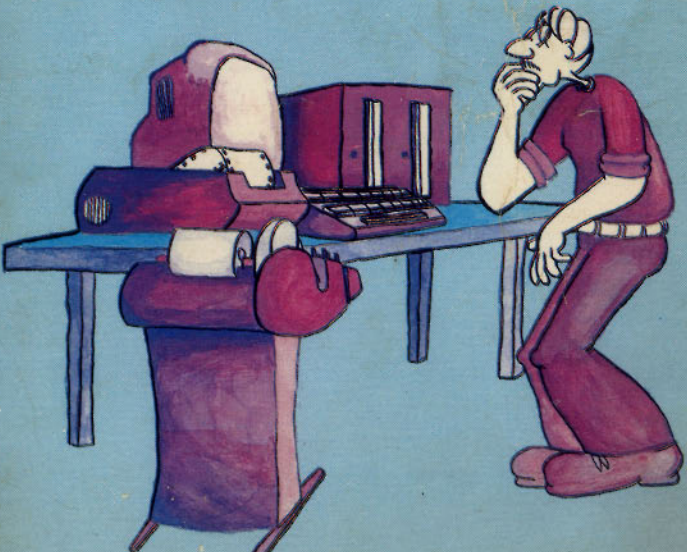
AN INTRODUCTION TO MICROCOMPUTERS

VOLUME 0
THE BEGINNER'S BOOK

2nd Edition



By Adam Osborne

AN INTRODUCTION TO MICROCOMPUTERS VOLUME 0
THE BEGINNER'S BOOK

931988-26-8

AN INTRODUCTION TO MICROCOMPUTERS

VOLUME 0
THE BEGINNER'S BOOK

2nd Edition

by Adam Osborne

Osborne & Associates, Inc.
Berkeley, California

For information on translations and book distributors outside of the United States of America, please write Osborne & Associates, Inc.

1st printing (1st edition) October 1977
2nd printing (2nd edition) May 1979

Library of Congress 76-374891
ISBN 0-931988-08-X

Photographs in this book courtesy of:

The Byte Shop
Digital Equipment Corporation
Heath Company
Heuristics, Inc.
Processor Technology Corporation
Qume Corporation
Technical Design Labs
Vector Graphics, Inc.

Published by Adam Osborne & Associates, Inc.
630 Bancroft Way, Berkeley, California 94710

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

QUICK INDEX

QUICK INDEX (Continued)

QUICK INDEX (Continued)

QUICK INDEX (Continued)

INTRODUCTION

This is a book about computers; it has been written for readers who know nothing about computers.

This book has been written for two audiences:

1) For those of you who have a real interest in learning how to use computers.

2) For everyone else — who must live with computers, like it or not, and had therefore better know a little about them.

For those of you who have a real interest in learning how to use computers, this is the first in a series of books. This book will explain superficially how computers work and what they can do. After reading this book you will be ready to move on to "Volume 1 — Basic Concepts", which gives you the information you will need in order to use computers.

But what if you can live a happy life and never program a computer — or even touch one? Why read this book? The answer, quite simply, is that computers in business and government have run wild; moreover, they have become the vehicles for a formidable new breed of criminals. And it is going to take a population who understand computers to bring them under control.

Let us begin by looking at computers in society.

Computers, like the automobile and electricity, are an integral part of daily life in any industrial society. During any normal day your life will be touched by computers many times.

Your name is probably on mailing lists — all maintained by computers; and that is why you receive junk mail. A large computer can print thousands of address labels in a minute. If a typist had to type address labels, you would never receive any junk mail; the sender could not afford the cost of the typist.

Credit cards exist because of computers. If human beings had to do all of the bookkeeping associated with credit card accounts by hand, the cost of accounting would make credit cards uneconomical.

Consider airline ticket reservations. You can walk up to a ticket counter and request a reservation on any flight, anywhere in the country; an operator is able to tell you instantly whether seats are available; and if you make a reservation, it is recorded — also instantly. If another customer, hundreds of miles away, requests the same seat ten seconds later, the customer will not be sold your ticket. Overbooking on airlines is intentional; computers tell the airlines exactly how many tickets have been sold, and on which flights.

The very size of government is a direct outgrowth of computers. In reality, government is little more than a vast accounting system; government collects money, budgets it and spends it. The very existence of sound government is based on its ability to track its income and its expenditures; government would surely collapse — jokes notwithstanding — if it did not have a pretty good idea of what it was doing. The sheer magnitude of government today is a direct result of its ability to manage huge cash flows — which it can do only by using computers.

And the future is going to bring us more computers, not less. The National US. Census of 1950 was made possible by ENIAC I, considered by some to be the world's first commercial computer. ENIAC I cost more than half a million dollars (that is 1950 dollars). Today you can buy the same computing power for $10 (that is 1977 dollars). In the late 50's and early 60's computers costing a million dollars (or more) started to handle data processing for very large companies — who could afford the high price. An equivalent computer system is available today for $2,000 or $3,000 — cheap enough to do the bookkeeping for the local drugstore. Indeed, computers now are so inexpensive that they drive video games, children's toys, oven timers, sewing machines and washing machines. In the not too distant future every automobile will have two or three computers controlling the engine and dashboard instruments.

We have all seen what electronics has done to the calculator and watch industries; electronic calculators and watches are driven by devices that are, in reality, small computers. The music industry will be the next to be revolutionized by computers and electronics. Soon you will be able to buy, at low cost, home recording equipment more sophisticated than recording studios possess today. Within ten years records and tape cassettes will be replaced by tiny electronic memories. That spells chaos for the music industry, and it puts a computer in every record player.(Make that "electronic memory" player.)

The ubiquitous presence of computers in our daily lives is the direct result of the computer's phenomenal efficiency — an efficiency which grows more phenomenal every day. But this efficiency is a two-edged sword. The very capabilities that make the computer such a powerful data processing tool also make it a formidable weapon for fraud, crime, and simple nonsense.

The first dishonest auto mechanic probably swindled his first customer a few months after the first automobile was sold; and the first dishonest programmer probably used a computer for criminal purposes soon after computer programs started to be written. Unfortunately too many of us continuously avoid the problems of computer misuse by assuming that computers are too complex for one person to misuse — and hide the fact from his fellow programmers. Nothing could be further from the truth. While you read this book, there are probably thousands of silent swindles in progress — the handiwork of dishonest companies or dishonest programmers within honest companies. And the way things stand today, only the very stupid, or the very unlucky will ever be discovered. We have, as a society, placed no obstacles in the path of anyone wishing to use a computer for any purpose, nor have we made the slightest attempt to monitor the manner in which computers are used. Not one in a hundred legislators know enough about computers to draft a realistic law regarding their use or control. Thus a recent swindle allowed a company to steal more than $200 million by the simple expedient of having their computer create fictitious records of insurance policies; after all, the computer printed it out, therefore the reader of the computer printout assumed that the insurance policies had to be real.

In reality the use of computers in our society is totally out of control. The things we do with computers far exceed our ability to monitor what is being done. Computers may well be the Achilles' Heel of our entire technological society; an enemy could bring this country to its knees faster by manipulating computer records than by dropping bombs. After all, when records of millions and billions of dollars are held in transient, magnetized zones on fast-moving computer equipment, the task of suitably manipulating these magnetized zones is not particularly formidable. A few well placed saboteurs could so scramble this country's financial records that no one would know any longer who owned or owed what money, where any government project stood, or what any business or agency was doing.

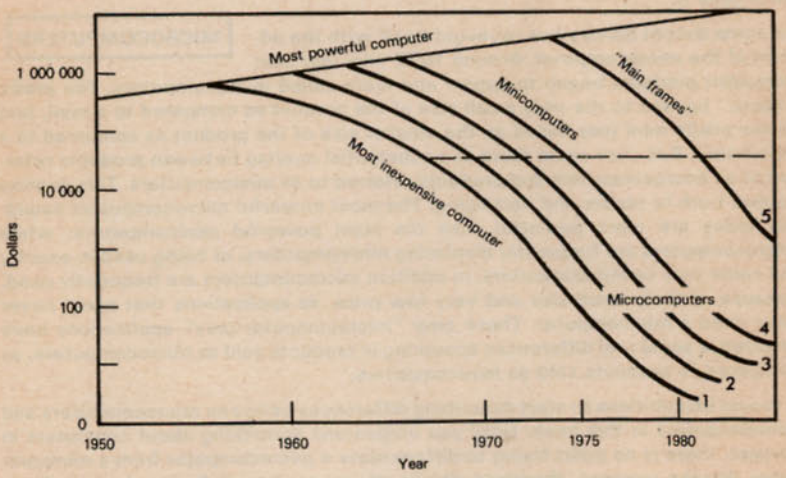
There is an urgent need for an intelligent population in any industrial society to understand just how simple computers really are and how easily they may be used or abused.

The myths surrounding computers result from the fact that not long ago computers really were million dollar monsters that filled a room. A little more than 20 years ago, the president of Sperry Rand — who got into computers before IBM — decided that computers would always be too expensive for general use and probably no more than 50 computers would ever be built; therefore Sperry Rand prematurely got out of the business. Now that was one of the biggest business blunders of all time. The computer which that company president saw filling a room and costing a million dollars today will fit on your fingernail and costs a very few dollars.

If a computing machine is so big that it fills a room, if it costs a million dollars to build — and hundreds of thousands more dollars to use, then very few of these machines will be built and they will be used only to solve exceptionally important problems. But take the same computing machine, reduce its price to a few dollars, and its size to that of your fingernail and you can afford to use it in home appliances and games.

That, in essence, is what happened to the computer. As we have already stated, today you can buy for $10 a computing device with essentially the same capabilities as the ENIAC I computer, which cost half a million dollars in 1950.

During the last 25 years there have been a whole series of new inventions which have dramatically reduced the cost of building computing circuits — and therefore computers. But surprisingly, there have been very few changes in fundamental computer concepts. Thus, with each new technological breakthrough, we have built the same computer as last time, but charged much less for it; and we have built a new, more powerful computer for the old price. This may be illustrated as follows:



As the years went by, the difference between the most powerful computer and the most expensive computer became more pronounced. In the illustration above, the lines marked ① , ② , ③ , ④ and ⑤ identify approximately equivalent computers, that is to say, computers with about the same performance. For example, look at line ② . What this line says is that there was a computer which sold in 1960 for $1,000,000 as the world's most powerful computer; an essentially equivalent computer was available in 1975 for approximately $1,000 — but it was no longer thought of as a powerful computer when compared to all of the new, much more powerful computers available in 1975.

MINICOMPUTERS

MAINFRAMES

With the huge differences in computer price and performance that appeared during the 60's it was inevitable that some product stratification would occur. Around 1965 the most inexpensive computers started to be called minicomputers. More powerful computers, by way of differentiation, were called mainframes. The prefix "mini" arose from the fact that the new, low cost computers were much smaller, physically, than their predecessors.

Now if you find ten people who are supposed to know, and you ask each to define the difference between a minicomputer and a mainframe computer, you will likely get ten totally different definitions. Price and packaging are really the only differences between mainframe computers and minicomputers. Minicomputers are much less expensive than mainframe computers and in general they are less powerful, although there is a very considerable overlap; the most powerful products sold as minicomputers are a good deal more powerful than the least powerful products sold as mainframes. Mainframes are packaged as business data processing or scientific data processing systems, while minicomputers are sold in a variety of ways, some of which are identical to mainframe computers while others are not. People who have worked for a long time with mainframe computers and minicomputers might claim to identify subtle differences between the two, but many of these differences are too subtle to recognize. A minicomputer is a minicomputer, and a mainframe is a mainframe, because that is what the manufacturer calls it.

MICROCOMPUTERS

To some extent history has repeated itself with the advent of the microcomputer. Around 1972 very low cost computer products began to appear and were called microcomputers. The prefix "micro" applied to the very small size of the product as compared to a mini, just as the prefix mini was based on the smaller size of the product as compared to a mainframe. But once again there is a substantial overlap between products referred to as microcomputers and products referred to as minicomputers. This overlap applies both to power and packaging. The most powerful microcomputers available today are more powerful than the least powerful minicomputers; while microcomputers are frequently displacing minicomputers, or being used in exactly the same way as minicomputers. In addition microcomputers are frequently used, because of their small size and very low price, in applications that could never have used a minicomputer. These new "microcomputer-only" applications have resulted in some real differences appearing in products sold as microcomputers, as compared to products sold as minicomputers.

It would be pointless to start describing differences between microcomputers and minicomputers in this book. Until you understand something about computers in general, there is no point trying to differentiate a microcomputer from a minicomputer. For the moment, therefore, this is all you need to understand: there is no fundamental difference between the largest mainframe computer and the smallest microcomputer; the major difference is in price, power and size.

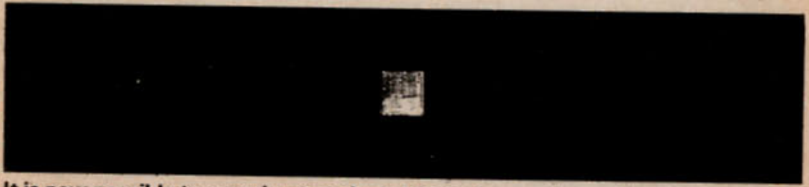
HOW THIS BOOK HAS BEEN PRINTED

This book covers a very broad range of information, therefore the text has been printed in boldface and lightface. The purpose of having two print faces is to let you pick your way through the book, bypassing information you understand and dwelling on information you do not understand. **Boldface text summarizes all major subject matter.** When you come across anything in the boldface that you do not understand, then read accompanying lightface for extra information.

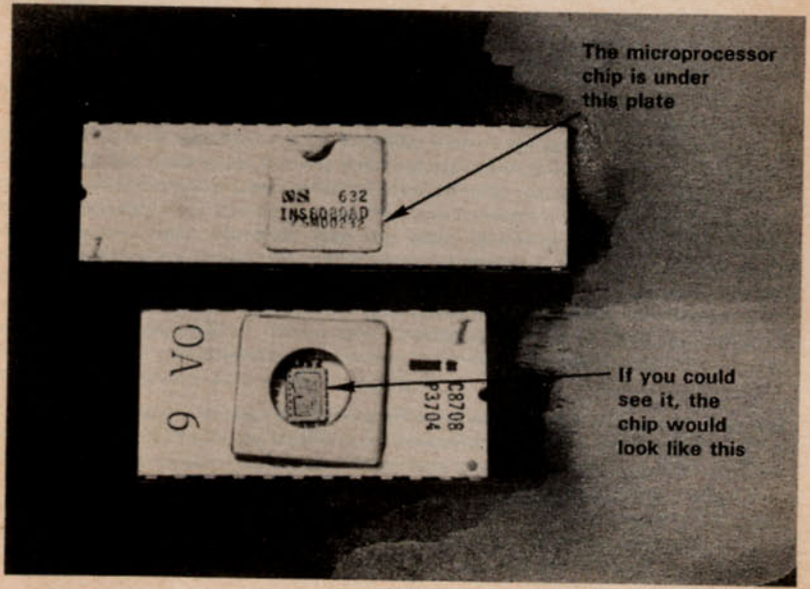
Chapter 1
THE PARTS THAT MAKE THE WHOLE

The age of the consumer computer industry is the result of a new technology which allows tens of thousands of microscopic electronic circuits to be crowded into a square space that may be less than 1/8 inch on each side. This new technology is referred to as Large Scale Integration (usually abbreviated to LSI). Here is an LSI device, reproduced in actual size:

LARGE SCALE INTEGRATION
LSI

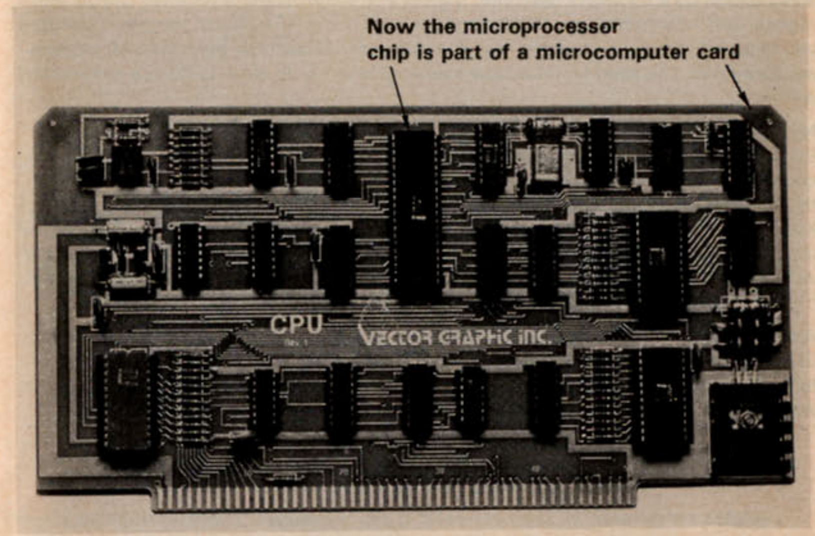


It is now possible to crowd onto a single LSI device, no bigger than the one illustrated above, all of the circuits needed to create the "brain" of a simple computer. We call this LSI device a microprocessor. The microprocessor is the "computer" within any microcomputer system. Here is an actual size microprocessor:


The microprocessor chip is under this plate

If you could see it, the chip would look like this

Now that you know what a microprocessor looks like you can forget about it. Not until much later in this book will we return to microprocessors. The "microprocessor" becomes a small part of a "microcomputer":


Now the microprocessor chip is part of a microcomputer card

A microcomputer, in turn, is a small part of a "microcomputer system".

When you first look at a "microcomputer system", it will look much like any other computer system. A typical microcomputer system is illustrated in Figure 1-1.

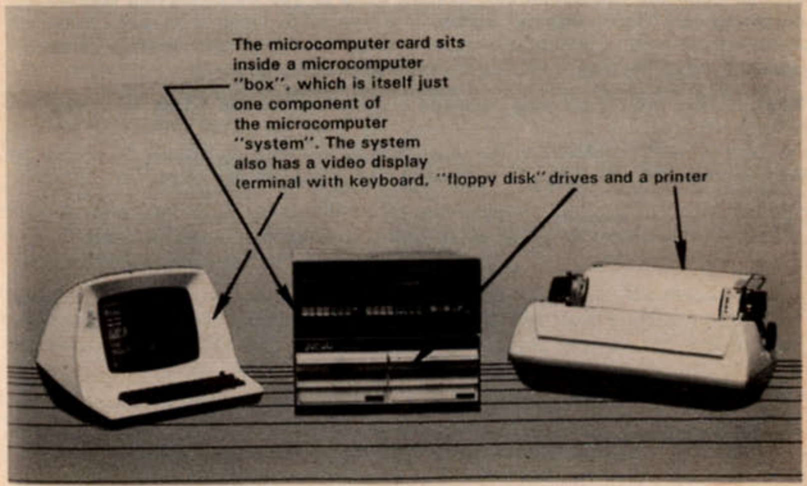

The microcomputer card sits inside a microcomputer "box", which is itself just one component of the microcomputer "system". The system also has a video display terminal with keyboard, "floppy disk" drives and a printer

Figure 1-1. A Typical Microcomputer System

A microcomputer system, as illustrated in Figure 1-1, is simply a means of getting a job done. At this most superficial level you need not even know that you are dealing with a computer — and in many cases that may be your preference. **If, for example, you are using the microcomputer system to handle your office payroll, then the thing becomes a business machine;** it gives you some means of entering payroll information, and hopefully it generates accurate paychecks and payroll records. Providing you receive the output you seek, you do not need to know how the computer works — any more than you need to know how a jet engine works in order to fly in a commercial airliner.

Now there is probably no reason why you should ever learn how a jet engine works. If the person who designed the jet engine were to sit beside you on your next flight, his knowledge would not get him to his destination any sooner than you, nor will it do anything to make his commercial air travel any more efficient than yours. But if you do not know how computers work, you could be sorry. Would you allow your bookkeeper to live in a world beyond your comprehension? Would you simply assume the bookkeeper will forever be honest? Of course not. So why assume your programmer will forever be honest — if he is never scrutinized, and works in a world only he understands?

And if you are not concerned with computers at work, then look at them in your daily life. For how long will you go on accepting the excuse, "it is a computer error"? You will have no option but to accept this excuse until someone teaches you how computers work — which is what we will now begin to do.

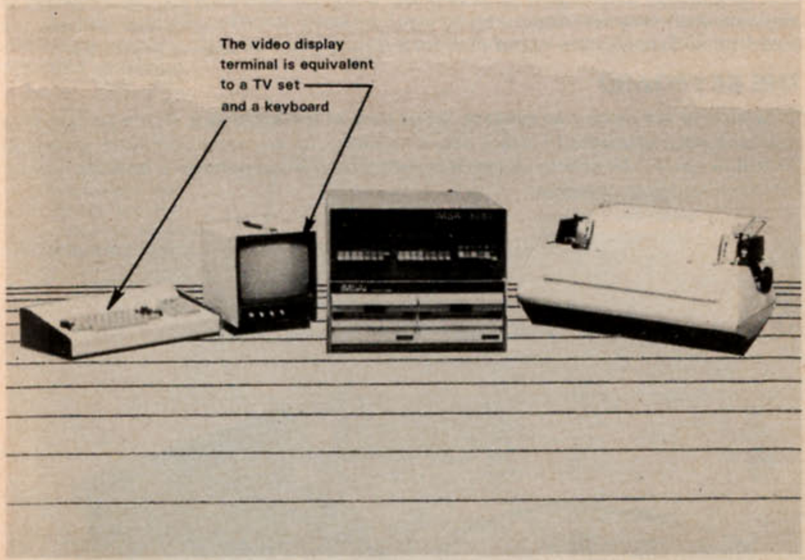
A MICROCOMPUTER SYSTEM

We are going to begin by looking at an entire microcomputer system, examining what each part of the system does.

THE VIDEO DISPLAY

The most noticeable parts of the microcomputer system illustrated in Figure 1-1 are the human interfaces. You hold a dialog with the microcomputer system in order to enter data; **the microcomputer system talks to you by displaying appropriate messages on a "video display" screen.** You respond by typing a reply at a keyboard. Anything you type in is displayed on the screen to reassure you that you made no mistakes — and the microcomputer correctly accepted your input.

There is very little difference between a video display and a television set. The principal difference is that the video display has better "resolution"; that is to say, you can have smaller characters at the video display, without the display becoming too fuzzy to read. But providing you are willing to put up with big letters, you can use a television set as a video display. This may be illustrated as follows:


The video display terminal is equivalent to a TV set and a keyboard

People on tight budgets frequently use a television set as their video display.

Now one of the problems associated with microcomputers and microcomputer systems is terminology and language. While a lot of terminology is pure jargon for its own sake, some terminology is necessary. Remember, English (and for that matter any other language) evolved in non-technological societies, with the result that we are often hard pressed to find words, or even phrases, that adequately represent technical concepts. In this book we will introduce you to computer terms, if only to make other books understandable.

For starters why are we talking about a "video display" rather than a television screen? The answer is that a television screen is designed primarily to display pictures. A video display is designed primarily to display printed words. And there are substantial differences between the electronics which are best suited to each case. A television screen is going to display pictures, but not much text; the television screen must therefore be able to display black, white and grey (or colors and depth of color), but it does not need high resolution for small characters and fine detail. A video display primarily displays text; it can get by with black and white but no grey (or colors, but no depth of color). However, a video display must have very high resolution in order to represent many small, well formed characters.

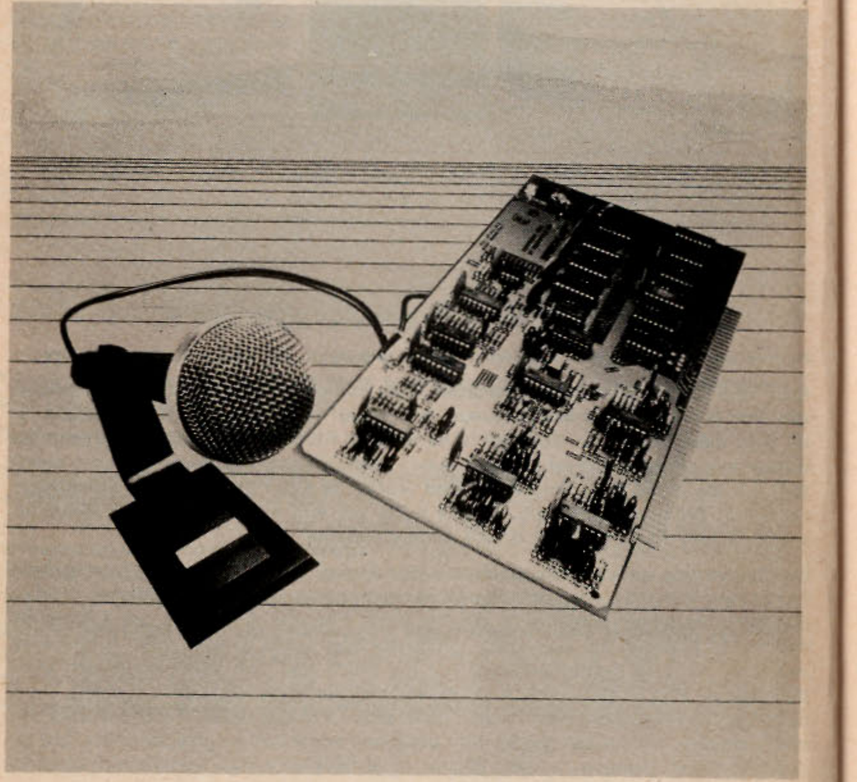
If television manufacturers built high quality television tubes, with the resolution demanded by video display terminals, then your television set could serve as a computer video display with no loss of display quality; but that would mean paying more for the television.

CRT
VDU

A video display is sometimes referred to as a Cathode Ray Tube (or CRT) or a video display unit (also known as a VDU). Calling a video display unit a VDU might seem logical enough, but calling it a "CRT" is somewhat meaningless to the average user. "CRT" refers to the way in which television and video display tubes are built today; "cathode rays" are used to create displays. In the not too distant future, cathode rays will probably be displaced by cheaper and more efficient technologies; but do not expect terminology to change with technology. Computer scientists are just as human as everyone else; they will likely refer to video displays as CRTs long after cathode ray based video displays have found their final resting place in science museums.

THE KEYBOARD

Returning to our computer system, let us look at the keyboard. At the present time you must use a keyboard to create new information for any small computer system. In the future you will be able to replace a keyboard with a microphone — and talk new information into your computer.

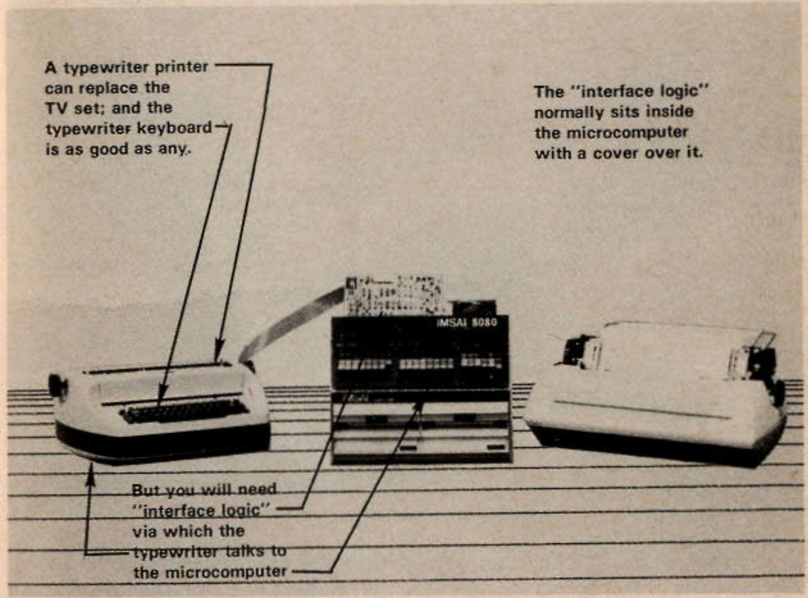


INTERFACE

There are innumerable keyboards on the market, and each bases its right to exist on special features or capabilities which you may find relevant or ridiculous. **You could replace the keyboard with a typewriter, providing someone builds an appropriate "interface" for the typewriter.**

And what does an interface consist of?

If you simply stand your typewriter next to your microcomputer and television set, hitting typewriter keys will have no effect on your television display. Someone must build the electronics that senses depressed keys and creates appropriate signals, causing the microcomputer and television to sense that a key has been depressed. This may be illustrated as follows:

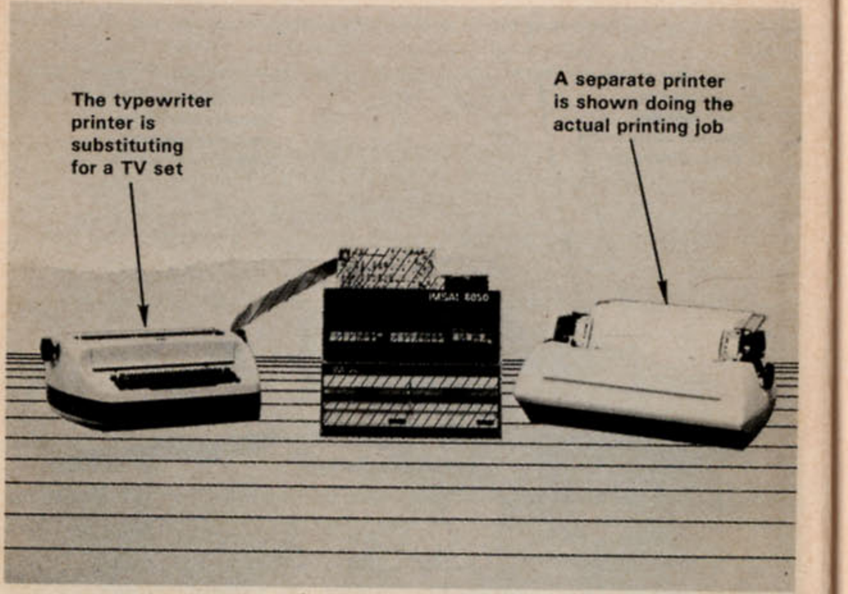

A typewriter printer can replace the TV set; and the typewriter keyboard is as good as any.
The "interface logic" normally sits inside the microcomputer with a cover over it.
IMSAI 8080
But you will need "interface logic" via which the typewriter talks to the microcomputer

As you might expect, the keyboard, the video display, in fact, every component of a microcomputer system must have its own electronics interface to the microcomputer.

Thus you can replace a keyboard and video display with a television set and typewriter.

THE PRINTER

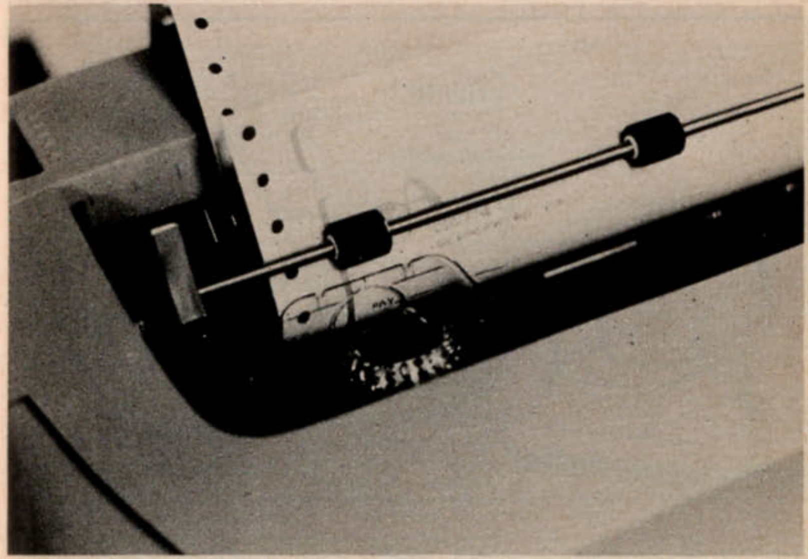
The typewriter offers something that a keyboard does not; the typewriter prints what you type. In our microcomputer system we show a printer performing this function:


The typewriter printer is substituting for a TV set
A separate printer is shown doing the actual printing job
IMSAI 8080

If we substitute a typewriter for the keyboard, we can eliminate the printer. So why do people buy printers? The answer is for print speed. Typewriters are clumsy mechanical devices that fall apart if you try to print more than 15 characters per second. The slowest printers are twice that fast, while a fast printer can print 600 lines a minute; a very fast printer will print thousands of lines per minute.

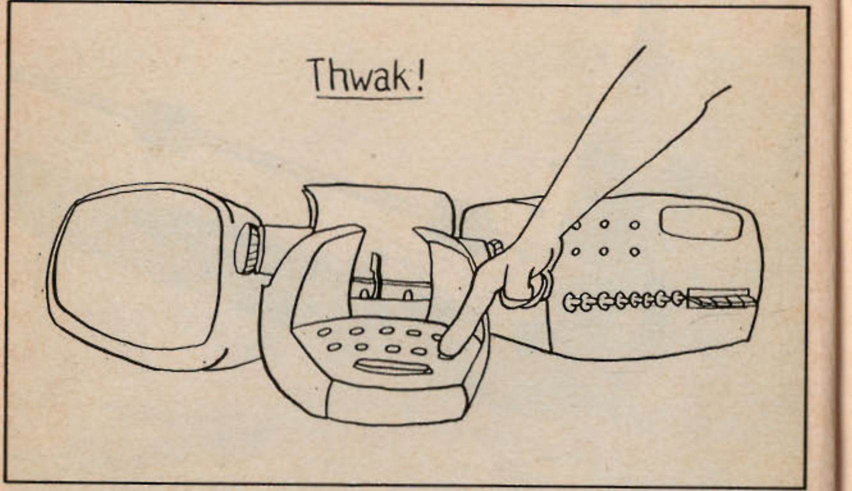
Is 15 characters per second not fast enough? Clearly nobody can type that fast; but remember, the printer is not simply going to print what you type. Consider a payroll program. The microcomputer is going to calculate pay data, then type out paychecks, unaided by human hands.

After entering all payroll data via the keyboard, you must feed a roll of paychecks into the typewriter and align the paycheck correctly:

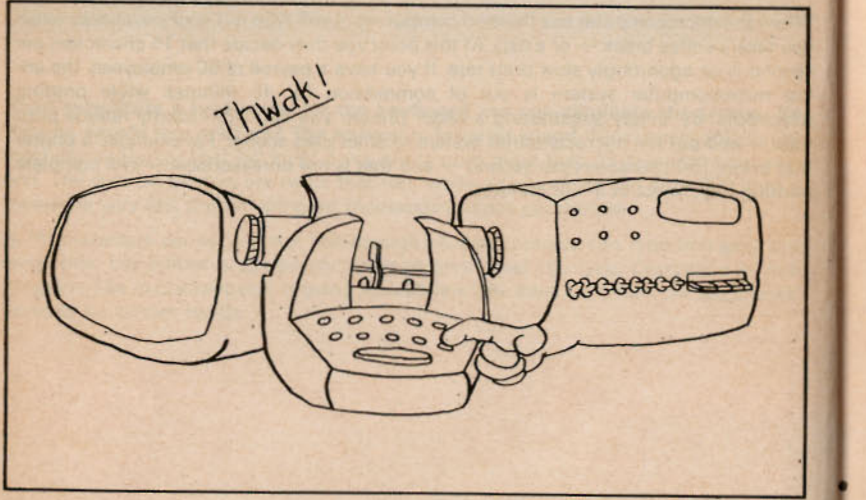


When the microcomputer has finished computing, it will type out your paychecks while you take a coffee break — or a nap. At this point you may decide that 15 characters per second is an agonizingly slow print rate. If you have a payroll of 50 employees, the entire microcomputer system is out of commission for 45 minutes while printing paychecks. By simply substituting a faster printer, you can significantly reduce print time — and put the microcomputer system to other uses sooner. For example, a printer that prints 150 characters per second — and that is not unreasonable — will complete printing paychecks in 4-1/2 minutes.

But print speed is not the only reason for separating the printer from the keyboard. When the two are linked, as they are in a typewriter, every time you hit a key you will print a character:

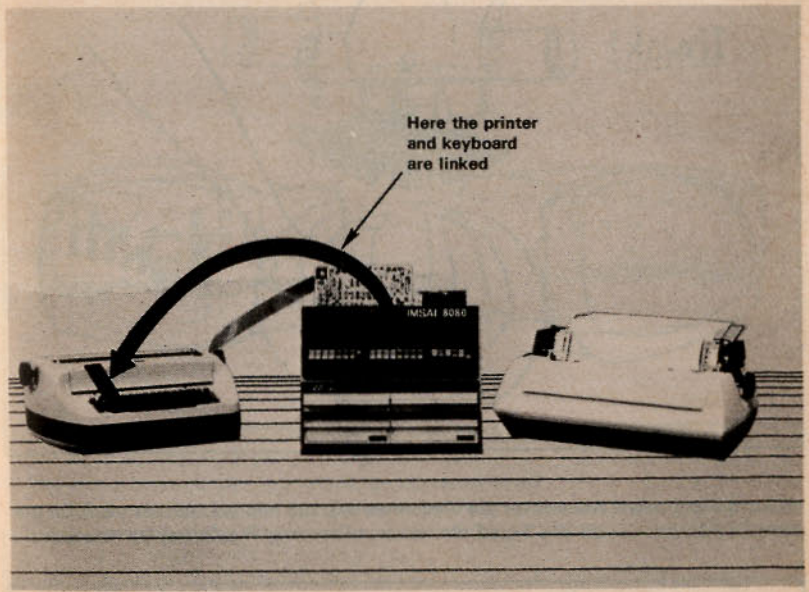


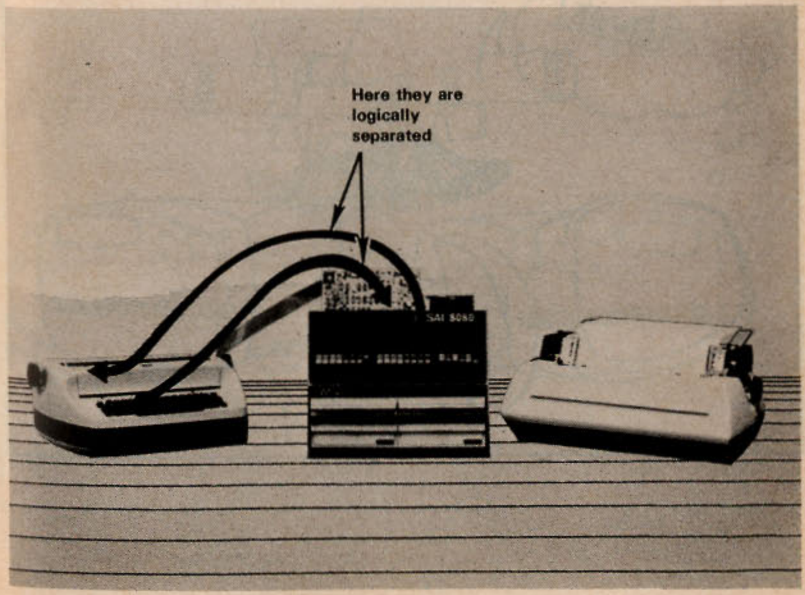
Every time the microcomputer prints a character, it will cause a key to be depressed.



