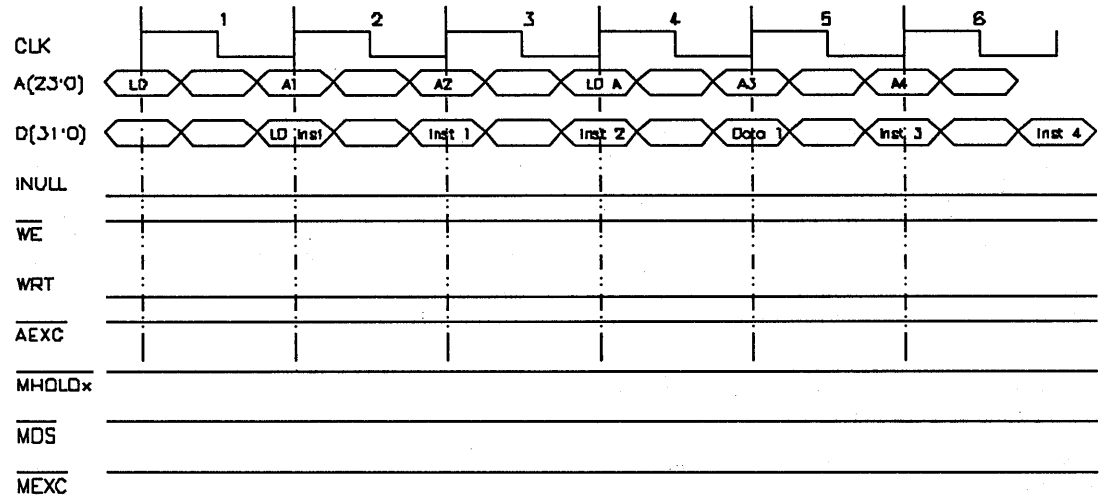


Figure 2. Load Bus Cycle

edge. To be used, they must be latched outside the processor.

The processor sends out a few latched signals. INULL is the only one of these signals used in this design. The processor holds INULL until the next rising clock edge, and the signal is *not* designed to be latched externally.

The Load, Load Double, Store, and Store Double bus cycles for the CY7C611 appear in *Figures 2, 3, 4,* and *5,* respectively. *Figure 6* illustrates a Load with Memory Exception, and *Figure 7* illustrates a Store with Memory Exception. Note that in both cases, MHOLD becomes active more than one clock after the address causing the memory exception leaves the bus. This is acceptable because the data corresponding to that address is just being clocked into the CY7C611's fetch pipeline stage, and thus can be easily invalidated.

Signal Description

The signals used in the design are:

1. /MHOLD(A/B) — Memory Hold (CY7C611 inputs)

These two signals are ORed together inside the processor. Either is asserted to freeze the processor's pipeline, which is the first thing that must be done to generate a memory exception. The CY7C611's outputs revert to and maintain the value they had at the clock's rising edge in the cycle in which either signal was asserted. These inputs are sampled at the processor clock's falling edge.

This design's state machines use both /MHOLDA and /MHOLDB. The store exception state machine uses /MHOLDA, and the load exception state machine uses /MHOLDB (*Figures8* and *9*).

2. /MDS — Memory Data Strobe (CY7C611 input)

The memory-exception generator asserts Memory Data Strobe to strobe the /MEXC memory exception signal into the CY7C611. /MDS can only be asserted when the pipeline is frozen via assertion of /MHOLDA or /MHOLDB. /MDS must be de-asserted before or simultaneously with the release of memory hold (*Figure 6* and *7*).

3. /MEXC — Memory Exception (CY7C611 input)

Assertion of this signal initiates an instruction access or data access exception trap and indicates to the processor that an attempt was made to access an invalid address. /MEXC serves as a qualifier for the /MDS signal and must be asserted when both /MHOLD(A/B) and /MDS are already asserted. When /MEXC is generated with /MDS, the contents of the data bus are ignored. /MEXC is latched on the clock's rising edge and is used in the subsequent cycle. /MEXC must be released in the same cycle that memory hold is released (*Figure6* and *7*).

4. INULL — Instruction Nullify (CY7C611 output)

The processor asserts INULL to indicate that the current memory access is being nullified. The signal is asserted in the same cycle in which the address being nullified is active, although the address is no longer on the address bus. The address is held in external latches. INULL is used to disable memory exception generation for the current memory access. This means INULL should not be asserted during a memory exception. INULL is asserted under the following conditions:

- During the second data cycle of any store instruction, to nullify the second occurrence of the store address, i.e., if the address was valid the first time, it is still valid the second time
- On all traps, to nullify the third instruction fetch after the trapped instruction

- On a load in which the hardware interlock is activated
- On JMPL and RETT instructions

INULL is used as a Mealy input to the memory-exception generator to inhibit memory exception generation during nullified load memory accesses. The signal has no effect on the store exception state machine (*Figures2* through *7*).

5. /WE — Write Enable (CY7C611 output)

/WE is asserted during the cycle(s) in which store data is on the data bus. The address-checking circuit uses this signal to inhibit generation of address exceptions after the store has begun (*Figures4,5* and *7*).

6. WRT — Advanced Write (CY7C611 output)

WRT is asserted in two cases: during the first store address cycle of integer single or double store instructions and during the second load/store address cycle of atomic load/store instructions. The memory-exception generator uses this signal to enable either the store (WRT = 1) or load (WRT = 0) state machines (*Figures2* through *7*).

Address-Checking Circuit

The address-checking circuit fits completely into a single CY7C332. This PLD was chosen because it has the required number of I/O pins, a very narrow capture window, and inputs that are configurable as latches. Configuring the inputs as latches allows you to make maximum use of the CY7C611's 10-ns address setup before the system clock's rising edge (more on this later).

The inputs to the address-checking circuit are the 22 most significant bits of the processor address bus, the system clock (SCLK), and /WE. The output of the address-checking circuit is a single line: Address EXCeption (/AEXC). /AEXC is inhibited when /WE is active. *Figures 4* and *5* show that /WE is active only when store data is on the data bus, i.e, after the first address cycle of a store. At this point, because it is too late to stop a store and generate a memory exception, /AEXC is inhibited. Note that /AEXC is inhibited only on the data portions of store bus cycles.

Memory-Exception Generator

The memory-exception generator occupies roughly 1/3 of a CY7C361 PLD and must accomplish two things. First, the circuit must respond to address exceptions generated by the address-checking circuit. Second, the memory-exception generator must know when not to respond to memory exceptions generated by the address-checking circuit.

The second case requires the use of a Mealy input/output pair in the CY7C361. The CY7C361 was chosen for its Mealy I/O capability and its input configurability. Each input can be configured as single registered, double registered, or combinatorial. This design uses both single-registered and combinatorial inputs.

At first glance, INULL looks like the perfect signal to inhibit memory-exception generation and reset the memory-exception generator to its initial state. But as *Figure 7* shows, if the store's first address cycle causes the address exception, /MHOLDx is asserted just after

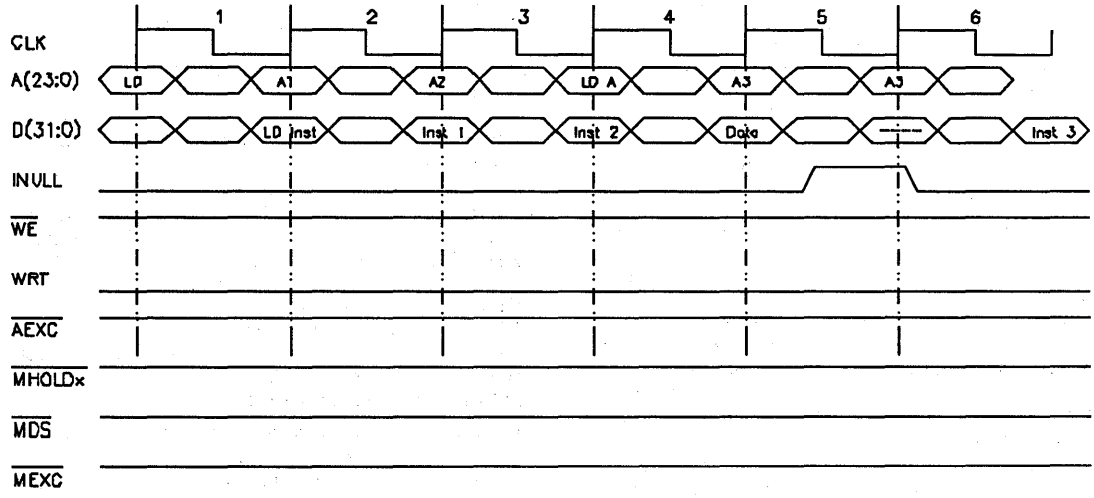


Figure 3. Load Double Bus Cycle

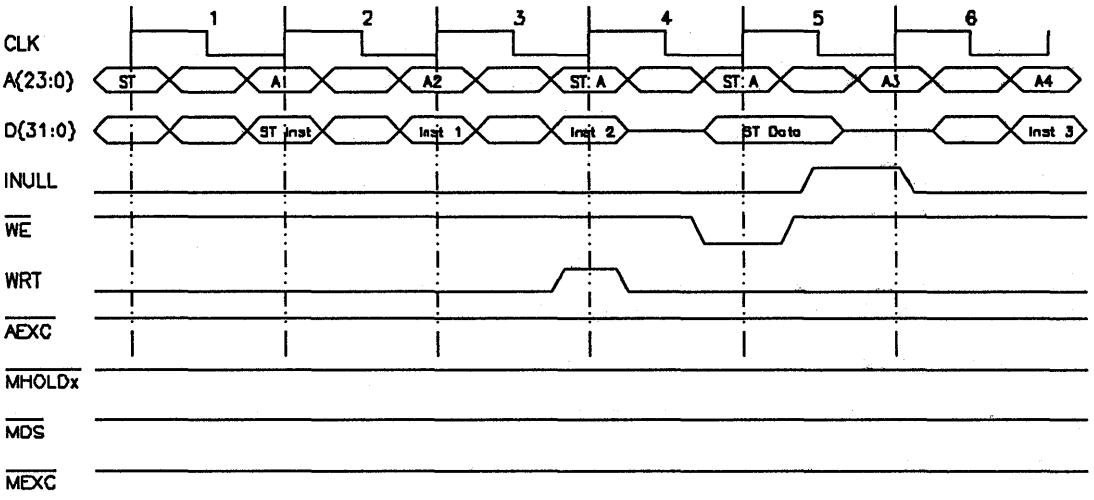


Figure 4. Store Bus Cycle

the next SCLK rising edge. At this point, INULL inhibits /MHOLDx and resets the exception circuit before it can generate an exception — an undesired chain of events.

To avoid this problem, the memory-exception generator is actually two state machines — one for stores and one for loads. The load-exception state machine has a Mealy output connected to the CY7C611's /MHOLDB input, and the store-exception state machine has a regular CY7C361 output connected to the /MHOLDA input. Thus, INULL can inhibit nullified load transactions but has no effect on stores.

The equivalent function for stores is accomplished in the address-checking circuit with the /WE input. When /WE is active, the address-checking circuit cannot generate address exceptions. Memory exceptions

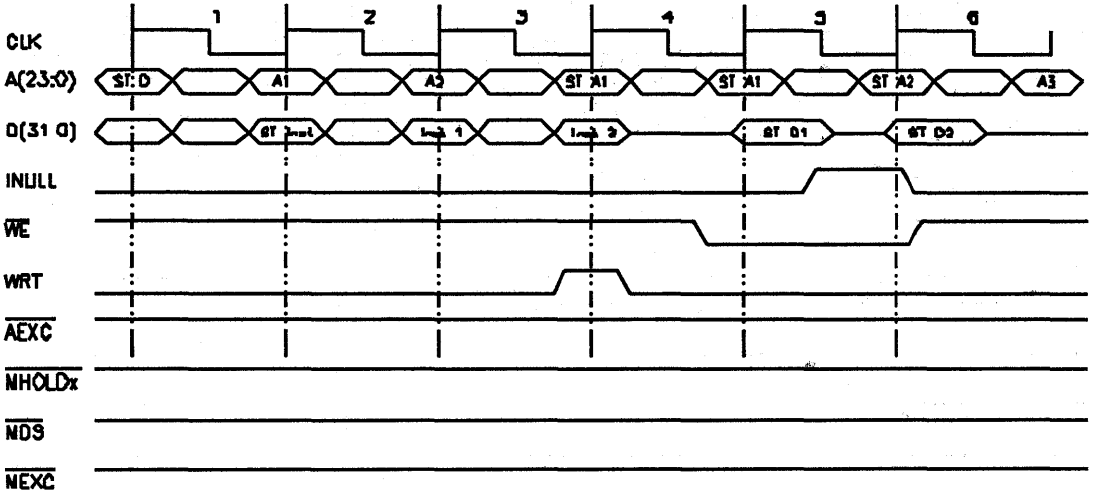


Figure 5. Store Double Bus Cycle

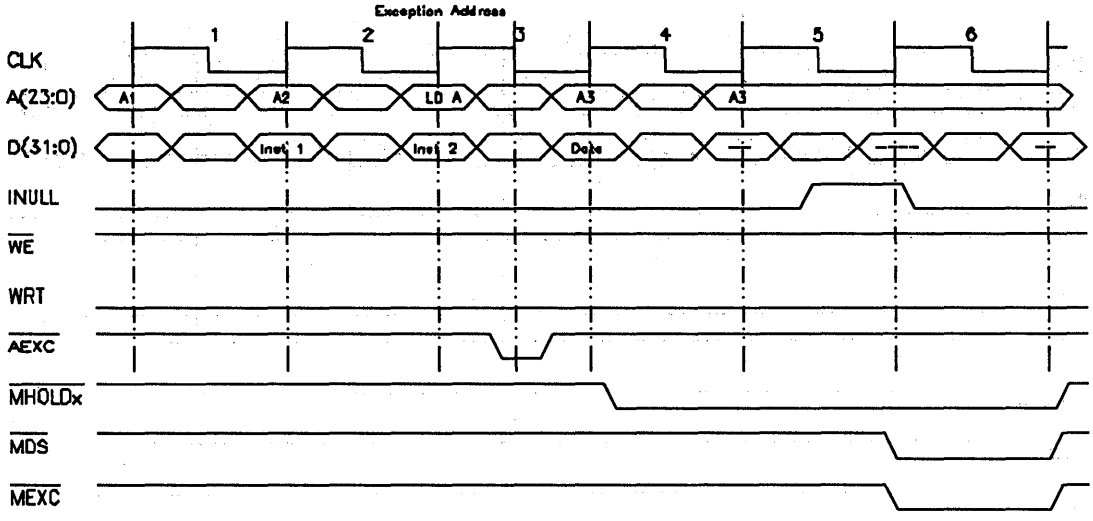


Figure 6. Load with Memory Exception Bus Cycle

can therefore be generated. Recall that /WE is only active during data portions of store transactions, when it is too late to generate memory exceptions for that specific transaction.

General Timing

A 20-ns CY7C332 and a 100-MHz CY7C361 are sufficient to support 25-MHz operation of the CY7C611. Changing the CY7C332 to the newly available 15-ns version allows higher processor clock rates. All timing described here is for a 25-MHz system.

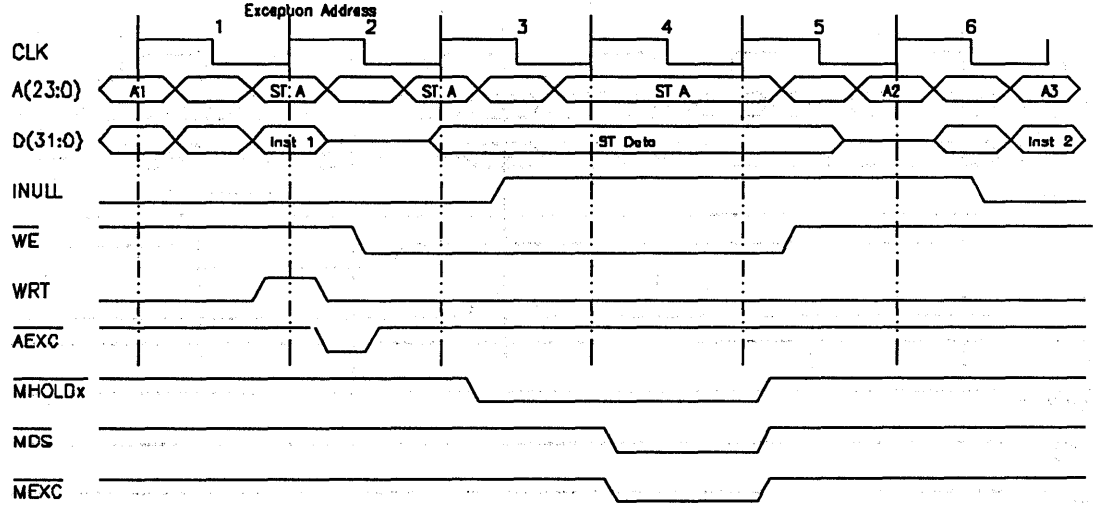


Figure 7. Store with Memory Exception Bus Cycle

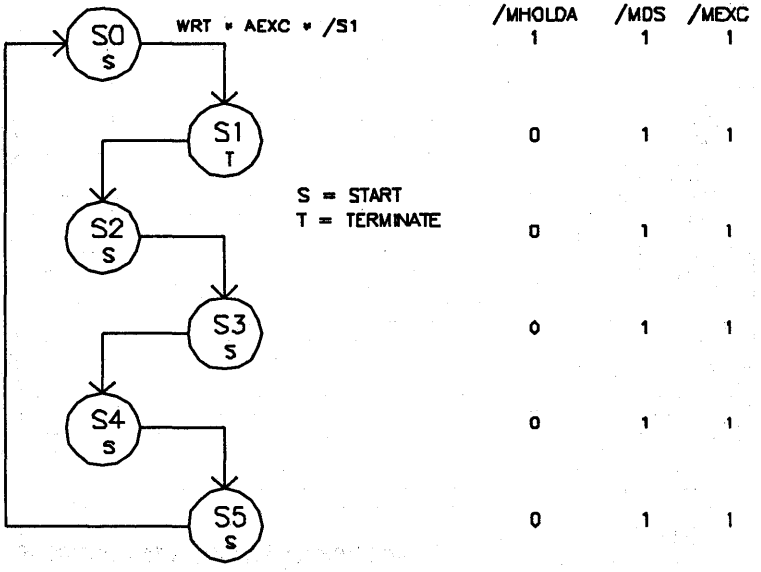


Figure 8. Store Exception State Machine

The address is captured in the transparent latch inputs of the CY7C332-20. The latches are transparent when SCLK is Low. This gives you use of the 10-ns setup of the address to the clock's rising edge. 10 ns after the latches close (SCLK rising), /AEXC is valid. This signal is clocked into the CY7C361-100 on SCLK's middle edge. If /AEXC and WRT are both active, the store-exception state machine is started. If /AEXC is active and WRT is inactive, the load-exception state

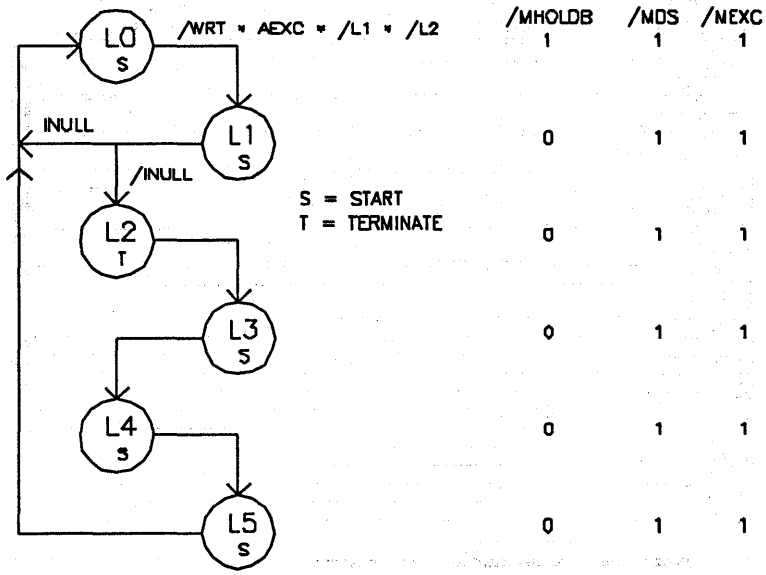


Figure 9. Load Exception State Machine

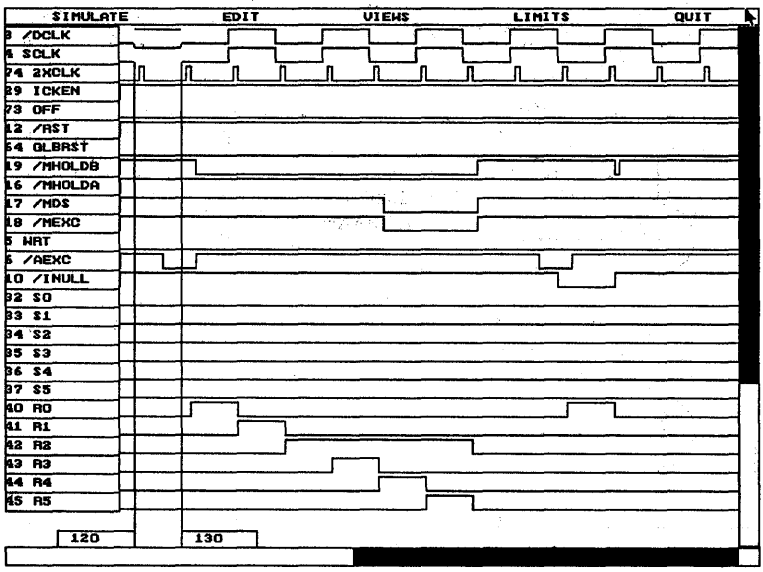


Figure 10. PLD ToolKit Simulator Output (Load)

machine is enabled. WRT is captured in a single CY7C361 registered input.