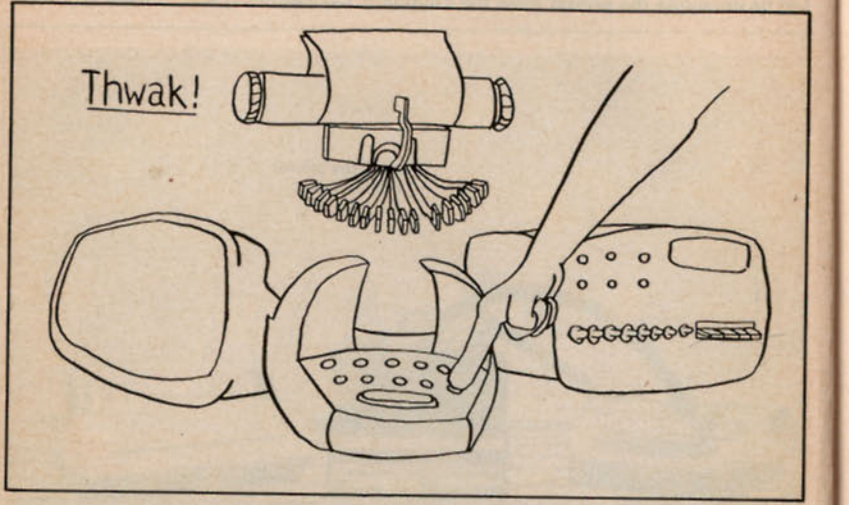
That means you cannot use the keyboard while you are printing; and conversely, you cannot print while you are using the keyboard.

Let us uncouple the printer from the keyboard. Conceptually this is what happens:

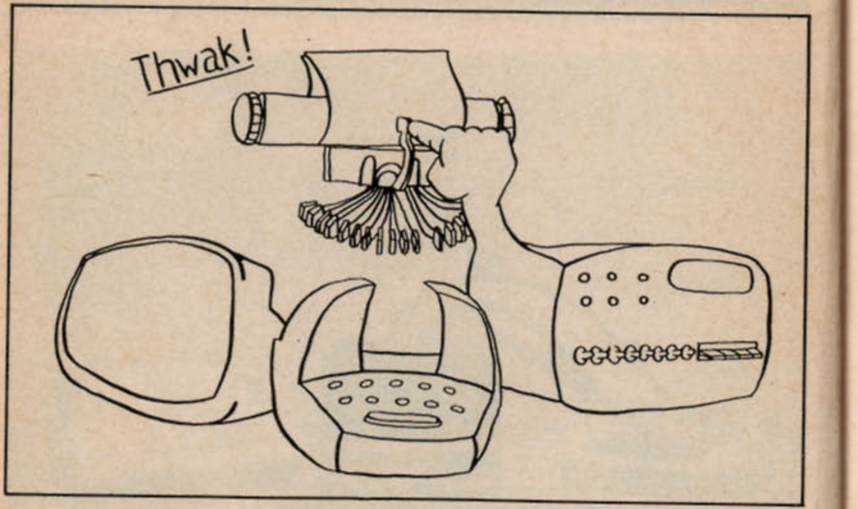

Here the printer and keyboard are linked


Here they are logically separated

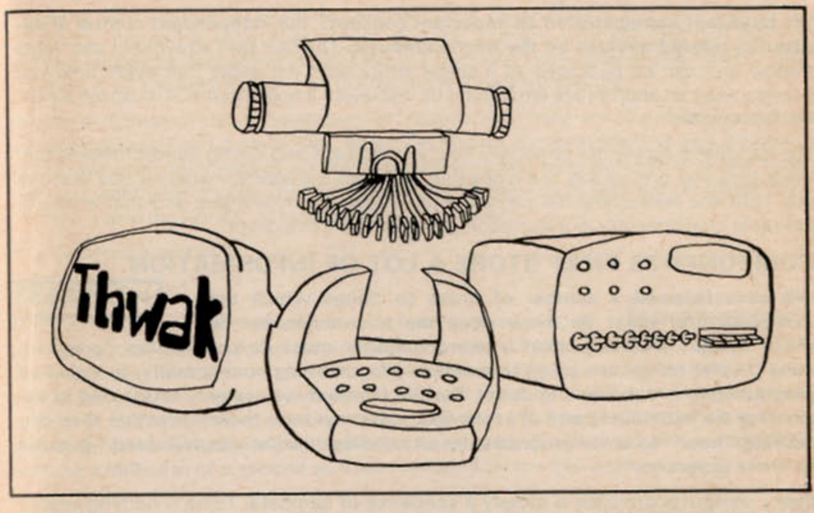
When the keyboard and printer are mechanically coupled, depressing a key automatically prints a character:



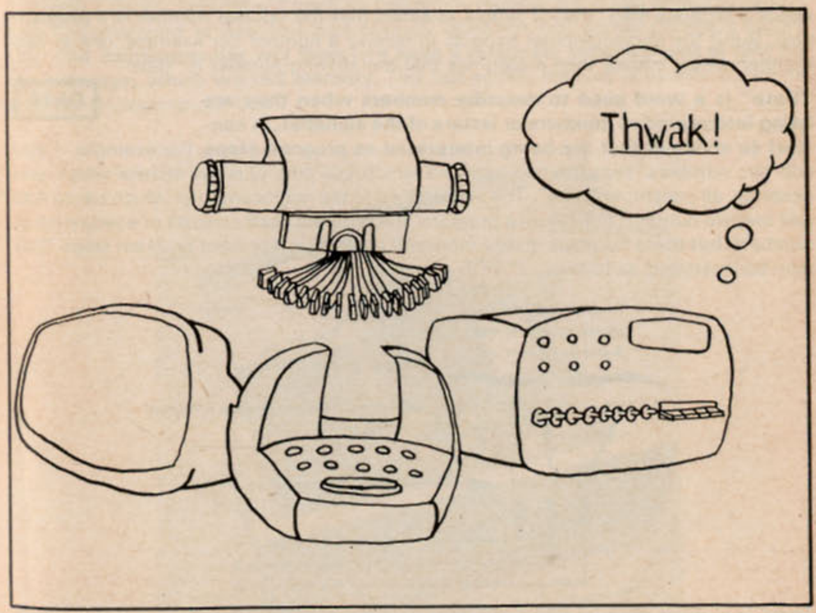
When the keyboard and printer are disconnected, the microcomputer must be programmed to print back the typed character, otherwise it will not be printed:



Alternatively the microcomputer can return the character to the video display:



Or the microcomputer may not return the character at all:



If the microcomputer returns a character, either to the printer or to the display, it is said to be "echoing" the character.

ECHO

We have just demonstrated an important concept: the independent control of apparently related devices by the microcomputer. The fact that a keyboard and video display unit can be packaged as a single entity does not imply that every time you depress a key an appropriate letter must be displayed. If echo occurs, it is caused by the microcomputer.

The fact that a typewriter physically links its keyboard and typing element means that every time you press a key the typewriter will print a character, whether you like it or not. Thus the microcomputer cannot control echo at a typewriter, and that is an undesirable characteristic in a microcomputer system component.

COMPONENTS THAT STORE A LOT OF INFORMATION

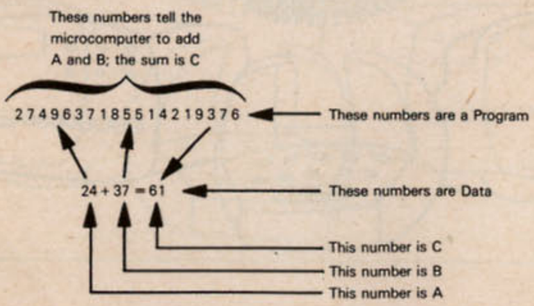
PROGRAMS

We have referred a number of times to things which the microcomputer must do. How does the microcomputer do these things? Defining what a microcomputer must do constitutes "programming". Later on we are going to examine programming conceptually; in Volume I programming is described in detail. For the moment we are only interested in examining the individual parts of a microcomputer system; therefore rather than discussing "how" to write programs, let us consider "what you will need" in order to write programs.

Every computer program is simply a sequence of numbers. There is nothing special about these numbers, they are just numbers, pure and simple; thus **a number stored inside a microcomputer may represent a program step, or it may represent "data".** It is all a question of interpretation. If the microcomputer fetches a number when it needs a program step, then it simply assumes that the fetched number is a program step. But if the microcomputer expects to receive a number (for example, one of two numbers being added) then it assumes that the arriving number is "data".

DATA

"Data" is a word used to describe numbers when they are being interpreted as numbers or letters of the alphabet, in contrast to numbers that are being interpreted as program steps. For example, if you add two numbers, then the two numbers which you add, plus the answer which you generate, all constitute "data". The instructions to the microcomputer which cause it to add the two numbers constitute a program; the program itself consists of a sequence of numbers, but these numbers do not represent data, they represent program steps. This may be illustrated as follows:



There is nothing unusual about numbers within a microcomputer representing either program steps, or data; we all do something similar every day. A number on a piece of paper might be part of a social security number; it could be a bank check number, the dollar amount of a check, or the dollar amount of a bill. It is only by inspection and interpretation that you can tell which is which. If, while reconciling your bank statement, you accidentally read your bank statement number rather than the check amount, you will get a very weird (and obviously incorrect) bank reconciliation; but there is nothing inherently impossible about making this mistake. Similarly, you will find that there is nothing inherently impossible about having a microcomputer read a data number and interpret it as a program step; but the results will be very strange.

Numbers represent just a small part of your world. They are the entire world of microcomputers. Even a small task, when defined for a microcomputer as a program, will create many hundreds of numbers. By the time you have defined all of the tasks that you want your microcomputer to perform, all these task defining programs may become many thousands, or even millions of numbers.

This presents a problem.

MEMORY

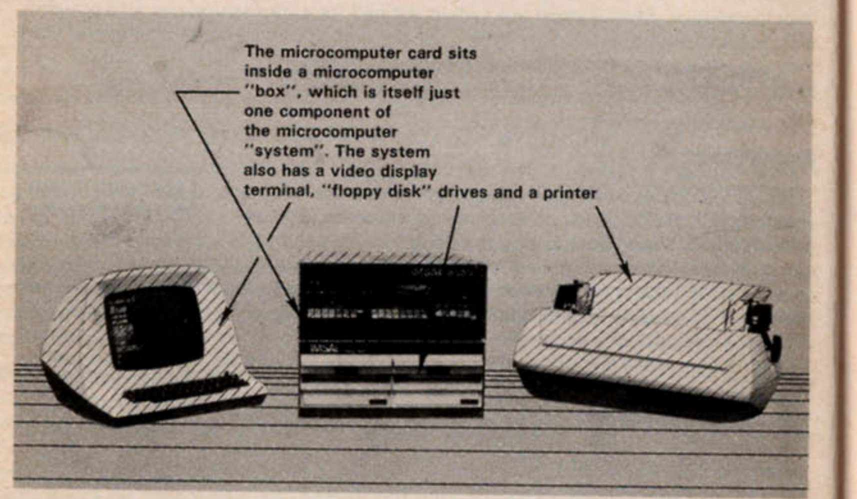
MICROSECOND

The microcomputer performs any specified task by executing a specific program. The program consists of a sequence of steps, or instructions; each instruction is identified by a unique number. But a microcomputer can go through hundreds of thousands of program steps, or instructions, within a single second. In fact, **the typical microcomputer will execute a single instruction in some time interval ranging from 1 millionth of a second to 10 millionths of a second. (We refer to a millionth of a second as a microsecond.)** If the microcomputer is to execute an instruction every few microseconds, then clearly the number representing the instruction must be stored in some type of ready reference, fast, access storage. The microcomputer must be able to fetch the number representing the instruction in even less time than the few microseconds available to execute the entire instruction. Typically a microcomputer will fetch an instruction number out of fast access storage in half a microsecond, or less. This type of ready reference, fast access storage is expensive. **The microcomputer will therefore have a relatively small amount of fast access storage, which we call memory.** This fast access memory is usually buried inside the microcomputer box:

Program steps and data which the microcomputer is currently using must be stored in this fast access memory.

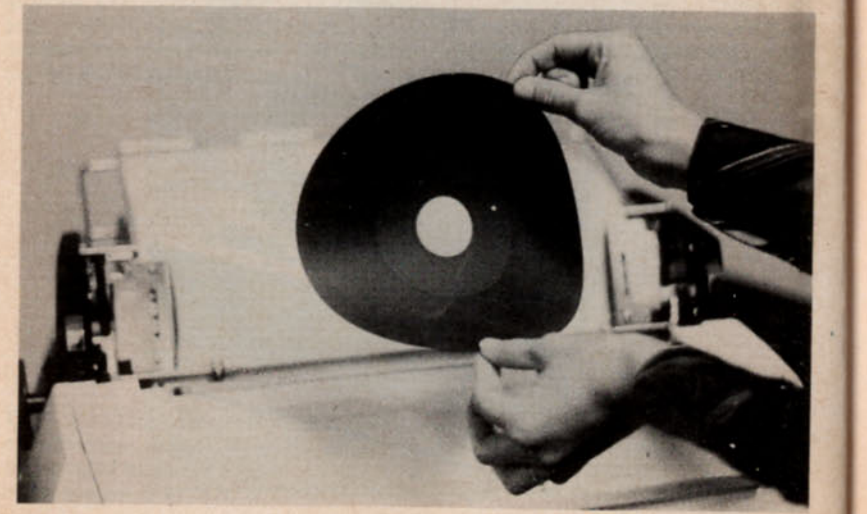
FLOPPY DISK UNITS

Programs and data which the microcomputer is not currently using are stored using some form of slower (and cheaper) bulk storage device. In Figure 1-2 this bulk storage device is shown as a "floppy disk" system:



The microcomputer card sits inside a microcomputer "box", which is itself just one component of the microcomputer "system". The system also has a video display terminal, "floppy disk" drives and a printer

There are conceptual similarities between a floppy disk system and a record player. The floppy disk system stores its information on floppy disks, which are so named because they are soft and bend easily:



Because the disks are floppy, they are housed inside stiff cardboard envelopes to keep them rigid:

MINI FLOPPY

There are two sizes of floppy disk: regular ones which are 8" in diameter, and "mini floppies" which are 5-1/4" in diameter.

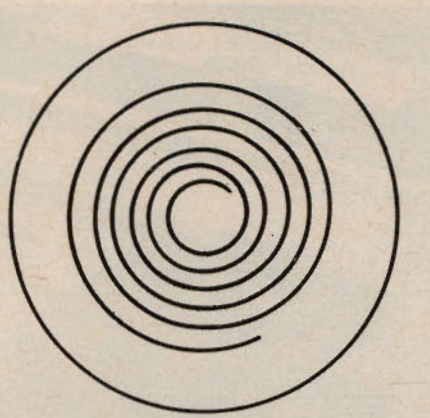
Whereas a record has a grooved surface, **the floppy disk has a smooth magnetic surface. On this smooth magnetic surface information is stored as sequences of magnetic pulses.**

TRACKS

The magnetic pulses are recorded along "tracks" on the surface of the floppy disk. In contrast, music is stored on the surface of a record within a continuous groove. A needle rides the groove in order to position itself. But the floppy disk surface is smooth and has no grooves. The track is instead an imaginary line along which the magnetic pulses lie. Information is written onto the track, and read off it, by a magnetic read/write head that is quite similar to the pick up arm of a record player. Of course, the read/write head of the floppy disk unit has no needle since there is no groove to track; instead it has little metallic pads that can create (to write) or sense (to read) magnetic pulses on the floppy disk surface.

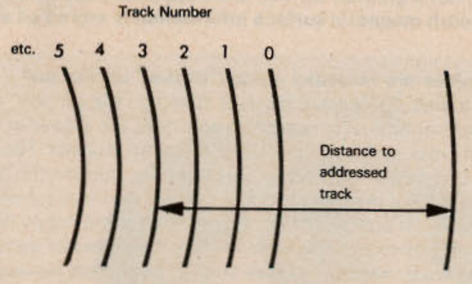
Now until you become a real microcomputer expert, you need not concern yourself with exactly how information is stored on a floppy disk. The discussion which follows will give you a general understanding of concepts, and that is all. Most microcomputer users never bother with this information, just as most music lovers never concern themselves with how music is recorded on tape, cassettes or records. You know that a record, when played, creates music; similarly a floppy disk, when played, creates numbers. You can also write numbers onto a floppy disk, just as you can record music on a magnetic tape or cassette.

It would be feasible to record information on the surface of a floppy disk, much as music is recorded on the surface of a record, by simply having one continuous groove (or, in this case, one continuous track) spiraling out from the center of the floppy disk surface:



The problem with this recording scheme is that it is very hard to pick your way around the recorded information. You can prove this for yourself by trying to find a particular word in a song on a record. If you struggle long enough, you will probably accomplish the task; but you will do so only by using human judgement. Electronics cannot use human judgement; it can only follow set rules. We therefore replace the single continuous spiraling track with a large number of concentric tracks.

We can now select one particular track by its track number — which can be addressed in terms of the exact distance of the track from the edge or center of the floppy disk:



Different manufacturers use different numbers of tracks on each floppy disk surface. The only standard that exists is for 8" floppy disks which frequently have 77 concentric tracks (numbered 0 through 76) recorded on one surface. Some floppy disks store information on one side of the disk only, while others store information on both sides.

You can store approximately 250,000 characters of information on one surface of an 8" floppy disk. But if you were to simply divide up this information on the basis of track number, you would have some problems. First of all, tracks increase in length as they move from the center of the surface towards the circumference — which means that no two tracks would store the same amount of information. Second of all, the amount of information stored on one track could be quite large; yet it would represent the smallest

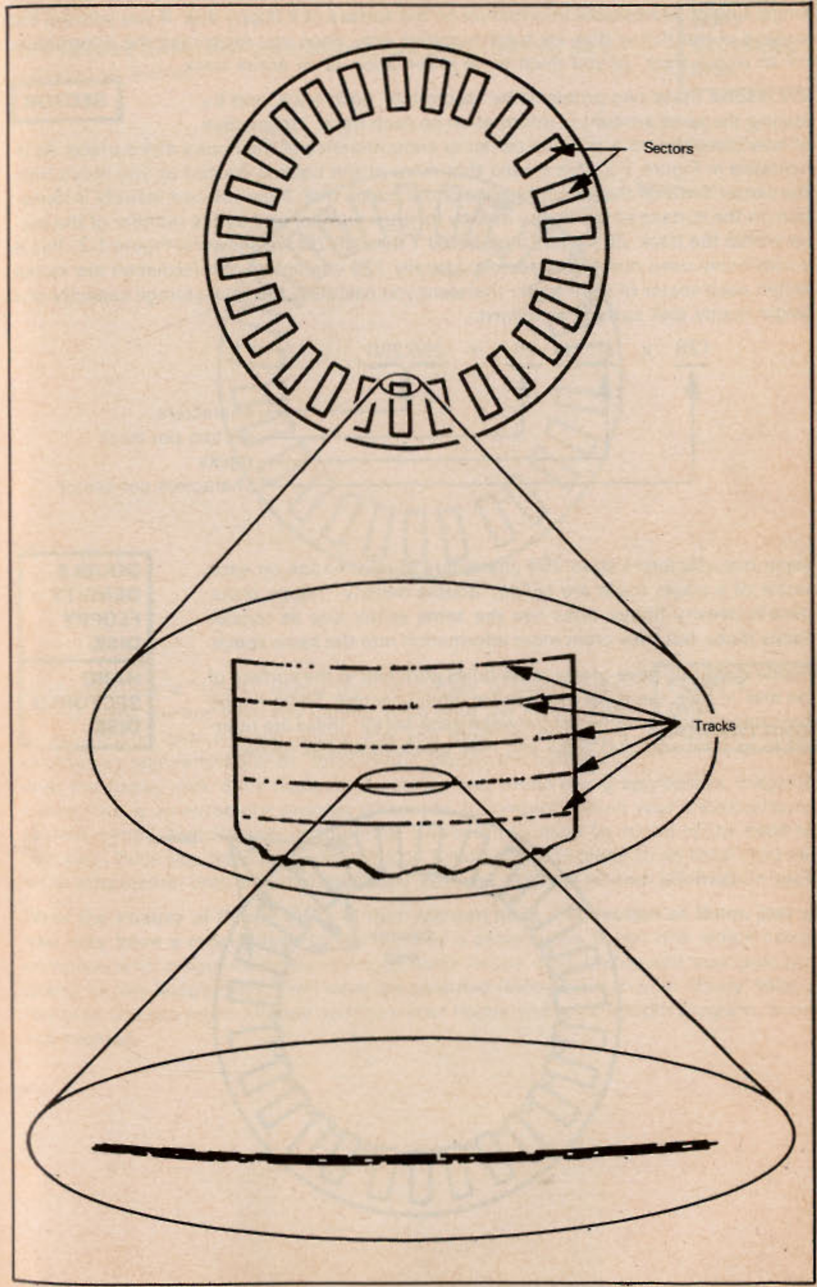


Figure 1-2. A Floppy Disk's Recorded Surface

single unit of addressable information on the surface of a floppy disk. If you address the surface of the floppy disk via track numbers only, then you must read the information off an entire track, or you must write information to an entire track.

SECTOR

We resolve these two problems by "sectoring" each track, and by storing the same amount of information on each track, irrespective of how close the track is to the center or circumference of the floppy disk surface. As illustrated in Figure 1-2, this means that more of the track is wasted as you move from the center towards the circumference of the floppy disk. Now you can identify information on the surface of the floppy disk by its track number and by the number of the sector within the track. 26 sectors (numbered 1 through 26) are shown in Figure 1-2; this is a commonly used number of sectors. Usually 128 characters of information are stored within each sector of each track; therefore you calculate the total storage capacity of a single floppy disk surface as follows:

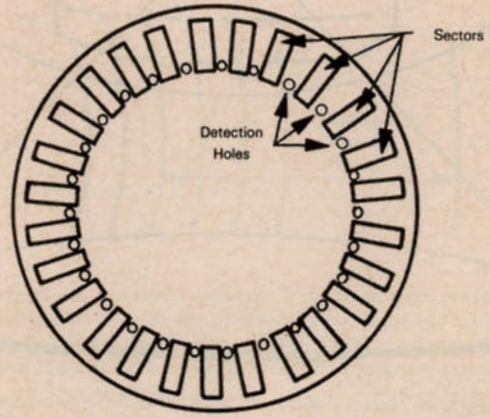
$$\underbrace{128}_{\text{characters per sector}} \times \underbrace{77}_{\text{tracks}} \times \underbrace{26}_{\text{sectors per track}} = \underbrace{256{,}256}_{\text{characters}}$$

DOUBLE DENSITY FLOPPY DISK

Some manufacturers store 256 characters of information on each sector of a track; these are called "double density" floppy disks. Double density floppy disks use the same sector size as regular floppy disks, but they cram more information into the same space.

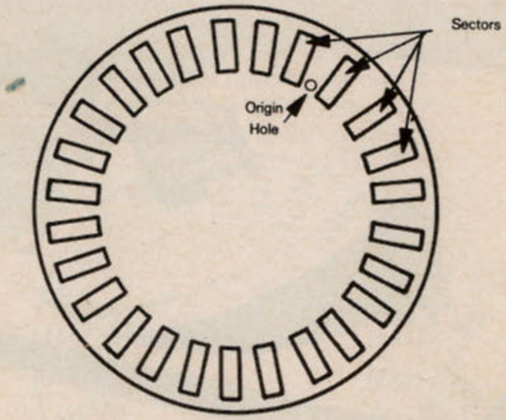
HARD SECTORED DISK

Floppy disks will have one or more holes punched in the surface of the disk to help the drive mechanism detect sectors. Some floppy disks have a hole punched in between each sector; these are referred to as hard sectored disks and may be illustrated as follows:



SOFT SECTORED DISK

Soft sectored disks have one punched hole only. This may be illustrated as follows:



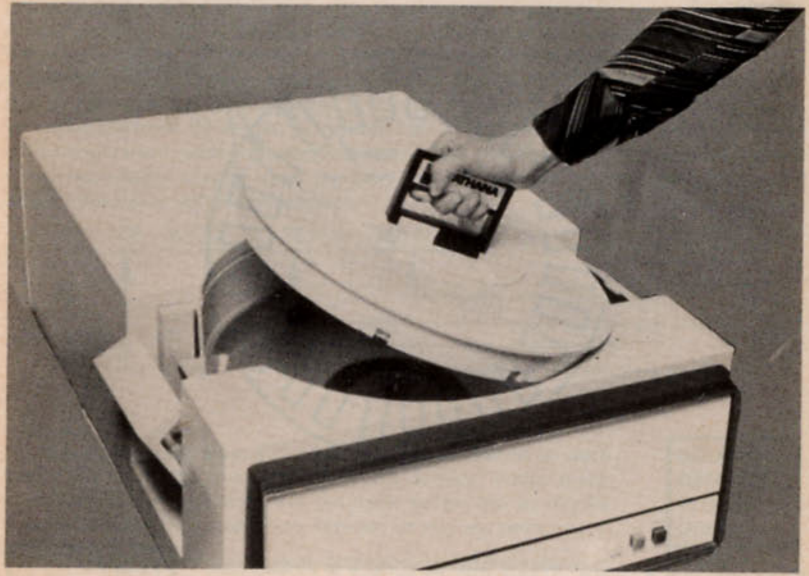
FLOPPY DISK FORMATTING

When you buy a "blank" floppy disk its surface is indeed completely blank; in this form you cannot write onto the floppy disk. Before writing onto a blank floppy disk you must go through a preliminary step referred to as "formatting". During the formatting step the floppy disk drive designates sectors and tracks using appropriate magnetic codes. This is an automatic process which you accomplish, using your microcomputer system, simply by following instructions; however you must be aware of the need for this step. Once you have formatted a blank floppy disk it is ready to be used. You can write information onto a formatted floppy disk and read the written information back.

Now the beauty of floppy disks is that you can have a collection of them, just as you may have a record library. You can buy a prerecorded floppy disk which comes complete with programs, or you can buy blank floppy disks and record your own programs on the floppy disks, then save the recorded floppy disks in your library. Now at any time you can fetch a floppy disk from your library and run a bought program, or one you created.

RIGID DISK UNITS

Floppy disks are not the only means of storing programs and data. Big computer systems use large rigid disks:



Large rigid disk systems offer a tremendous variety of options and sizes; but they all store information on a much bigger disk, which is not floppy:

The "rigid" disk is equivalent to the "floppy" disk of smaller systems. Data is stored on the rigid disk.

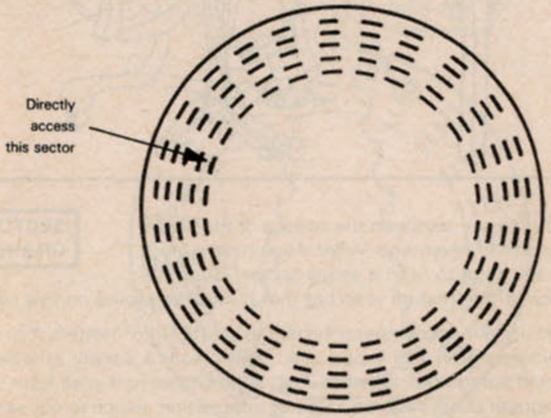
Rigid disk units store a great deal more information than floppy disk units — but they are more expensive.

In reality, rigid disks are cheaper if you look at their cost in terms of price per unit of information stored. But with a small microcomputer system you simply do not need the storage capacity of a large rigid disk unit; moreover, many large rigid disk units transfer information faster than a microcomputer can handle it.

DISK ACCESS

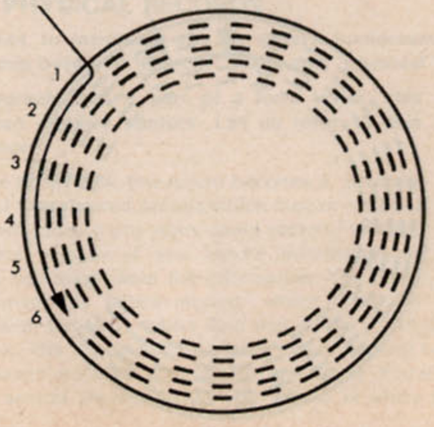
RANDOM ACCESS

Floppy disk units and fixed disk units are both referred to as "random access" bulk storage devices. The name "random access" means that you can go directly to any sector of any track in order to read or write information:

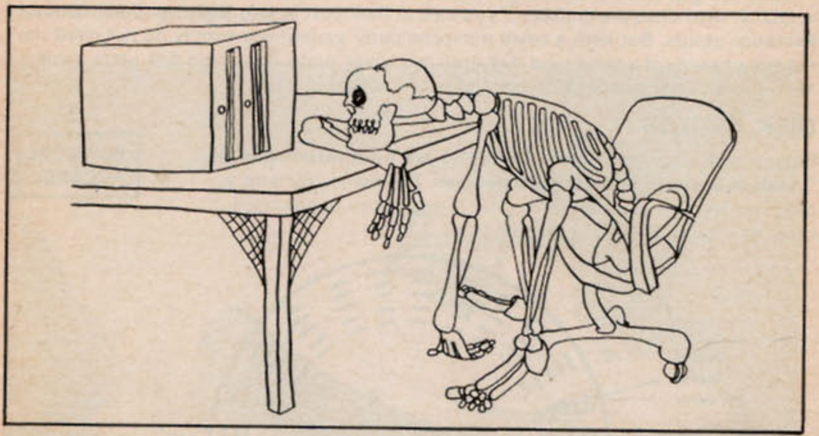


Suppose, for example, you wish to read the contents of track 12, sector 8; the floppy disk read head can go directly to this sector without first accessing tracks 1 through 11, and sectors 1 through 7 of track 12.

Random access is an extremely useful capability in any bulk storage device. The ability to directly access any sector in order to read or write will speed up data access operations. Suppose you had to read sectors sequentially; you would not have much trouble with the first few sectors:



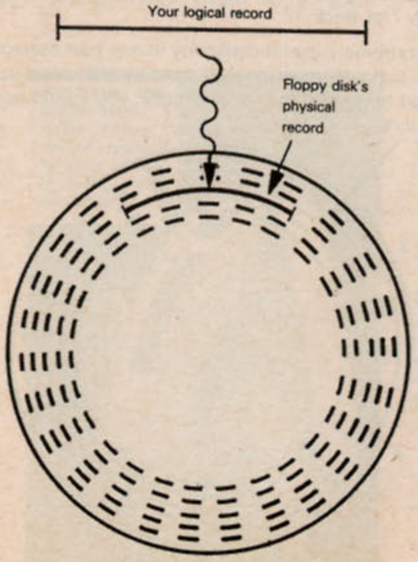
But it would take a painfully long time to reach one of the last sectors on the surface of the floppy disk.



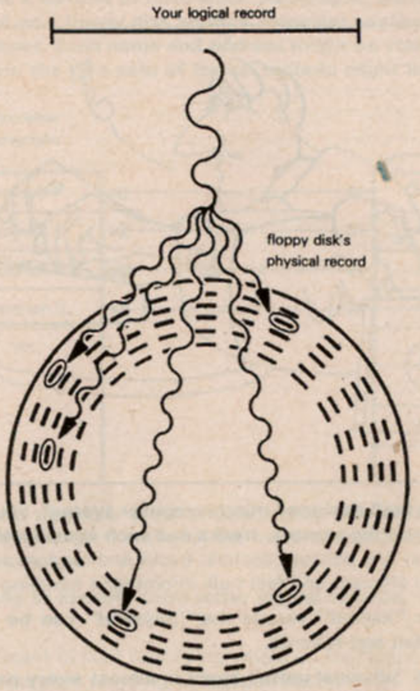
SECTOR CHAINING

Being able to go directly to any sector on the surface of the floppy disk has another less obvious advantage. What if you have a block of information that is too large to fit in a single sector? Suppose, for example, your block of information is so big that it must be stored on five sectors.

For example, you could use a microcomputer system to create "form" letters. You could store 50 standard paragraphs on the floppy disk, then create a variety of letters by simply stringing selected paragraphs together. That is how most junk mail form letters are created. Each paragraph could become a unit of information, which in our example is stored on five floppy disk sectors. Now your immediate reaction may be to use five contiguous sectors:



But since you can access sectors randomly, it makes no difference whether the five sectors are contiguous, as illustrated above, or whether they are scattered randomly across the surface of the floppy disk:



CHAINED SECTORS

When you store a single unit of information on more than one sector, the sectors are said to be "chained".

LOGICAL AND PHYSICAL RECORDS

This is a good point to introduce you to a very fundamental computer concept: the relationship between "logical" ideas and "physical" reality.

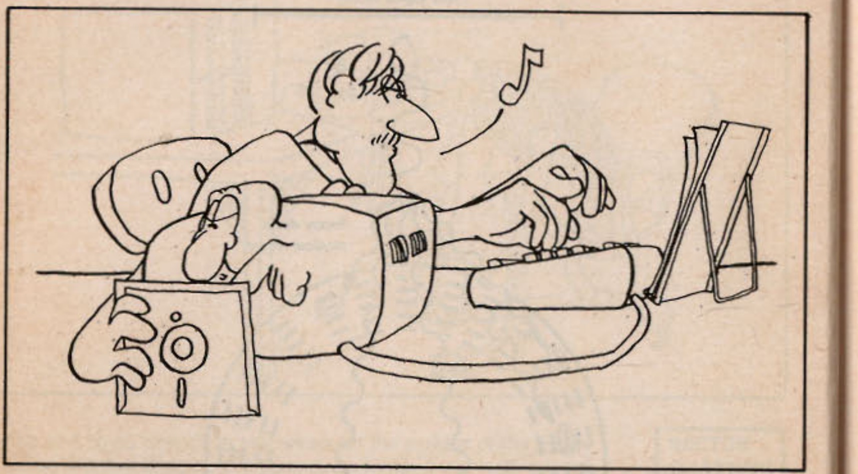
RECORD

Consider one paragraph that is part of a form letter; this paragraph is written on five sectors. Let us refer to this paragraph as a "record".

PHYSICAL RECORD

LOGICAL RECORD

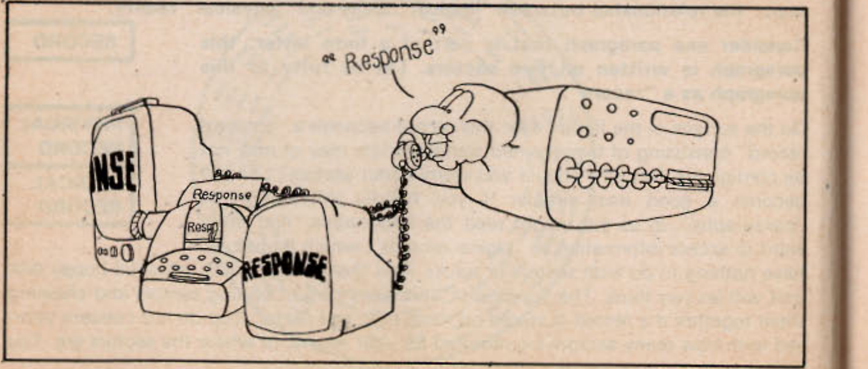
On the surface of the floppy disk, this record becomes a "physical record" consisting of five chained sectors which may or may not be contiguous. But why should you worry about sectors? Life will become a good deal simpler if you handle information as "paragraphs", or as you might read the information. You really want to access information as "logical records", which (hopefully) have nothing to do with sectors or tracks. And that is how a well designed floppy disk unit will let you think. The floppy disk unit takes care of finding sectors and chaining them together if a record is stored on more than one sector. You do not concern yourself with how many sectors are required for your record, or where the sectors are. You do not even have to know that tracks and sectors exist. **You deal with the "logical idea" of a paragraph and the microcomputer takes care of the "physical reality" of sectors and tracks.**



Thus, when using a well designed microcomputer system, you can think in terms of logical records, ignoring sectors, tracks and such complications. But you should always remember that it is because sectors, tracks and random access exists that the floppy disk unit is an efficient and fast bulk information handling device.

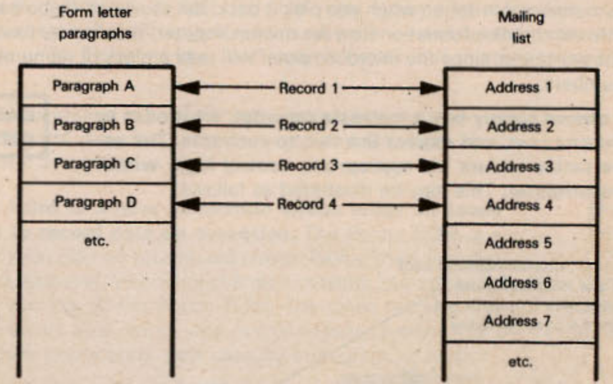
The concept of the "logical" versus the "physical" can be extended to cover more than information and records.

"Logical units" and "physical units" apply in almost every part of any computer system. Generally stated a "logical unit" is a piece of information, an idea, or an operation, as you, a human being, will use it. A "physical unit" is the actual physical implementation — the physical reality behind the idea, information or function. For example, in our earlier discussion we saw how a video display could be replaced either by a television set or by a typewriter printer. In this case all three — the video display, the television set and the typewriter printer — are the reality, physical units representing the idea of a computer response logical unit.



RECORDS AND FILES

Suppose you have a number of form letter paragraphs, each of which is recorded as a logical record on a floppy disk surface. Consider another alternative: a list of names and addresses. Each name and address might be stored as a single, logical record. This is how the two sets of logical records might be compared:



In order to identify a particular piece of information, you must now know the logical record number and whether it is one of the form-letter logical records, or one of the mailing list logical records.

How are you going to identify form letter logical records, as against mailing list logical records?

Clearly we do not want to start specifying where on the floppy disk the information is stored; the whole purpose of going to logical records in the first place was to avoid worrying about the floppy disk surface. We therefore combine all of the records into a file. A file is simply a collection of one or more records. Once again you deal with logical files and leave the microcomputer the task of determining where the physical file actually exists on the floppy disk surface. Now **you have a "form letter" logical file, where each paragraph is a logical record, and you have a "mailing list" logical file, where each name and address becomes a logical record.**

Files and records represent the fundamental structure used to store bulk information in computer systems — from the smallest microcomputer system to the largest mainframe computer system.

This concept of logical files and logical records is not so very different from daily office life. For example, suppose you need to look at a letter you received from XYZ Corp., containing price quotes. You could ask a secretary to fetch the 27th letter in the third filing cabinet drawer; more likely you would ask your secretary to go to the "XYZ Corp." file and retrieve the "price quotes" letter. The "XYZ Corp." file is the equivalent of a logical file; each letter in the file is the equivalent of a logical record. Your secretary is the equivalent of the microcomputer intelligence which is capable of accessing a specific item of information, given a description of what the information is.

CASSETTE UNITS

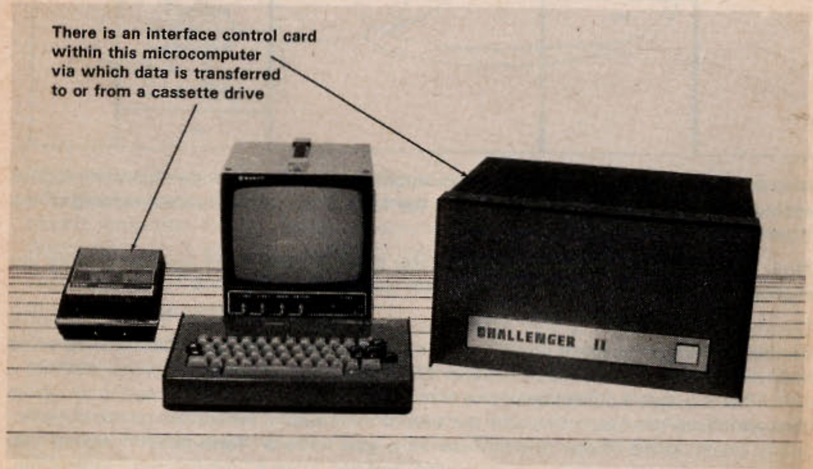
If you are a typical microcomputer user, you will find that (at least initially) you cannot afford a floppy disk system. What are the less expensive options? There are cassettes and there is paper tape.