If INULL becomes active and the bus cycle is *not* a store, the /MHOLDB output is inhibited, and the load-exception state machine is returned to the initial state. Note that the signal DCLK (Duplicate CLocK) qualifies various inputs to the CY7C361. This is required because if the clock doubler is enabled, as it is here, all the inputs and the macrocells are clocked off the doubled clock; however, some inputs are only valid on a single edge of the clock. (For additional information on the CY7C361, refer to the application note, "Understanding the Cypress CY7C361."

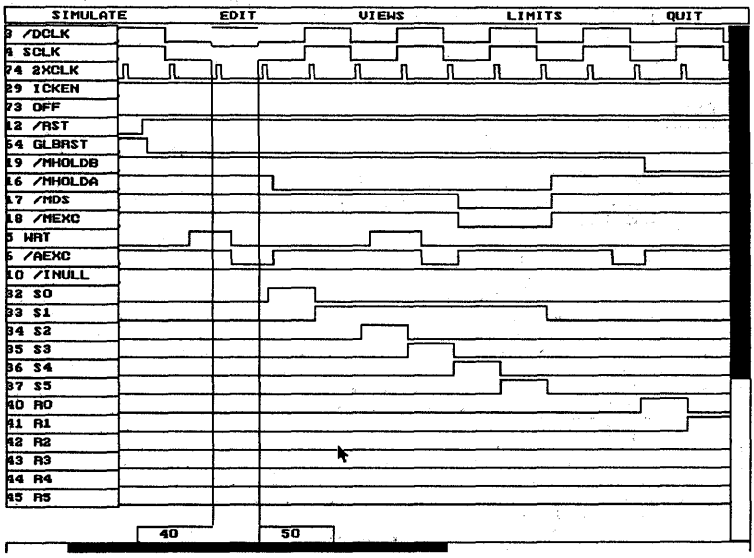


Figure 11. PLD ToolKit Simulator Output (Store)

Waveform diagrams for both load and store exceptions appear in *Figures 10* and *11*. The diagrams illustrate the state sequencing inside the CY7C361 and the output decode from the internal states. Notice how the terminate function inhibits generation of further memory exceptions until the current exception has been completed. These diagrams were captured from the Cypress PLD ToolKit simulator.

Appendix A contains the ABEL source code for the CY7C332 portion of the design, and *Appendix B* contains the PLD ToolKit source code for the CY7C361 portion of the design. These files are also available via the Cypress bulletin board. Contact your local Cypress office for details on how to access the bulletin board.

Appendix A. ABEL Source Code - Address Check Circuit

```
Declarations
"Inputs ********************************************

SCLK                    Pin 1;
A23,A22,A21,A20,A19,A18      Pin 2,3,4,5,6,7;
A17,A16,A15,A14,A13,A12      Pin 9,10,11,12,13,14;
A11,A10,A9,A8,A7,A6          Pin 15,16,17,18,19,20;
A5,A4,A3,A2                 Pin 23,24,25,26;
!WE                     Pin 28;
A11ireg,A10ireg,A9ireg       Node 91,92,93;
A8ireg,A7ireg,A6ireg         Node 94,95,96;
A5ireg,A4ireg,A3ireg         Node 97,98,99;
A2ireg,!WEireg              Node 100,102;

A23 istype 'latch'; A22 istype 'latch'; A21 istype 'latch'; A20 istype 'latch';
A19 istype 'latch'; A18 istype 'latch'; A17 istype 'latch'; A16 istype 'latch';
A15 istype 'latch'; A14 istype 'latch'; A13 istype 'latch'; A12 istype 'latch';
A11ireg istype 'latch'; A10ireg istype 'latch'; A9ireg istype 'latch';
A8ireg istype 'latch'; A7ireg istype 'latch'; A6ireg istype 'latch';
A5ireg istype 'latch'; A4ireg istype 'latch'; A3ireg istype 'latch';
A2ireg istype 'latch'; WEireg istype 'latch';

"Output ********************************************

!ADREXC                 Pin 27;

"Macros for readability
 ADR = [A23,A22,A21,A20,A19,A18,A17,A16,A15,A14,A13,A12,A11ireg,
A10ireg,A9ireg,A8ireg,A7ireg,A6ireg,A4ireg,A3ireg,A2ireg];

TOPPROM =  ^ h07ffff/4; "Top of boot PROM (read only)
RSV0L =  ^ h080000/4;  RSV0H =  ^ h0fffff/4;  "First reserved space
IOSTAT =  ^ h100000/4; "I/O stat register (read only)
IOCONT =  ^ h100004/4; "I/O control register (write only)
NUSD0L =  ^ h100008/4; NUSD0H =  ^ h1fffff/4; "First unused memory space
RAML =  ^ h200000/4;  RAMH =  ^ h5fffff/4;   "RAM
NUSD2L =  ^ h600000/4; NUSD2H =  ^ hbfffff/4; "2nd unused space
IOL =  ^ hc00000/4;   IOH =  ^ hdfffff/4;    " I/O space
RSV1L =  ^ he00000/4;  RSV1H =  ^ hffffff/4;  "2nd reserved space

C,H,L,Z,X = .C.,1,0,.Z.,.X.;
```

Appendix A. ABEL Source Code - Address Check Circuit

```
EQUATIONS

A23.C =  !SCLK; A22.C =  !SCLK; A21.C =  !SCLK;
A20.C =  !SCLK; A19.C =  !SCLK; A18.C =  !SCLK;
A17.C =  !SCLK; A16.C =  !SCLK; A15.C =  !SCLK;
A14.C =  !SCLK; A13.C =  !SCLK; A12.C =  !SCLK;
A11ireg.C =  !SCLK; A10ireg.C =  !SCLK;
A9ireg.C =  !SCLK; A8ireg.C =  !SCLK;
A7ireg.C =  !SCLK; A6ireg.C =  !SCLK;
A5ireg.C =  !SCLK; A4ireg.C =  !SCLK;
A3ireg.C =  !SCLK; A2ireg.C =  !SCLK;
WEireg.C =  !SCLK;

!ADREXC =  WEireg & (
     (0 ADR) & (ADR  TOPPROM)
   #  (RSV0L  ADR) & (ADR  RSV0H)
   #  (IOSTAT = =  ADR)
   #  (IOCONT = =  ADR)
   #  (NUSD0L  ADR) & (ADR  NUSD0H)
   #  (NUSD2L  ADR) & (ADR  NUSD2H)
   #  (RSV1L  ADR) & (ADR  RSV1H) );


Test_Vectors  ([ADR, !WE, SCLK] - !ADREXC)
          [TOPPROM, 0, 0] - 1;
          [TOPPROM, 0, 1] - 1;
          [TOPPROM+ 1, 0, 0] - X;
          [TOPPROM+ 1, 0, 1] - 1;
          [TOPPROM+ 1, 1, 0] - X;
          [TOPPROM+ 1, 1, 1] - 0;

end
```

Appendix B. PLD ToolKit Source - Memory Exception Generator

```
CY7C361;
CONFIGURE;
!DCLK(node=3),
SCLK(node=4),
2XCLK(node=74,dbl_clk),
ICKEN(node= 29), {Enable for WRT input clk}
OFF(node = 73),
!RST(node=12),
GLBRST(node=64),

!MHOLDB(node=19,and), {This output will default to active low}
!MHOLDA(node=16),!MDS,!MEXC, {Active low pure outputs}

WRT(node=5,ireg),
!AEXC(node=6),
!INULL(node= 10),    {A 0 on this input will force !MHOLDB HIGH}

S0(node= 32,start),   {STORE STATE MACHINE}
S1(cin,term),
S2(cin,start),
S3(cin,start),
S4(cin,start),
S5(cin,start),

R0(node= 40,start),   {READ STATE MACHINE}
R1(start),
R2(cin,term),
R3(cin,start),
R4(cin,start),
R5(cin,start),

EQUATIONS;
GLBRST  = <prod> RST;
ICKEN = <inv_sum>; {always on}
{************************************************************
S states are states for Store operations, R states are states for Read
operations. DCLK is used to ensure sampling only when inputs are valid.
************************************************************}
S0 = <prod> WRT * AEXC * !DCLK * !S1; {These conditions start
                    state machine}
S1 = < prod> S5;    {S1 is term, so S5 terminates this output}

R0 = <prod> !WRT * AEXC * DCLK * !R1 * !R2; {These conditions
                        start state machine}
R1 = <prod> R0 * !INULL * !DCLK;{R2 is term, so R5 terminates
                    this output}
R2 = < prod> R5;          {R5 turns off the state machine}

MHOLDA = < inv_sum> !S1 * !S0;          {Active for 0, 1}
MDS = < inv_sum> !S4 * !S5 * !R4 * !R5; {Active for 4 and 5}
MEXC = < inv_sum> !S4 * !S5 * !R4 * !R5; {"              "}
MHOLDB = < inv_sum> !R2 * !R1 * !R0;    {Active for 0, 1, 2}
```

Section Contents

Bus Products

CYPRESS SEMICONDUCTOR

Features of the VIC068 VMEbus Interface Controller

This application note describes some features of the Cypress VIC068 and provides information on how to use the device.