The cassette units used to store information for a microcomputer are exactly the same as the cassette units that you use around the house. You can buy a cassette recorder at the supermarket and use it to store information for your microcomputer system.

Hopefully you will use a high quality cassette recorder and very high quality cassette tapes with your microcomputer system, since the microcomputer system is not going to tolerate errors. If you have a "glitch" in a cassette with music recorded on it, then you will have a moment of irritation when you play it back, but that is all. If you have a glitch in a cassette that holds information for your microcomputer, then it may render the entire cassette worthless, since the microcomputer will read a block of no numbers, or erroneous numbers.

CASSETTE INTERFACE

Now you cannot simply buy a cassette recorder, sit it next to your microcomputer, and expect the two to converse. The actual conversation occurs via appropriate control logic within your microcomputer. This may be illustrated as follows:


There is an interface control card within this microcomputer via which data is transferred to or from a cassette drive
CHALLENGER II

There are two very different ways in which cassette recorders store information on magnetic tape. The old way was to store information digitally, that is to say as a sequence of magnetized dots on the cassette tape surface:

Magnetic tape
1 0 1 1 1 0 0 1 1 0 0
Numeric interpretation

Nearly all industrial cassette recorders store information digitally as illustrated above. **The new cassette recorders that are being sold in computer stores** do not record information digitally; rather they **record information as sounds** — in exactly the same way as they would record voices or any other noise. One particular tone represents a 0 digit while another tone represents a 1 digit.

The fact that there is an "old" and a "new" way of recording data on cassettes does not mean that the "new" way is better, or that it has superseded the "old" way. "Old" and "new" refer to chronology. In fact, the "old" way stores a lot more data on cassette tape, it allows data to be written and read much faster — and it requires much more expensive tape units. Therefore the "old" way is better, but more expensive than the "new" way.

The principal advantage of using tones to record data is that you can use any standard household cassette recorder, either to record and play back music, or as part of your microcomputer system.

FLOPPY ROMS

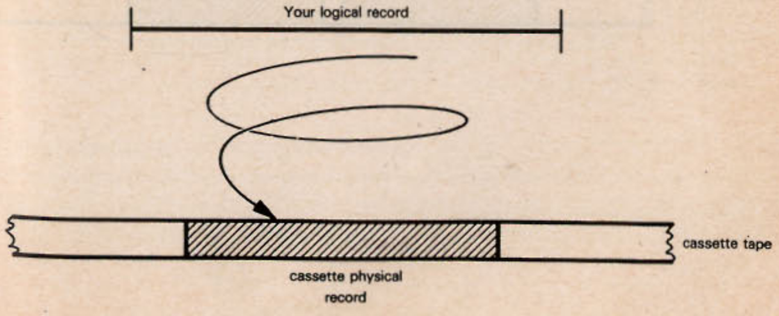
The floppy ROM is a new innovation which relies on tones being used to record data on cassettes. The floppy ROM is a record which you play on your record player. Rather than listening to the record, however, you connect a tape cassette to your record player and tape the information coming off the floppy ROM. The tones recorded on the cassette are interpreted as binary data within your microcomputer system. The binary data may constitute program instructions, data used by a program, or both.

ROM are the initials for Read-Only Memory. The record contains information, therefore it is part of the microcomputer's "memory". You can read from the record, but you cannot write to it; therefore it is a "read-only" component of the microcomputer's memory.

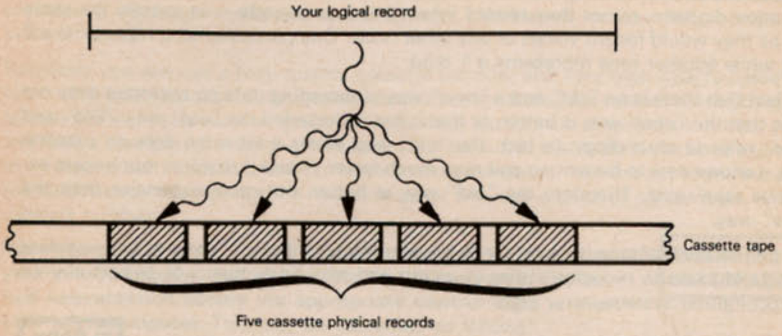
The floppy ROM was first introduced commercially by Interface Age Magazine which uses the floppy ROM as a means of giving its readers computer readable programs in a magazine. Previously programs had to be printed in a programming language which left the reader the task of entering the program into his microcomputer — which is a laborious and error prone operation.

CASSETTE RECORDS

Information is stored on cassette tape as a sequence of physical records. There is no standard cassette formatting scheme such as the sectors and tracks we described for a floppy disk. The physical records on a cassette tape may all have the same length or they may have various lengths; also, there are no restrictions placed on physical record length. A physical record may be one character long, or it may be as long as the physical length of the cassette tape will allow. The logical record we described when discussing floppy disk sectors and tracks can be recorded on cassette tape as a single physical record:



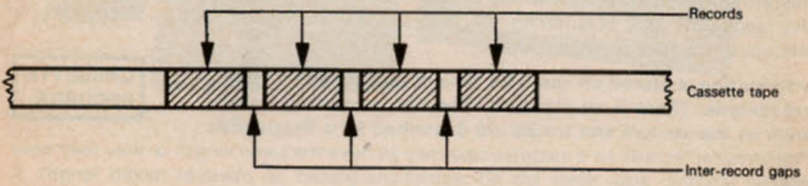
Or it may be spread over a number of physical records:



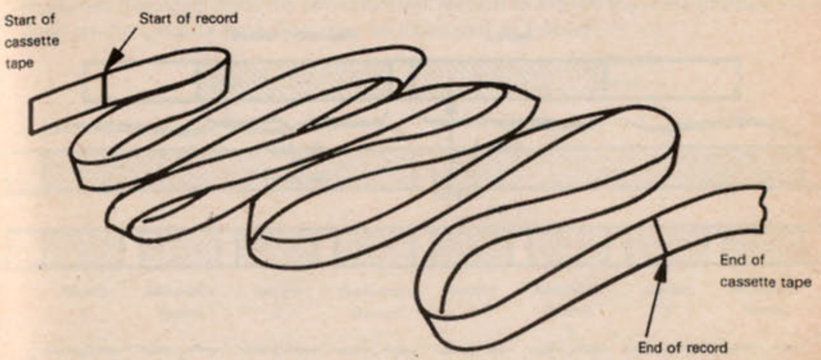
The most important difference between a cassette unit and a floppy disk unit is that the cassette unit is a sequential (or serial) access device, whereas the floppy disk unit is a random access device. When we say that a cassette unit is a sequential device, we mean that you cannot jump around on the surface of the cassette the way you can on the surface of a floppy disk. If you wish to access the 25th record on a cassette tape, then you must first count your way past the first 24 records. Thus information stored far down a cassette may take a long time to reach. For example, it could take 10 or 15 minutes to reach the last record on a cassette.

CASSETTE INTER-RECORD GAP

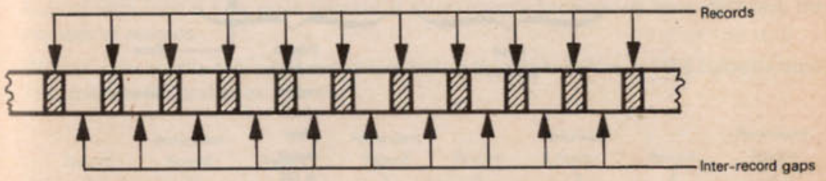
If you have more than one record on a cassette tape, then there must be inter-record gaps separating individual records. This may be illustrated as follows:



There is no useful information stored in an inter-record gap. As we increase the number of records on a cassette tape, therefore, we increase the number of inter-record gaps and we decrease the total amount of useful information which we can store on the cassette tape. **To get the most information on a cassette tape,** therefore, **we would store that information as one continuous record** with no inter-record gaps:



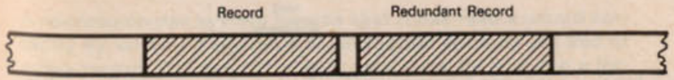
At the other extreme, if you have numerous very short records, you will waste most of the cassette tape on inter-record gaps:



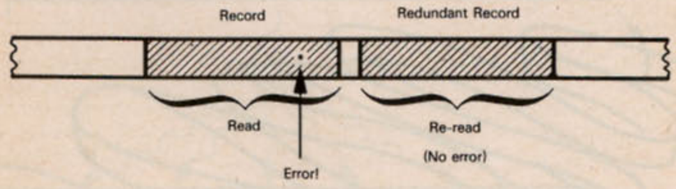
CASSETTE TAPE RELIABILITY

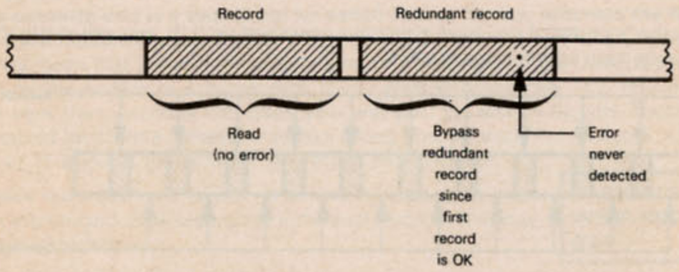
REDUNDANT RECORDING

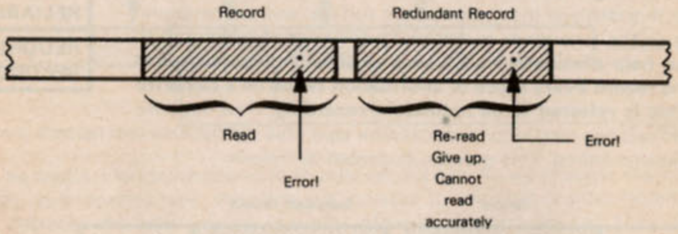
So why does anyone bother storing lots of short records? The answer is to fight errors. Cassettes are not very reliable; it is easy to scratch or damage the cassette tape surface, and the magnetic coating on the tape tends to wear off as you use the cassette. **In order to help eliminate errors, well designed microcomputer systems record every piece of information twice on a cassette tape; this is referred to as redundant recording.** Now suppose you have just one long record per cassette tape; this will become two records if you use redundant recording. This may be illustrated as follows:



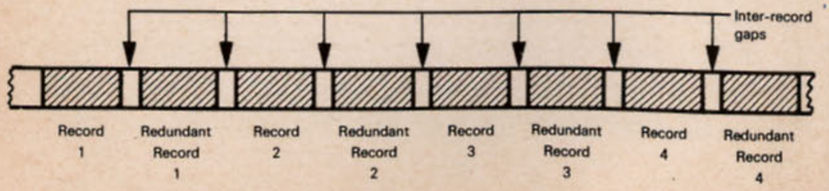
You can have one or more errors in one of the two records and still read the record correctly:



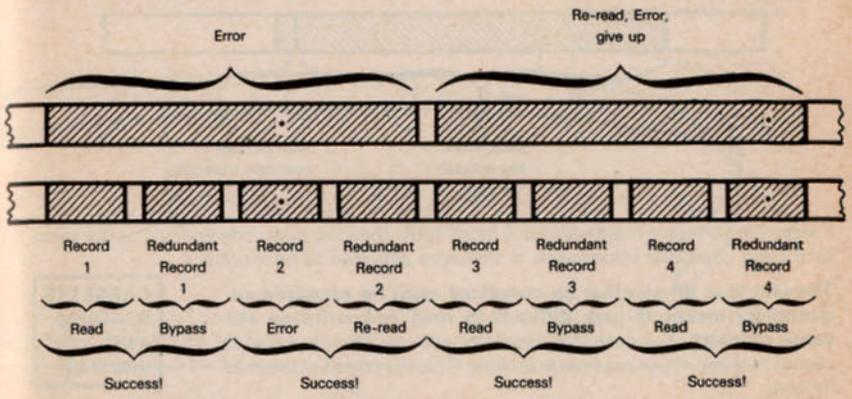




Now suppose we break up the one long record into four shorter records. Each of the four shorter records will have its own redundant record, which means that there will be eight records on the cassette tape. This may be illustrated as follows:

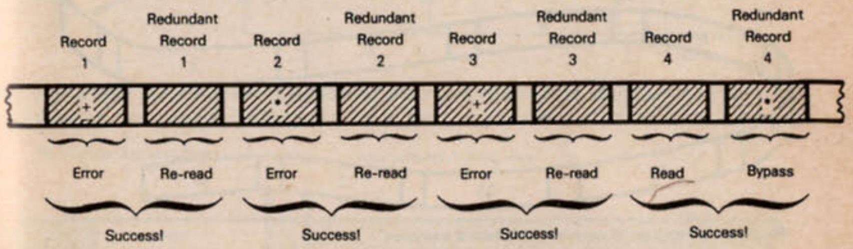


The two errors which would otherwise render the cassette tape unusable can now be tolerated. Remember, errors on a cassette tape will occur because the cassette tape is physically damaged. Thus the two errors will reoccur in exactly the same physical locations on the cassette tape. This may be illustrated as follows:



Our cassette tape is now more tolerant of errors simply because we have increased the number of records.

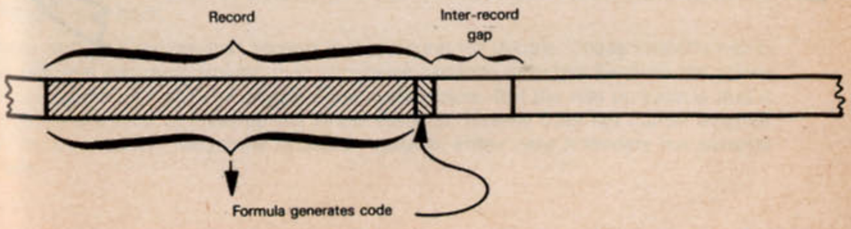
We can now handle two additional errors before having to throw out the cassette tape. This may be illustrated as follows:



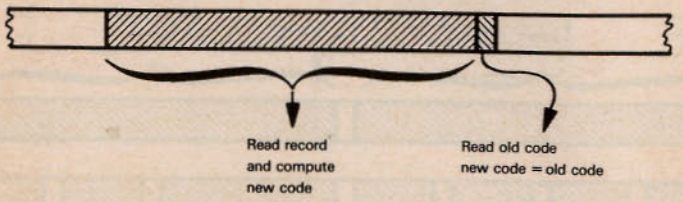
In summary, as the number of records on a cassette tape increases, the total amount of information you can record decreases, but your tolerance of errors goes up — provided you use redundant recording.

ERROR DETECTION CODES

A microcomputer system insures that it has read a record correctly by writing a special error detection code on the end of every record. This special code is calculated by applying a formula to all of the numbers in the record. This may be illustrated as follows:



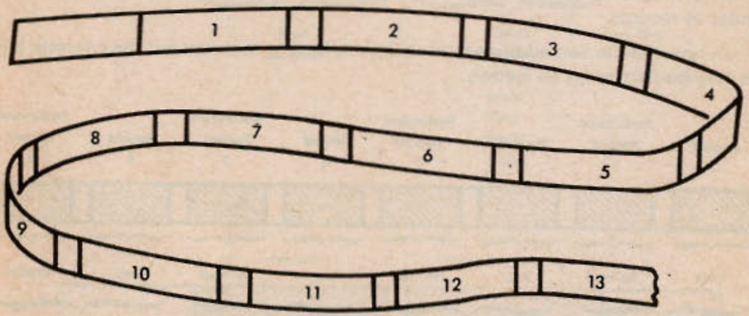
When the microcomputer system reads back a record, it calculates a new error detection code, this time based on the numbers it reads back. Then it reads back the old error detection code. If the record has been read back correctly, the new and old error detection codes will be identical:



If any numbers have been read back incorrectly, then the new code will not be the same as the old code; the information is therefore assumed to be incorrect.

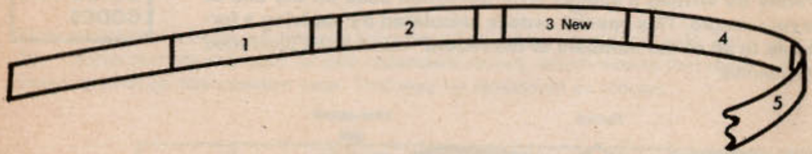
CASSETTE READING AND WRITING

The fact that information on cassettes must be accessed sequentially makes it very difficult to read and write on the same cassette. Suppose, for example, you are storing a list of names and addresses on a cassette tape. This may be illustrated as follows:

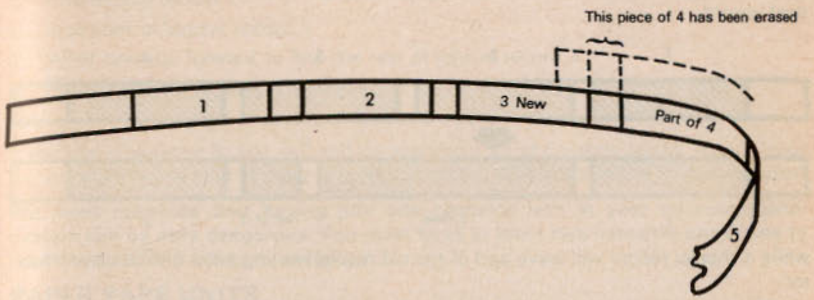


The numbers 1 through 13 represent 13 individual addresses.

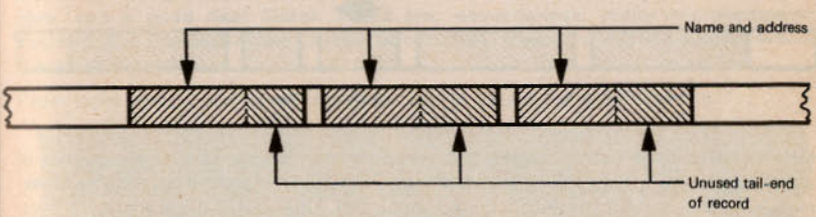
Now suppose you want to change the third name and address (3). This might seem to be a simple enough job; you simply write the new name and address over the old one:



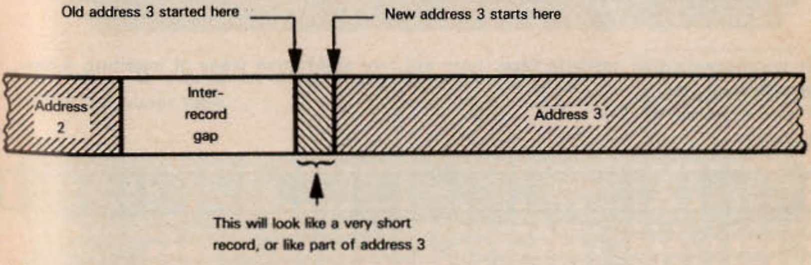
But wait a minute. This could become a very tricky operation. Suppose the new name and address is longer than the old one; you will now wipe out part of the next name and address:



We can get around this problem by reserving the same amount of space for every name and address, irrespective of the number of characters actually in the name and address. Excess characters will be left blank. This may be illustrated as follows:



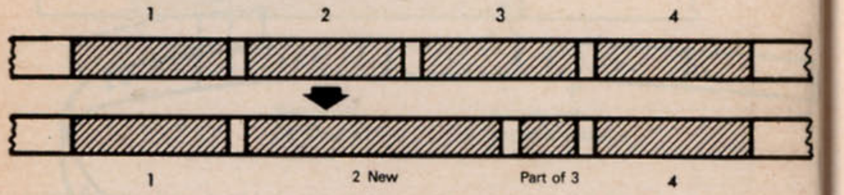
Even this is not an adequate solution, since it demands that your cassette drive mechanism be very precise. Suppose you start rewriting a very small distance too late:


Old address 3 started here
New address 3 starts here
Address
2
Inter-
record
gap
Address 3
This will look like a very short
record, or like part of address 3

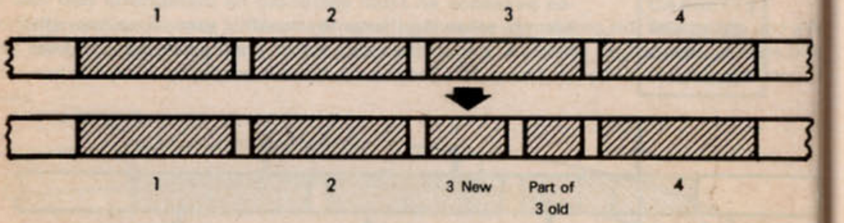
When the microcomputer system reads Address 3 off the cassette, it will pick up the remnant of the old name and address — and get a read error. If you start writing too early, then you have a glitch at the end of the record; and again you will obtain an error when you read back.

Clearly, writing over previous records is just asking for trouble. Cheap cassette recorders will give you more trouble than expensive cassette recorders because the cheap cassette recorders have less precise drive mechanisms. But the real problem is that it only takes one error to mess up the whole cassette. It really does not matter whether this one error occurs frequently or occasionally; in either case it renders the cassette useless.

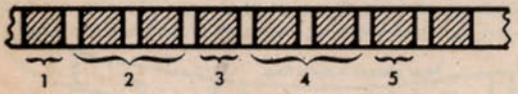
If you have long records and short records all mixed up on one cassette, then reading and writing on the same cassette becomes a hopeless task. If each logical record is stored on the cassette tape as a single physical record, then clearly you cannot write different length records into the same space. A longer record will wipe out part of the next record:



while a shorter record will leave part of the old record waiting to be picked up as an error:

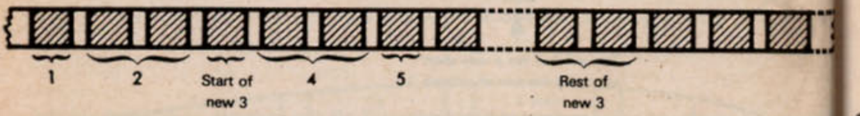


If logical records of various lengths are stored on the cassette tape as a sequence of equal length physical records, you could write and read on the same cassette by breaking up the logical record as we did on the floppy disk. This may be illustrated as follows:

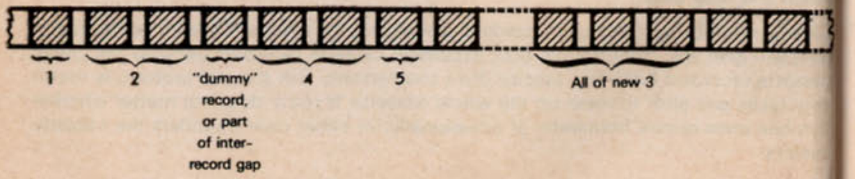


Beginning with this cassette tape, here are two alternative ways of inserting a new record 3:

Alternative A



Alternative B



The problem with these schemes is that you will grow old watching the cassette wind and rewind. If you have alternative A, here are the steps that would be needed to read back the record illustrated above:

1) Find logical record 2
2) Find start of logical record 3
3) Wind cassette forward to find the rest of logical record 3
4) Rewind to start of logical record 4

etc.

If you use alternative B, you will end up wasting a lot of cassette space. Also, if your records are in some kind of order (e.g., alphabetical), they will soon be in total disorder.

We must conclude that having just one cassette unit in your microcomputer system can be very dangerous. You must have at least two cassette units, one to read from and the other to write to.

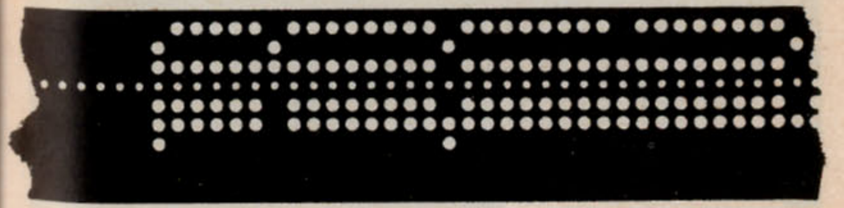
PAPER TAPE UNITS

There is a bulk storage device that is even more primitive than cassettes; it is paper tape. There was a time when paper tape represented the only low cost means of storing computer information, but today cassettes are about as inexpensive, and a good deal faster. There are, nevertheless, many microcomputer systems that use paper tape to store information; therefore this is a subject which we must discuss.

Paper tape, as its name would imply, consists of a long thin paper tape:

PAPER TAPE CHARACTERS

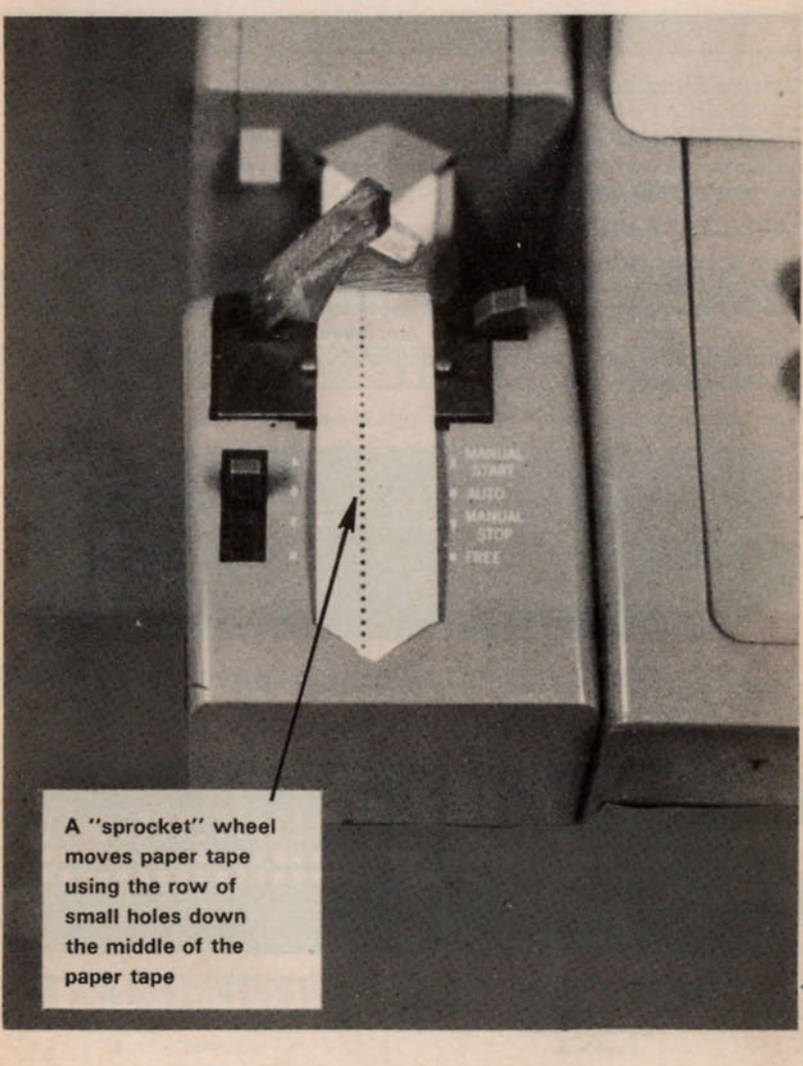
Information is stored on paper tape by punching holes across the paper tape:



Every row of vertical holes on the paper tape represents one character.

PAPER TAPE SPROCKET FEED

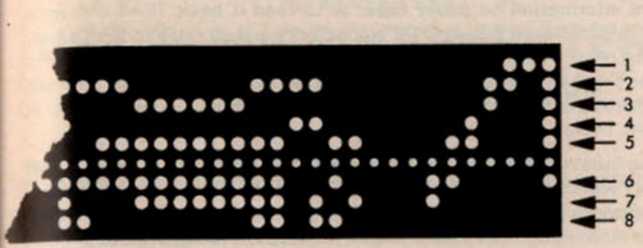
In addition to the holes which represent characters, **most paper tapes have a line of small sprocket feed holes** punched approximately down the center, as illustrated above; these holes are used to move the paper tape:



A "sprocket" wheel moves paper tape using the row of small holes down the middle of the paper tape

PAPER TAPE CHANNELS

There are eight positions on each vertical line where a hole may be punched or not punched.:



The eight hole positions are **called channels.**

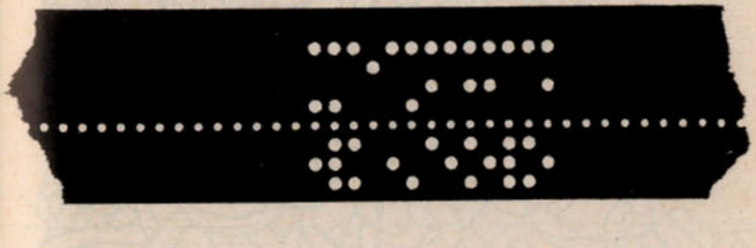
The combination of punched and unpunched positions constitutes a code, which in some fixed fashion identifies a character. When we talk about "characters" we are referring to any letter of the alphabet, any numeric digit, or any special character such as a period, a comma, an exclamation mark, a question mark, etc.

We will discuss character codes later on, in Chapter 4.

There are ten rows of holes punched per inch of paper tape. Thus one inch of paper tape will record ten characters of information. For example the name:

Joe Bitburger

will require 1-3/10 inch of paper tape if recorded exactly as illustrated above. This may be illustrated as follows:



The principal disadvantage of paper tape is that you will need enormous quantities of it. Although cassette storage densities vary greatly, you will use approximately 2000 feet of paper tape to record the same amount of information as a single 90 minute cassette, or one side of a single density floppy disk.

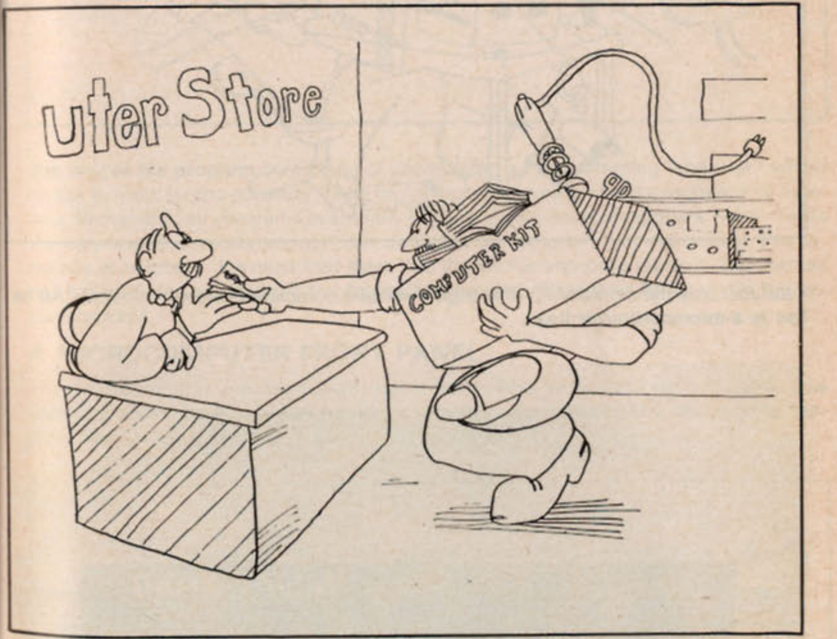
A further disadvantage of paper tape is that it takes a long time, relatively speaking,either to record information on paper tape, or to read it back. Read and write times will typically vary between 10 and 100 characters per second. While this may seem to be quite fast, cassettes can be written to and read from at between 100 to 1000 characters per second; typical floppy disk read and write rates range between 1000 and 10,000 characters per second. You can read and write using rigid disks, at rates in excess of one million characters per second.

Regarding time, you must understand the totally different frames of reference that apply once you start working with computers. Why is 10 characters per second considered slow? The reason is that you will usually read or write hundreds or thousands of characters at a time; and more often than not you will have nothing to do but watch while the read or write operation occurs. Even ten seconds is a long time to wait and watch if you find yourself doing it repeatedly.

Chapter 2
USE A MICROCOMPUTER AND WATCH IT GROW

Let us now look at the many ways in which you can use a microcomputer system.

In order to help us in this task, meet Joe Bitburger, an intrepid computer "hobbyist". Having worked occasionally with computers, one day Joe is seized by a fit of irrationality, during which he buys a microcomputer at his local computer store.



Joe's act is not unusual; it is also not very reasonable. Joe has a great deal to learn.

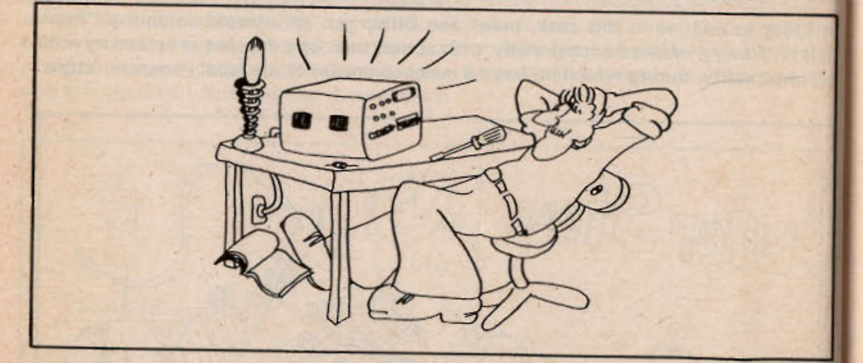
When I say that Joe works occasionally with computers, what I mean is that in the course of his job he occasionally writes computer programs using a computer language called FORTRAN. Every morning Joe dutifully takes a deck of cards (that is computer cards, not playing cards) to the computer center at his office. All being well, some time shortly after lunch he stops back at the computer center to pick up a large wad of paper on which his results are printed.

Perhaps one of the things that guided Joe into his irrational act was the fact that having written computer programs for more than two years, he had never been close enough to a computer to touch it.

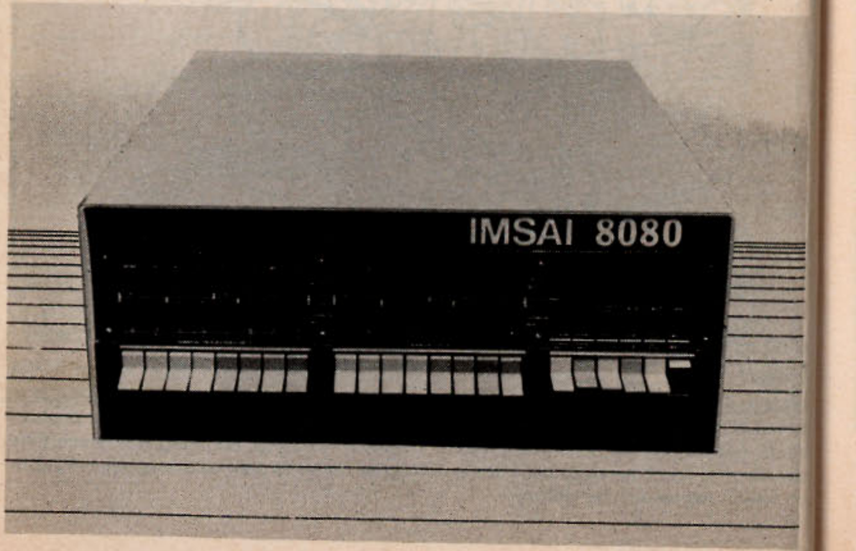
Joe proudly takes his microcomputer home — to assemble and use.

At this point we must bypass many weeks (or months) of Joe Bitburger's life. You see, Joe bought a microcomputer kit. As any computer hobbyist will tell you, the process of assembling such a kit will cause you many hours of anxiety, during which you will cast doubt on the parentage of the kit builder and the kit manual writer.

Ultimately, after many trips back to the computer store, and many scoldings from the store owner for poor soldering, reading and other practices, **Joe Bitburger gets his microcomputer to work.**

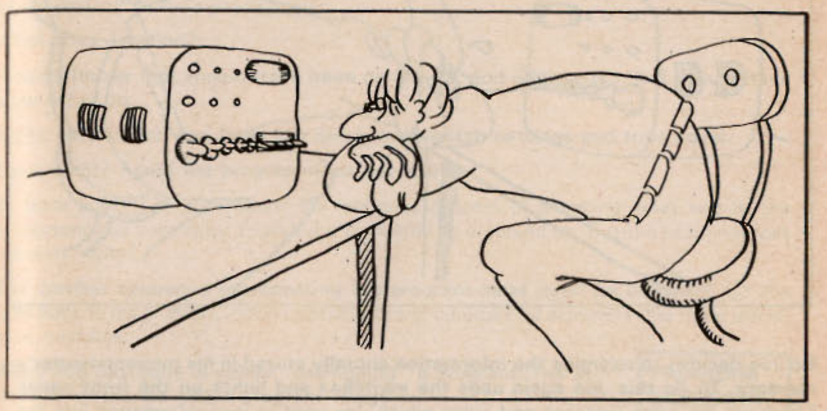


Poor Joe, only the exhilaration of a mighty venture completed keeps him happy. **All he has is a microcomputer box:**



CREATING A PROGRAM AND MAKING IT WORK

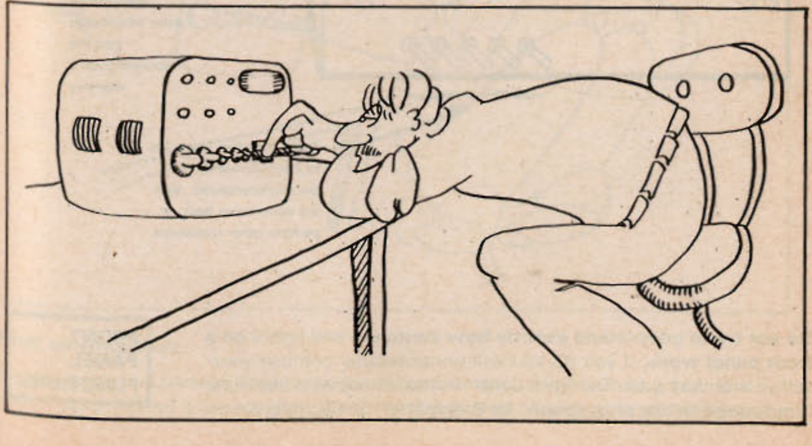
Having built his microcomputer, Joe wants to do something with it. There is precious little it can do. Having few choices, **Joe decides to write a program that makes a light run around on the front panel:**



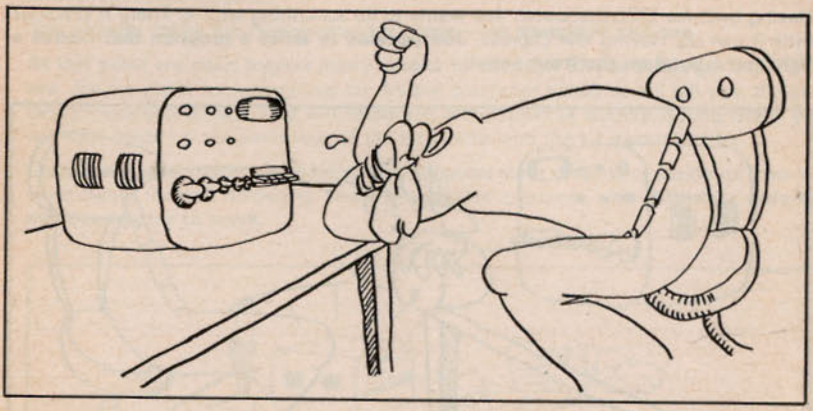
Joe writes his program on a piece of paper using a "programming language" which he has to learn for the occasion. Then he converts the program into a sequence of numbers. Remember, all programs eventually become a sequence of numbers, since that is the only way the microcomputer can understand a program. Having created this sequence of numbers, Joe must load them into the microcomputer's memory. But Joe has no keyboard. Joe has nothing but his microcomputer. What does he do under the circumstances?