The VIC068 was designed by a consortium of VMEbus manufacturers in partnership with Cypress. The major goals of this consortium were to achieve a standardized, reasonably priced VMEbus interface that was not dominated by any board manufacturer. Manufacturing this specialized chip requires a high-speed process (125 MHz) and high-power I/O pins (64 and 48 mA).

The VIC068 adheres to the ANSI/IEEE Standard 1014, which minimizes the problems of interfacing among the VMEbus boards of various manufacturers. A block diagram detailing the device's functional blocks appears in *Figure 1*.

VIC068 Highlights

With very precise timing, based on a 64-MHz clock that is used internally to make decisions on 8-ns intervals, you can reach the theoretical limits of the VMEbus transfer rates — a block transfer rate of 40 MBytes/s.

Because all logic resides in a single chip, the VIC068 greatly reduces the board space necessary to interface to the VMEbus. Even a highly sophisticated interface with an A32/D32 system controller and block transfer support requires no more than 60 square cm or 20 percent of a double eurocard (6U card).

Special care has been taken to speed up the VIC068's VMEbus access. Although many of today's CPU boards use megabytes of high-speed local RAM to limit the number of VMEbus accesses, the accesses that do occur for I/O or data reads and writes must be done efficiently to avoid slowing the rest of the system.

For both types of data transfers, the VIC068 offers special support. For single-write cycles, you can program the VIC068 to operate in the so-called master or slave write-posting mode. In the master write-posting mode (*Figure 2*), the local VMEbus write cycle is terminated locally as soon as data is latched in the VMEbus latches. This allows the local CPU to continue with instruction fetches or other operations while the VIC068 transfers data over the VMEbus.

In slave write-posting mode (*Figure 3*), the same function happens with write cycles from the VMEbus to the local bus. As soon as the data is latched, the VMEbus cycle is terminated and the local cycle can finish independently of further VMEbus traffic. Both modes reduce CPU overhead and VMEbus utilization, providing higher bandwidth in single-cycle writes.

The VMEbus prohibits a similar function in single-cycle reads because every read cycle on the VMEbus could turn out to be a read-modify-write (RMW) cycle. This cannot be foreseen because the only difference is that the address strobe is held Low between the two cycles. Therefore, if the VMEbus address strobe were released during the two portions of the same RMW cycle, another VMEbus master could break into that cycle and modify the same data.

To move blocks of data over the VMEbus, the VIC068 uses the block transfer mode. In its standard form, this mode allows a processor to transfer up to 256 bytes with just one starting address supplied to the VMEbus. Additionally, the VIC068 uses a type of pipelining to accelerate VMEbus throughput. On a block transfer read cycle, the slave VIC068 automatically prefetches the n+1 data byte during the same read. The nth data byte is transferred across the VMEbus, and the n-1 byte is latched in local RAM. As shown in *Figure 4*, this operation uses all three buses in overlapped and parallel operation to speed up the transfer. Write transfers use the same mechanism.

The limiting factor on the VMEbus transfer rate is either the VMEbus's many timing restrictions or the source or destination memories. If the memory consists of dynamic RAM, the restriction is probably the cycle time of the chips used, often as slow as 200 ns. To overcome this limitation, the VIC068 offers a programmable access mode so that attached DRAM can be used in page mode.

After a starting row address cycle (RAS), all subsequent cycles need only a column address (CAS) to reduce the access time, often by as much as half. For a slave interface, the VIC068 contains all the necessary counters and timing elements for local AS, DS, and address generation.

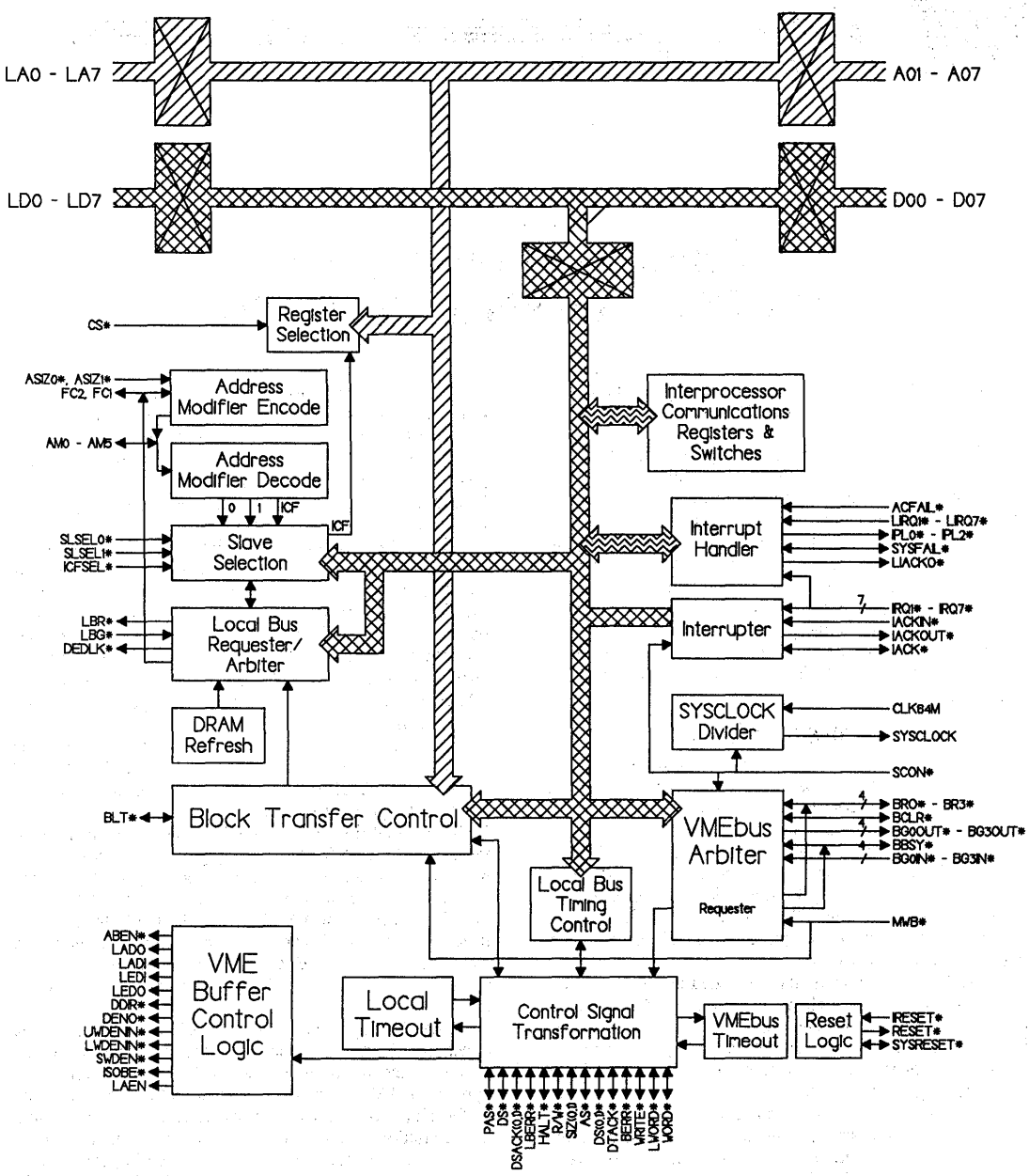


Figure 1. VIC068 Functional Block Diagram

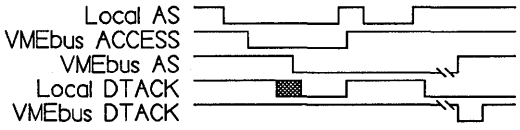


Figure 2. Master Write Posting

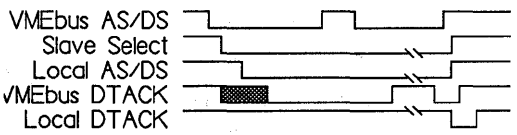


Figure 3. Slave Write Posting

A master block transfer needs two or three additional latches for the higher address lines during the local DMA part of the block transfer. Thus, even with low cost DRAMs, the VIC068's block transfer rate can reach 40 MBytes/s, limited only by the VMEbus specification and the physical characteristics of the VMEbus.

This transfer rate decreases the time needed to load programs or move data to graphics boards, as well as increasing the VMEbus's bandwidth, thereby allowing more CPUs to work together in a multiprocessor system.

Mailbox Signaling

To add greater capability to multiprocessor systems, the VIC068 has four interprocessor communication global switches (ICGSs) and four interprocessor communication module switches (ICMSs). These are all byte-wide mailbox registers that generate a local interrupt when accessed from the VMEbus. The ICGSs of one group reside at the same address and are accessed with a write cycle, which behaves as a broadcast to all members of the group. Because the ICMSs are at different addresses, one dedicated processor can be activated with a local interrupt request (IRQ).

A processor can inform a logical group of processors about a new task via a broadcast using the ICGSs and can then communicate with single processors about the task using the ICMSs.

Eight-byte-wide interprocessor communication registers (ICRs) are also available. Five of these registers serve as general-purpose read/write registers, and three are dedicated to control local activities (Halt, Reset, Mask ID, etc.). The ICRs can be read and written from the local side or the VMEbus without interfering with each other.

Interrupt Generator

The VIC068 handles up to seven simultaneous pending IRQs with separate vectors. The VIC068 also provides independent local IRQ vectors, if external IRQs are served.