A MICROCOMPUTER FRONT PANEL

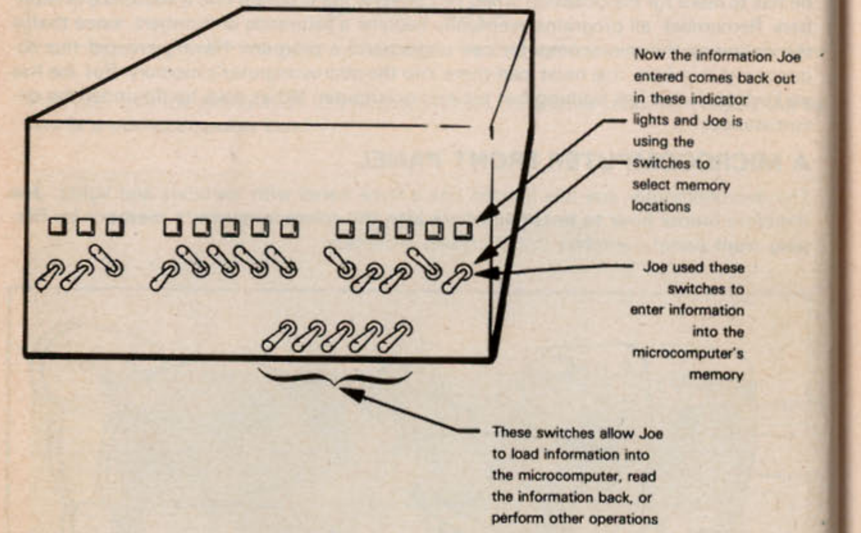
The microcomputer Joe has bought has a front panel with switches and lights. **Joe** therefore **learns how to enter numbers into the microcomputer's memory by flipping front panel switches** in the correct sequence.



Twenty minutes, and a very sore finger later, Joe has finished. Eagerly **he sets the microcomputer to run. Guess what? It does not.**



So Joe decides to examine the information actually stored in his microcomputer's memory. To do this Joe again uses the switches and lights on the front panel. These are the same switches and lights which Joe used to load information into the microcomputer memory. Conceptually this may be illustrated as follows:


Now the information Joe entered comes back out in these indicator lights and Joe is using the switches to select memory locations
Joe used these switches to enter information into the microcomputer's memory
These switches allow Joe to load information into the microcomputer, read the information back, or perform other operations

FRONT PANEL FUNCTIONS

Do not try to understand exactly how switches and lights on a front panel work; if you do you will unnecessarily confuse yourself — and that is all. The front panel illustrated above is like none that appears in any photograph; for that matter, no two microcomputer front panels are alike and some microcomputers do not even have a front panel. **Front panels are used to load information into the microcomputer's memory, to examine the contents of the microcomputer's memory and to control microcomputer operations in general.**

In order to examine microcomputer memory contents Joe simply reads his manual and follows step-by-step instructions. That is what you would do, when and if you found yourself in Joe's predicament.

What does Joe find?

Two switches that should have been on are off, and one switch that should have been off is on.

Not to be daunted, **Joe fixes the erroneous switch settings and tries again.**

Guess what? **Again the program does not work.**

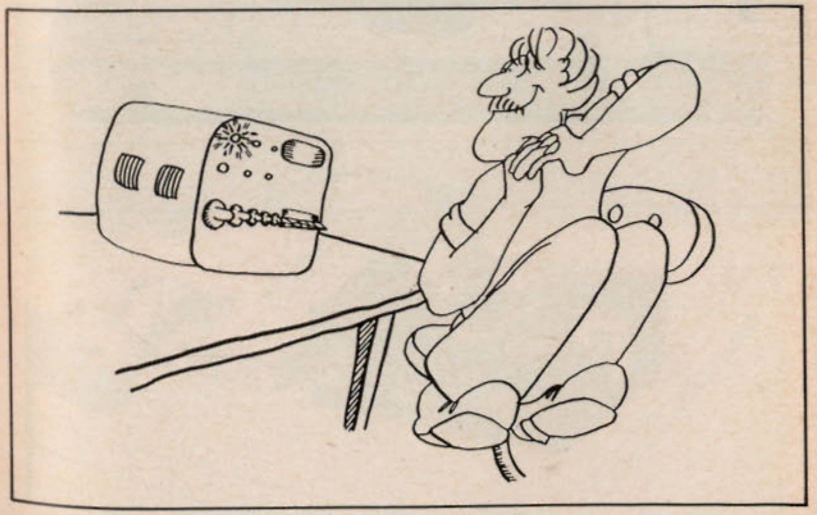
So once again Joe goes about the laborious process of checking what was in the microcomputer's memory. Everything is exactly as it should be. But the program does not work; why?

The number sequence representing the program must itself be incorrect. So Joe goes back to his program; upon checking it over carefully he discovers that there are indeed mistakes.

PROGRAM DEBUGGING

What Joe has done is called "debugging" a program. The individual errors are called program "bugs".

This sequence of correcting errors in switch settings, and then finding more errors in the program, may occur many times. In fact, **it is two days before Joe finally gets his program to run. Triumphantly he sits in front of his microcomputer — watching a light chase around and around the display.**



What an anticlimax.

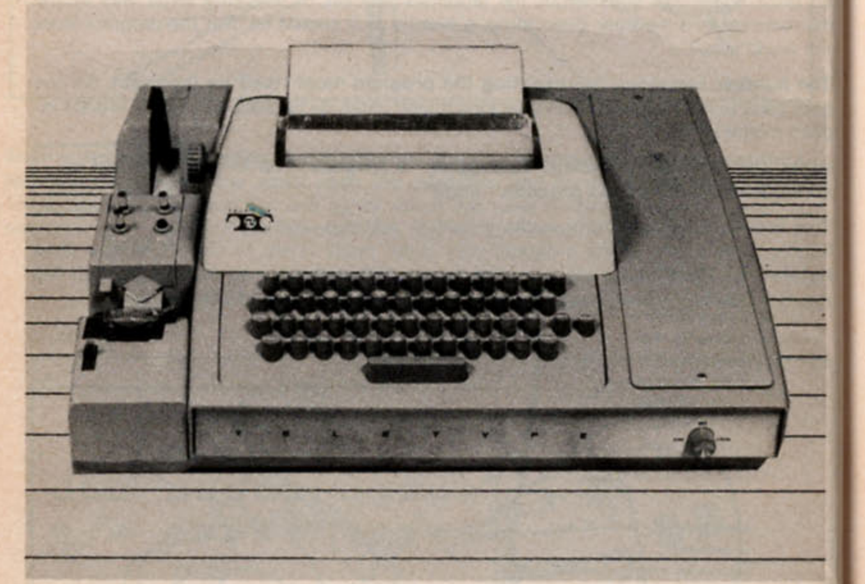
Joe has spent hundreds of dollars and hours, just to watch a light running around a display? This is indeed a far cry from the great things that happen at work between the time Joe leaves his program card deck at the computer center, and the time he gets back a stack of printed results.

Clearly Joe must get some eyes and ears for his microcomputer.

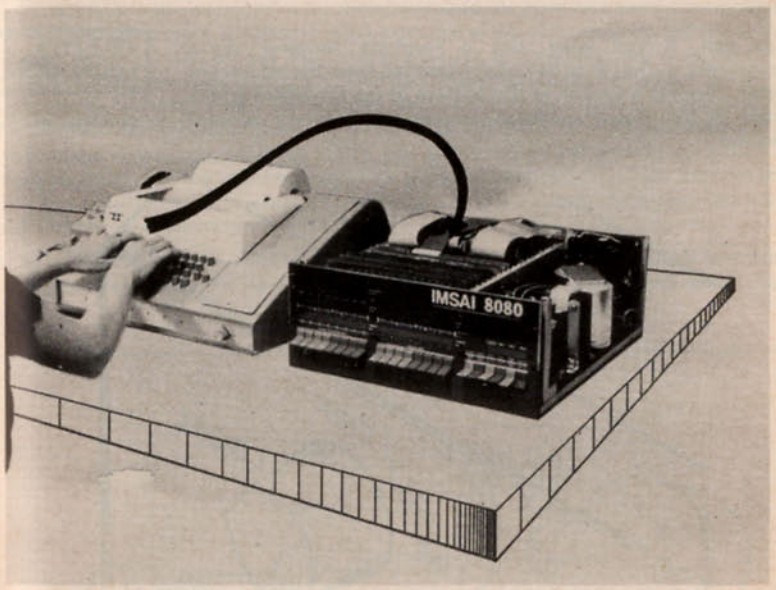
Joe borrows a Teletype terminal from a friend.

THE TELETYPE TERMINAL

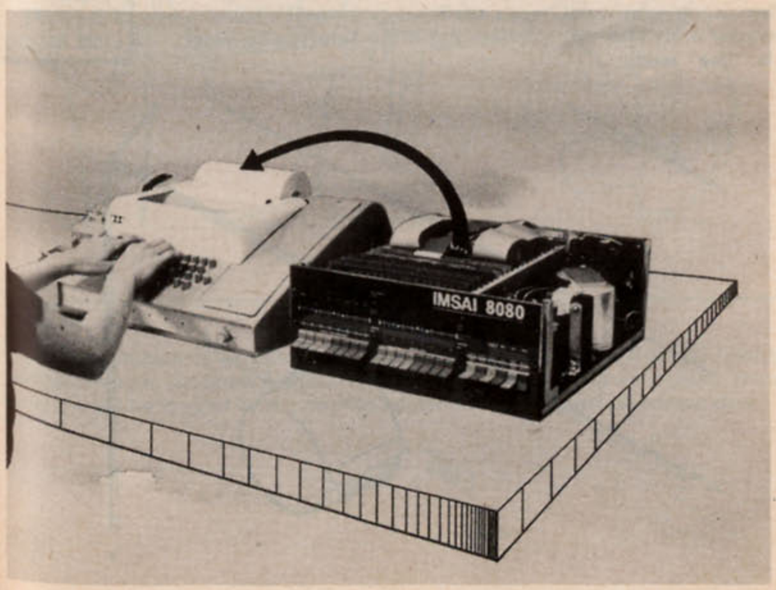
Now the Teletype terminal is a very interesting device. It has been around for more than twenty years, virtually unaltered; it has received more abuse (physical and verbal) than any other piece of computer equipment, and yet there are probably more Teletype terminals around than any other kind of computer terminal. The reason that Teletype terminals are so popular, and so enduring, is that they give you a little bit of everything you need to support a computer; and they rarely break down. **Take a look at a Teletype terminal:**



A Teletype terminal has a keyboard which you can use to enter information into your microcomputer memory:



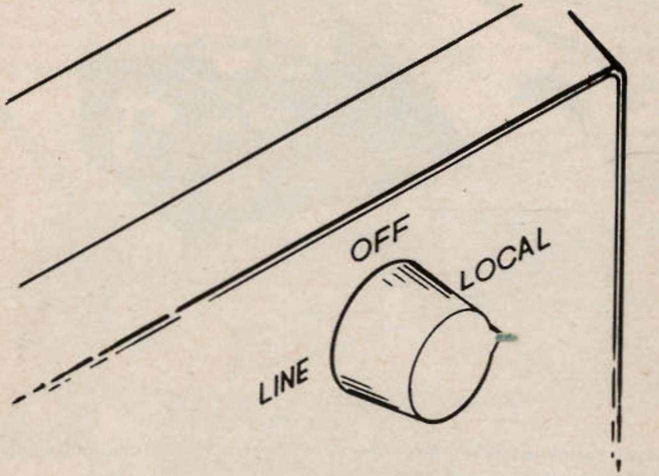
The Teletype terminal has a printer which you can use to print out information, or to carry on a dialogue with the microcomputer:



The Teletype terminal printer serves the combined functions of the video display and printer. In Chapter 1 we discussed how a typewriter printer does much the same thing. But the Teletype printer is not necessarily connected to the Teletype keyboard.

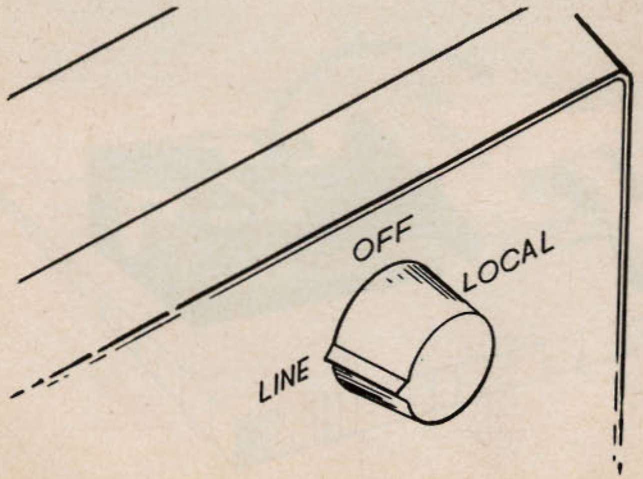
TELETYPE LOCAL MODE

You can, if you wish, use a Teletype terminal like a typewriter — that is to say, disconnected from the microcomputer. There is a switch at the bottom of the keyboard which, when in the "local" position, causes the Teletype terminal to operate in this fashion:



TELETYPE LINE MODE

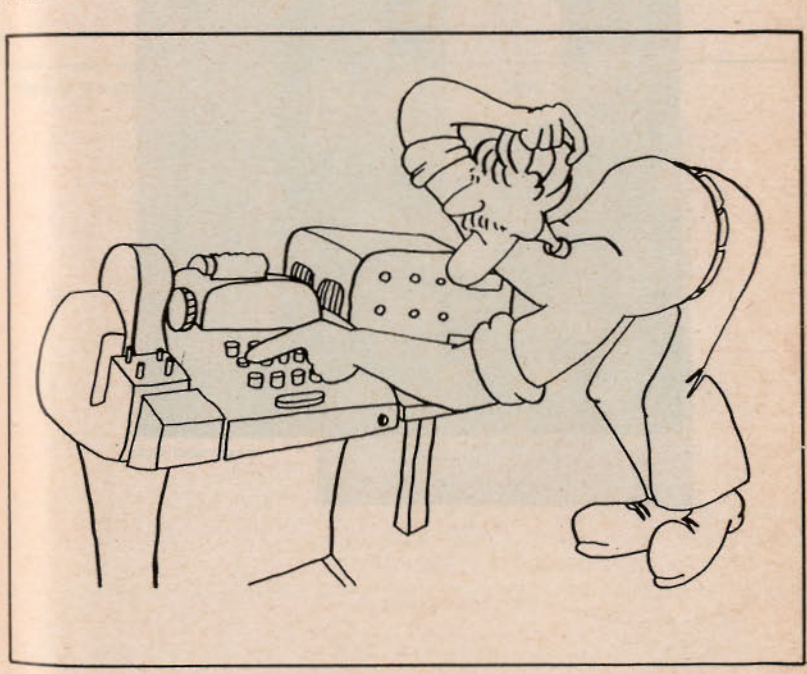
When operating in "local" mode, the Teletype terminal will print a character every time you type one. **But if the Teletype terminal is in "line" mode:**



ECHOING

the Teletype terminal is under the microcomputer's control. When you press a key, a code representing the key you press will be transmitted to the microcomputer, which will read the code and store it in memory, providing an appropriate program is being executed at the time you press the key. If the program which accepts the key input also prints back the character you type, then the Teletype terminal appears to respond much as a typewriter would. Remember, we referred to this as "echoing". But if the program which receives the information you enter at the keyboard does not send back this information to the printer, then no printback or echo occurs. If you wanted to be smart, you could write a program to echo back a character other than the one you enter. You could, for example, echo back the next letter of the alphabet — B following A, C following B, etc. It all depends on what you have programmed your microcomputer to do.

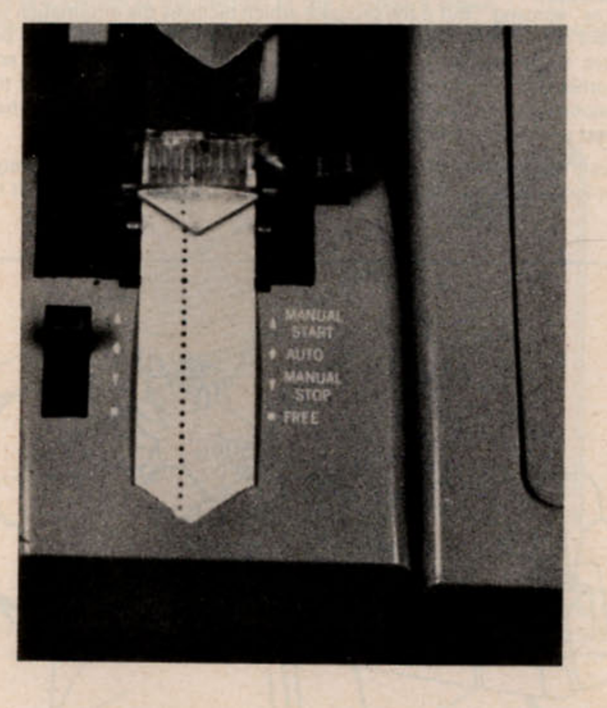
If you press a Teletype key while your microcomputer is executing a program that does not expect input from the Teletype terminal, then the data you enter will fall on deaf ears.



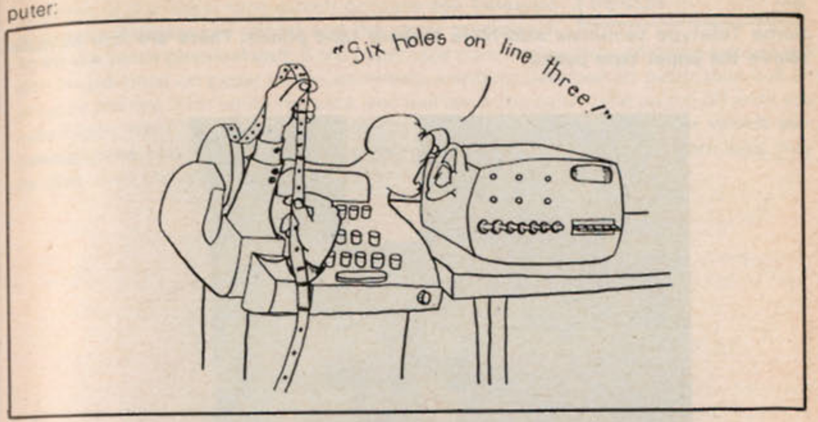
If the microcomputer is not executing a program that expects input from the Teletype terminal, then you can press as many keys as you wish, but nothing will happen. The information will be ignored by the microcomputer and nothing will be printed back.

TELETYPE PAPER TAPE READER

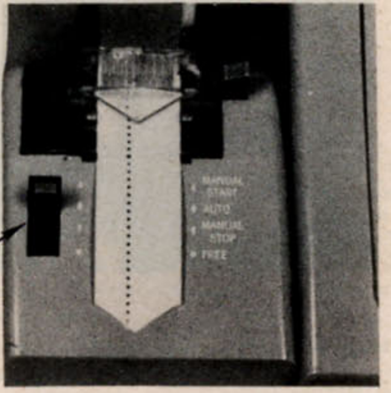
Some Teletype terminals have a paper tape reader:



As the paper tape moves through the reader, the reader detects the presence or absence of holes in each vertical line and sends an appropriate code to the microcomputer:



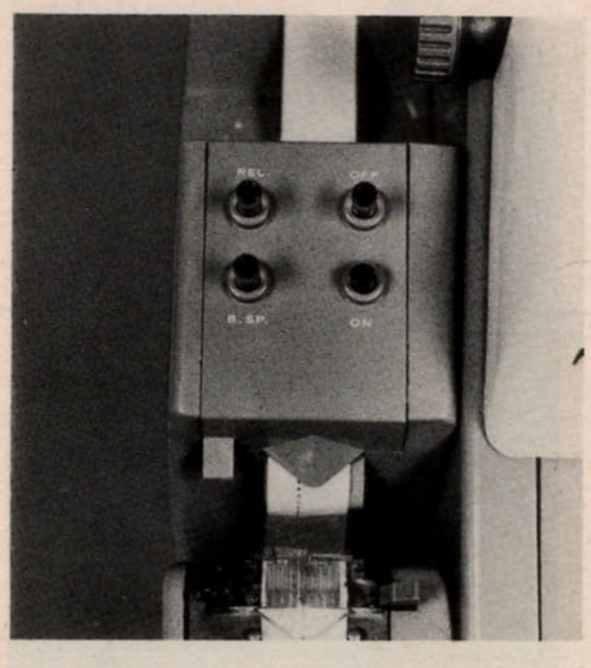
There is a switch that you can use to start the paper tape reader:



Use the RUN switch to manually start and stop the paper tape reader

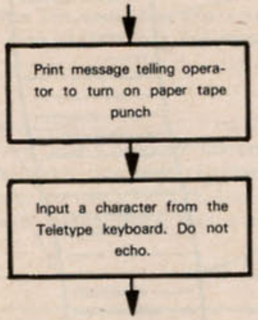
Some programs that use the paper tape reader simply wait for you to start the reader by switching it on. More complicated programs can switch the paper tape reader on for you.

Some Teletype terminals also have a paper tape punch. There are four buttons above the paper tape punch:



TELETYPE PRINTER AND PAPER TAPE PUNCH

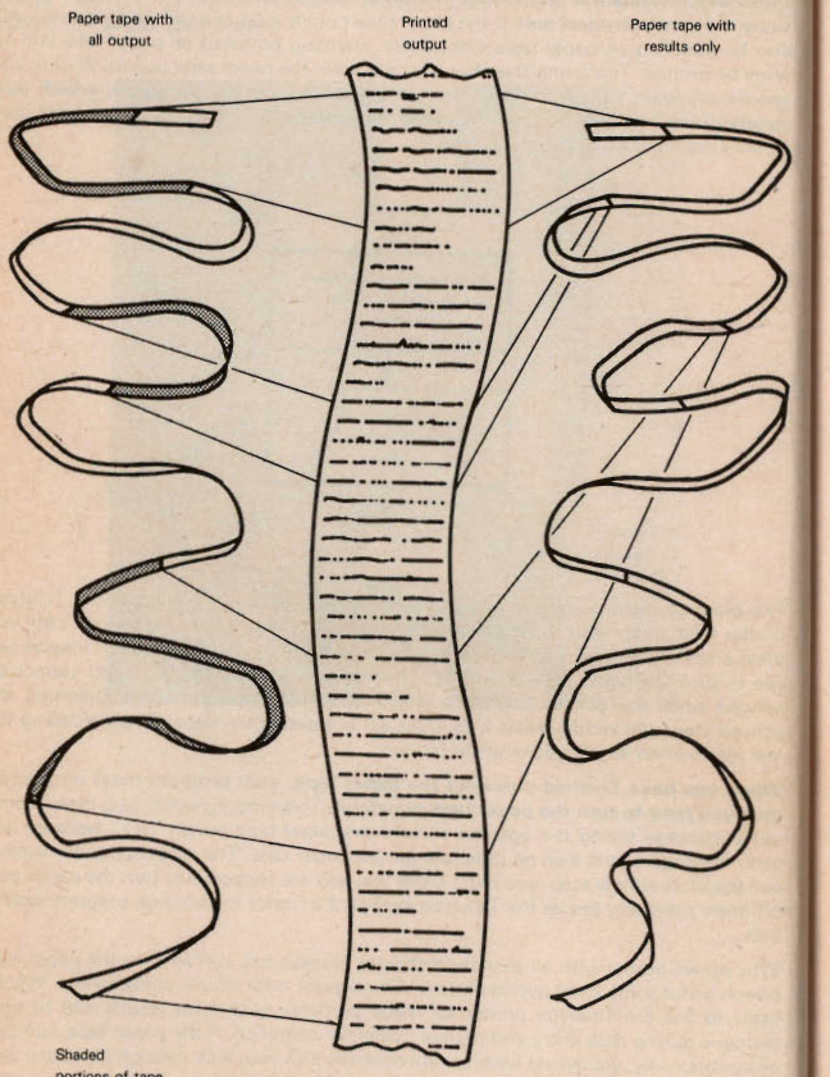
The "ON" and "OFF" switches, as their names would imply, turn the paper tape punch on and off. These are necessary switches at the paper tape punch because **the Teletype terminal considers the printer and the paper tape punch to be the same physical unit.** If the paper tape punch is on, then anything printed will also be punched on paper tape. Conversely, anything punched on paper tape will always be printed. This being the case, you will leave the paper tape punch off until you specifically want to punch paper tape; then **you will write a program which purposely gives you time to turn the paper tape punch on.** Typical program logic may be illustrated conceptually as follows:


Print message telling operator to turn on paper tape punch
Input a character from the Teletype keyboard. Do not echo.

The program logic illustrated above causes a message to be printed at the Teletype printer telling you to turn the paper tape punch on. The program then stops until you press a key — any key — at the Teletype keyboard. Thus you have as much time as you like to turn the paper tape punch on. The key you press at the keyboard cannot be echoed since the echoed character would be printed and punched. Punching the echoed character would create a spurious set of holes in the paper tape preceding the real information which you want to output.

When you have finished punching the paper tape, your program must once again **give you time to turn the paper tape punch off.** This time, however, you cannot print out a message telling the operator to turn the paper tape punch "off", because any such message would also be punched on the paper tape. The microcomputer system will therefore simply stop; you must know that you are supposed to turn the paper tape off, then press any key at the Teletype keyboard in order to continue program execution.

Why do we bother with all of these elaborate precautions, just to have the paper tape punch on for some short period of time? The answer is for simple convenience. We are likely to use the Teletype printer for many purposes — to print results and to print dialogue during data entry and normal computer operation. If the paper tape punch is on continuously, you would have to subsequently pick your way through the paper tape identifying those pieces of paper tape which contain results that you wish to save, and those pieces of paper tape which contain dialogue and junk. This may be illustrated as follows:



Shaded portions of tape represent unwanted dialogue

TELETYPE PUNCH REL CONTROL

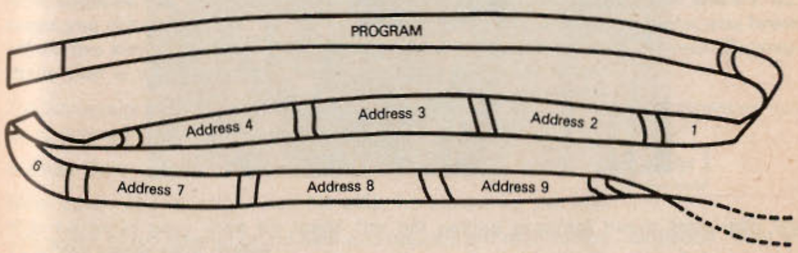
TELETYPE PUNCH B.SP CONTROL

There are two additional control buttons at the paper tape punch. One of these buttons is a Release button (REL); it allows you to slide paper tape in and out when you are threading paper tape into the punch. The other button is a backspace button (B.SP); it allows you to move the paper tape backward, one character position at a time.

USING A SIMPLE MICROCOMPUTER SYSTEM

Given the luxury of a Teletype terminal, Joe can now make his microcomputer system do useful things. The first useful thing Joe does is write a program to help him pay his bills; this is very appropriate since the microcomputer has created more than its fair share of bills.

Now how in the world is a microcomputer going to help Joe pay his bills? Simple — it can save Joe from writer's cramp. **Joe's program contains a list of names and addresses for everyone who routinely demands checks from Joe;** the phone company, the mortgage company and the cable TV company, to name just a few. **Joe creates his program on paper tape; conceptually this is how the paper tape looks:**



Now it does not take Joe very long to figure out that messing around with programs to control the Teletype terminal is a waste of time. Remember, it requires a program executing in the microcomputer to accept information you enter at the Teletype keyboard or paper tape reader, and to optionally echo this information at the printer. **When Joe first connects his teletype to his microcomputer, there are no programs in the microcomputer to take care of the Teletype terminal. Thus the first thing Joe has to do is write such programs and enter them into the microcomputer memory by flipping switches at the front panel.**

The problem with this procedure is that **whenever Joe switches power off at the microcomputer, he also empties everything out of the microcomputer's memory.** Thus every time Joe turns his microcomputer on, he has to go through the laborious process of entering the Teletype terminal control program via the front panel switches.

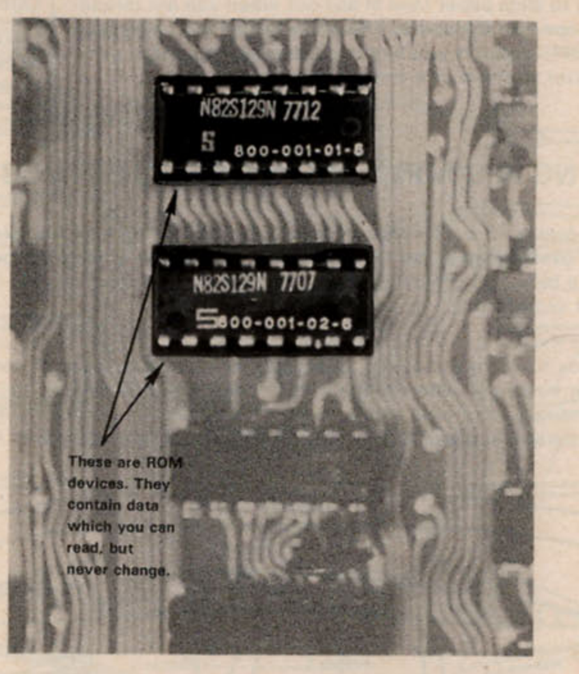
Before long Joe ceases to look upon **the Teletype terminal control program** as a program at all — it **becomes a necessary part of his microcomputer system.**

Joe goes back to the computer store looking for help. And he gets it.

READ-ONLY MEMORY

BOOTSTRAP LOADER

Everyone who has a Teletype terminal needs a program to control it. Joe discovers that such a program is available for sale, loaded into a small memory chip that can never lose its contents. **The program is called a "bootstrap loader". Joe buys one and this is what he gets:**

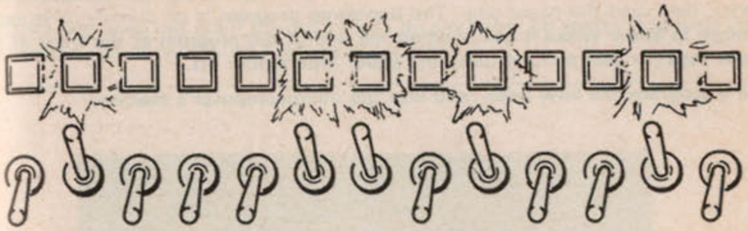


These are ROM devices. They contain data which you can read, but never change.

The term "bootstrap" comes from the concept of a man lifting himself out of a hole by pulling at his bootstraps. Via the bootstrap program, the computer starts itself.

The bootstrap program is stored in a read-only memory chip which, as its name implies, is a memory device whose contents you can read, but into which you can never write. The contents of a read-only memory chip are fixed forever and can never be changed.

In order to understand the difference between read-only memory and read/write memory, imagine the memory chip as consisting of thousands of switches with a light above each switch:



A memory chip is conceptually equivalent to such switches and lights. In reality the memory chip is covered with microscopic electronic structures that look nothing like switches or lights.

When you write into a memory device, what you are doing may be likened to flipping selected switches; a light turns on above each "on" switch.

When you read information from a memory device, what you are doing is equivalent to examining the lights to see which are "on" and which are "off".

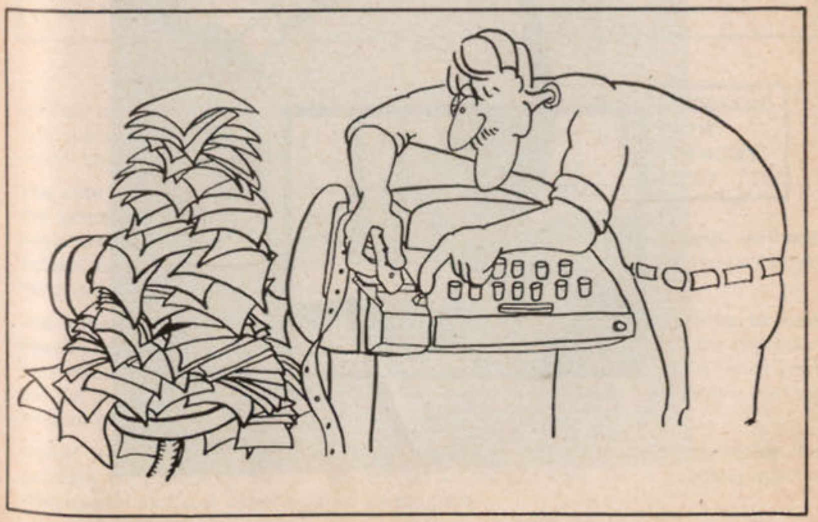
Now suppose you write into the memory chip by flipping switches on and off, but when you are certain that all the switches are in their correct positions, you break off all the switches. Now the lights which are on will stay on forever. Conceptually that is what you have in a read-only memory device.

The advantage of a read-only memory device is that it keeps its contents whatever you do to it — short of breaking it.

A Read-Only Memory device is frequently called a ROM.

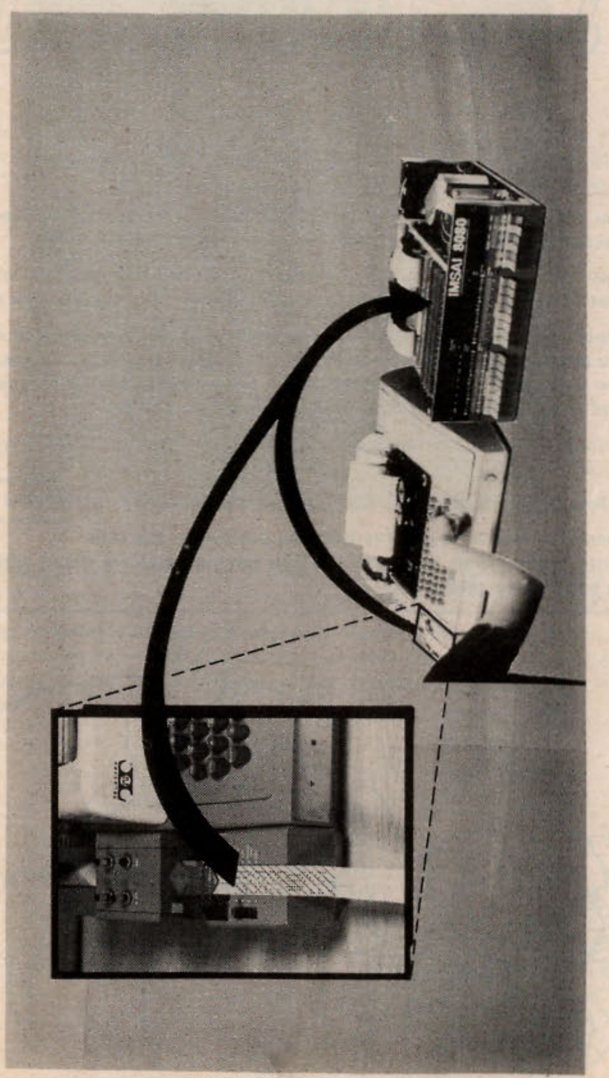
ROM

Now when Joe wants to pay his bills, he stacks up the bills that he must pay, then he loads the bill paying program paper tape into the Teletype paper tape reader:



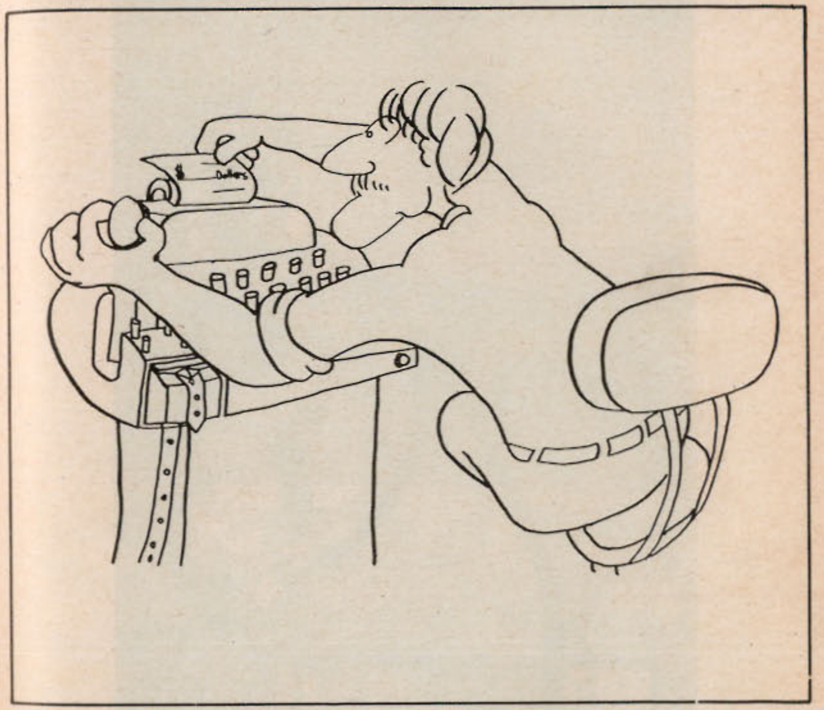
Next Joe flips a few switches, carefully following instructions, and the bootstrap program in Read-Only Memory (ROM) takes over. One by one, instructions from the bootstrap program rush to the microcomputer, causing it to turn on the paper tape reader, then read the paper tape. **The bootstrap program** is no dummy; it **is smart enough to know when it has reached the end of the program at the front of the paper tape; at that point it turns the paper tape reader off.**

Joe's program has now been read into the microcomputer's memory:



The bootstrap program, knowing its work is done, **relinquishes control to Joe's program.** Now instructions from Joe's program rush one after the other into the microcomputer causing it to do Joe's bidding.

Initially Joe does not want his program to do anything; he wants it to wait while he feeds a roll of checks into the Teletype printer:



KEYBOARD ENTRY WITHOUT ECHO

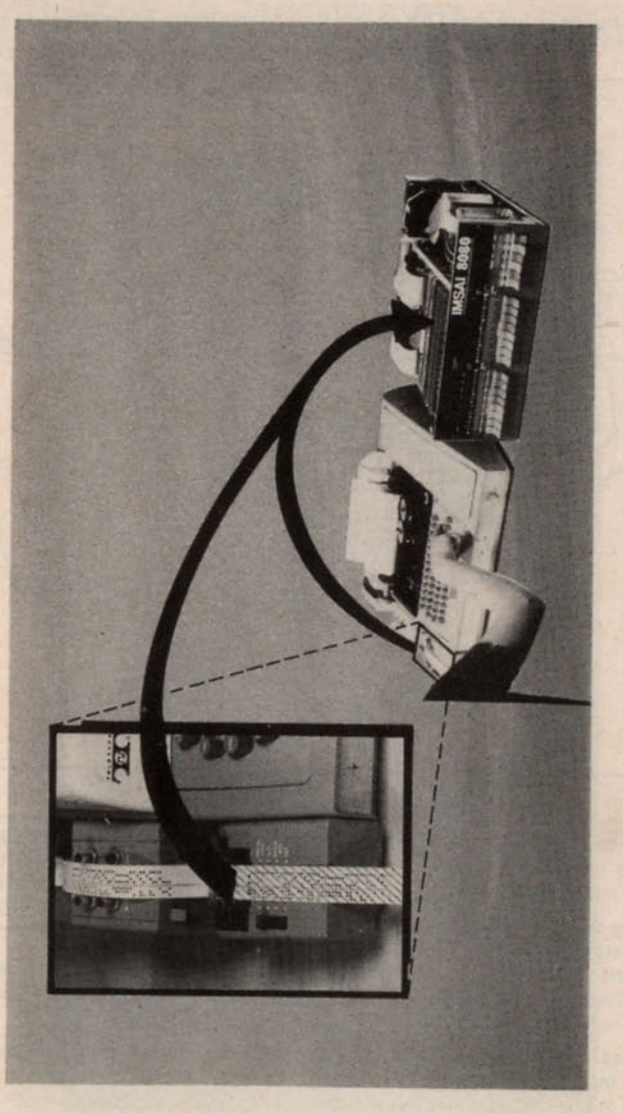
In order to create time during which he can load checks into the Teletype printer, Joe's program has logic much like the logic which turns the Teletype paper tape punch on and off.

The only way Joe's program can know that the checks are in the printer is for Joe to press some key at the Teletype keyboard. He writes his program to wait for Teletype keyboard input, without echo. Clearly, if he presses a Teletype key and the program echoes back the character, the printed character will appear on a check — and will mess it up.

Joe adds an enhancement to the program steps which give him time to load checks into the Teletype printer. Joe's program is designed to check the character entered at the keyboard and see if it is the letter "A". Only if the letter "A" is entered will the program continue; if any other letter is entered the program will simply wait for another character to be entered from the Teletype keyboard.

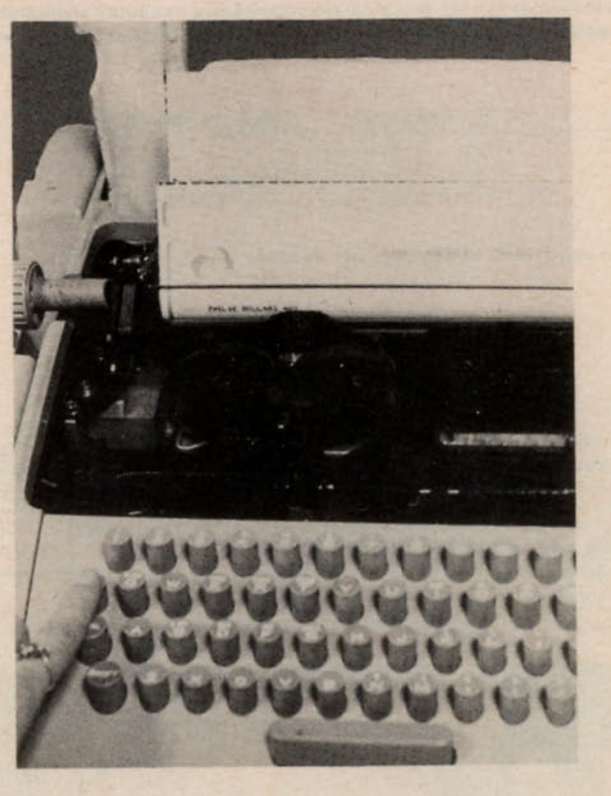
Now if Joe accidentally presses any Teletype key, he will not inadvertently restart the program. Only pressing the "A" key will restart the program. Joe is guarding against making silly mistakes when he runs his program.

As soon as the letter "A" has been entered at the keyboard, Joe's program turns the Teletype paper tape reader on in order to read the first name and address, which you will remember is stored on the paper tape, directly after the program:



Joe's program reads this name and address from the paper tape and stores it as data in memory.

Next Joe's program advances the check in the Teletype printer until the "dollar amount" space is directly behind the printing element. The program makes the microcomputer wait for Joe to type in the money to be paid, first as words, then as numbers:



KEYBOARDS

Let us look at what the microcomputer does every time Joe types a character at the Teletype keyboard. Remember, so far as the microcomputer is concerned, Joe is a real slowpoke — in fact he is a regular snail. The fastest rate at which Joe can type is three characters per second. **A typical microcomputer executes an instruction every five microseconds, which means that it executes 200,000 instructions per second.**

$$\text{5 microseconds} = \frac{5}{1000000} \text{ second}$$

$$\text{Instructions per second} = \frac{1000000}{5} = 200000$$

Since Joe can type 3 keystrokes per second,

$$\text{Instructions per keystroke} = \frac{200000}{3} = 67777$$

The microcomputer can execute 67,777 instructions, on average, between each Teletype key depression. Clearly Joe's program has time to do more than read the incoming character and echo it back — that requires less than 20 instructions.

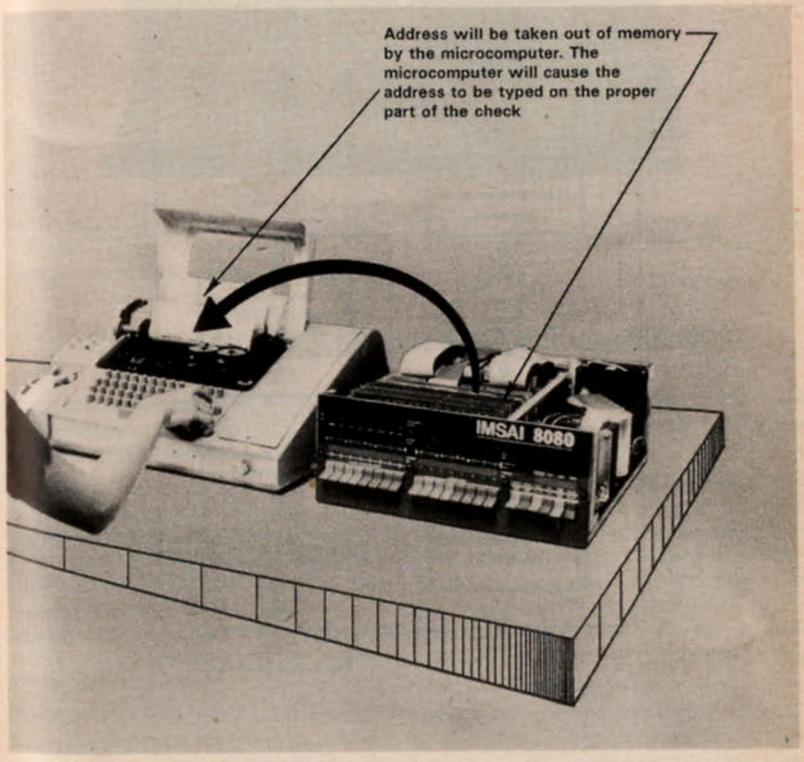
ERROR RECOVERY

Given all of this surplus time, **Joe decides to use the keyboard to help him get out of trouble** — because if a mistake can be made, he will surely make it. **Joe also uses the keyboard to control his program.** Upon receiving a character from the keyboard, Joe's program checks to see if the character is an "Escape" — this is a special key on the Teletype keyboard:

RESTARTING

Upon detecting an "Escape", the program advances the checks in the teletype printer to the beginning of the next check; returns the printer to the beginning of the line; then **restarts handling the current name and address.** Thus any time Joe makes a mistake he can simply escape and restart.

Next Joe's program checks for a "Carriage Return" character. **Upon detecting a Carriage Return character, Joe's program reads the name and address which is now in memory, and prints it on the check:**



Address will be taken out of memory by the microcomputer. The microcomputer will cause the address to be typed on the proper part of the check

PROGRAM LOGIC FLOWCHART

FLOWCHART SYMBOLS

In order to help you understand the logic of Joe's program, without understanding how Joe wrote the program, look at Figure 2-1 which illustrates a program logic flowchart. The squares, circles and diamonds used in Figure 2-1 are part of a standard set of flowchart symbols which everyone uses in the same way. The complete set of symbols is given in Appendix B.

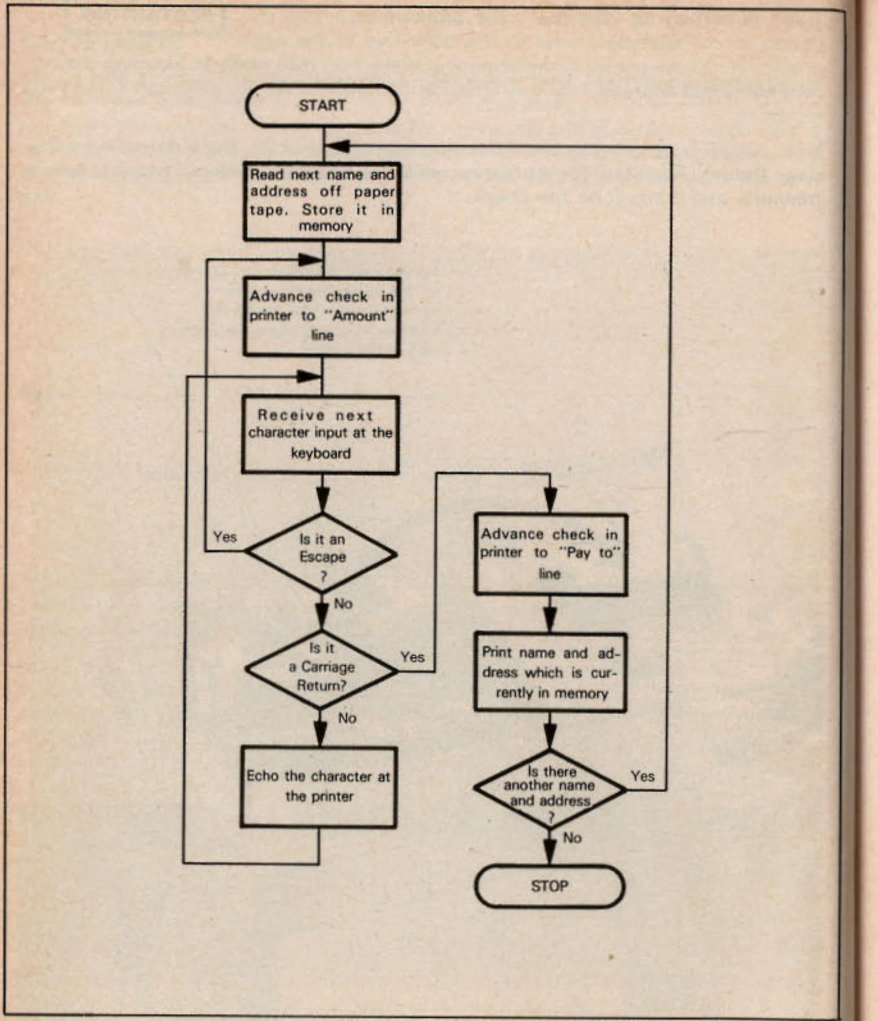

START
Read next name and address off paper tape. Store it in memory
Advance check in printer to "Amount" line
Receive next character input at the keyboard
Is it an Escape ?
Yes
No
Is it a Carriage Return?
Yes
No
Echo the character at the printer
Advance check in printer to "Pay to" line
Print name and address which is currently in memory
Is there another name and address ?
Yes
No
STOP

Figure 2-1. A Flowchart For Joe's Bill Paying Program

If you step back and look at Joe's program, it consists of a sequence of major steps — where one name and address is read, and one check is printed during each major step. And **this is what happens during a major step:**

1) The microcomputer positions the check for Joe to type in the dollar amount.

2) Joe types in the dollar amount.

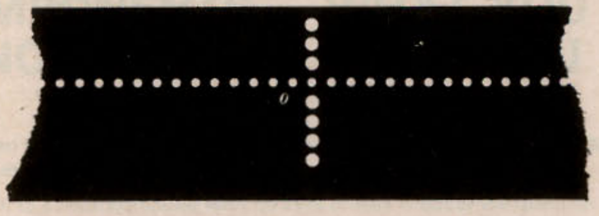
3) Joe types a carriage return to end the dollar amount. The microcomputer automatically executes a carriage return, then types the payee's name and address.

4) The microcomputer reads the next name and address from the paper tape, and it advances the roll of checks to the "dollar amount" line of the next check.

If Joe ever makes a mistake, he presses the Escape key; that allows him to restart the current check — that is, the one within which he made the mistake.

How does Joe's program know when there are no more names and addresses?

Joe's paper tape reading program also checks incoming characters. Joe selects a row of 8 holes as a special "end of data" character. Upon detecting a row of 8 holes the program assumes that all names and addresses have been read; therefore the program stops:



At this point Joe has a set of checks which he can sign and slip into envelopes with windows. There are no names and addresses to write on the front of the envelope and Joe has saved a lot of time.

SOME MICROCOMPUTER APPLICATIONS

Joe's bill paying program is just one simple example of the way you could use a microcomputer system; but it does demonstrate the manner in which the various parts of a microcomputer system interact as a program executes.

In fact, until Joe has a very large number of bills to pay, it will probably be faster to pay them by hand. Programming the microcomputer to do this job will not save any time. We have used the bill paying program as an example not to show you why a microcomputer system is an economical thing to buy, but rather to show you how a microcomputer system is made to work. But even though Joe, who may pay 10 or 15 bills a month, could not possibly justify paying these bills by computer, the same program Joe wrote could handle hundreds of bills as easily as it handled his 10 or 15. With hundreds of bills to pay, the microcomputer system would save a lot of time as compared to doing the same job by hand.

But there is an even more important point worth noting: by simply changing the paper tape, Joe can make his entire microcomputer system do a totally different job. Thus if the microcomputer system is not economical to do one job, it may be economical to do ten. Suppose, for example, that Joe has spent $1,000.00 on his microcomputer system. $1,000.00 plus one program on paper tape generates a microcomputer system that costs a $1,000.00 for one job but suppose Joe has ten different programs, each with its own paper tape. By spreading the $1,000.00 over all ten jobs he could argue that he is only spending $100.00 per program. As Joe finds more and more uses for his microcomputer system, the effective cost per use steadily goes down. And this is what makes microcomputer systems so popular; they will do anything you can define via a program. The only limitation placed on the number of programs you write is the time it takes your microcomputer system to execute all of them.

Joe has many other applications in mind for his microcomputer system. In addition to paying his bills, it can keep his address book updated and balance his checkbook — just to mention a few business type programs.

There are also many non-business type things that the microcomputer can do; for example, it can play games. A microcomputer system can be used to play simple games like those you hook on to your television set, or it can be programmed to play complex games like chess. Joe plans to invent brand new games.

But in the back of Joe's mind **there is an even more interesting application for his microcomputer — synthesizing music.** You can drive speakers directly from a microcomputer; thus you can write programs which create sound. That is something Joe is very interested in trying once he understands his microcomputer a little better.

Chapter 3
MICROCOMPUTER SYSTEM COMPONENTS — WHAT YOU SEE IS NOT ALWAYS WHAT YOU GET

One day an unfortunate thing happens to Joe; his friend wants the Teletype terminal returned.

Going back to a microcomputer with no eyes or ears is out of the question. Not only is the cost of the microcomputer a high price to pay, just to watch lights flash on and off, but Joe has now invested much time writing useful programs. So Joe tightens his belt, grabs his checkbook and goes back to the computer store.

Joe wants to expand his microcomputer system.

The options facing Joe are mind boggling; and they will probably boggle your mind too — until you understand exactly what it is you are looking at. Then a little order will begin to appear out of the chaos.

When putting together a microcomputer system, you must first select the functions you would like your microcomputer to perform; receiving data input, printing results and storing information — these are all "functions". Next you must select the physical unit you want (or can afford) in order to support each function you have selected. For example, you can store information on paper tape, cassettes or floppy disks. Each time you select a particular physical unit, you must decide on the options and additional features you are prepared to pay more for.

We are now going to identify the components you can buy for your microcomputer system, the functions they serve within a microcomputer system, and the options commonly offered.

We will begin by looking at the functions performed and the physical components that you can buy in order to perform each function.

PHYSICAL AND LOGICAL UNITS IN MICROCOMPUTER SYSTEMS

If you refer back to Chapter 1, in Figure 1-1 you will see the following components illustrated:

1) The microcomputer itself
2) A keyboard
3) A video display
4) A printer
5) Bulk storage

Do you remember how, in Chapter 1, we described records and files as either "physical" or "logical"? The "physical" record or file is the form in which infor-

mation is actually stored by a bulk storage device, whereas a "logical" record or file is the way in which you as a microcomputer user visualize and use the stored information. We can extend this concept a step further by taking the five components of a microcomputer system illustrated in Figure 1-1, and looking at the functions they perform.

The keyboard becomes an "Information Entry" logical unit; the information entry logical unit does not have to be a keyboard: it could be microcomputer console switches, or any other piece of hardware capable of reading information — even a paper tape reader.

INFORMATION ENTRY LOGICAL UNIT

The video display becomes an "Operator Message" logical unit; it could be a video display, a teletype printer, a typewriter printer or any piece of hardware capable of delivering messages in a human readable form.

OPERATOR MESSAGE LOGICAL UNIT

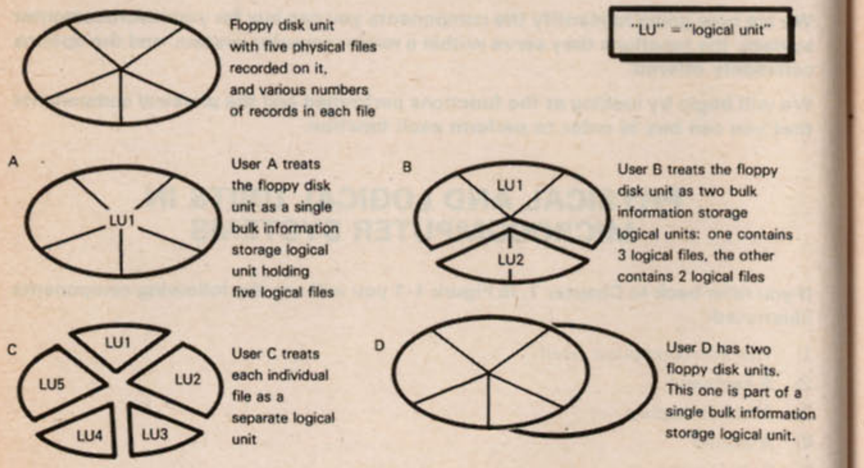
The printer becomes a "Results Output" logical unit. We have seen how results are generated by a typewriter, a Teletype printer or a stand alone printer: there are many other ways in which you may want to generate your results. For example, you could write results on cassette tape, with the idea of printing the results later, when the microcomputer system would otherwise be idle.

RESULTS OUTPUT LOGICAL UNIT

So far we have seen three implementations for a "bulk Information Storage" logical unit: floppy disk units, cassette tape units and paper tape units. We also examined rigid disk units as an alternative to floppy disk units.

A bulk information storage device is not always treated as a single logical unit. Sometimes individual files may become individual logical units, or a group of logical files could constitute a logical unit. Thus one physical unit may become many logical units. On the other hand, more than one physical unit may constitute a single logical unit. Any physical/logical unit relationship that is feasible is also possible. This may be illustrated as follows:

BULK INFORMATION STORAGE LOGICAL UNIT



"LU" = "logical unit"

Floppy disk unit with five physical files recorded on it, and various numbers of records in each file

User A treats the floppy disk unit as a single bulk information storage logical unit holding five logical files

User B treats the floppy disk unit as two bulk information storage logical units: one contains 3 logical files, the other contains 2 logical files

User C treats each individual file as a separate logical unit

User D has two floppy disk units. This one is part of a single bulk information storage logical unit.

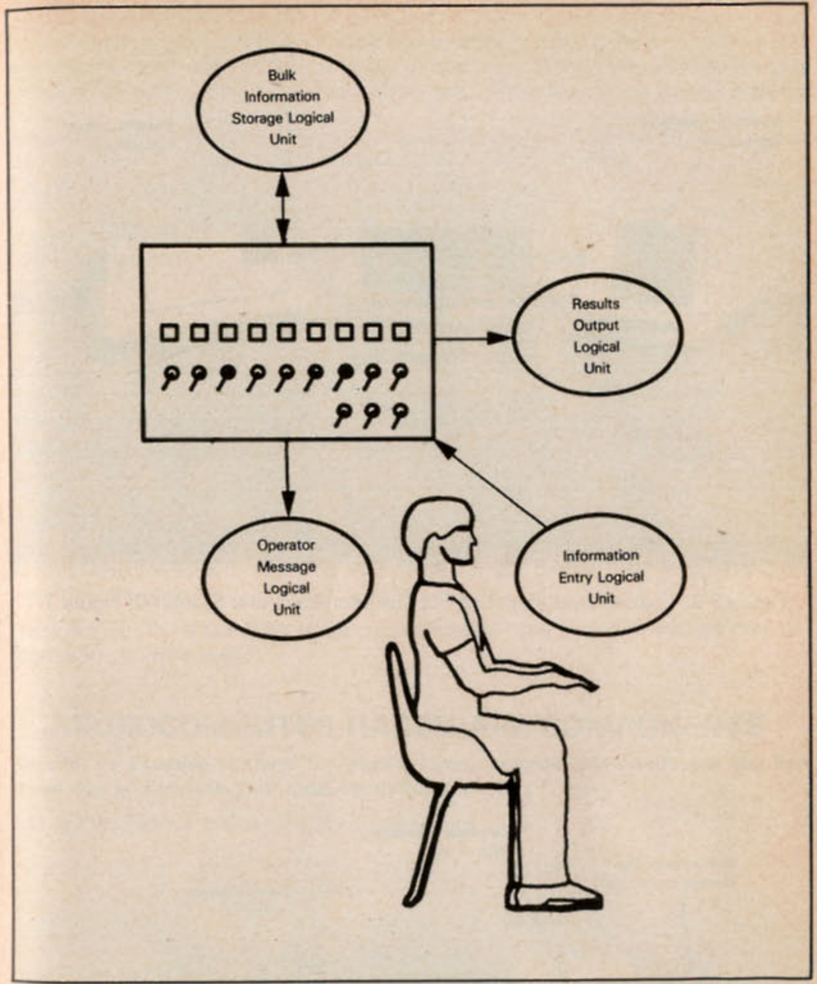

Bulk Information Storage Logical Unit
Results Output Logical Unit
Operator Message Logical Unit
Information Entry Logical Unit

Figure 3-1. Logical Units Surrounding A Microcomputer

Only the microcomputer itself has no substitute. In this one instance, therefore, the physical and logical units must always be one and the same thing.

Let us look at some illustrations that more clearly define the relationships between physical and logical units.

First of all, Figure 3-1 illustrates the four logical units surrounding the microcomputer. In Figure 3-2 these four logical units are shown superimposed on the microcomputer system configuration which we initially introduced in Chapter 1, Figure 1-1. Figure 3-3 illustrates how logical units and physical units are related in Joe Bitburger's very simple microcomputer system, consisting of the microcomputer and a Teletype terminal.

While the concept of logical and physical units may appear abstruse, in reality we can find many parallels in our daily lives.

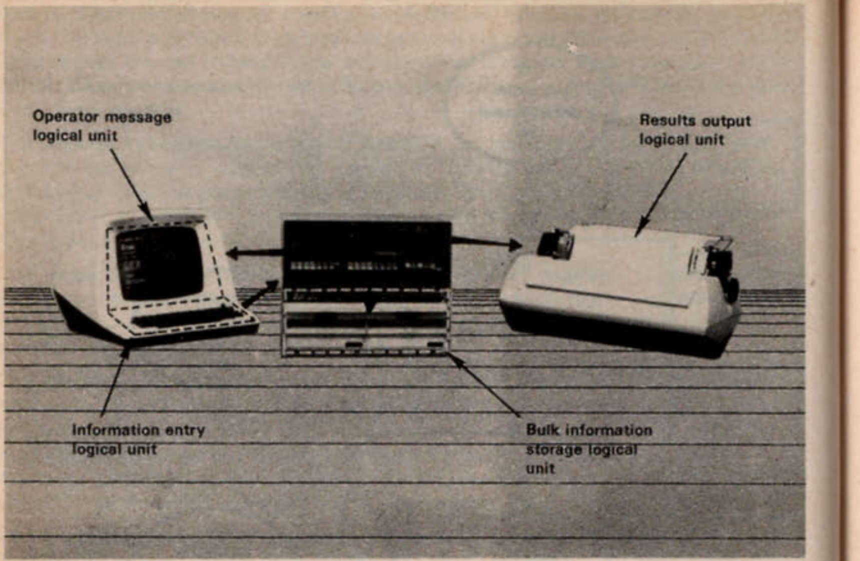

Operator message logical unit
Results output logical unit
Information entry logical unit
Bulk information storage logical unit

Figure 3-2. Logical Units Identified For The Microcomputer System Of Figure 1-1

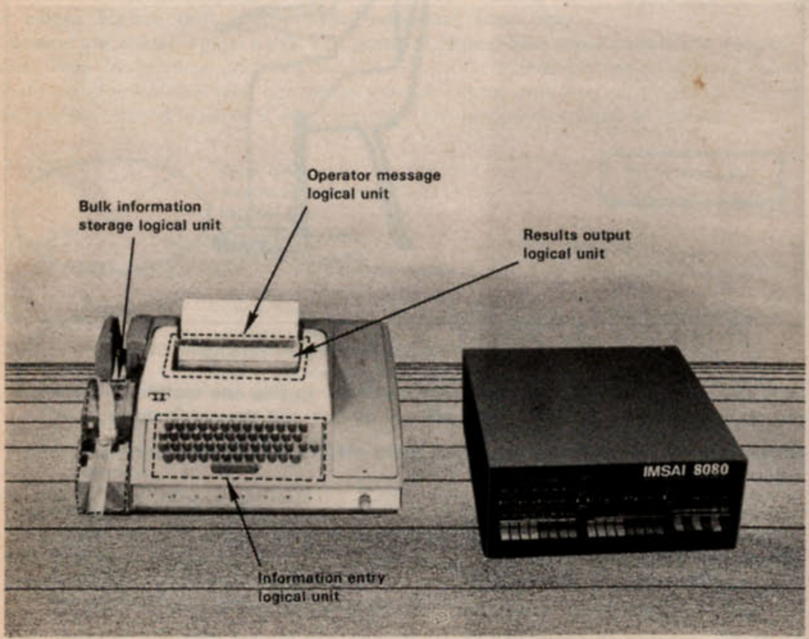

Operator message logical unit
Bulk information storage logical unit
Results output logical unit
IMSAI 8080
Information entry logical unit

Figure 3-3. Logical Units Identified For A Teletype Terminal

Joe Bitburger requires a "wake up" logical unit in the morning, or he will not get out of bed in time, so an "alarm clock" physical unit implements the "wake up" logical unit. One day the alarm clock breaks down. Fortunately the season of the year is such that the alarm clock has been ringing just as the sun rises. So Joe leaves his curtains open while the alarm clock is being fixed; he uses the rising sun through his open window as the physical unit implementing his wake up logical unit. Now had Joe programmed himself (badly) to respond to a ringing noise in the morning, the switch to light coming through the window would not work. But Joe programmed himself intelligently, to anticipate any type of wake up stimulus; therefore he has very little trouble switching from the ring of an alarm clock to sun coming through an open window as his wake up stimulus.

Joe usually has a cup of coffee for breakfast. In order to make a cup of coffee Joe needs a "water heater" logical unit. The "water heater" logical unit may be a "kettle" physical unit; but when the kettle springs a leak, Joe substitutes with a "saucepan" physical unit.

After breakfast Joe has to cope with the "transportation to work" logical unit. Normally a "bus" is the physical unit which implements Joe's "transportation to work" logical unit. One day the bus drivers go out on strike, so Joe rides his bicycle to work. The "bicycle" now becomes the physical unit implementing the "transportation to work" logical unit. Had Joe programmed himself to live by an exact bus schedule, he would now have a problem. But Joe programmed himself to consider various means of getting to work; he allowed sufficient time for a missed or late bus, so now he has sufficient time for his bicycle.

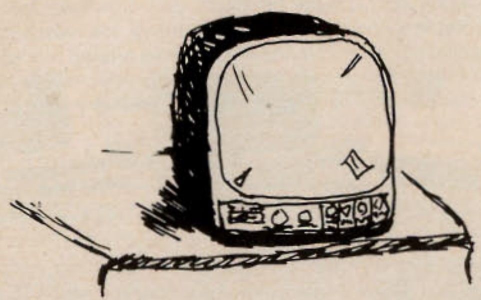
These are just three examples of logical and physical unit analogies that we may encounter in our daily lives.

MICROCOMPUTER HARDWARE COMPONENTS

Now let us examine some of the physical unit combinations which you will find when you go down to your local computer store.

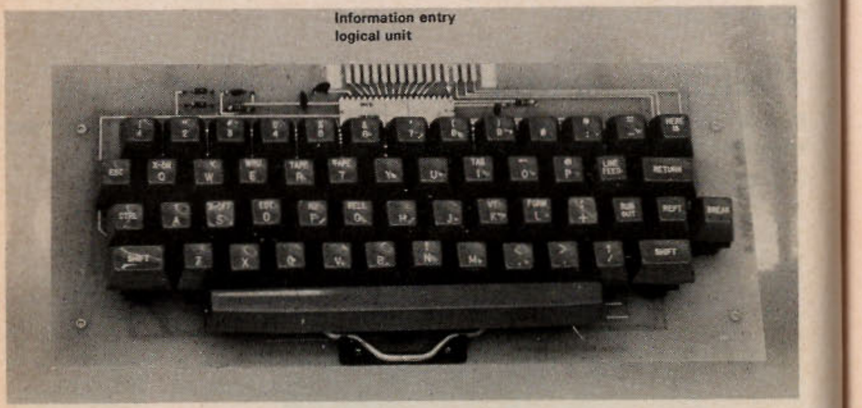
You will see keyboard/video display combinations:

This combination is so common that many people think the two must go together; but they need not. You can buy a video display on its own:

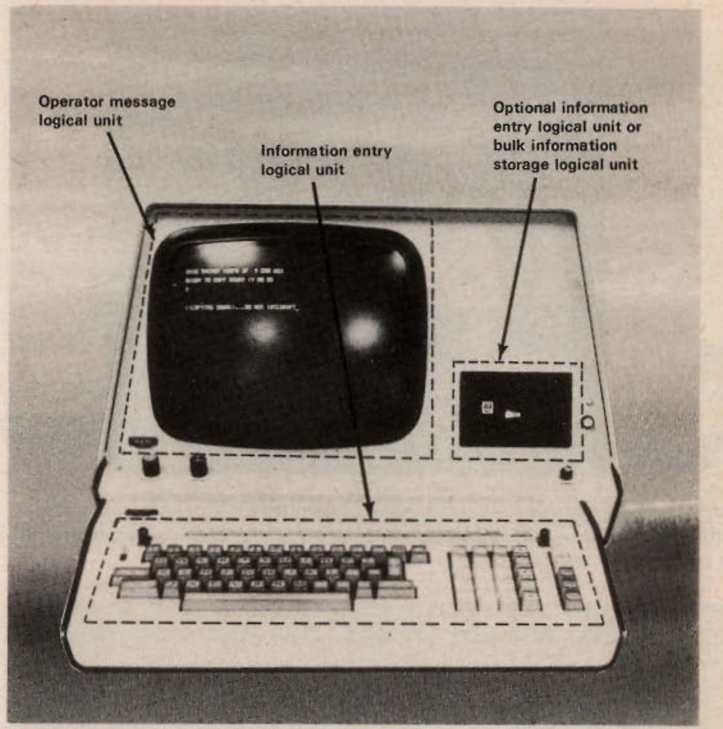


Or you can use your television set as a video display.

You can buy a keyboard on its own:

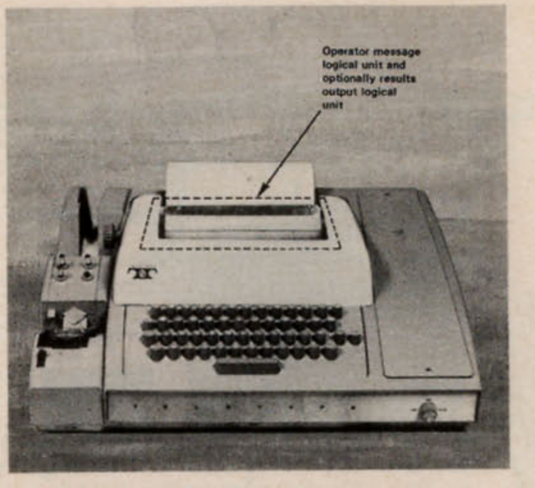


There are keyboard/video display units that include a cassette drive; a cassette drive can serve as part, or all of your bulk information storage logical unit:

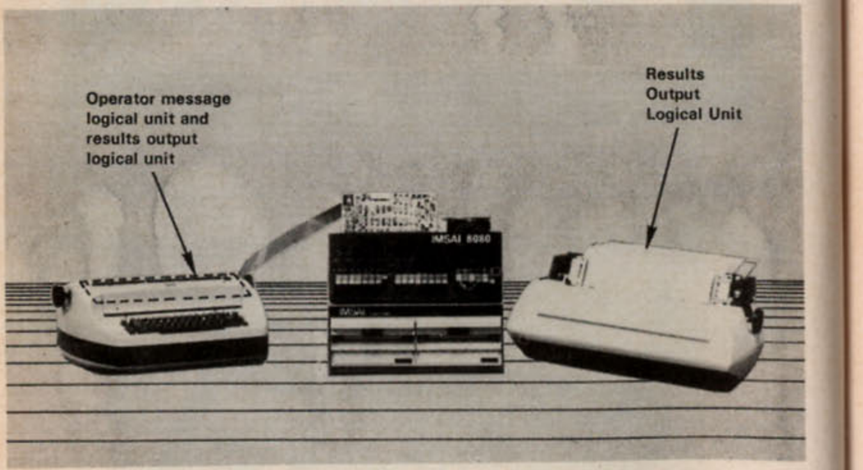


You can even buy a keyboard that looks as though it is an integral part of the microcomputer:

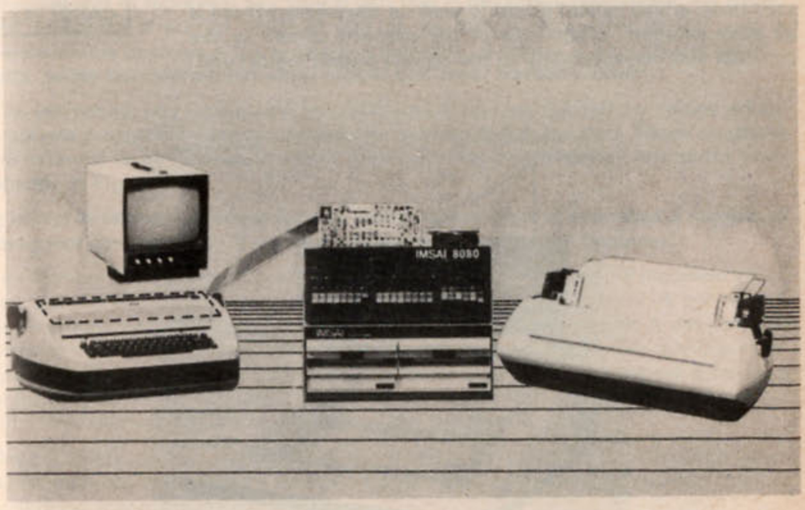
Printers can also be confusing. Sometimes the operator message logical unit and results output logical unit become the same physical unit:


Operator message logical unit and optionally results output logical unit

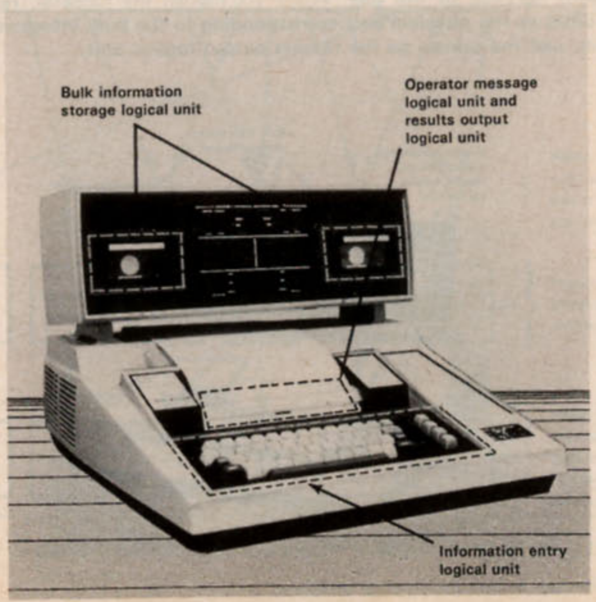
But you can take the same microcomputer system, add a separate printer and now the results output logical unit and the operator message logical unit, even though they are both printers, are separate physical units:


Operator message logical unit and results output logical unit
Results Output Logical Unit
IMSAI 8080

Add a video display and you can use either the typewriter or the printer as the results output logical unit, while the typewriter or the video display can serve as the operator message logical unit:



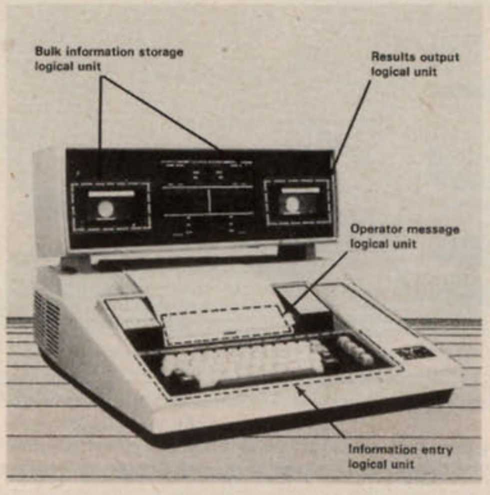
There is one very popular series of terminal made by Texas Instruments, the Silent 700; one version of this terminal combines keyboard, printer and cassettes. This terminal may serve as information entry, operator message, results output and bulk information storage logical units:


Bulk information storage logical unit
Operator message logical unit and results output logical unit
Information entry logical unit

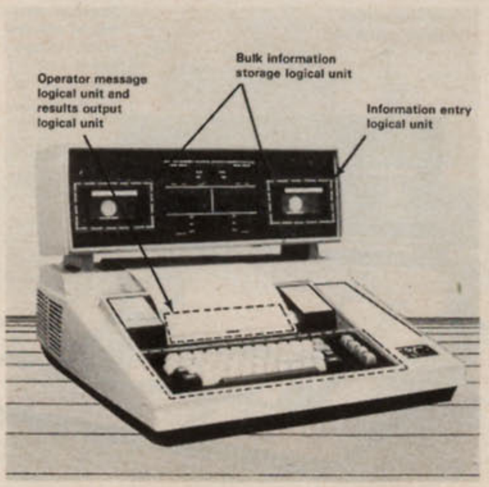
LOGICAL UNIT REASSIGNMENTS

SAVING RESULTS ON CASSETTE

When using the Texas Instruments Silent 700 terminal, consider another interesting possibility; you have results which you wish to print, but you do not need the results at the present time, or you cannot afford to wait for the printer. There is an easy solution. Assign one of the cassette drives as the physical unit corresponding to the results output logical unit:


Bulk information storage logical unit
Results output logical unit
Operator message logical unit
Information entry logical unit

Now your results will be written very quickly to a cassette. **Remove the cassette and save it until the end of the day.** When you reload the cassette, **re-assign the cassette drive as the physical unit corresponding to the bulk information storage logical unit, and the printer as the results output logical unit:**


Operator message logical unit and results output logical unit
Bulk information storage logical unit
Information entry logical unit

Your results can be printed in your absence

DEVICE DRIVERS

The beauty of a microcomputer system is its versatility. Once you have the necessary pieces of hardware, all it takes is one little program in order to use any physical unit to represent any logical unit that is physically reasonable. For example, assigning a printer physical unit as an information entry logical unit is unreasonable because it is physically impossible. A printer is only able to receive data, while the information entry logical unit must transmit data.

You can assign any reasonable physical unit to any logical unit by having an appropriate program in your microcomputer system to link the two. These programs are referred to as "device drivers". Device driver programs are illustrated functionally in Figure 3-4.

It is worth taking a moment to look at Figure 3-4 since it embodies a number of ideas and concepts which are difficult to grasp if you are a beginner.

The physical units shown at the bottom of Figure 3-4 are all pieces of hardware which you can see and touch. Every one of these physical units must have its own device driver program which caters to the specific needs of the physical unit. A device driver program is simply a computer program — a sequence of numbers, like any other program. What distinguishes a device driver program from any other program is the logic of the program — the fact that this logic harnesses the physical capabilities of a physical unit to perform the functions required of a logical unit.

Device driver programs are not new to you at this point; remember, Joe Bitburger eventually went out and bought a device driver program (in a read only memory chip) to control his teletype.