Miscellaneous Features

The VIC068 furnishes several features for VMEbus support:

- SYSFAIL generation
- Software reset
- ACFAIL
- BERR register for detailed information

For local support, the VIC068 provides these features:

- Seven local IRQ sources, all level, polarity, edge and vector programmable.
- Local bus time out (2 - 512 ms)
- With/without VMEbus request time included
- 31 different local IRQ vectors
- VIC ID register

In addition to the VIC068, the following parts or equivalents are required for a minimum hardware interface:

- Three address latches and drivers (74xx543)
- Three data latches and drivers (74xx543)
- Four isolation buffers (74xx245)

You might also need the following:

- One to two PLDs for slave address decoding
- Two to three latches for master block transfer
- 1/2 PLD for block transfer glue logic

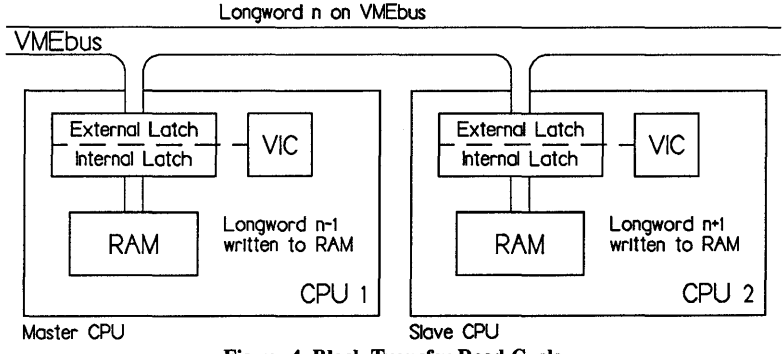


Figure 4. Block Transfer Read Cycle

Interfacing

To connect a processor other than the 680x0 to the VIC068, it is often easiest to map the processor control signals into the control signals available on a 680x0 type of processor. This type of transition interface offers the advantage of compatibility with a large family of 680x0-compatible peripheral parts, which you can then use elsewhere in the design.

Figure 5 shows a sample interface, whose four address latches store the multiplexed M-bus of the MC88000 processor. Four data latches store the data bytes after the acknowledge of the 680x0 bus and then start calculating parity for the processor's M-bus. The reason for this approach lies in some older peripheral I/O chips, which change their data lines when they should remain stable (i.e., transmit data buffer empty, etc.).

Two other data latches emulate the MC68020's dynamic bus sizing. The last buffer, between D0 - D7 of the 680x0 bus and AD16 - AD23 of the M-bus, emulates the 680x0 bus's IRQ cycles with normal read cycles of the MC88000.

Acknowledgment

Cypress Semiconductor wishes to thank Jürgen Bullacher of Eltec GMbH and Eltec International S.A.R.L. for submitting this article.

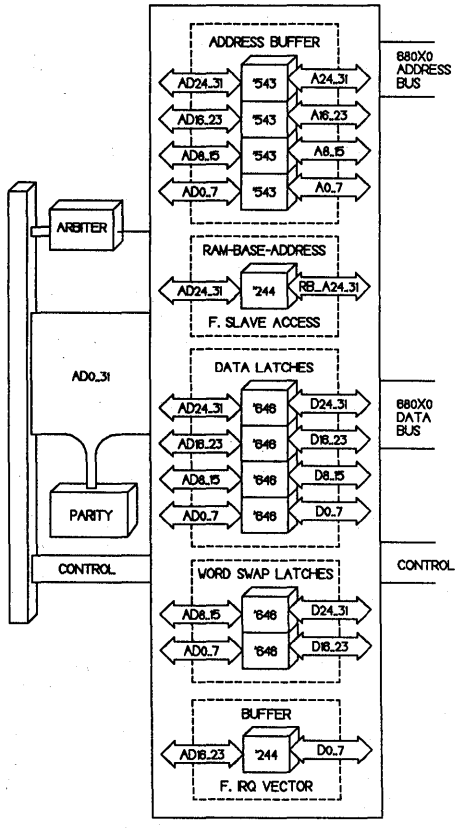


Figure 5. Sample Interface

CYPRESS SEMICONDUCTOR

Interfacing the VIC068 to the MC68020

This application note explains some of the features of the Cypress VIC068 and provides the first-time VIC068 user with simple implementations of these features. The VIC068 offers the most highly integrated VMEbus interface available today. It reduces the number of parts needed and saves board space. The emphasis in this application note is on interfacing the VIC068 as VMEbus A24/A16 D16/D08(E0) master/slave to the Motorola 68020.

Reset Operation

The VIC068 performs three distinct reset operations:
Internal reset — activated by the IRESET pin, which initializes most of the internal registers
System reset — essentially the same as IRESET, but is activated by writing ($F0) to the system reset register, or by asserting IRESET when the VIC068 is the VMEbus controller (SCON pin asserted)
Global reset — initializes all the VIC068 registers

After a reset, the 680X0 processor reads its initial stack pointer (SSP) and program counter (PC) from addresses $0 through $7. One way to handle this is to remap the boot-up ROMs to the low addresses for the first few cycles of the processor.

Figure 1 shows a circuit you can use to do this. The circuit uses a serial-in/parallel-out shift register (the 74HC164) to generate the MAP signal. This active-Low signal can be used with address-decode logic to force boot ROM access to the lower addresses during initial power up. Asserting the 74HC164 CLEAR pin drives all the parallel outputs Low, which asserts the selected MAP signal. With the two serial inputs tied High, each Low-to-High transition of the 68020 AS clocks the High through the shift register and out each of the parallel outputs. By picking the proper output for the MAP signal, you can decode from 1 to 8 of the initial processor cycles. You can use the MAP signals on memory configurations that are 8, 16, or 32 bits wide by using the QH, QD, or QB outputs, respectively.

Using The Processor RESET Instruction

The OR gate in *Figure 1* ensures that the 74HC164 is cleared only when HALT and RESET are both asserted. This allows the use of the 68020 RESET instruction without inadvertently re-asserting MAP. An alternative to this approach is to use two small-signal diodes (1N4148) and a pull-down resistor in place of the OR gate. This change reduces the design's parts count by eliminating the 74HC32.

A ROM remapping circuit must be used whether the RESET instruction is issued or not because of the way the VIC068 arbitrates local bus contention between the 68020 and the VMEbus. Contention occurs when both master and slave operations are requested concurrently (MWB asserted and SLSEL0, SLSEL1, or IFCSEL asserted). The VIC068 indicates this contention by asserting DEDLK. You can deal with the condition by setting bit 4 of the VIC068's interface configuration register ($AF) to assert

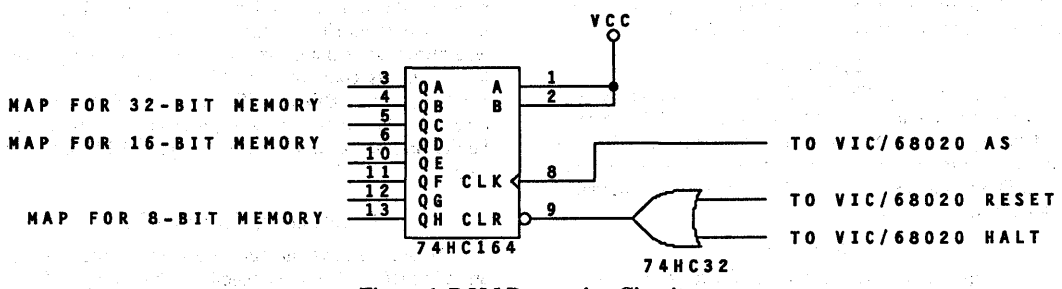


Figure 1. ROM Remapping Circuit

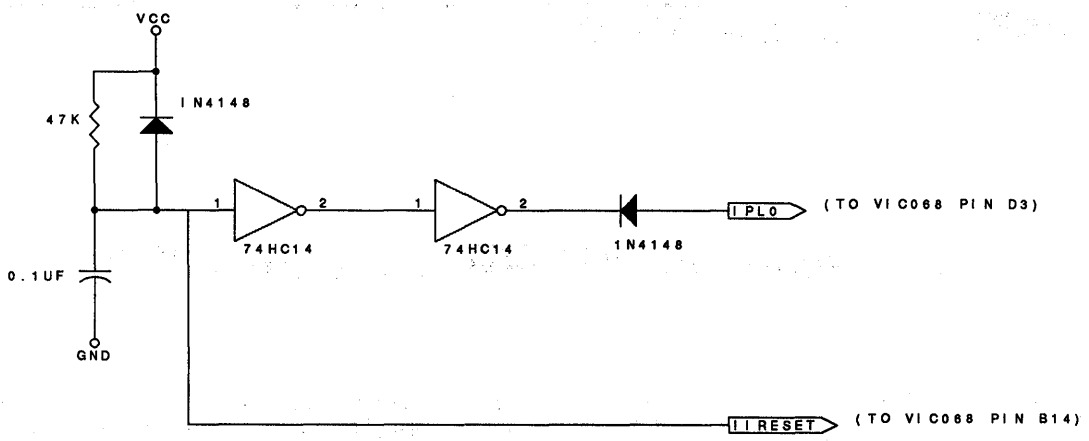


Figure 2. Global Reset Circuit

HALT along with LBERR when DEDLK occurs (68020 bus retry sequence). The VIC068 then waits for the 68020 to de-assert the MWB input. Once this happens, the VIC068 releases LBERR but continues to assert HALT to keep the 68020 off the local bus. The VIC068 then allows the slave operation to complete and deasserts HALT. The 68020 can now retry the contested bus cycle.

Internal Reset

At first glance, the IRESET might seem the logical choice for implementing the power-on reset. Because the IRESET input has some built-in hysteresis, a simple RC circuit would be appropriate for applying the power-on signal.

IRESET does not initialize the local bus timing register nor any of the slave select registers, however. Additionally, the VIC068 powers-up with the DRAM refresh option enabled (bit 4 of the arbiter/requester configuration register $B3 High). This condition is acceptable if you are using DRAM but adversely affects the external reset circuit in *Figure 1*. Specifically, for the first DRAM refresh cycle, the VIC068 deasserts RESET but maintains HALT in the active (Low) state and toggles AS. This action causes shift operations in the 74HC164. You can activate DRAM refresh after reset by writing a 1 to bit 4 of the arbiter/requester configuration register ($B3).

System Reset

The assertion of SYSRESET on the VMEbus typically activates system reset, but only when a global reset is not taking place. When the VIC068 is configured as the system controller (SCON pin asserted), it drives the SYSRESET pin for the required 200 ms during an internal or global reset.

Global Reset

The global reset is the most useful for power-up purposes because it places all the VIC068 registers in a known state. You initiate a global reset by asserting IPL(0) concurrent with or just after asserting IRESET. Because IPL(0) is also one of the encoded interrupt lines for the 68020, you must assert this signal with an open-collector device.

Figure 2 shows a typical power-up circuit for asserting IRESET and IPL(0). By using a device such as the 74HC14, you get the hysteresis necessary for the shallow charging slope of the RC circuit connected to IRESET. And because of the 74HC14's inherent propagation delay, you can easily meet the requirement for asserting IPL(0) after IRESET.