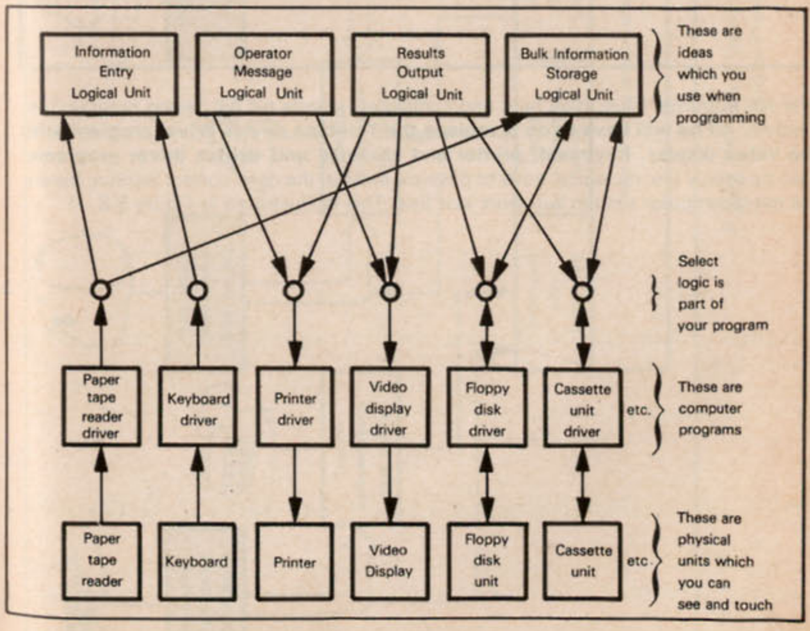
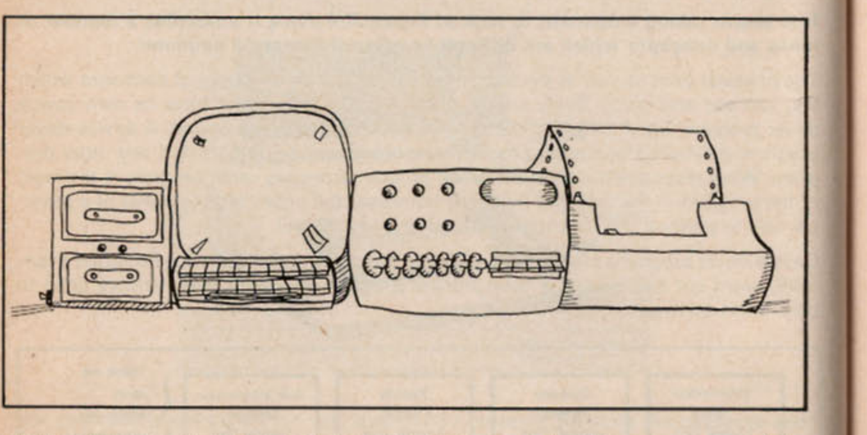

Information Entry Logical Unit
Operator Message Logical Unit
Results Output Logical Unit
Bulk Information Storage Logical Unit
These are ideas which you use when programming
Select logic is part of your program
Paper tape reader driver
Keyboard driver
Printer driver
Video display driver
Floppy disk driver
Cassette unit driver
etc.
These are computer programs
Paper tape reader
Keyboard
Printer
Video Display
Floppy disk unit
Cassette unit
etc.
These are physical units which you can see and touch

Figure 3-4. Logical Units And Physical Units Connected Using A Device Driver

A logical unit is nothing more than an idea. You can neither see nor touch a logical unit. For example, in Joe Bitburger's bill paying program, the paper tape reader becomes the bulk information storage logical unit since the paper tape reader provides Joe Bitburger's names and addresses. The concept of stored information, in Joe Bitburger's case, names and addresses, is an idea. This idea is connected to a physical reality, a paper tape reader, by the Teletype device driver program.

Now **suppose Joe Bitburger did a good job of writing his bill paying program. He would have written his bill paying program looking at all devices beyond the microcomputer as logical units. Now that Joe has lost his Teletype, suppose he replaces it with a video display terminal, a keyboard, a stand alone printer and a pair of cassette drives.**



Will Joe Bitburger have to go back and completely rewrite his bill paying program? Indeed no. **All he will have to do is replace the Teletype device driver program with the video display, keyboard, printer and cassette unit device driver programs;** then by simply linking logical units to physical units in the new, correct fashion, his entire microcomputer system will work just fine. This is illustrated in Figure 3-5.

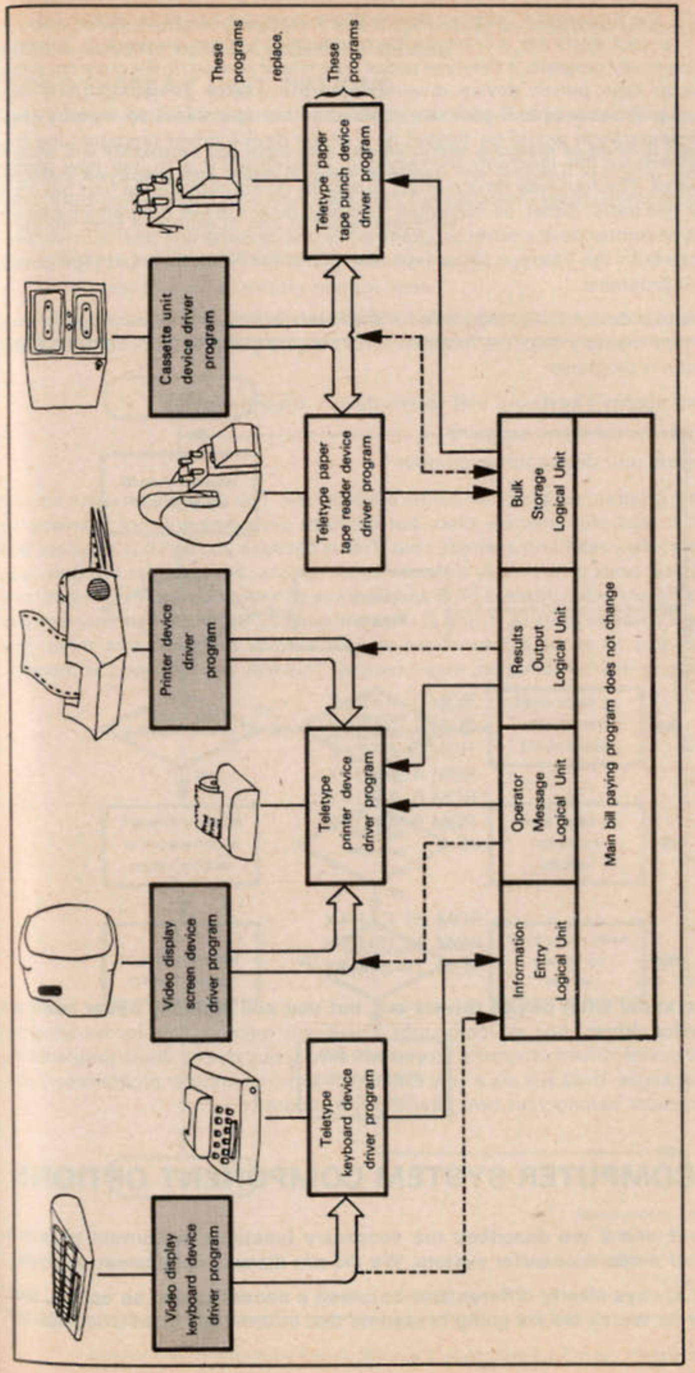

These programs replace.
These programs
Teletype paper tape punch device driver program
Cassette unit device driver program
Teletype paper tape reader device driver program
Printer device driver program
Teletype printer device driver program
Video display screen device driver program
Teletype keyboard device driver program
Video display keyboard device driver program
Bulk Storage Logical Unit
Results Output Logical Unit
Operator Message Logical Unit
Information Entry Logical Unit
Main bill paying program does not change

Figure 3-5. Using Device Driver Programs To Replace Physical Units

In Figure 3-5, Joe Bitburger's Teletype device driver program has been broken up into four parts. The four parts are a Teletype keyboard device driver program, a Teletype printer device driver program, a Teletype paper tape reader device driver program and a Teletype paper tape punch device driver program. In reality all four parts of the Teletype device driver program would be included on a single read-only memory chip, and the whole package would be treated as a single device driver program. Joe Bitburger must remove this read-only memory chip and replace it with four new device driver programs. The four new device driver programs do not replace the four old ones on a one for one basis; rather, as illustrated, two new device driver programs replace a single Teletype printer device driver program, while one cassette unit device driver program replaces both the Teletype paper tape reader and the Teletype paper tape punch device driver programs.

Whereas the four device driver programs for the Teletype looked like one program on a single read-only memory chip, the four new programs are going to look like three separate and distinct programs:

- The video display's keyboard and screen device driver programs
- The printer device driver program
- The cassette unit device driver program

Each separate program supports a separate physical unit. The separate program may be available as a read-only memory chip, but all three programs will not normally be available on a single read-only memory chip. This is because you can buy separate and distinct physical units in too many different combinations. Suppose, for example, you have three different video displays (A, B and C), three different printers (P, Q and R), and three different cassette units (X, Y and Z). You will need 27 different read-only memory chips to give you all of the combinations of device driver programs that these nine physical units, in any combination, might require. This may be illustrated as follows:

ROM 1	A+P+X
ROM 2	A+P+Y
ROM 3	A+P+Z
ROM 4	A+Q+X
ROM 5	A+Q+Y
ROM 6	A+Q+Z
	-
	-
	-
ROM 25	C+R+X
ROM 26	C+R+Y
ROM 27	C+R+Z

You need to know what device drivers are, but you will probably never need to create a device driver. Just as Joe bought a read-only memory chip for his teletype bootstrap (or device driver) programs, so you will simply buy device driver programs as prewritten packages. Until you are a very experienced microcomputer programmer, you will never consider writing your own device driver program.

MICROCOMPUTER SYSTEM COMPONENT OPTIONS

In Chapters 1 and 2 we described the necessary functions performed by each component of a microcomputer system. We did not discuss component options.

You cannot always clearly differentiate between a necessity and an option, but for better or for worse we are going to assume that information on microcomputer system components in Chapters 1 and 2 constitute necessities, whereas what we are now about to describe constitute options.

VIDEO DISPLAY UNIT OPTIONS

Let us begin by looking at options which you may find in video display units.

Figure 3-6 illustrates the logic of a device driver program to control the most elementary keyboard and video display terminal. The program in Figure 3-6 does very little; the program waits for a keystroke from the keyboard; upon detecting a keystroke, program logic branches in one of three directions:

- It displays a displayable character
- It responds to a video display control code
- It transmits any microcomputer system control code to the microcomputer

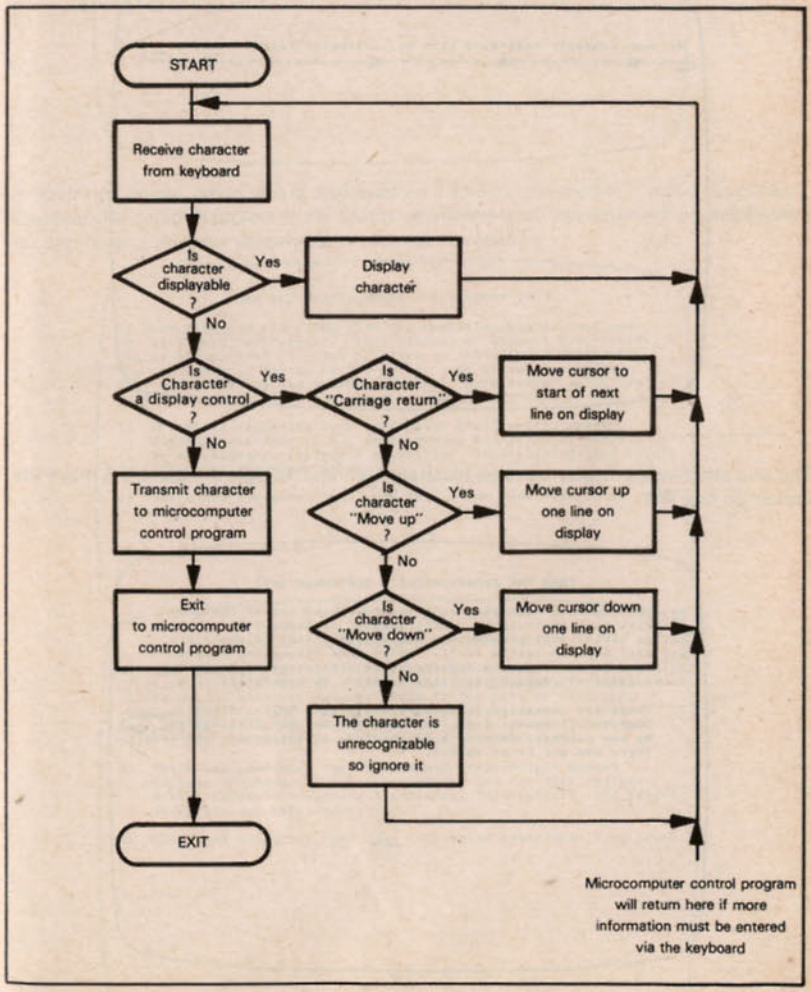

START
Receive character from keyboard
Is character displayable ?
Yes
No
Display character
Is Character a display control ?
Yes
No
Is Character "Carriage return" ?
Yes
No
Move cursor to start of next line on display
Is character "Move up" ?
Yes
No
Move cursor up one line on display
Is character "Move down" ?
Yes
No
Move cursor down one line on display
The character is unrecognizable so ignore it
Transmit character to microcomputer control program
Exit to microcomputer control program
EXIT
Microcomputer control program will return here if more information must be entered via the keyboard

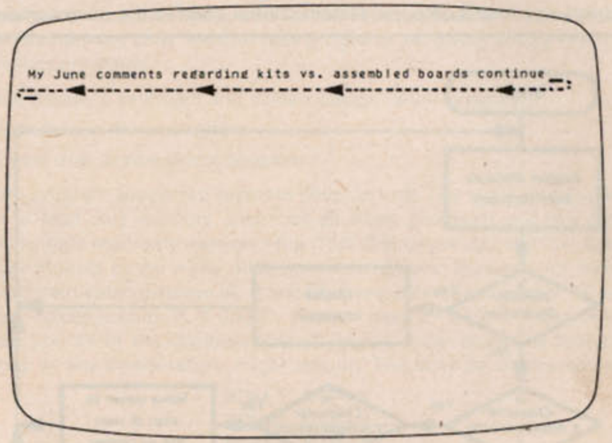
Figure 3-6. Flowchart For A Simple Video Display Driver Program

If the keystroke enters a letter of the alphabet, a number or any other displayable character, the program writes the character back to the video display; remember this is called "echo".

SCREEN CURSOR

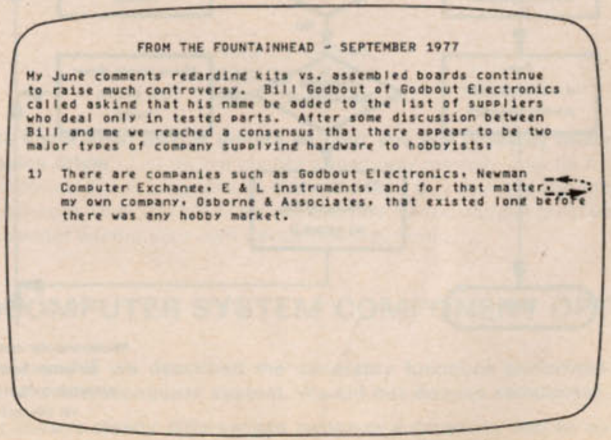
The keystroke may constitute a video display control. Controls illustrated in Figure 3-6 include a carriage return, moving up one line or moving down one line. Since there is no print mechanism to tell you where you are on the display, all displays use what is called a "cursor"; a cursor is a small spot, or square of light sitting where the next character will be displayed.

When you press the carriage return key at the video display terminal, the cursor moves to the beginning of the next line down:



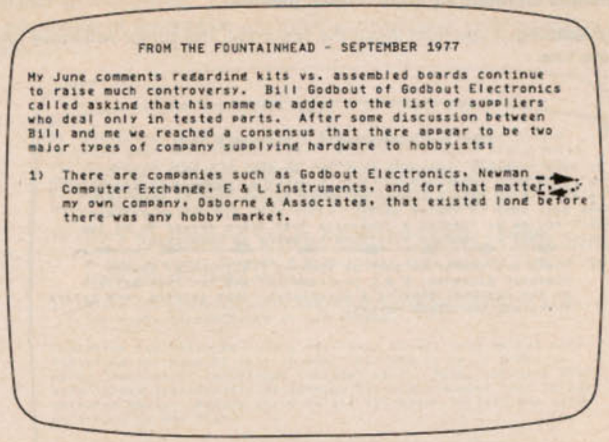
```
My June comments regarding kits vs. assembled boards continue
```

The two other video display controls illustrated within the logic of Figure 3-6 move the cursor up one line:



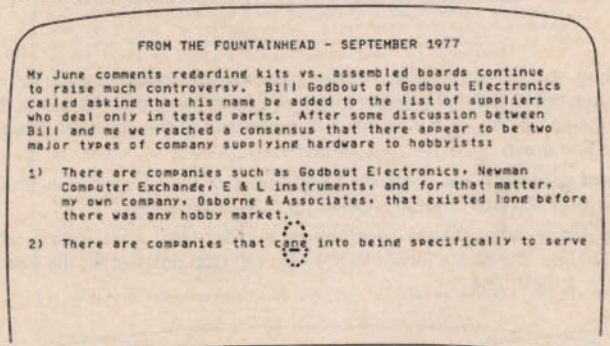
```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.
```

Or down one line:

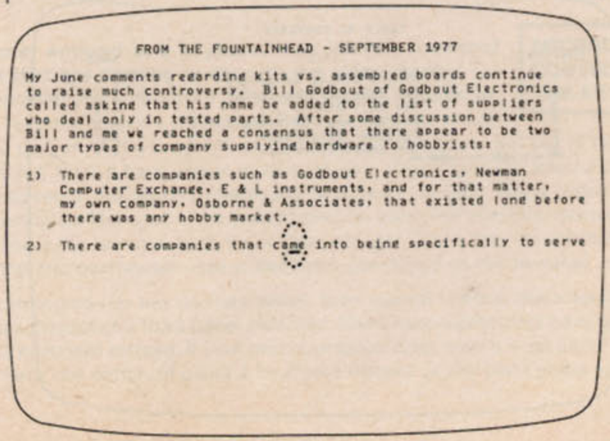


```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.
```

As illustrated above, when text is displayed on a screen, you modify it using the cursor. Wherever the cursor happens to be, that is where the next character will be displayed. You can correct an error anywhere in your text as follows:



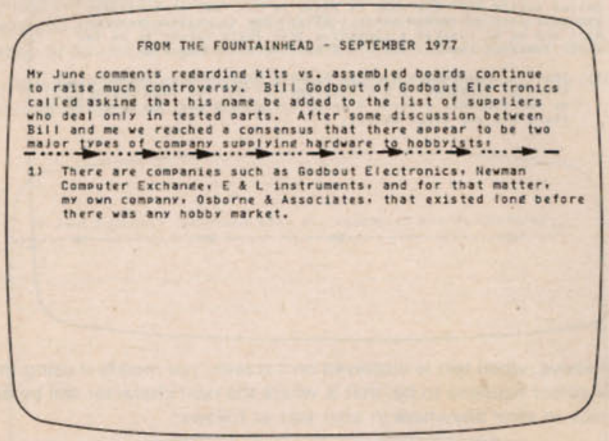
```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.

2)  There are companies that cane into being specifically to serve
```



```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.

2)  There are companies that came into being specifically to serve
```

A number of video display controls which are not shown in the logic of Figure 3-6 are nevertheless common options. These include:

1) **Forward spacing.** If you hold down the space bar, the cursor will move towards the end of the line:

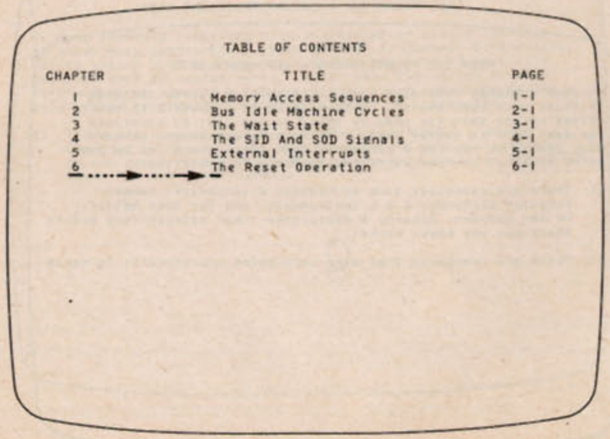


```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.
```

Depending on how your video display has been designed, the cursor may stop when it reaches the end of a line, it may move from the end of the line to the beginning of the same line, or it may move from the end of the line to the beginning of the next line down.

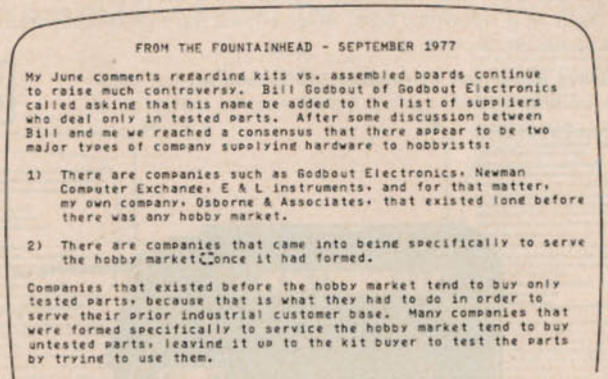
2) **Backward spacing.** If you press the backspace key, the cursor simply moves in the opposite direction from that illustrated for forward spacing.

3) **Tabbing.** Many video display terminals let you set tabs; after setting tabs, if you press a tab key, the cursor jumps to the next tab stop position on the video display within the current line:

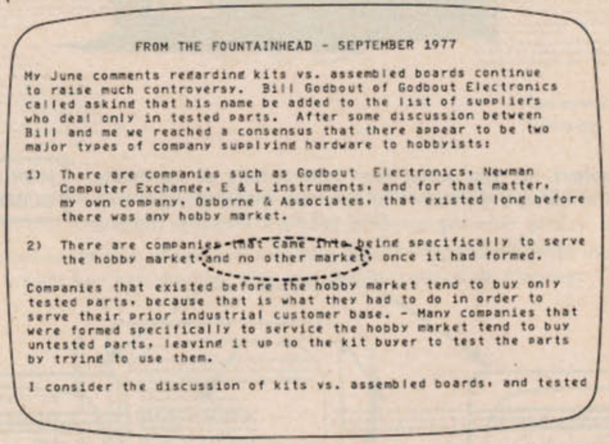


```
                    TABLE OF CONTENTS

CHAPTER                    TITLE                         PAGE

   1               Memory Access Sequences                1-1
   2               Bus Idle Machine Cycles                2-1
   3               The Wait State                         3-1
   4               The SID And SOD Signals                4-1
   5               External Interrupts                    5-1
   6               The Reset Operation                    6-1
```

4) **Text insertion and deletion.** Video displays have a very important advantage as compared to any printer; you can electronically move text. Many video displays take advantage of this and let you insert or delete text. Inserted text is illustrated as follows:



```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.

2)  There are companies that came into being specifically to serve
    the hobby market once it had formed.

Companies that existed before the hobby market tend to buy only
tested parts, because that is what they had to do in order to
serve their prior industrial customer base.  Many companies that
were formed specifically to service the hobby market tend to buy
untested parts, leaving it up to the kit buyer to test the parts
by trying to use them.
```



```
                FROM THE FOUNTAINHEAD - SEPTEMBER 1977

My June comments regarding kits vs. assembled boards continue
to raise much controversy.  Bill Godbout of Godbout Electronics
called asking that his name be added to the list of suppliers
who deal only in tested parts.  After some discussion between
Bill and me we reached a consensus that there appear to be two
major types of company supplying hardware to hobbyists:

1)  There are companies such as Godbout  Electronics, Newman
    Computer Exchange, E & L instruments, and for that matter,
    my own company, Osborne & Associates, that existed long before
    there was any hobby market.

2)  There are companies that came into being specifically to serve
    the hobby market and no other market, once it had formed.

Companies that existed before the hobby market tend to buy only
tested parts, because that is what they had to do in order to
serve their prior industrial customer base. - Many companies that
were formed specifically to service the hobby market tend to buy
untested parts, leaving it up to the kit buyer to test the parts
by trying to use them.

I consider the discussion of kits vs. assembled boards, and tested
```

MICROCOMPUTER CONTROL FROM THE KEYBOARD

A character entered at a keyboard may identify a control function that has nothing to do with the video display. Do you remember Joe Bitburger's programs always contain an "Escape" or error recovery character? This is a character which restarts a portion of the program, or otherwise allows Joe to get out of trouble. Video terminals will always have one or more such keys. These keys are special because they do not represent displayable characters or screen controls. Some video display terminals dedicate a few special controls to specific tasks. For example, there may be a key to disconnect the terminal from the microcomputer. There may be another key to stop or start the microcomputer once it has been connected to the terminal.

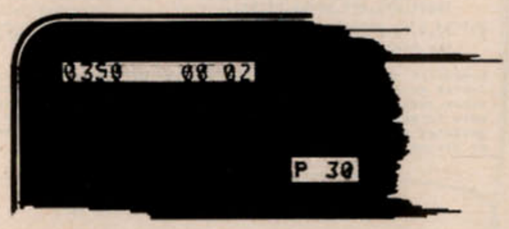
If your microcomputer has no front panel, then special control characters entered at a keyboard substitute for front panel switches. Many microcomputers have no front panel and use a keyboard instead. If your microcomputer does have a front panel, you will frequently have the option of using a keyboard instead of the front panel switches.

UPPER AND LOWER CASE DISPLAYS

Upper and lower case displays are another video display option. The simplest and cheapest video display terminals display upper case letters only; more expensive video display terminals display upper and lower case characters. The keyboard will have a shift key, much as a typewriter does, which allows you to move between upper and lower case character displays.

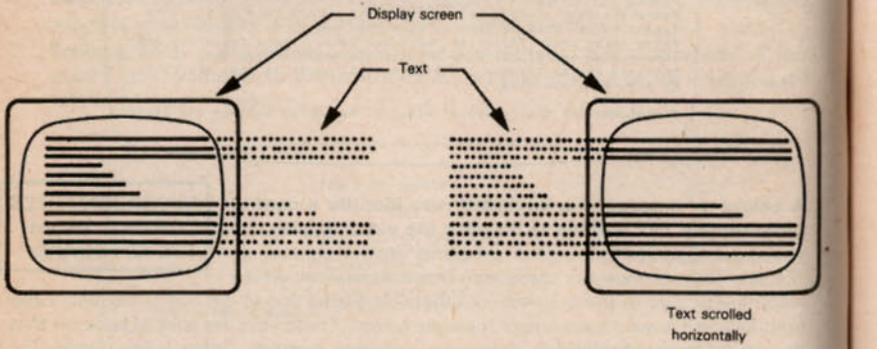
REVERSE DISPLAY

Some displays allow you to reverse the screen so that black characters are displayed on a white background for part or all of the screen:



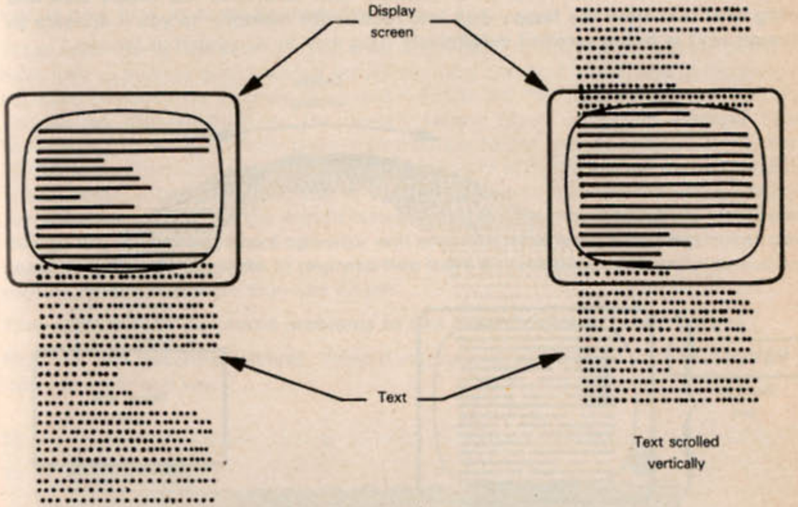
HORIZONTAL SCROLLING

Some displays let you scroll text horizontally or vertically. Horizontal scrolling lets you look at text lines that are longer than the screen is wide. Now the screen acts as a window on the line. This may be illustrated as follows:



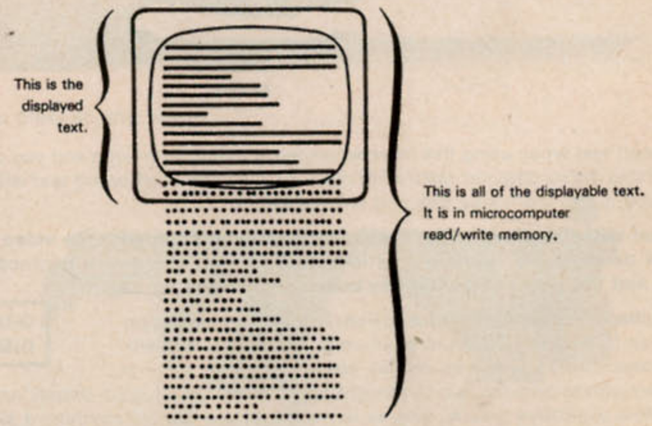
VERTICAL SCROLLING

Horizontal scrolling is not a common video display option. **Vertical scrolling is much more common.** In this case you imagine the text as having more lines than the screen will display; the screen allows you to move the text up and down. This may be illustrated as follows:



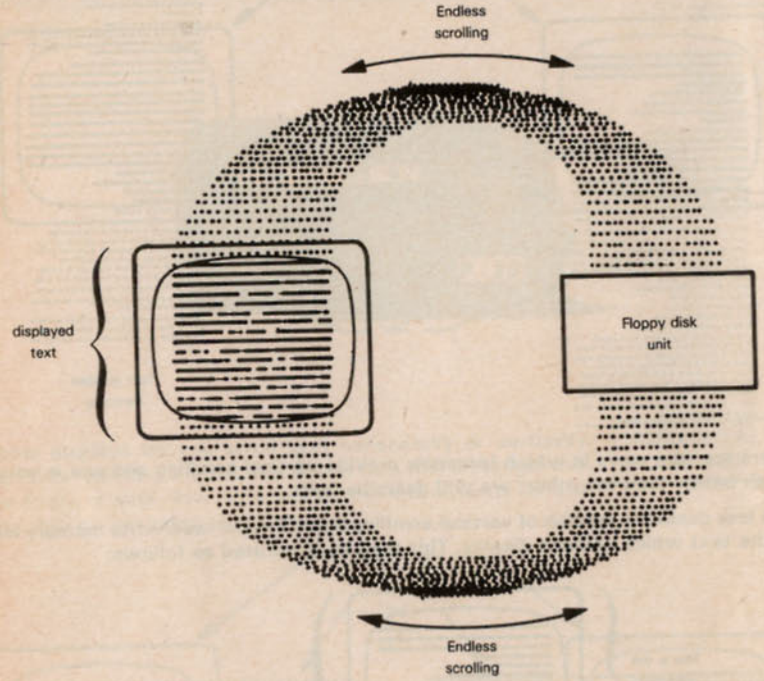
There are two ways in which terminals provide vertical scrolling and one is very much better than the other; we will describe both.

The less desirable version of vertical scrolling holds in local read/write memory all of the text which you can display. This may be illustrated as follows:


This is the displayed text.
This is all of the displayable text. It is in microcomputer read/write memory.

If you have this less desirable version of scrolling, you cannot scroll above the top, or below the bottom of the text currently held in read/write memory. If you insert text and overflow the available read/write memory, then you will simply lose information from the top or bottom of your text.

If your microcomputer system has a floppy disk unit, then a well designed scrolling option will connect the floppy disk unit to the read/write memory within which displayed text is being stored. Now if you scroll above or below the text in read/write memory, programs within the microcomputer system will automatically store some of the read/write memory text back into the floppy disk and bring new text from the floppy disk into read/write memory; to you it appears as though text is being scrolled indefinitely. This may be illustrated as follows:



If you insert text when using this more advanced scrolling technique and you overflow the available microcomputer read/write memory, then the overflowing text will simply get written out to the floppy disk and will not be lost.

Note that these scrolling options really have nothing to do with the video display terminal or keyboard; they are options that rely on the entire microcomputer system and the way in which it has been programmed for you.

GRAPHIC DISPLAY

Graphic display is another useful option. A graphic video display allows you to display pictures in addition to characters. Inexpensive graphic display terminals display black and white, but not grey; they create pictures out of straight lines, circles, dots, solid shapes and block forms. More expensive graphic display terminals let you display anything a television screen can, and with finer detail.

There are color video displays as well as black and white video displays, just as there are color and black and white television sets. A color video display gives you the added capability of specifying the color in which a character or graphic segment will appear.

LIGHT PEN

Some very expensive video displays allow you to draw on the display with a "light pen". The electronic logic behind the video display "remembers" the points on the screen which the light pen beam illuminates.

KEYBOARD OPTIONS

Let us now look at some keyboard options.

What about the speed at which you enter keystrokes? Is the microcomputer going to have time to process one character before you type another? In the case of the simple keyboard device driver program illustrated in Figure 3-6, the answer is almost certainly yes. As we discussed earlier, the world's fastest typist enters (on average) nine keystrokes per second, which gives your microcomputer time to execute more than 20,000 instructions between keystrokes. The program in Figure 3-6 will use 100 or 200 of the 20,000 instructions, and that is all. But nine keystrokes per second represents average keystroke rate for the world's fastest typist, not maximum keystroke rate. **Even though the fastest keyboard operator will enter no more than nine keystrokes per second, it is quite possible to depress two keys almost simultaneously,** providing you are typing with more than one finger.

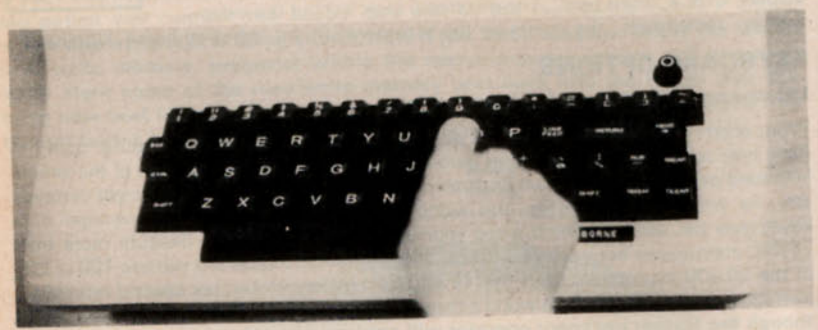
Two options help you avoid problems in this area: "rollover" and "buffers".

ROLLOVER

First we will describe rollover. What if you type keys so fast that you enter one key:

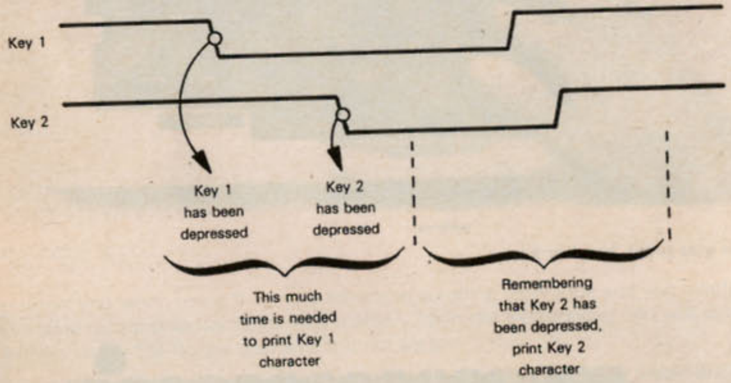
Then you press another key:

Then you release the first key:



The instant at which a keystroke is recorded is very important. In any ordinary typewriter, once you press a key, you type the character corresponding to the depressed key. If you press a second overlapping key, as illustrated above, an electric typewriter will lock out the second keystroke and nothing will happen. In a mechanical typewriter the second printhammer will hit the back of the first printhammer, and the second character will not be printed.

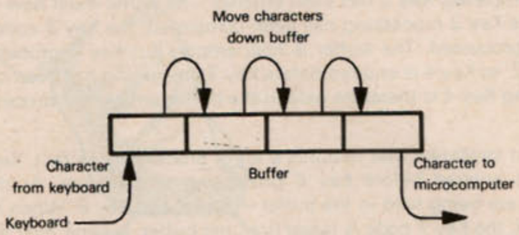
But there is no need to be so restrictive in the world of electronics. While one key is depressed, electronic logic can detect another key being depressed, and can "remember" the second key depression until the first key character has been printed. This may be illustrated as follows:


Key 1
Key 2
Key 1 has been depressed
Key 2 has been depressed
This much time is needed to print Key 1 character
Remembering that Key 2 has been depressed, print Key 2 character

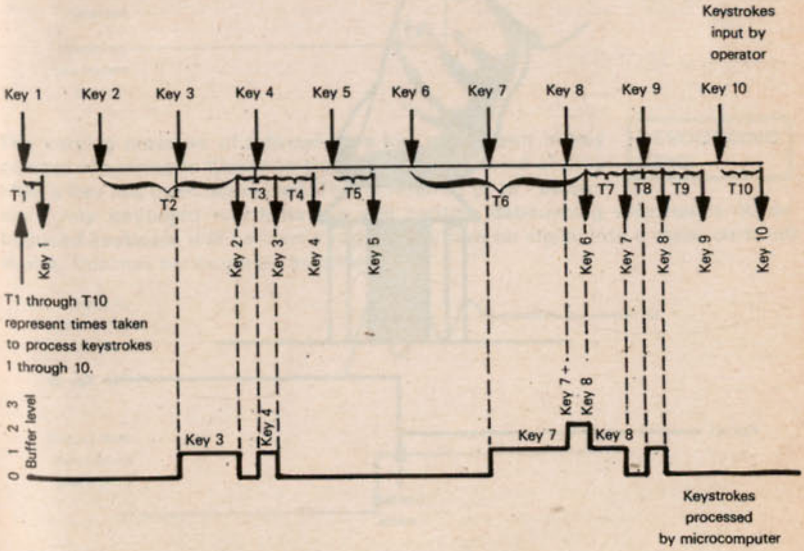
Rollover is a highly desirable option to have in any keyboard, since it has nothing to do with computer execution speed. If you have agile fingers, you might frequently find yourself overlapping keystrokes. Unless your keyboard has rollover, it will lose the second keystroke of the overlap.