In using global reset, bear in mind that when the VIC068 powers-up it ignores the VMEbus SYSRESET. The VIC068 releases HALT and RESET after the 200-ms time out even if the current VMEbus master asserts SYSRESET past this required minimum time. This automatic release is a useful feature because it eliminates reliance on the system controller to release SYSRESET to start the power-up sequence.

The VIC068 generates a LBERR if you try to access the VMEbus or any of the VIC068 registers before SYSRESET is de-asserted. One solution to this problem is to structure the software so that the VIC068 registers are set up as late as possible in the power-up sequence. You can also temporarily point the 68020 BERR exception vector to an address containing an RTE instruction and let the 68020 cycle in a BERR/RTE loop until SYSRESET is deasserted. The latter approach provides an opportunity to be the first board in a system to request VMEbus mastership.

Connecting The Bus Lines

Figure 3 shows the standard buffer configuration for an A24/D16 VMEbus connection. This design also supports A16 and D08(E0) operation.

The D16/D08(E0) Data Bus

Connect the VIC068 to the 68020 as you would any 16-bit peripheral device. The 74FCT543 data buffer connects between the 68020 data bus's upper byte (D31 - 24) and the VMEbus D15 - 8 data lines. The lower byte (LD7 - LD0) is buffered through the VIC068 to the VMEbus low byte (D7 - D0). Several control signals connect directly from the VIC068 to the 74FCT543: DENO (data enable out) to OEAB (Output enable A-to-B), LWDENIN (lower word data enable) to OEBA (Output enable B-to-A), LEDO (latch enable data out) to LEAB (Latch enable A-to-B), and LEDI (latch enable data in) to LEBA (latch enable B-to-A).

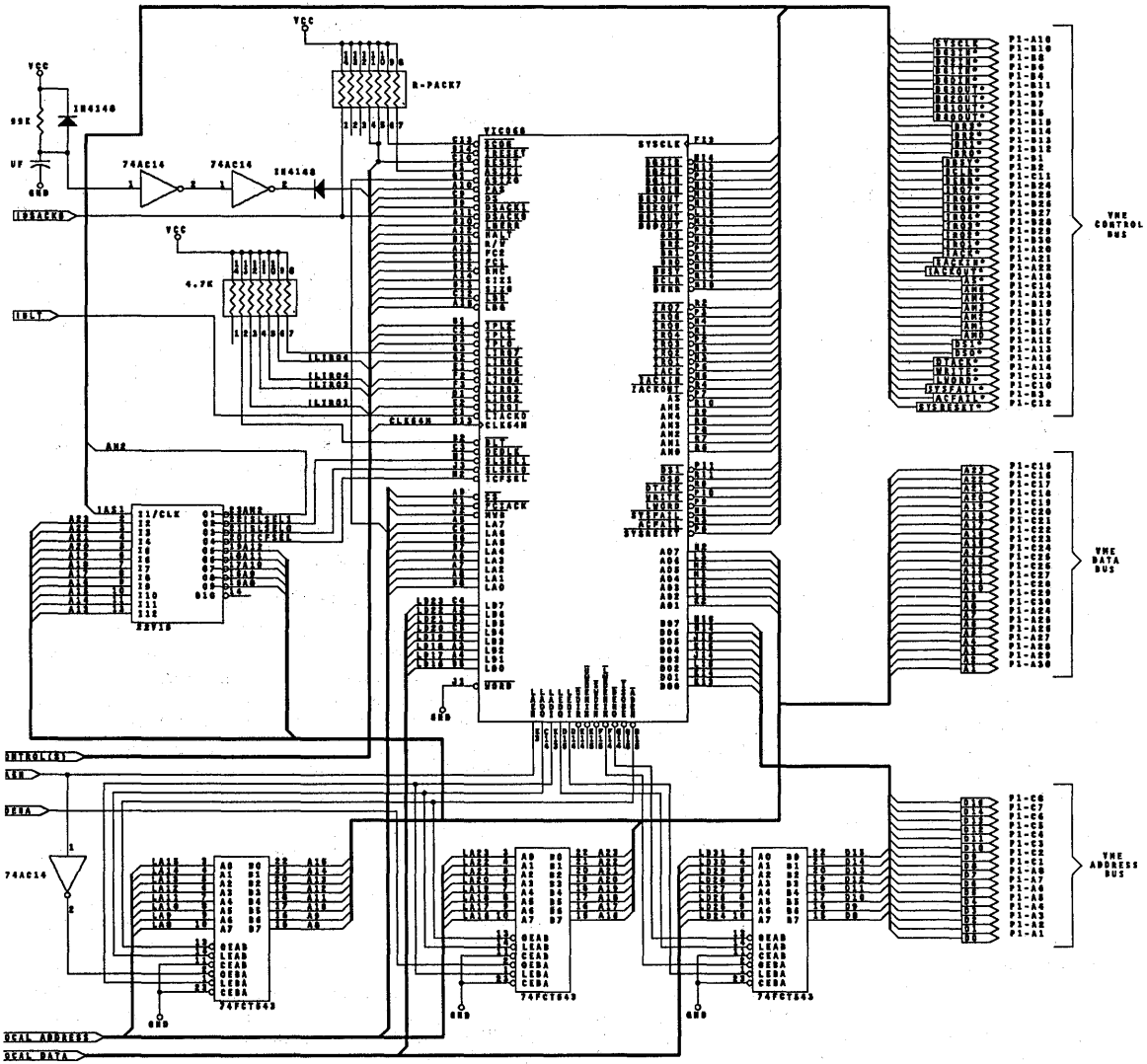


Figure 3. Address and Data Bus Connections

The Address Bus

The A24/A16 configuration requires the use of two more 74FCT543 devices to buffer and control the VMEbus A23 through A8 signals. The 74FCT543 LEAB, LEBA, and OEBA inputs connect directly to the VIC068 LADO (latch address out control), LADI (latch address in control), and ABEN (enable address out control) outputs, respectively. The output of the VIC068 LAEN (local-address enable control) must be connected to the 74FCT543 OEBA input through an inverter because LAEN is an active-High output and OEBA is an active-Low input.

Connecting The DSACK Lines

During the normal local bus operation, the 68020's slave devices (i.e., memory, UART, parallel port) must tell the processor the size of their data bus. This is done by asserting the DSACK1 inputs, which tells the 68020 that the port is a 16-bit device. Asserting DSACK0 instead indicates that the port is an 8-bit device. Asserting both DSACK1 and DSACK0 indicates that the port is 32 bits wide. To configure the VIC068 as a 16-bit port, simply connect the 68020 DSACK1 to the VIC068 DSACK1.

So long as there you have no requirement for VMEbus access to 8-bit devices on the local bus, you do not need to do anything with the VIC068 DSACK0 pin except terminate it (pull it High).

When you do need to access 8-bit devices, a small problem arises with the way the VIC068 acknowledges register accesses and interrupt-acknowledge cycles. During these cycles, the VIC068 always asserts both DSACK1 and DSACK0, whether the WORD input is asserted or not. And in VMEbus master cycles, when talking to an 8-bit device on the VMEbus, the VIC068 responds with DASCK0 to acknowledge the 8-bit transfer completion.

The solution to the DSACK0 problem is simple but can be complicated to implement: You must break the DASCK0 connection between the VIC068 and the 68020 during interrupt acknowledge or VIC068 register access (CS) cycles. The circuit needed to do this is a bidirectional, open-collector buffer between the VIC068 and 68020. The buffer should be inactive in both directions only when the VIC068 FCIACK or CS inputs are asserted. In *Figure 4*'s PAL equations, the DSACK0_020 and VIC068 DSACK0 equation illustrates how to handle the DSACK0 connection.

Master Operation

VMEbus master operation with the VIC068 is easily accomplished with the use of the MWB (module-wants-bus) input. The VMEbus can be requested at any level (0 - 3). The VMEbus can also be dynamically changed via the arbiter/requester configuration register ($B3), which eliminates the need for hardware jumpers. All VMEbus release modes are supported through the release control register ($D3). Support for write posting means that the local processor can write to the VMEbus without having to wait for the current bus master to release the bus or for the arbitration logic to assert the correct BGIN 9 (bus grant in) line. The VIC068 takes cares of this overhead for the local processor, improving system throughput.

To request VMEbus mastership, the 68020 asserts the MWB input. You can think of MWB as a VMEbus chip select.

When interfacing to the VMEbus as an A24 or A16 device, you can have access to the whole VMEbus address space by decoding a 32-Mbyte area of the 68020 address space for VMEbus operations. The ASIZ1-0 pins tell the VIC068 whether the current cycles represent an A32, A24, or A16 operation. You can use the upper 16-Mbyte address space (A24 High) for VMEbus A23 operation and the lower half (A24 Low) for VMEbus A16 operation by following three steps: decode A31 through A25 to generate MWB, tie the ASIZ1 input High, and connect the 68020 A24 address line to the VIC068's ASIZ0 input. *Figure 4* demonstrates this way of decoding MWB.

When the VIC068 recognizes a valid slave access, the device asserts LBR (68020 BR input) and waits for LBG assertion (68020 BG output). Once the VIC068 receives LBG, the device becomes the local bus master at the conclusion of the current cycle and completes the requested VMEbus slave operation. If the VIC068 is the only DMA device on the local bus, there is no need to generate BGACK (bus grant acknowledge) for the 68020. But if any other devices are capable of local bus mastership, you have to provide the arbitration logic and the BGACK signal for the 68020. Keep in mind, too, that other DMA devices must be able to recognize and deal appropriately with the 68020 bus-cycle entry operation (BERR and HALT asserted).

Slave Operation

The VIC068 can provide full VMEbus slave operation by dual-porting local memory with little or no 68020 overhead. The normal slave access operation starts by providing SLSEL0 or SLSEL1 through VMEbus address decoding. The circuits in *Figures 3* and *5* use a 22V10 PAL for this purpose. Always qualify VMEbus address decoding with the AS and/or DS1-0.

Decoding SLSEL0, SLSEL1, and IFCSEL

Figure 5 illustrates a typical PAL specification that you can use to provide address decoding for SLSEL0, SLSEL1, and IFCSEL. The VIC068 uses all the address modifier lines (AM5 - 0) to qualify the access mode. Address decoding can ignore these inputs. The VIC068 then decides if the access mode is legal and completes the cycle or generates the VMEbus BERR signal, depending on the value programmed in the slave select registers. You can also qualify the select outputs with the address modifiers and let the initiating device time-out if the access is not legal.