KEYBOARD BUFFER

The next technique used to make keyboards error proof is to have some storage location within which character codes are stored while waiting to be transmitted to the microcomputer. This may be illustrated as follows:



The storage location illustrated above is referred to as a buffer. Although the buffer is shown having room to store four characters, a real buffer may have any number of characters, typically ranging between two and eight. A buffer gives the microcomputer more time to handle occasional characters that require long response times. Now for the simple keyboard driver program illustrated in Figure 3-6 a buffer could serve no possible useful purpose. But as a keyboard driver program becomes more complex, it is possible for a few keystrokes to require more processing time than is available. Now the buffer comes into play. If the average keystroke requires more than 20,000 instructions worth of processing, the buffer will fill up and overflow, however large it is. But if only some keystrokes require more than 20,000 instructions worth of processing, then the buffer will work just fine. This may be illustrated as follows:



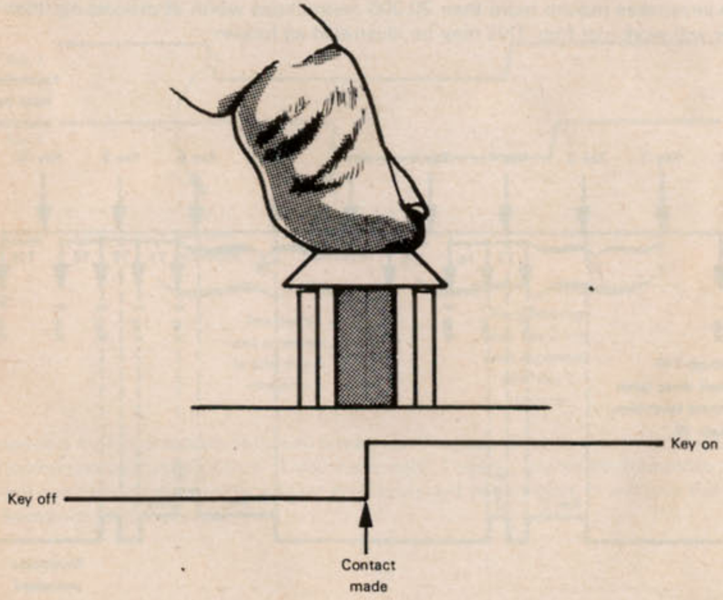
Let us examine the illustration above. Keystrokes are shown being input at regular time intervals (at the top of the illustration).

Key 1 is processed very quickly, during time interval T1; the buffer is not needed. Key 2, on the other hand, requires substantial processing time; in fact, before Key 2 processing has been completed, Key 3 has been entered. The buffer must now hold the Key 3 code. As soon as Key 2 processing has been completed, the Key 3 code is taken from the buffer and processed. The buffer is now empty. But Key 3 processing has been delayed by Key 2, so Key 4 is entered before Key 3 processing has been completed. The code representing Key 4 is therefore held in the buffer until Key 3 processing has been completed.

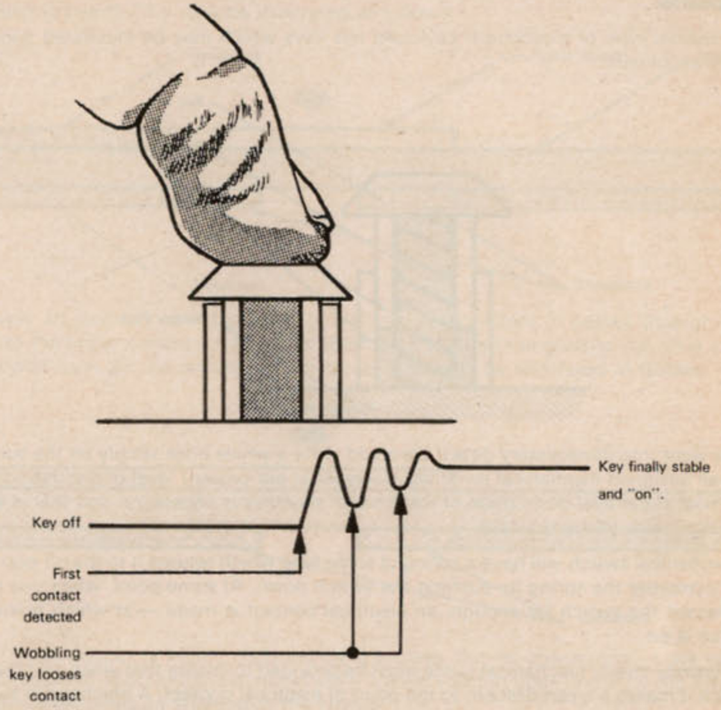
Key 6 is the next keystroke that requires a lot of processing. In fact, Key 7 and Key 8 have both been entered before Key 6 processing is complete. At this point, two character codes are being held in the buffer — Key 7 and Key 8. When Key 6 processing is completed, the Key 7 code is taken from the buffer, leaving only the Key 8 code. Key 7 processing is completed before Key 9 is entered, therefore the buffer is empty. But Key 8 processing, having itself been delayed, causes the Key 9 code to be stored for a short period of time.

Had it not been for the buffer, Key 3, Key 7 and Key 8 would all have been missed.

Our discussion of multiple keystrokes raises an obscure aspect of keyboards which we must consider: exactly when is a key going to be detected as depressed? Answering this question is not quite as simple as it sounds. You might simply assume that pressing a key causes some type of contact to close, at which point the key is considered depressed:

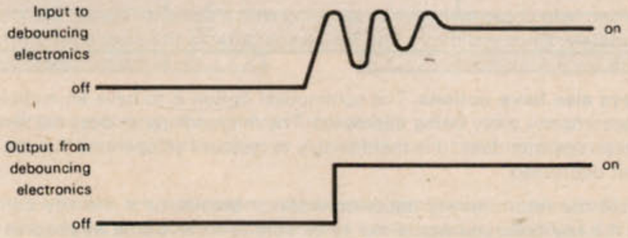


Unfortunately, electrical contacts are not always clean. The simple contact shown above might more accurately occur as follows:



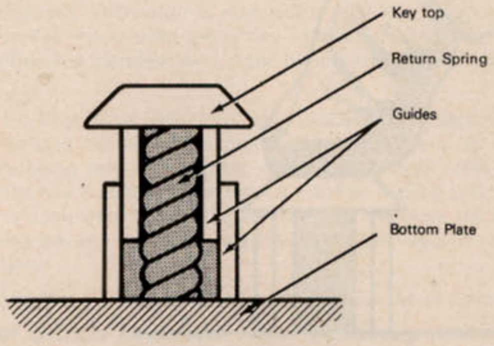
DEBOUNCING KEYS

The varying pressure of a finger on a key may result in key contact appearing to flicker on and off for some period of time after a key has been depressed. This is referred to as "bouncing". Any keyboard worth having will contain debouncing electronics. A debounced keyboard will convert a ragged key turn on signal into a clean on-to-off signal. This may be illustrated as follows:


Input to debouncing electronics
on
off
Output from debouncing electronics
on
off

Rollover, buffers and debouncing are electronic options associated with keyboards. But it is equally important that you evaluate mechanical aspects of keyboards.

The normal type of mechanical keyboard has keys which may be illustrated conceptually as follows:



If we went into unnecessary detail, we could write a whole book simply on the subject of well designed mechanical switches. However at our present level of discussion only the most superficial description of mechanical switches is necessary; and that is what the illustration above provides.

A mechanical switch will have a spring of some type which returns it to the off position. You compress the spring by pushing the switch down. At some point, when you have depressed the switch far enough, an electrical contact is made — at which point the switch is on.

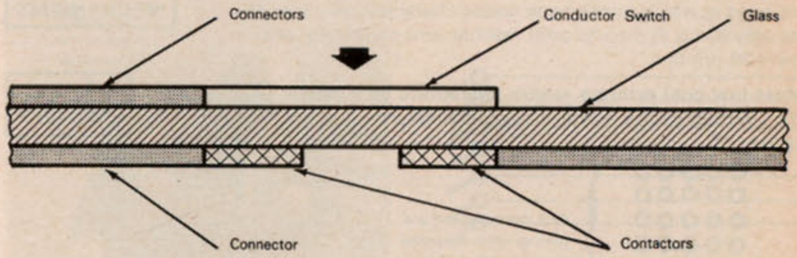
A relatively sturdy mechanical guide must be provided to ensure that when you press a switch it makes a clean descent to the point of electrical contact. A mechanical switch may appear to be a very elementary device, but in reality it is not. For example, no one ever presses a switch straight down; invariably the angle at which your finger touches a switch causes the switch to be pushed to one side, as well as being pushed down. The switch must be designed with this in mind. Consider also the electrical contact; it might appear that an electrical contact is generated very simply by having metal on the key (or key stem) make contact with metal on the guide (or bottom plate). But this type of dry contact, if not designed well, will give you nothing but trouble. Just a small amount of corrosion on the metal surfaces is sufficient to render the switch ineffective. Sometimes you can help cheap switches by spraying with a suitable cleaner. But do not do so indiscriminately, since you might find the cleaner attacks the plastic used to construct the switch.

Mechanical keys also have options. The commonest option is to have an audible or tactile "click" accompany a key being depressed. The microcomputer does not need a "click", the human operator does; it is there simply to reassure an operator that the key has indeed been depressed.

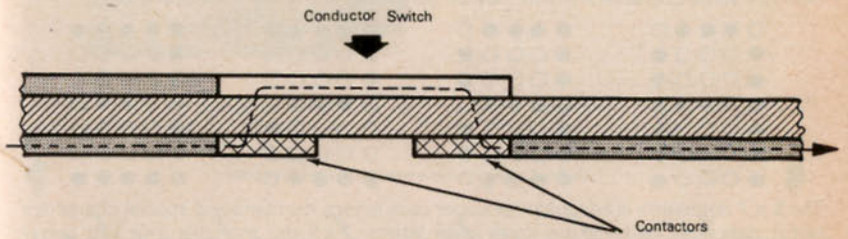
A key may fail if the return spring becomes weak or breaks, or if the key contact becomes dirty. If a key fails, you would like to be able to fix it. **Some keyboards are built as integral units, such that even one key failing requires the whole keyboard to be replaced. These integral keyboards are cheaper to buy in the first place, but obviously more expensive to replace if anything goes wrong.**

TOUCH SWITCHES

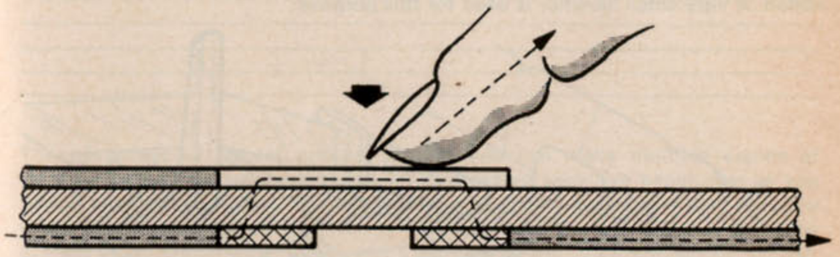
In the future, mechanical keyboards may conceivably be replaced by touch switches. Touch switches consist of glass or other inert plates upon which patterns are painted using electrical conducting paints. This may be illustrated as follows:



Normally an electric current is input to one of the contactors; it passes through the glass to the upper conductor, and from there back through the glass to the other contactor and back out via the second connector. This may be illustrated as follows:



The current output is highly distorted by the glass, but its characteristics are quite recognizable to appropriate electronic logic. When you put your finger on the switch your body becomes an addition to the circuit:



Your finger touching the switch dramatically alters the current output; external electronic logic detects this change in current output and translates it into the switch being "on".

Touch switches are not commonly seen in appliances or electronic devices, but they are not new. Touch switches have been used in elevators for years.

Touch switches are not popular at the present time but because they are rugged, low cost and very reliable they are likely to replace mechanical keyboards in the future. The real problem with touch switches at the present time is the fact that when you touch one it makes no audible or tactile click and that upsets many longtime keyboard users.

PRINTER OPTIONS

There are three printer options that contribute most significantly to price: print mechanism, print line (and paper) width and print speed.

PRINT MECHANISMS

Let us first consider print mechanisms. There are innumerable ways in which printers can create characters on paper. We are only going to discuss print mechanisms commonly seen in low cost printers.

MATRIX PRINTERS

Most low cost printers create characters as a matrix of dots. The simplest matrix printers create characters out of a 5 by 7 dot matrix:

```
○○○○○
○○○○○
○○○○○
○○○○○
○○○○○
○○○○○
○○○○○
```

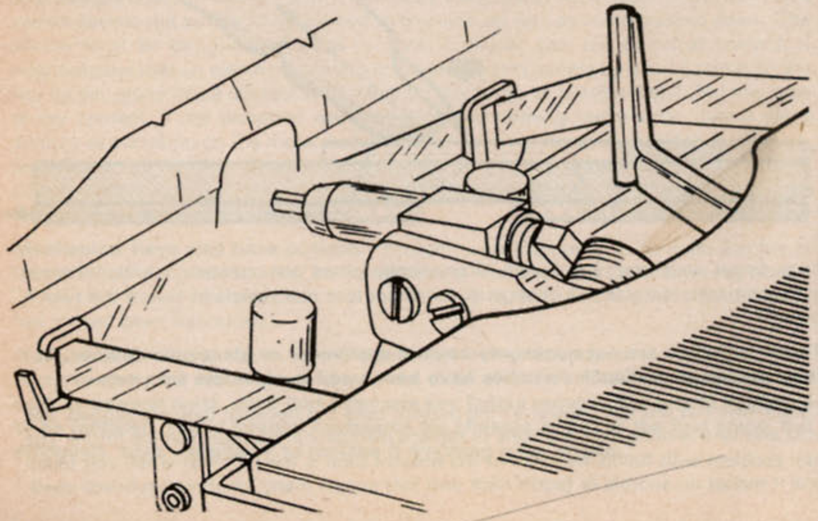
Each matrix position is a point at which the printer can print a single dot.

Here are some characters which may be generated out of a 5 x 7 dot matrix:

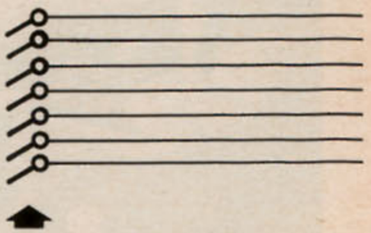
```
○●●●○   ●●●●○   ○●●●●   ●●●●●
●○○○●   ●○○○●   ●○○○○   ●○○○○
●○○○●   ●○○○●   ●○○○○   ●○○○○
●●●●●   ●●●●○   ○●●●○   ●●●●○
●○○○●   ●○○○●   ○○○○●   ●○○○○
●○○○●   ●○○○●   ○○○○●   ●○○○○
●○○○●   ●●●●○   ●●●●○   ●●●●●
```

The 5 x 7 dot matrix is adequate for upper case letters, numbers and special characters but it generates odd looking lower case letters. 7 x 9 dot matrices give you better resolution.

Dot matrix printers may be impact or non-impact printers. An impact printer, as its name would imply, makes marks on paper by hitting the paper through an inked ribbon. A very small hammer is used for this purpose:

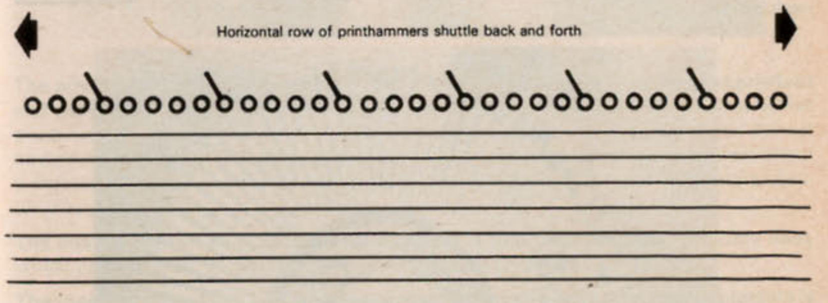


Two techniques are commonly used to create dot matrix characters. One technique uses a vertical line of hammers which sweep across the paper, line by line, creating characters:



7 Printhammers sweep across the paper creating a single line of characters.

Another printer has a horizontal row of hammers which shuttle backwards and forwards creating characters as follows:



The advantage of impact printers is that you can make multiple copies of whatever you print, using carbon paper or pressure sensitive paper, just as you would with a typewriter. The disadvantage of impact printers is that they are relatively slow since printing involves moving printhammers and firing them. The average matrix printer can print anywhere between 30 and 300 characters per second.

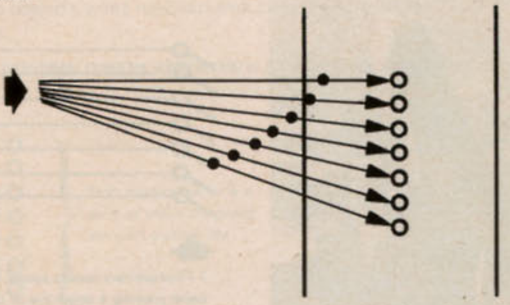
There are also a number of non-impact matrix printers which create marks on paper using a variety of ingenious techniques, none of which involve actually hitting the paper with a hard object. These non-impact matrix printers are very fast because they do not move mechanical parts in order to print dots; but you cannot use them to print multiple copies since nothing hits the paper hard enough to effect copies. **Some very inexpensive non-impact matrix printers use special heat-sensitive paper.** These printers have a print head which moves across the paper sending out pulses of heat wherever a dot should be created.

Printers which use this technique are called thermal printers.

THERMAL PRINTERS

INK JET PRINTERS

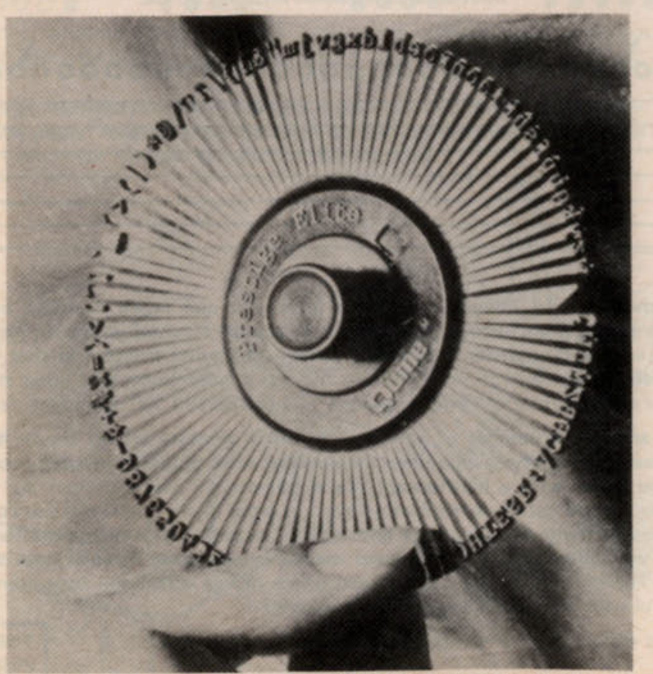
Ink jet printers are the most common non-impact printers available today. Believe it or not, an ink jet printer prints by firing a high speed stream of tiny ink droplets at the paper. Each drop of ink carries a small electric charge. The charged ink drop passes through a magnetic field which deflects it so that it hits the paper at just the right position. This may be illustrated as follows:



The drops of ink which an ink jet printer fires are tiny and dry almost immediately. Ink droplets are fired so quickly that an ink jet printer can print thousands of characters per second.

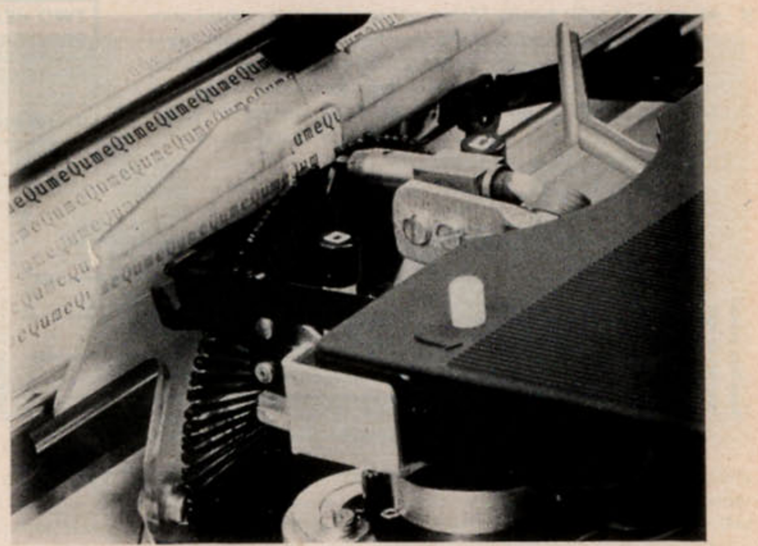
DAISY WHEEL PRINTER

Two manufacturers, Diablo and Qume Corporation, make a "daisy wheel" printer. This printer uses a print element which looks like a 96-petaled daisy. There is a character on the end of each petal:



COURTESY OF QUME CORPORATION

Daisy wheel printers spin the daisy wheel print element at high speed, while moving it across the paper. Whenever the correct petal is in front of a character position, a print hammer is fired, pushing the petal against a ribbon — which prints the character upon the paper. This may be illustrated as follows:



The advantages of daisy wheel printers are that they create very elegant printout and they are very reliable and relatively inexpensive. Text printed by a daisy wheel printer is as attractive as that printed by a Selectric or Executive typewriter; that cannot be said for matrix printers. Daisy wheel printers can work for hundreds of hours without a mechanical failure, whereas typewriters, when modified to work as printers, continuously malfunction.

The one disadvantage of daisy wheel printers is that they are slow. Typically daisy wheel printers work at speeds between 30 and 45 characters per second.

The reason daisy wheel printers create more attractive text than matrix printers is because the daisy wheel printers generate solid characters.

BELT PRINTERS

Another more expensive way of generating solid characters is to use a steel belt. A belt printer will rotate the steel belt at high speed, with one or more hammers hitting the steel belt whenever the appropriate character is in front of the correct character position.

The most popular belt printer available today is part of the Teletype model 40 terminal.

All of the low cost printers available use one of the printing techniques described above. The lowest cost printers use impact matrix print techniques. Daisy wheel printers and belt printers are more expensive.

COMMON PRINTER OPTIONS

Whatever the printer mechanism, there are some common options which you should know about when choosing a printer:

PRINT LINE LENGTH

1) **Print line length.** 80-character lines are most common. Some printers generate 132-character lines. Some low cost printers generate lines as short as 20 characters; they use very narrow paper strips.

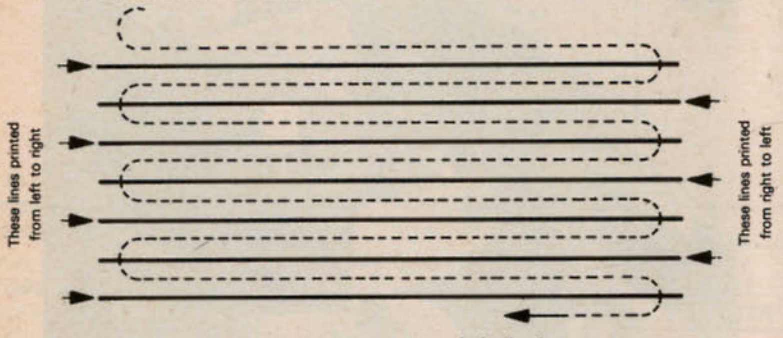
PRINTER CHARACTER SET

2) **Upper case only or upper and lower case.** Inexpensive printers can only print upper case letters of the alphabet. More expensive printers can print upper case and lower case characters.

TWO HEAD PRINTERS

REVERSED LINE PRINTING

3) **Number of print heads.** Matrix printers and daisy wheel printers that sweep across the paper horizontally usually have higher cost, double speed versions. Double speed versions use two print heads, each of which sweeps across half of the line. Most printers with print heads that sweep across the paper horizontally increase their print speed by going backwards and forwards across the paper:


These lines printed from left to right
These lines printed from right to left

print head path

In contrast, a typewriter always prints from left to right, performing a carriage return at the end of every line.

PROPORTIONAL SPACING

4) **Proportional spacing.** Currently this option is available only with daisy wheel printers. Proportionally spaced text allows each character a different width, depending on the shape of the character. Text in this book is proportionally spaced.Here is a line of text where each character occupies the same width on the page:

```
(2)  The ending location is an address
```

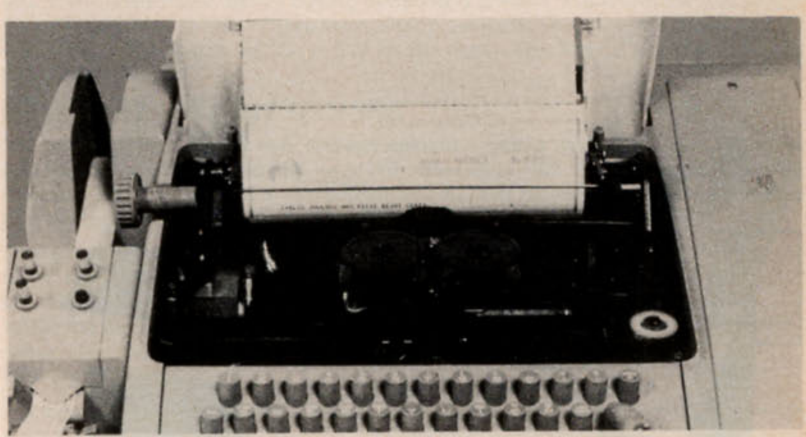
Here is the same text with proportional spacing:

(2) The ending location is an address

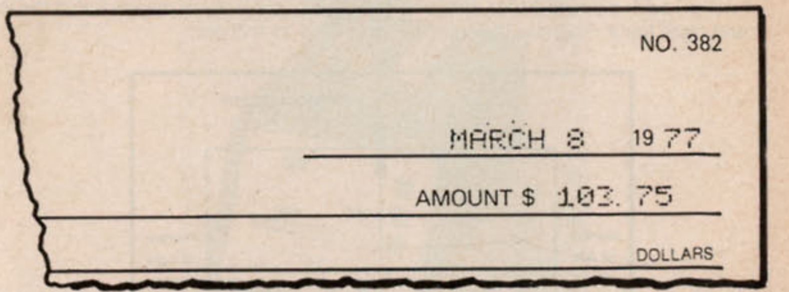
5) **Right adjusting.** If you look at the text in this book, with the exception of this paragraph, both margins form exactly straight lines. You would expect to start every line at the same point on the paper, but most low cost printers will not end every line at the same place. "Right adjusting" is the name given to printing text with lines that begin and end in a straight line.

FRICTION FEED PAPER ADVANCE

6) **Sprocket feed.** If you are going to print continuous forms, such as Joe Bitburger's checks, you can quickly run into paper positioning problems. Low cost printers advance paper using friction feed: rollers on both sides of the paper force it to advance:



Even a small amount of slippage in a friction feed, as illustrated above, can eventually result in paper not being where it should be; successive lines are then printed further and further away from the desired location. In a check, for example, there is a line printed on the check where the dollar amount is supposed to appear. If every check slips by 1/20 inch, that is to say, it advances 1/20 inch less than it should, then after five checks have been printed the dollar amount will appear 1/4 inch above its assigned location.

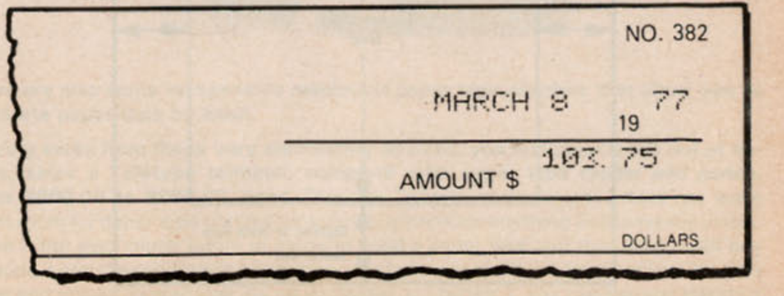


NO. 382

MARCH 8 19 77

AMOUNT $ 103. 75

DOLLARS



NO. 382

MARCH 8 19 77

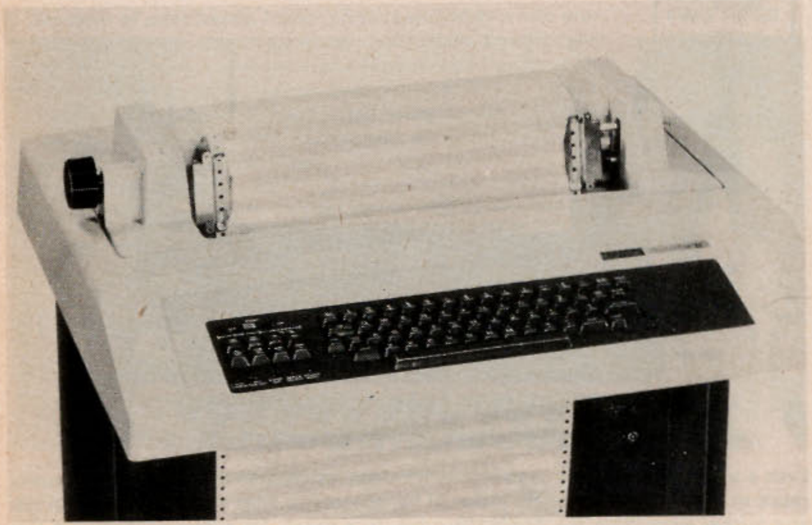
103. 75

AMOUNT $

DOLLARS

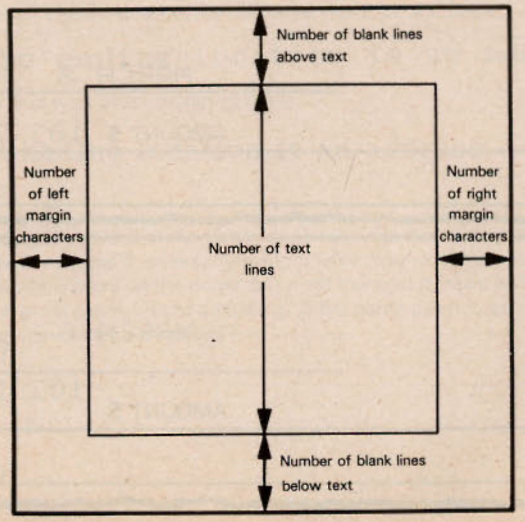
SPROCKET FEED PAPER ADVANCE

Before very long the checks will become illegible. Whenever you are going to use continuous forms on a printer you should have a sprocket feed:



A sprocket feed will advance the paper exactly as far as it should be advanced every time.

7) **Paper format.** Some printers give you mechanical methods of specifying printed page format. Printed page format includes such things as the number of printed lines on each piece of paper, the width of the top and bottom margins plus the left and right hand margins. This may be illustrated as follows:



Number of blank lines above text
Number of left margin characters
Number of right margin characters
Number of text lines
Number of blank lines below text

After evaluating all of the printer options we have described, you must still look at the ruggedness of the printer. Remember, a printer is a mechanical device and it is the mechanical devices of your computer system that will usually break down. If a printer has not been built ruggedly, it does not matter how many fancy features it offers you, it will break down. Before buying inexpensive printers, make sure that the printer manufacturer has not taken cheap short cuts. In particular, watch out for plastic parts in the printer mechanisms. Plastic is fine for the outer case, but the print mechanisms should be all metal.

BULK STORAGE UNIT OPTIONS

We have discussed three bulk storage devices — paper tape units, cassette units and floppy disk units. Rigid disk units are rarely used with microcomputer systems because they overwhelm a microcomputer, both in terms of data transfer speed and the amount of information stored.

PAPER TAPE UNIT OPTIONS

The only paper tape readers and punches you will see any more are low performance, very inexpensive units. This is because cassette units are so clearly superior that no one is going to pay the same price for a paper tape unit as a cassette unit. Consequently, the options available to you if you want to use paper tape are quite limited.

The Heathkit microcomputer systems use a paper tape punch; it is one of the few options you have.

There are also some inexpensive mechnical paper tape punches that allow you to perforate paper tape by hand.

Moving away from these very elementary devices, you will quickly run out of options, since a Teletype terminal, complete with paper tape reader and punch, costs $600.00 to $900.00, used. This low Teletype terminal price does not leave much room for paper tape unit manufacturers to produce anything economically viable. There is not even much future in trying to build a paper tape unit that people will buy because it punches and reads paper tape very fast; you can buy cassette drives that are faster and just as inexpensive. As we explained in Chapter 1, you can use any standard supermarket cassette unit together with appropriate interface electronics as a bulk storage device within a microcomputer system. The cassette drive itself, even if it is quite good, will cost less than a hundred dollars; cassette drive interface electronics, already inexpensive, are rapidly going down in price. Thus **the market for paper tape units is very limited.**

CASSETTE UNIT OPTIONS

There is a significant market for cassette drives serving as bulk storage devices in microcomputer systems. At the present time the least expensive floppy disk unit, together with interface electronics, costs close to $1000.00. Prices for floppy disk units will come down rapidly, particularly as volume grows, but in the meantime cassette units will be the dominant low cost, bulk storage unit.

When you are choosing a cassette unit for your microcomputer system, there are two important considerations:

1) **Make sure that the cassette drive itself is of good quality.** Do not use a cheap cassette drive since it will give you nothing but headaches.

2) **Make sure that the interface logic for your cassette drive stores information in relatively short records, using redundant recording.** We discussed the advantages of this procedure in Chapter 1. You should also be very wary of cassette units that advertise very high data transfer speeds. As a rule, reliability goes up as data transfer speed goes down. A slow drive is a reliable drive.

DEVICE COMPATIBILITY

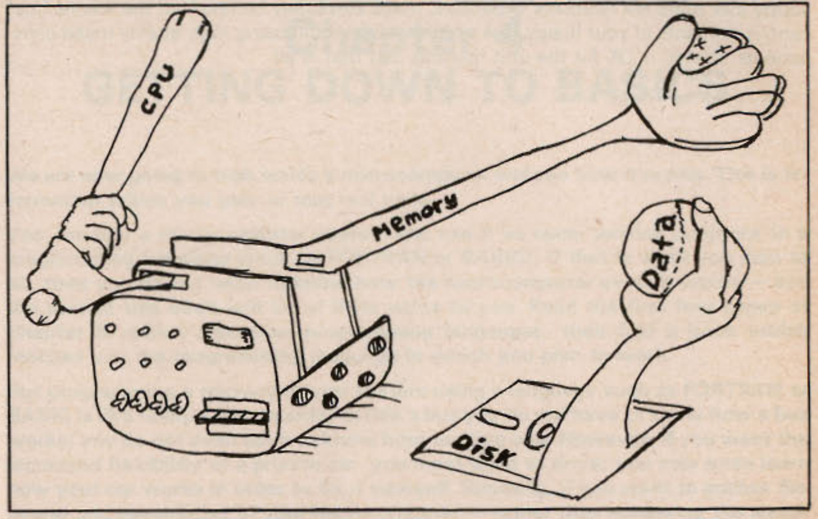
When you buy a cassette unit for your microcomputer system, it is worth checking into how many other brands of cassette units can read the cassettes that your drive creates, and can write cassettes that your drive will read. This is referred to as device compatibility.

When you buy a cassette unit for your microcomputer system you will normally be able to obtain a list of all cassette unit brands that are compatible with yours. Compatibility is a desirable feature since it means you can exchange cassettes with a wider variety of other users.

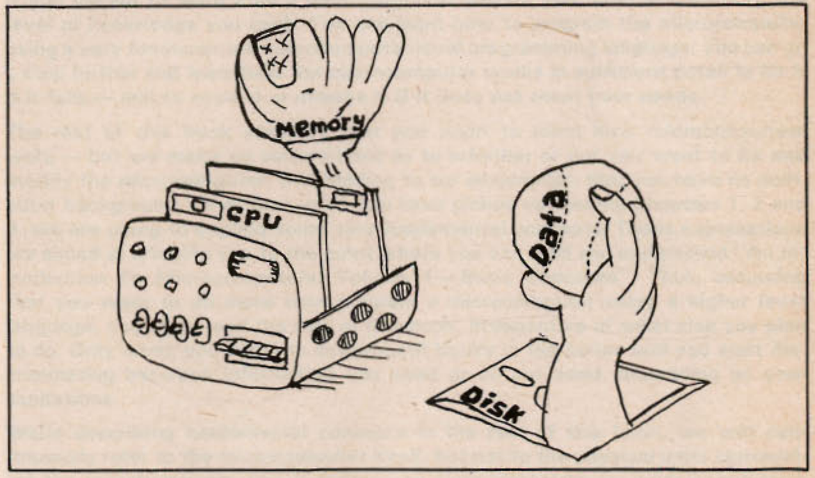
FLOPPY DISK UNIT OPTIONS

Floppy disk unit options have (for the most part) been covered in Chapter 1. You must first decide whether you want a mini-floppy disk unit or a standard floppy disk unit. Having made this decision, you should determine how many floppy disk units are cr mpatible with the one you buy. Clearly you are better off if you can read information that some completely different floppy disk unit wrote out, and conversely, the different floppy disk unit should be able to read information that your drive wrote out.

One important option to look into when buying a floppy disk unit is the manner in which interface logic transfers information between floppy disk and memory. Some floppy disk unit controllers make all data transfers via the microcomputer:



This is a slow and undesirable way of moving data. Better floppy disk units are able to move information directly between read/write memory and your floppy disk, bypassing the microcomputer:



This is the preferable way for a floppy disk controller to work. By accessing memory directly, the floppy disk interface logic is able to move data faster, while leaving the microcomputer free to execute programs while the data transfer is occurring.

You should also examine the way in which a floppy disk drive is constructed. Remember, the floppy disk drive is a mechanical, not an electronic unit. Not only is a floppy disk drive a mechanical unit, it is also a high precision mechanical unit that must be capable of moving a read head quickly, with a precision of thousandths of an inch, over the surface of a moving floppy disk. If the floppy disk drive has been built with cheap plastic parts within its control mechanisms, it will quickly wear out. Since the floppy disk units are relatively expensive, make absolutely certain that the mechanical control elements of your floppy disk unit have been built using high quality metal components; plastic is OK for the unit housing, but that is all.

Chapter 4
GETTING DOWN TO BASICS

We are now going to look inside a microcomputer and see how it works. This is information which you may or may not need.

You can buy a microcomputer system and use it as such, writing programs in a programming language (such as FORTRAN or BASIC); if that is what you plan to do, then you do not need to know how the microcomputer system works — and the rest of this book will be of little value to you. Read the first few pages of Chapter 5, which discusses programming languages, then find a book which teaches you the programming language in which you plan to work.

But programming a microcomputer system using a language such as FORTRAN or BASIC is like riding a bus. In order to ride a bus you do not have to know how a bus works; you do not even have to know how to drive one. However, if you want the increased flexibility of a private car, you must learn to drive; you may even learn how your car works in order to fix it yourself. Similarly, if you want to access the power and capabilities of your microcomputer — rather than accessing the lesser power and more limited capabilities of a programming language — then you must learn how the microcomputer works.

If you decide to learn how a microcomputer works, you must still identify the level of knowledge you seek. You can learn how to program the microcomputer using a very fundamental, microcomputer-level programming language; you can go a step further and learn how the microcomputer works in sufficient detail to fix it if it fails — and to expand or change it if it does not meet your needs.

The rest of this book assumes that you want to learn how microcomputers work — but we make no assumptions as to whether or not you want to fix and modify the microcomputer. Still holding to our assumption that you have no computer background, other than what you have picked up reading Chapters 1, 2 and 3, we are going to explain some very fundamental concepts. These explanations are aimed at bringing you to the point where you can read and understand "An Introduction To Microcomputers: Volume 1—Basic Concepts". Thus, assuming that you want to do more than program a microcomputer using a higher level language, you must read the rest of this book, irrespective of what else you plan to do. Only when you come to subsequent books in the series will you start discriminating between information you need or do not need, depending on your aspirations.

While describing fundamental concepts in the rest of this book, we will continuously refer to the microcomputer itself, but not to the physical units surrounding the microcomputer. This is appropriate, since ultimately you must learn how to program a microcomputer, and optionally how to build one. It will be a long time, if ever, before you need more information on floppy disks, keyboards, displays and other physical units, or their interface logic. You will buy the physical unit and its interface logic as a single unit, and you will program the physical unit from the microcomputer.