The IFCSEL input gives the VMEbus access to some of the VIC068 control registers and the interprocessor communication registers. These registers are available only through an A16 privileged-mode access.

```
module_CYCLE_DECODE;
    Cycle_decode device 'PV22V10';
    VCC,GND                              pin 24,12;

"inputs (15)
    A31,A30,A29,A28,A27,A26,A25,A19      pin 1,2,3,4,5,6,7,8;
    SLSEL1, SLSEL0                       pin 9,10;
    FC2,FC1,FC0,AS,LBG                   pin 13,14,15,16,17 "for FCIACK and VIC_Cycle output

"outputs (6)
    VIC_DSACK0,DSACK0_020      pin 18,19;    "To VIC DSACK0 and local system DASCK0
    VIC_CYCLE                  pin 20;       "current bus cycle is VMEbus
    FCIACK                     pin 21;       "Interrupt Acknowledge Cycle
    PRE_MWB,MWB                pin 22,23;    "VIC module-wants-bus (with and without AS)

"output type declarations
    VIC_CYCLE,PRE_MWB,MWB      istype 'com';
    FCIACK,VIC_DSACK0,DSACK0_020   istype 'com';
    VIC_CYCLE.OE,FCIACK.OE         istype 'com';
    PRE_MWB.OE,MWB.OE          istype 'com';
    VIC_DSACK0.OE,DSACK0_020.OE istype 'com';

equations in CYCLE_DECODE
"Enable ALL outputs except DSACK's
    VIC_CYCLE.OE    =1;
    PRE_MWB.OE =1;
    MWB.OE          =1;
    FCIACK.OE       =1;

"This signal tells everybody that the VIC068 is controlling the current bus cycle
    !VIC_CYCLE   =!LBG & AS                  "signal is asserted while AS is still high
                 #!VIC_CYCLE & !LBG &!AS "maintain signal through entire cycle

"Interrupt acknowledge cycle (68020 to VIC). Use VIC_CYCLE to insure this is not a VMEbus master cycle
    !FCIACK = A31 & A30 & A29 & A28 & A27 & A26 & A25 & A19 & FC2 & FC1 & FC0 & !AS & VIC_CYCLE;

"VME A24 access is at addresses $04000000 - $04FFFFFF. A16 access is at addresses $0500000 - $05FFFFFF (ASIZ0 is tied to LA24)
    !MWB = !A31 & !A30 & !A29 & !A28 & !A27 & A26 & !A25 & VIC_CYCLE &!(FC2 & FC1 & FC0);

"This is the same signal as MWB but the AS input is removed to provide an early VMEbus master cycle indication input to other PLDS
    !PRE_MWB = !A31 & !A30 & !A29 & !A28 & !A27 & A26 & !A25 & VIC_CYCLE &!(FC2 & FC1 & FC0);

"This signal is connected directly to the VIC DSACK0. It generates the VIC DSACK0 for VMEbus slave accesses to 8 bit device
    !VIC_DSACK0 = !VIC_CYCLE & !DSACK0_020;

"This enables VIC_DSACK0 only when VIC is the local bus master (slave accesses)
    VIC_DSACK0.OE = !VIC_CYCLE & (!SLSEL0 # !SLSEL1);

"This signal is connected to the 68020 DSACK). It generates the 68020 DSACK0 for VMEbus master accesses to 8 bit devices
    !DSACK0_020 = !MWB & VIC_CYCLE & !VIC_DSACK0;

"This enables the 68020 DSACK0 only when the VIC is the VMEbus master
    DSACK0_020.020 = !MWB & VIC_CYCLE;
end_CYCLE_DECODE
```

Figure 4. Abel Equations for PALC22V10 Cycle Decoding

```
module_VME_SLAVE;
title 'VMEbus slave access address decoding'

    VMEbus_Slave device 'PV22V10';

    VCC, GND            pin 24, 12;

"inputs (18)
    A23,A22,A21,A20     pin 1,2,3,4;
    A19,A18,A17,A16     pin 5,6,7,8;
    A11,A10,A9,A8       pin 14,15,16,17;
    AS                  pin 18;          "VMEbus address strobe
    AM2                 pin 22;          "High for supervisor, Low for user

"outputs (3)
    SLSEL0              pin 23;          "slave select for A16 access
    SLSEL1              pin 20;          "slave select for A24
    ICFSEL              pin 21;          "slave select for register access

"output type declarations
    SLSEL0              istype 'com';
    SLSEL1              istype 'com';
    ICFSEL          istype 'com';

    SLSEL0.OE           istype 'com';
    SLSEL1.OE           istype 'com';
    ICFSEL.OE           istype 'com';

"group assignment
    addr = [A23,A22,A21,A20,A19,A18,A17,A16,A15,A14,A13,A12,A11,A10,A9,A8,X,X,X,X,X,X,X,X];

equations in VME_SLAVE

"enable ALL outputs

    SLSEL0.OE           = 1;
    SLSEL1.OE           = 1;
    ICFSEL.OE           = 1;

"VIC068 slave select input 1, used for supervisor mode
    !SLSEL1 = !AS & (addr = ^h400000) & (addr = ^h4FFFFF) & AM2"ram access
            # !AS & (addr = ^h500400) & (addr = ^h53FFFF) & AM2"EEPROM access except for bootstrap
            # !AS & (addr = ^h540400) & (addr = ^h57FFFF) & AM2"bootstrap duplicate area
            # !AS & (addr = ^h580400) & (addr = ^h5BFFFF) & AM2"bootstrap duplicate area
            # !AS & (addr = ^h5C0400) & (addr = ^h5FFFFF) & AM2"bootstrap duplicate area
            # !AS & (addr = ^h404400) & (addr = ^h404FFF) & AM2"ram access in user mode

"The VIC068 slave select 0 input is used for user mode access
    !SLSEL0 = !AS & !A15 & A14 & !A13 & !A12 & !A11;

"VIC068 VME register access at addresses $4F00 thru $4FFF in supervisor mode only
    !ICFSEL = !AS & !A15 & A14 & !A13 & !A12 & A11 & A10 & A9 & A8;

end_VME_SLAVE;
```

Figure 5. Abel Equations for VIC068 Slave Decode

The PAL specification in *Figure 5* configures SLSEL0 to dual-port a 4-Kbyte (minus 256 bytes) space of local RAM as an A16 non-privileged access input and decodes IFCSEL in the SLSEL0 area's upper 256 bytes. You can use this 256-byte space for mailbox communication between boards in a multi-master system.

SLSEL1 is decoded as an A24 supervisory-only access and provides full dual-porting of the 68020 board's E^2PROM program memory. This allows the VMEbus system controller to put the system in a reset and hold state by asserting bit 6 of the VIC068's interprocessor communications register 7. The VMEbus master can then reprogram the entire program memory space. Once that operation is complete, the controller can use the interprocessor communications register 7 to release the reset and hold state. The board comes up running the newly installed program.

Take care when decoding SLSEL0, SLSEL1, and IFCSEL. The VIC068's operation is undefined when more that one of these inputs is active simultaneously.

Decoding for Supervisor/User Mode

You can use the VMEbus AM2 signal to select user (AM2 Low) or supervisor (AM2 High) modes. The AM2 input is used as part of the slave-select decoding shown in *Figure 5*.

Dealing with A24 and A16 Slave Accesses

Regardless of the access address size, the 74FCT543 address buffer outputs are enabled. Typically, the backplane pulls unused VMEbus address lines High passively, but most masters drive these lines regardless of the bus-cycle-address size. If this is not desirable, control the output-enable signals with the upper address line buffers using the VMEbus address modifiers. *Table 1* illustrates how to use AM5 and AM4 to determine the bus-cycle-address size.

You can derive individual enables for each of the VMEbus address latches by ANDing one or both of these address modifiers with the VIC068 LAEN (local-address enable) signal; modify both if operating in an A32 system.

Remember to provide a stable level for the local-address lines because nothing drives them during VMEbus accesses. You can ensure a stable level using 4.7-KΩ pull-up or pull-down resistors on the local-bus A31 - A16 lines. The local-bus address buffers can be set to the desired address state and enabled with the same latch-enable signals.

Dual-porting Local Memory

The PAL specification in *Figure 4* generates a signal called VIC_CYCLE than can serve as part of the local-address decoding to re-map local memory for dual-porting on the VMEbus. This approach allows memory placement at a VMEbus address independent of the local address.

Interrupts

The VIC068 interrupt structure is very versatile. One of the most useful features is the ability to redefine interrupt levels, and thus priorities, under normal program control. The VIC068 supports all seven levels of VMEbus interrupt as well as the seven local-interrupt levels. Interrupts are also available to notify the 68020 of VMEbus status and error conditions.

Figure 4 shows how to decode the 68020 interrupt acknowledge bus cycle to generate the VIC068 FCIACK input. You can omit A19 - A16 from the equation if you do not use breakpoints, a memory management unit (MMU), or a coprocessor (68881/68882).

Using LIACK0

The LIACK0 output is typically connected to the 68020 AVEC input to initiate autovectoring of interrupts to which the VIC068 has not been programmed to respond. You can also use LIACK0 with the IPL(2-0) outputs to generate interrupt-acknowledge signals to other 680X0-compatible interrupting devices.

LIRQ7 - 1 Inputs

The LIRQ7 - 1 inputs are the interrupt request inputs to the VIC068. The control register for each input allows you to determine the input's polarity (High/Low) and sensitivity (level or edge). The control register also allows you to define whether the VIC068 supplies the vector during interrupt acknowledge cycles or asserts LIACKO (local-interrupt acknowledge out), sets the level of interrupt the 68020 sees on IPL2 - 0, and enables or disables the interrupt. You do not need to terminate these inputs if you leave them unconnected, but you must pull them up externally if they are used.

The local interrupts (IPL2 - 0) are grouped and have a common vector base register ($57). This vector base is added to the encoded interrupt level programmed in each of the interrupt control registers to supply a unique vector to the 68020 for each interrupt input.

LIRQ2 is a special case because it can be used as an interrupt clock tick timer. You enable the timer through bits 2 and 3 of slave-select control register 0($C3). When enabled, LIRQ2 becomes the timer output, and the local-interrupt control register 2 ($2B) becomes the timer's interrupt-control register. The timer's periodic interrupt can be set to 50, 100, or 1000 Hz. If you plan on using the tick timer, do not connect the external interrupts to LIRQ2 because this pin becomes an output.

Table 1. Determining Bus-Cycle-Address Size

AM5	AM4	Cycle
H	H	A24 Access
H	L	A16 Access
L	L	A32 Access

Glossary

AHDL: Advanced Hardware Description Language, a high-level, modular language used to create logic designs for MAX EPLDs.

arbitration: The process of deciding which one of two or more competing entities will be allocated a resource.

arbitration time: The time taken to determine which of simultaneous contenders for a service takes priority.

associativity: The number of lines per set in a cache.

cache: A small, fast memory located between the CPU and main memory. A cache's purpose is to store copies of the instructions and/or data the CPU is most likely to need in the near future so that the CPU can access them more quickly than if they were stored only in main memory.

cache coherency: The state of a multiprocessor, multicache system in which the value of a shared variable is the same in all caches in which copies of the variable exist. In a multiprocessor, multicache system, a shared variable can be copied into more than one cache. If the copy is modified in one cache, steps must be taken to modify or invalidate the copies in the other caches to preserve cache coherency and prevent the other processes from using a non-current data value.

cache lock: In some caches, some lines can be locked into the cache. This means that the locked lines are never replaced by new lines. Users can lock critical programs in the cache to ensure that performance on these programs is high and deterministic.

cache tag: A table of the current contents of a cache. The tag itself is made up of a varying number of address bits that uniquely identifies each line in the cache as coming from a specific main memory line.

CAS: Column address strobe. In dynamic RAMs, this signal is asserted to strobe the column address of the current access into the device after the row has been input.

CMOS levels: There are two sets of CMOS specifications: HC and HCT. The older HC devices are generally not TTL compatible, and the newer HCT (also FACT, FCT, etc.) are TTL compatible. (See **TTL levels.**) Because the minimum V_{OH} level for TTL is 2.4V, TTL is not guaranteed to drive an HC input High. A 4 to 10,000Ω pull-up resistor to V_{CC} at the TTL device's output enables the device to achieve the HC V_{IH} level of 3.5V.

coherency (consistency): Agreement between shared contents of members of the memory system.

Compiler Netlist File (.CNF): A binary file that contains the data for a design, created by the ADF2CNF Converter (for state machine and Boolean equation designs) or the Compiler Netlist Extractor module of the MAX+PLUS Compiler (for schematic designs).

crosstalk: The temporal change in either the magnetic field or the electric field of a signal on one conductor that results in an unwanted signal being coupled to other conductors.

DMA: Direct memory access, a design technique that offloads some of the I/O processing from the CPU. A DMA controller allows the CPU to continue operation while the controller controls block transfers between I/O and memory or between separate memories in a multiprocessor system.

ECL levels: ECL voltage levels are specified at various temperatures. Only the values at 25°C are listed here. There are two families of ECL circuits: 10K and 100K. The older 10K devices are not temperature compensated. The newer 100K are.

effective access time: A cache performance metric giving the average time required to service a memory reference.

finite state machine: A synchronous sequential circuit whose current state is defined by the contents of its flip-flops and whose next state is determined by its current state and the inputs to the circuit.

FPLS: Field-programmable logic sequencer, a programmable device with an AND array, an OR array, and registers with feedback.

Graphic Design File (.GDF): A file created by the MAX+PLUS Graphic Editor or the ADF2CNF Converter. GDFs created by the Graphic Editor contain a schematic design in addition to the symbol that represents the file's inputs and outputs. GDFs created by the ADF2CNF Converter contain a symbol that represents the inputs and outputs of the original converted ADF or SMF. When the Compiler processes the file, this type of GDF acts as a pointer to the real design data that resides in a Compiler Netlist File.

ground bounce: When many outputs of a device change from High to Low there is a rush of current into the output drivers. If the inductance to ground is sufficient the virtual ground level is raised due to this inductance. The voltage spike caused by this phenomenon is called ground bounce.

Hierarchy Interconnect File (.HIF): A binary file created by the MAX+PLUS Compiler's Netlist Extractor module. The file specifies the hierarchical interconnections between files of one design. This file is required for design simulation even if the design consists of one file only.

LAB: Logic array block. In Cypress MAX PLD devices, the LAB represents a separate functional block in the device. Each type of MAX PLD has a different number of LABs.

latch-up: The fabrication of CMOS integrated circuits results in parasitic PNP and NPN transistors that form an SCR (PNPN). When the voltage at an input pin or an output pin of the device exceeds certain voltage levels, the SCR can be triggered and turn on (i.e., latch-up occurs), which creates a low-impedance path between V_{CC} and ground. When this occurs, if the V_{CC} current is not limited, the device is either destroyed or severely stressed because of the heat generated by the short circuit between V_{CC} and ground.

line (block): The basic unit of information exchange between a cache and main memory or between a parent cache and its child(ren) cache(s).

line size: The number of bytes or words in one cache/main memory line. In a cache system, a line is the quantum of data identified by the cache tag and is the smallest quantum of data that can be transferred between the cache and main memory. Whenever a new entry is placed into the cache, one line is transferred. Common line sizes are 16 and 32 bytes.

lockword (also called lockvariable): A memory location associated with a resource. The usual convention is that when the lockword is zero, the resource is available, and when the lockword is not zero the resource is being used by another process and is therefore not available.

macrocell: A low-level block of logic in programmable logic devices. This block can include one or more registers, along with configurable feedback and/or output paths.

master device: A device that controls the timing for data exchanges between two devices. Or when devices are cascaded in width, the master device is the one that controls the timing for data exchanges between the cascaded devices and an external interface. The controlled device is called the slave device.

Mbus: Memory Bus. Mbus is the Sun Microsystems standard bus interface between main memory and the CPU. It supports both uniprocessor and multiprocessor configurations.

Mealy machine: A state machine in which outputs depend on the current state and current inputs.

metastability: The property characterizing a device that is not in a known, fixed state, but in the metastable state — the state between the logical One and Zero levels. The metastable state is entered when the input voltage level at the input of a storage device is between these states when the clock changes. The time required to go from this indeterminate state to the logical One or Zero state is defined as the metastable settling time.

miss rate: A cache performance metric giving the probability that a reference will produce a miss.

Moore machine: A state machine in which the outputs depend only on the current state.

overshoot: The amount by which the amplitude of a signal transitions above its final value on a Low-to-High transition.

page table: A set of tables stored in main memory that translate virtual addresses to physical addresses.

Petrie net: A design methodology that employs token passing rather than encoded states.

physical address: The actual hardware address of a piece of information in main memory.

PIA: Programmable interconnect array. In Cypress MAX devices, the PIA is the routing path between separate logic array blocks (**LAB**s). The PIA is automatically routed and provides uniform timing throughout the devices.

placement algorithm: The method used to determine where a block may reside in a cache; often selects the set of a reference.

Programmer Object File (.POF): A binary file created by the MAX+PLUS Compiler for use with the MAX+PLUS Programmer. The POF contains the data used to program the MAX EPLD.

RAS: Row address strobe. In dynamic RAMs, this signal is asserted to strobe the row address into the device; the address inputs are time-multiplexed.

reference: A request by the processor to read or write a memory location.

Report File (.RPT): A file generated by the MAX+PLUS Compiler's Fitter module indicating how EPLD resources are used by the design. This report is particularly important if you have not specified all pin assignments. The RPT file contains header information, utilization information, and a graphical representation of pin assignments.

retransmit: Applies to the CY7C42X and CY7C43X families of large FIFOs when they are operating in either the stand-alone or width-expansion modes. When the retransmit pin is pulsed Low, the internal read pointer is set to the first physical location of the FIFO. Subsequent Low pulses on the read pin cause the FIFO's contents to be output until the read pointer equals the write pointer.

semaphore: A software technique for providing explicit mutual synchronization of parallel sequential (software) processes. Semaphores are initialized with the value Zero or One before the processes are started. After initialization, the processes access the semaphores only via two specific operations — the so-called synchronizing primitives. The operations carried out on semaphores are referred to as P and V, which are the first letters of the Dutch words corresponding to WAIT and SIGNAL, respectively.

set: A collection of cache locations in which a line may reside.

set associativity: A property that allows a cache to be divided into sets, each of which contains one or more lines. This property enables a line of main memory to map to more than one line in the cache; the line of main memory can map to one line in each of the sets. When searching the cache, the tags of one line from each of the sets are compared to the reference tag concurrently to determine to which set, if any, the main memory line was mapped.

Simulator Channel File (.SCF): A MAX+PLUS Simulator input and output file that contains a waveform representation of the input nodes that drive simulation, as well as the output nodes that are simulated. These waveforms represent High, Low, high-impedance, and undefined logic levels.

Simulator Netlist File (.SNF): The MAX+PLUS file that contains all data required to simulate a design. This file is created by the optional Simulator Netlist Extractor module of the MAX+PLUS Compiler.

slave device: A device that allows another device to control the timing for data exchanges between the two devices. Or when devices are cascaded in width, the slave device is the one that allows another device to control the timing for data exchanges between the cascaded devices and an external interface. The controlling device is called the master device.

Text Design File (.TDF): A MAX+PLUS text file created with the Advanced Hardware Description Language (AHDL). A TDF can be incorporated into the hierarchy of a design at any level.

transparent write: Possible only on separate I/O RAMs. During a transparent write, the data appears at the outputs as the data is written into the array.

TTL levels: The maximum value of the input Low (Zero) voltage level is V_{IL} = 0.8V. The minimum value of the input High (One) voltage level is V_{IH} = 2V. The maximum value of the output Low voltage level is V_{OL} = 0.4V. The minimum value of the output High voltage level is V_{OH} = 2.4V. The threshold level is approximately 1.5V. These values apply over the operating temperature range of interest.

undershoot: The amount by which the amplitude of a signal transitions below its final value on a High-to-Low transition.

Vector File (.VEC): A MAX+PLUS file that contains vectors that specify the logic levels of input nodes in a MAX EPLD design, which the Simulator uses to test the design's logical operation. This file describes the input conditions as well as output nodes to be simulated.

virtual address: An address generated by a program and later translated into a real address for main memory.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

• CYPRESS •

SEMICONDUCTOR

3901 NORTH FIRST STREET

SAN JOSE CA 95134

408 · 943 · 2600

1-891APPBK 40000