NUMBERS AND LOGIC

Every piece of logic within a microcomputer system may be reduced to a network of switches, each of which is "on" or "off". This is a concept which is not entirely new to you, since we used it to describe memories, and specifically, read-only memories. But **let us look at how a network of switches can ultimately generate the power and versatility of a computer.**

BINARY DATA

Consider numbers. As we have often stated, instructions, programs and data of all kinds become a sequence of numbers. **How are we going to represent so many numbers, given nothing more than switches that may be on or off?**

The digit "0" can be represented by an "off" switch:

"1" can represent an "on" switch:

Now what? A computer that can only count to 1 will not be very useful. **Computers can, indeed, create just two separate and distinct numeric digits:**

Zero	0
One	1
Then what?	

But humans have a very similar problem. We are limited to just ten separate and distinct numeric digits:

Zero	0
One	1
Two	2
Three	3
Four	4
Five	5
Six	6
Seven	7
Eight	8
Nine	9
Now start combining digits!	
Ten	10
Eleven	11
etc.	

DECIMAL NUMBERS

The human number system is referred to as the decimal number system. The decimal number system appears world-wide, among totally unrelated tribes and nations — wherever societies have learned to count. Most probably this is because we all have 10 fingers and first learned to count on our fingers. There is nothing "unique" or "natural" about the decimal counting system; in fact, it is a rather clumsy way of doing things. We will see later that there are much neater ways of counting — where neatness is measured in terms of making arithmetic easy.

Just as the human counting system has a name — it is the decimal system — so **the computer counting system** has a name — it **is the binary system.**

Let us look at the binary system in detail.

THE BINARY NUMBER SYSTEM

The binary number system has just two separate and distinct digits: 0 and 1. To represent numbers greater than 1 we follow the example of decimal numbers — and use more than one digit. Consider **the number two;** in binary format it **is represented by the digits 10:**

DECIMAL		BINARY	
Zero	0	Zero	0
One	1	One	1
Two	2	Two	10
Three	3		
Four	4		
Five	5		
Six	6		
Seven	7		
Eight	8		
Nine	9		
Ten	10		

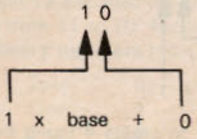
In both the decimal and binary counting systems the two-digit combination "10" represents a number that is one greater than the largest single-digit number. In the case of decimal numbers the largest single-digit number is nine; therefore 10 represents one more than nine, which is ten. In the binary number system the largest single-digit number is one; therefore 10 represents one more than one, which is two.

The numeric value of the digit combination "10" is very important. This digit combination has the value ten in the decimal system, which is where the decimal system gets its name. Similarly, in the binary system, the digit combination "10" has the value two, which is where the word "binary" comes from.

NUMBER BASE

These values, ten for the decimal system and two for the binary system, are called the "base" for the numeric system.

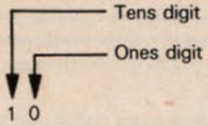
The base number, that is, the value associated with the digit combination "10", is interpreted for decimal or binary numbers as follows:



ONES DIGIT

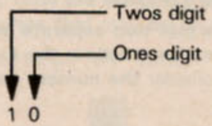
TENS DIGIT

The digits of a two-digit decimal number are referred to as the "ones" digit and the "tens" digit:



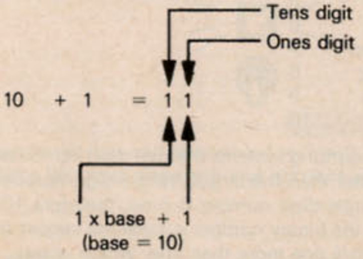
TWOS DIGIT

For a two-digit binary number, the digits are referred to as the "ones" digit and the "twos" digit.



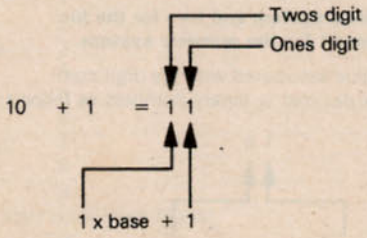
To represent three using binary numbers, we can still draw a parallel with our decimal counting system. **The next decimal number after decimal 10 is created by adding 1 as follows:**

Ten + One = Eleven



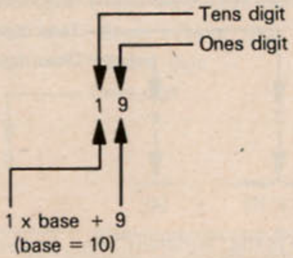
Similarly, we advance from two to three using binary numbers by adding 1 to binary two:

Two + One = Three



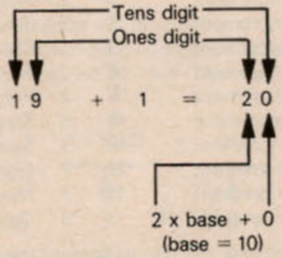
What happens when we want to create 4 using binary numbers? The parallel with decimal counting is not so immediately visible. Following decimal 11, we still have a long way to go before problems arise. We can keep on adding 1 until we reach decimal nineteen (19):

Nineteen

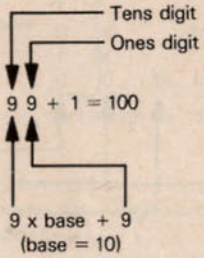


Then we go to decimal twenty (20):

Nineteen + One = Twenty



It is not until you reach decimal 99 that you have to add a third decimal digit and create 100:

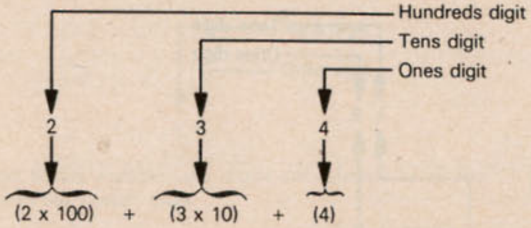


Now in the binary system, following binary 11 (which is decimal three) we have a problem. If we add 1 to 11 we cannot get 12, because the digit 2 does not exist in the binary system. Moreover, we cannot go from 11 to 20, because once again we are using 2, which is an illegal binary digit. **Binary numbers** must therefore follow **11 with 100:**

Zero	0
One	1
Two	10
Three	11
Four	100

Let us examine the meaning of three-digit numbers.

When you see the number 234 you automatically interpret it as two hundred and thirty-four:

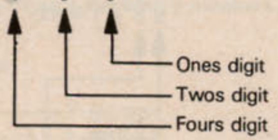


But there is a special significance to the "hundreds" digit, just as there is to the "tens" digit. You can increment the tens digit nine times; on the tenth increment you must increment the hundreds digit. Consider tens digit increments, beginning with the decimal number three:

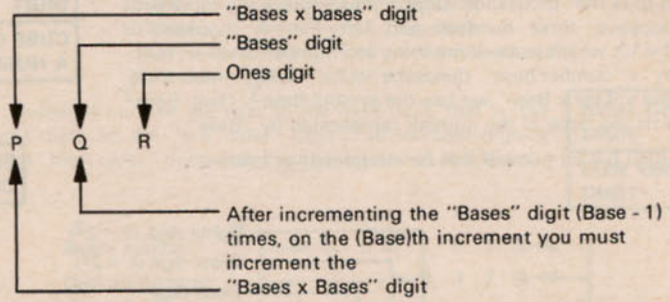
			3	Three
First increment		1	3	Thirteen
Second increment		2	3	Twenty three
Third increment		3	3	Thirty three
Fourth increment		4	3	Forty three
Fifth increment		5	3	Fifty three
Sixth increment		6	3	Sixty three
Seventh increment		7	3	Seventy three
Eighth increment		8	3	Eighty three
Ninth increment		9	3	Ninety three
Tenth increment!!	1	0	3	One hundred and three

In the case of binary numbers, you can increment the "twos" digit just once; on the second increment, you must create a "fours" digit.

			0
			1
First increment		1	0
		1	1
Second increment	1	0	0

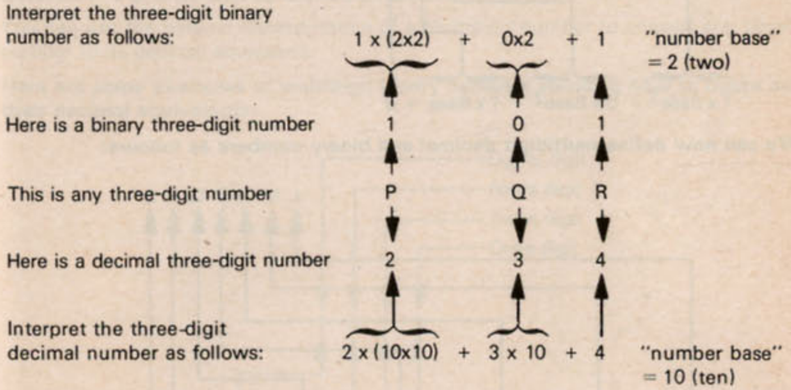


Here is the rule: the number of times you can increment a digit is equal to one less than the number base; then you must increment the next higher digit. Thus in a decimal number you can increment any digit nine times (0 to 9); then you must increment the next higher decimal digit. In a binary number, you can increment once (0 to 1) then you must increment the next higher binary digit. Thus digits can be represented as follows:


"Bases x bases" digit
"Bases" digit
Ones digit
P
Q
R
After incrementing the "Bases" digit (Base - 1) times, on the (Base)th increment you must increment the
"Bases x Bases" digit

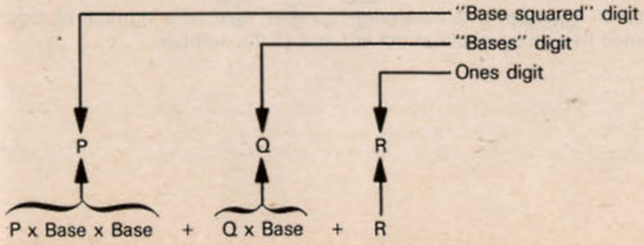
In the illustration above, P Q and R represent any number system's digits. Substitute 2 or 10 for "base" and the illustration will represent "binary" or "decimal" numbers.

The second digit of a multidigit becomes a "number base" multiplier within an equation which tells you the value of the multidigit numbers. Similarly a third digit becomes a multiplier for the number base multiplied by itself. This may be illustrated as follows:


Interpret the three-digit binary number as follows:
1 x (2x2) + 0x2 + 1
"number base" = 2 (two)
Here is a binary three-digit number
1
0
1
This is any three-digit number
P
Q
R
Here is a decimal three-digit number
2
3
4
Interpret the three-digit decimal number as follows:
2 x (10x10) + 3 x 10 + 4
"number base" = 10 (ten)

SQUARE OF A NUMBER

A number multiplied by itself is referred to as the "square" of the number. Thus **we can represent any three-digit number by the following general-purpose equation:**



For "Base x Base" we use the symbol Base^2. Thus Base^2 represents the square of the number represented by "Base".

We can extend the same reasoning to larger numbers.

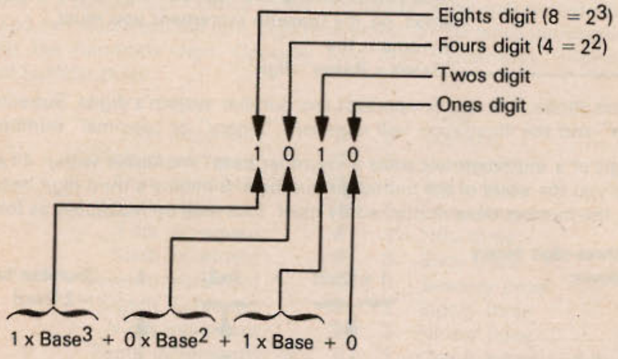
In decimal arithmetic a fourth digit identifies thousands and is referred to as the "thousands" digit. For example, 2345 represents two thousand, three hundred and forty-five. A thousand is 10 x 10 x 10, which is the same thing as "number base" x "number base" x "number base"; this is the "cube" of the number base. For "Base x Base x Base" we use the symbol Base^3. Thus Base^3 represents the cube of the number represented by "Base".

THOUSANDS DIGIT

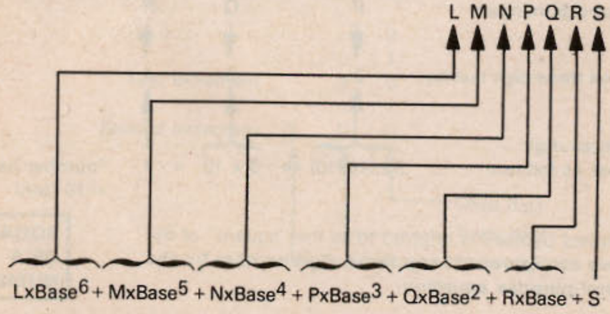
CUBE OF A NUMBER

A four-digit binary number will be interpreted as follows:

EIGHTS DIGIT



We can now define multidigit decimal and binary numbers as follows:


L M N P Q R S
LxBase^6 + MxBase^5 + NxBase^4 + PxBase^3 + QxBase^2 + RxBase + S

The general definition above is for a seven-digit number; any other number of digits could be represented by adding digits at the left end of the number.

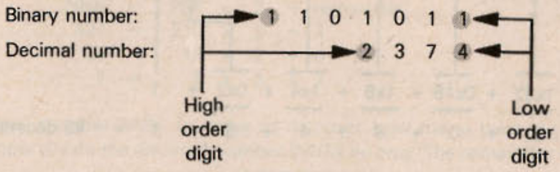
Each time you multiply Base^n by Base you get Base^{n+1}. Thus "Base" through "Base^6" have the following values:

Binary		Decimal
Two	Base	Ten
Four	Base^2	One Hundred
Eight	Base^3	One Thousand
Sixteen	Base^4	Ten Thousand
Thirty-two	Base^5	One hundred thousand
Sixty-four	Base^6	One million

In any multidigit number we refer to the ones digit, that is, the rightmost digit, as the "low order" digit. The leftmost digit is called the "high order" digit. This may be illustrated as follows:

HIGH ORDER DIGIT

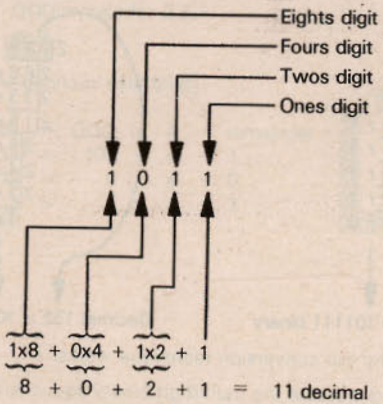
LOW ORDER DIGIT

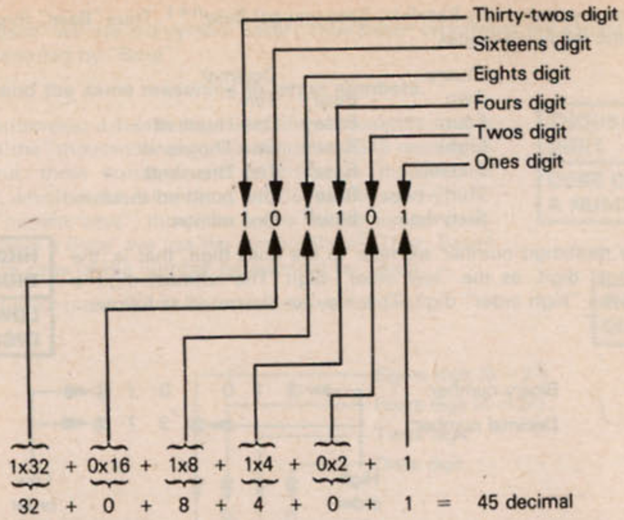


BINARY TO DECIMAL CONVERSION

You can use the general representation of a multidigit number to convert any binary number to its decimal equivalent.

Here are some examples of multidigit binary numbers showing how to figure out their decimal equivalents:

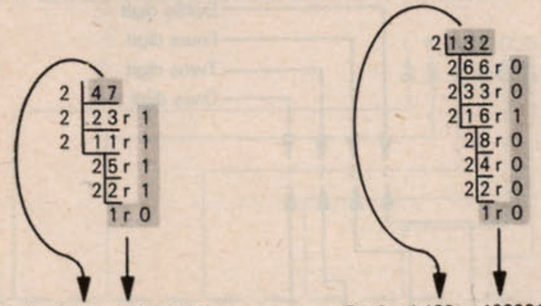




DECIMAL TO BINARY CONVERSION

There is a very simple technique for converting any decimal number to its binary equivalent; we will simply define this technique, then we will explain why it works.

Here is a definition of the technique: to convert a decimal number to its binary equivalent, repeatedly divide the decimal number by two until nothing is left of the number. Here are two examples:



Decimal 47 = 101111 binary   Decimal 132 = 10000100 binary

Now we will explain why the conversion technique works.

The steps illustrated above create the multidigit binary equivalent of the decimal number; the least significant (that is, the rightmost) binary digit is created first. This digit is the remainder once you know how many twos digits there are.

Let us use the symbol NNN to represent any decimal number. What happens when we divide NNN by 2? We will get half of NNN, plus a remainder of 0 or 1. Let us use the symbol PPP to represent half of NNN. In the general case this is how we illustrate NNN being divided by 2:

2 | NNN
PPP remainder 0 or 1

Here are some specific cases:

2 | 421 (NNN = 421)
210 remainder 1 (PPP = 210)

2 | 36 (NNN = 36)
18 remainder 0 (PPP = 18)

2 | 7 (NNN = 7)
3 remainder 1 (PPP = 3)

In each of the above illustrations the decimal number NNN is shown as having PPP twos digits, plus 0 or 1:

NNN	=	PPP	x	2	+	remainder
421	=	210	x	2	+	1
36	=	18	x	2	+	0
7	=	3	x	2	+	1

For any decimal number NNN, in order to discover how many twos digits there are (PPP), you simply divide the decimal number (NNN) by two. The remainder (0 or 1) is the ones digit.

What about PPP? It is a decimal number. If by chance PPP is 0 or 1, then it is also a valid binary number; any larger number is not a valid binary number. If the number of twos digits computed in the first step above (PPP) is more than 1, that means there are some fours digits in the binary number. In order to calculate how many fours digits there are you could simply divide the initial decimal number by four:

4 | NNN
QQQ remainder 3,2,1
or 0

Let us look again at our previous examples:

NNN	=	QQQ	x	4	+	remainder
421	=	105	x	4	+	1
36	=	9	x	4	+	0
7	=	1	x	4	+	3

But note that QQQ, the number of fours digits, must be half of PPP, the number of twos digits; that is to say, dividing NNN by 4 is the same as dividing half of NNN by 2:

```
4 |NNN                          is the same as   2 |NNN
   QQQ  remainder 3, 2, 1 or 0                   2 |PPP  remainder 1 or 0
                                                    QQQ  remainder 1 or 0

4 |421                          is the same as   2 |421
   105  remainder 1                              2 |210  remainder 1
                                                    105  remainder 0

4 |36                           is the same as   2 |36
   9    remainder 0                              2 |18   remainder 0
                                                    9    remainder 0

4 |7                            is the same as   2 |7
   1    remainder 3                              2 |3    remainder 1
                                                    1    remainder 1
```

The advantage of dividing by two twice, is that all remainders are 0 or 1 — valid binary digits. Look at those remainders illustrated above:

1	is the same as	01	(binary)
0	is the same as	00	(binary)
3	is the same as	11	(binary)

Decimal seven (7_{10}) has become binary 111_2. That is to say, decimal seven equals one fours digit, plus one twos digit, plus one ones digit:

$$7_{10} = 4_{10} + 2_{10} + 1$$

Decimal numbers 36_{10} and 421_{10} must continue to have higher level binary digits created since 9_{10} and 105_{10} are not valid binary digits. Higher level binary digits are created by continuing to divide by 2; here is the complete conversion for 36:

```
2 |36
2 |18    remainder 0 (no ones digits)
   2 |9  remainder 0 (no twos digits)
   2 |4  remainder 1 (one fours digit)
   2 |2  remainder 0 (no eights digits)
      1  remainder 0 (no sixteens digits)
      ↑
      one thirty-twos digit
```

Thus, $36_{10} = 32_{10} + 4_{10} = 100100_2$

DECIMAL NOTATION

In the illustrations above we have introduced a new form of shorthand which is commonly used in computer books. **Decimal numbers are identified by a 10 subscript at the end of the number:**

Decimal 4713 is represented as 4713_{10}

BINARY NOTATION

Binary numbers are identified by a 2 subscript at the end of the number:

Binary 11010 is represented as 11010_2

Since you are repeatedly dividing the decimal number by 2, the remainder can only be 0 or 1 — and the remainder tells you how many ones digits, twos digits, fours digits, and so on, there are in the binary equivalent of the decimal number. If QQQ is 2 or more, then there are more than 0 or 1 fours digits and you divide the fours digits (QQQ) by 2 to determine how many eights digits there are; the remainder, when you divide the fours digits (QQQ) by 2, tells you whether there are 0 or 1 fours digits. If there is more than one eights digit, then you go on to the sixteens digits; and if there is more than one sixteens digit you go on to the thirty-twos digits; and so on.

Table 4-1 summarizes all possible four-digit binary numbers and gives their decimal equivalents.

Looking at Table 4-1, it is easy to see how switches can be used to represent numbers of any size. A 0 becomes an "off" switch while a 1 becomes an "on" switch. By simply increasing the number of switches used to represent a number, you can indefinitely increase the size of the numbers which switches can represent. **Table 4-2 shows you the largest number that you can represent in the binary counting system as you increase the number of digits in the number.**

Table 4-1. All Four-Digit Binary Numbers And Their Decimal Representations

Decimal Numbers	Binary Numbers
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Table 4-2. The Largest Number That Can Be Represented By Binary Numbers With 1 Through 16 Digits

Number of Binary Digits	Maximum Binary Value	Decimal Equivalent
1	1	1
2	11	3
3	111	7
4	1111	15
5	11111	31
6	111111	63
7	1111111	127
8	11111111	255
9	111111111	511
10	1111111111	1023
11	11111111111	2047
12	111111111111	4095
13	1111111111111	8191
14	11111111111111	16383
15	111111111111111	32767
16	1111111111111111	65535

Notice that each time you add a new switch (or binary digit) you double the maximum number size which can be represented.

BITS, NIBBLES AND BYTES

BIT

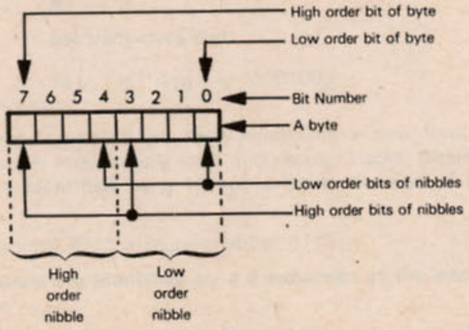
A BInary digiT is always referred to as a bit. Thus a bit can have a value of 0 or 1.

BYTE

Although numbers can be created from any number of binary digits, or bits, as illustrated in Table 4-2, there are certain numbers of bits which you will frequently encounter. Most frequently you will deal with 8-bit combinations. **An 8-bit unit is referred to as a byte.** There are a few obscure computers that use the word "byte" to describe some other number of bits (most frequently 6 bits), but in the world of microcomputers, the byte is always an 8-bit unit. Thus a byte can represent numbers in the range 0 through 255.

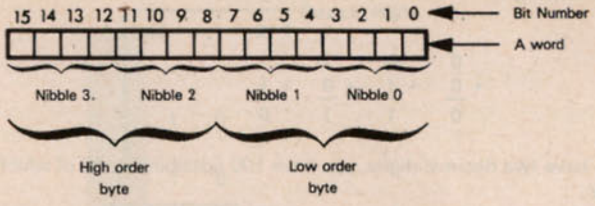
NIBBLE

4-bit units are sometimes referred to as nibbles. Thus a byte consists of two nibbles:



WORD

16-bit units are sometimes called words. Thus a word consists of two bytes or four nibbles:

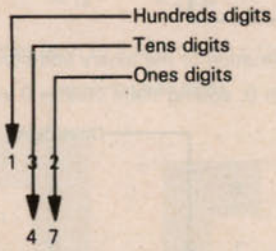


BINARY ARITHMETIC

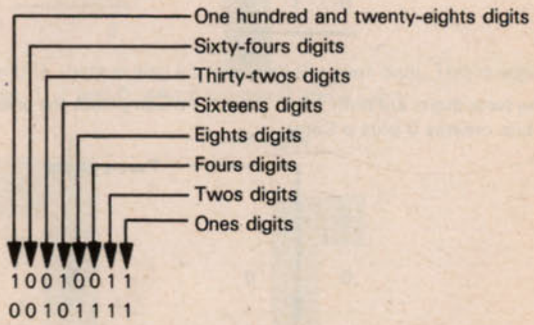
Let us look at parallels between **arithmetic using binary numbers** and arithmetic using decimal numbers.

BINARY ADDITION

When you perform decimal addition, you align the digits of the two numbers being added as follows:



You do essentially the same thing for binary addition:



You add the numbers, one digit at a time, starting with the low order (ones) digit. But binary addition is beautifully simple, as compared to decimal addition. When you add two binary digits there are just four possibilities:

$$\begin{array}{rrrr} & & & 1 \xleftarrow{\quad} \text{Carry } 1\\ 0 & 0 & 1 & 1\\ +\underline{0} & +\underline{1} & +\underline{0} & +\underline{1}\\ 0 & 1 & 1 & 0 \end{array}$$

When you have two decimal digits, you have 100 possibilities, 45 of which will generate a carry.

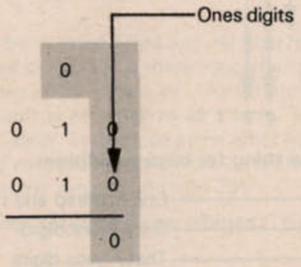
Let us look at a few examples of binary addition, first the very simple 2+2=4. Using binary numbers, this is what you get:

Decimal	Binary
0	000
1	001
2	010
3	011
4	100

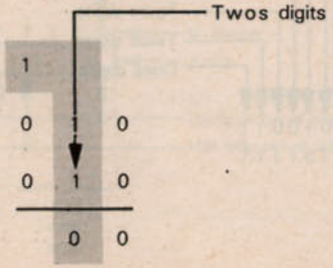
$$\begin{array}{rr} \text{Decimal} & \text{Binary}\\ 2 & 010\\ +\ 2 & +\ 010\\ \hline =4 & =100 \end{array}$$

Here is a digit-by-digit explanation of the binary addition:

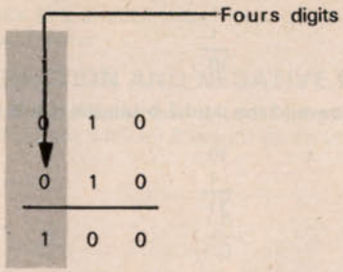
1) The ones digits are both 0; adding them creates 0 and no Carry:



2) The twos digits are both 1 and there is no Carry from the ones digit. Adding the two 1 bits creates 0 plus a Carry:



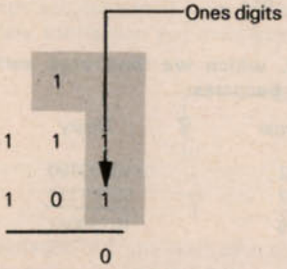
3) The fours digits are both 0, but there is a Carry from the twos digits. The two zeros create 0, but adding 1 (the Carry) to 0 creates 1, with no Carry:



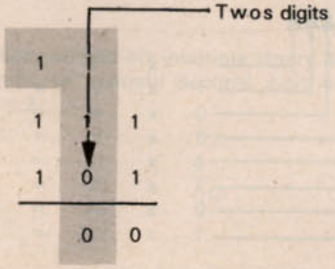
Now let us look at a slightly more complex example of binary addition: 7+5=12. This may be illustrated as follows:

$$\begin{array}{rr} \text{Decimal} & \text{Binary}\\ 7 & 111\\ +\ 5 & +\ 101\\ \hline =12 & =1100 \end{array}$$

The ones digits are both 1; they sum to 0 and create a Carry:



The twos digits are 1 and 0; there is also a Carry from the ones digit. This is equivalent to adding the three bits: 1, 1 and 0, which creates 0 and a Carry:



The fours digits are both 1, but there is also a Carry from the twos digits. Adding three 1 bits is simple enough if you do it in two steps. First add two 1 bits:

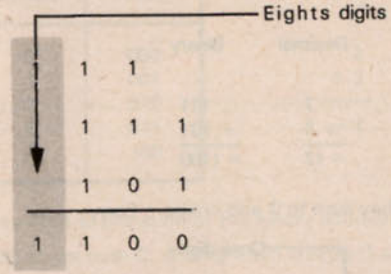
$$\begin{array}{r}1\\1\\\hline 10\end{array}$$

This creates 0 and a Carry. Now add the third 1 bit to the result:

$$\begin{array}{r}10\\1\\\hline 11\end{array}$$

You have 1 with a Carry.

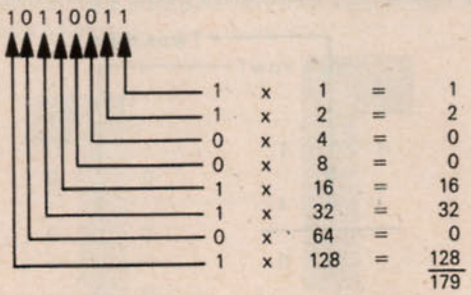
The eights digits are both 0, but there is a Carry. The eights digits therefore sum to 1:



The addition of 132 and 47, which we illustrated earlier when describing microcomputer logic, in binary becomes:

Decimal	Binary
132	10000100
+ 47	00101111
= 179	10110011

The decimal to binary conversions for 132 and 47 have already been described. **We can check that the binary sum is indeed equivalent to decimal 179 as follows:**



10110011

1	x	1	=	1
1	x	2	=	2
0	x	4	=	0
0	x	8	=	0
1	x	16	=	16
1	x	32	=	32
0	x	64	=	0
1	x	128	=	128
				179

If you do not understand binary addition at this point, you should continue doing examples until you do understand binary addition. For each example write two decimal numbers; then create their binary equivalents. Add the binary equivalents, and convert the sum back to a decimal value. If you do not get the correct decimal sum, then you have made an error.

BINARY SUBTRACTION AND NEGATIVE NUMBERS

Binary subtraction is far simpler than decimal subtraction because you have just four possibilities. You may subtract 0 from 0 which leaves 0:

$$\begin{array}{r}0\\-0\\\hline 0\end{array}$$

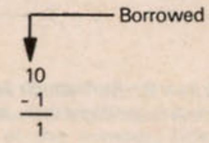
You may subtract 1 from 1 which again leaves 0:

$$\begin{array}{r}1\\-1\\\hline 0\end{array}$$

If you subtract 0 from 1 the result is 1:

$$\begin{array}{r}1\\-0\\\hline 1\end{array}$$

But what happens when you subtract 1 from 0? Just as you would do for decimal subtraction, so for binary subtraction you must borrow from the next highest bit as follows:



10 is the binary representation of 2; the illustration above subtracts 1 from 2, leaving a result of 1. If there is no higher bit to borrow from, then the result is -1:

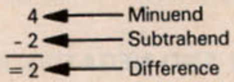
$$\begin{array}{r}0\\-1\\\hline -1\end{array}$$

Extending subtraction to multi-bit (multiple binary digit) numbers is as simple as extending subtraction to multiple decimal digit numbers. Here is the binary equivalent of 4 - 2 = 2:

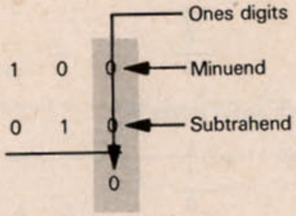
$$\begin{array}{r}100\\010\\\hline 010\end{array}$$

When you subtract two numbers, you subtract a subtrahend from a minuend. This may be illustrated as follows:

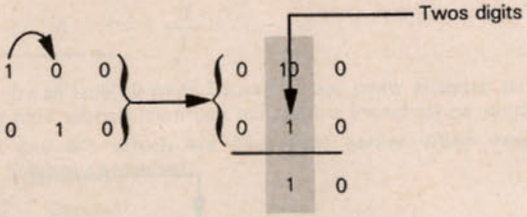
SUBTRAHEND
MINUEND



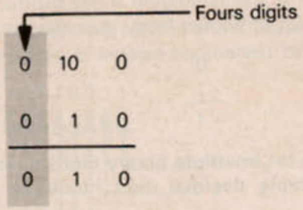
Looking at the binary subtraction of two from four, the ones digits are both 0; therefore the difference is 0:



In the case of the twos digits, we must subtract 1 from 0. Therefore we borrow 1 from the minuend fours digit and obtain a difference of one:



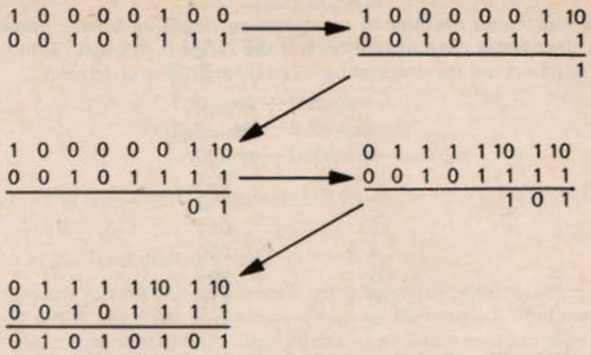
The minuend fours digit is now 0; the 1 which was there has been borrowed by the minuend twos digit. Thus for the fours digits we subtract 0 from 0, creating a difference of 0:



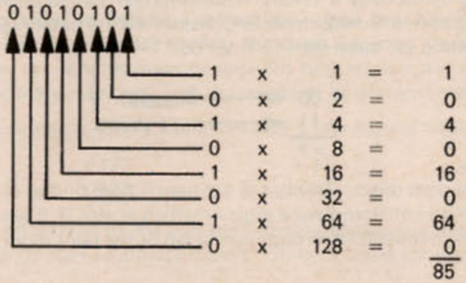
Going to a more complex example here is the binary representation of $132_{10} - 47_{10} = 85_{10}$:

Decimal	Binary	
11	111	◄—Borrows
132	10000100	◄—Minuend
- 47	- 00101111	◄—Subtrahend
= 85	01010101	◄—Difference

Here is a step-by-step illustration of the binary subtraction:



We can check that the binary result is correct by generating its decimal equivalent:



Now this may surprise you but most computers cannot subtract. They can only add. Fortunately this is no problem. Subtracting one number from another is equivalent to adding the negative of the number. This may be illustrated as follows:

4 + -(2) = + (-2)

The above solution might look ridiculous, but in fact it is not; it looks ridiculous only because you are used to decimal numbers. Consider the subtraction of 3 from 9. You could write this as 9-3, or you could write it as 9+(0-3). In fact 9+(0-3) is more complicated than 9-3. To solve 9+(0-3) you must subtract 3 from 0 before adding the result to 9; but when you subtract 3 from 0 you get -3 which puts you right back to subtracting 3 from 9 — which was what you were going to do in the first place.

Subtraction in the world of switches and binary numbers cannot be resolved until we find a way of representing negative numbers. A switch is a two-state device and we cannot simply add a new state for the switch to represent a negative sign:

Until we find some means of representing negative binary numbers, we cannot even try to come up with the binary equivalent of 9+(0-3).

In order to find some method of representing negative binary numbers, let us begin with the simple case of numbers in the range 0 through -7; in their positive form these numbers are represented by three binary digits as follows:

Decimal	Binary Equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

We cannot arbitrarily select a method of representing negative binary numbers; our requirement is that we must be able to subtract by adding the negative representation of the number.

The logical way of finding a binary representation for negative numbers is to try subtracting the positive number from 0. Consider +3. Its binary form is 11_2. Let us see what happens when we subtract 11_2 from 00:

```
  00  ◄——— Minuend
- 11  ◄——— Subtrahend
  =?
```

Starting with the ones digits, we want to subtract 1 from 0; that is impossible, so we try to borrow 1 from the minuend twos digit — which is also 0. If we assume that we can borrow 1 from the minuend fours digit (to the left of the twos digit), then this is what we get:

```
┌──────────── Twos digit borrowed 1
│ ┌────────── Twos digit
│ │ ┌──────── Ones digit
▼ ▼ ▼
1 0 0  ◄—— Minuend
- 1 1  ◄—— Subtrahend
  = ?
```

Now the minuend ones digit can borrow 1 from the minuend twos digit:

```
┌──────────── Twos digit (10 - 1 = 1)
│ ┌────────── Ones digit borrowed 1
│ │ ┌──────── Ones digit
▼ ▼ ▼
 1 10  ◄—— Minuend
- 1 1  ◄—— Subtrahend
=   ?
```

We can successfully perform the subtraction:

```
  ┌──────── Twos digit
  │ ┌────── Ones digit
  ▼ ▼
  1 10  ◄—— Minuend
 - 1 1  ◄—— Subtrahend
   0 1  ◄—— Difference (negative)
```

The difference in the ones digit is computed as $10_2 - 1_2 = 1_2$. This is equivalent to the decimal $2_{10} - 1_{10} = 1_{10}$.

The difference in the twos digit is simply 1 - 1 = 0.

We have succeeded in subtracting 3 from 0 using binary arithmetic but it involved a sleight-of-hand which we must now account for; we borrowed 1 from the next high order digit of the minuend (the fours digit in this case) when no such digit existed.

We have another problem that now needs resolution. The two bits 01 are shown representing the value -3; but they also represent the value +1.

These two problems have no solution within the context of binary counting as we have defined it thus far. **In order to handle subtraction — and the inevitable negative numbers that can result — we must modify the rules adopted thus far for binary counting.** But the new set of rules must be logically and numerically consistent with the needs of positive binary numbers, and the needs of binary arithmetic.

Fortunately there is a simple solution. Let us look at a few signed decimal numbers:

+ 10	+ 123	+ 47	+ 83742
-10	-123	-47	-83742

The one new feature introduced by the numbers illustrated above is the sign: there is a plus sign (+) and a minus sign (-). Binary signed numbers could conceivably be illustrated as follows:

+ 1011	+ 1110101	+ 110	+ 1011010
-1011	-1110101	-110	-1011010

SIGN OF BINARY NUMBERS

What we have is a plus (+) or minus (-) sign preceding the string of bits (binary digits). Now we cannot represent plus and minus signs as separate and distinct entities within computer logic. Remember computer logic consists of nothing other than two-state switches. We must therefore take the same two-state switch which represents 0 and 1 digits, but now use it to represent plus and minus signs. If **we use an "off" switch to represent a plus sign, and an "on" switch to represent a minus sign,** then the plus sign (+) and the zero digit (0) are both represented by an "off" switch; the minus sign (-) and the one digit (1) are both represented by an "on" switch. Now **our signed binary numbers could be represented as follows:**

01011	01110101	0110	01011010	Positive numbers
11011	11110101	1110	11011010	Negative numbers

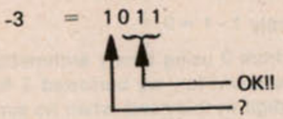
Unfortunately the method of representing negative numbers illustrated above is not going to work. Consider the simple example of 5 - 3 = 2. Remember, we said that in order to subtract we must be able to add the negative representation of the number. Thus 5 - 3 = 2 becomes 5+(-3)=2. Does this work? Let us try and see:

If +5 = 0101 +3 = 0011 and -3 = 1011

$$\textbf{Then } +5 + (-3) \textbf{ becomes } \begin{array}{r} 0101 \\ +\ 1011 \\ \hline 10000 \end{array}$$

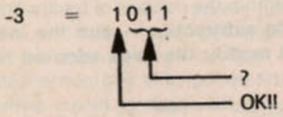
It does not work. Either the method we have adopted for representing the sign does not work:

-3 = 1011



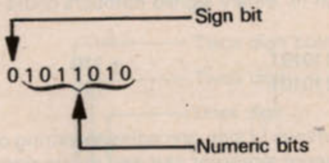
Or the method we have adopted for representing the numeric portion of the negative number does not work:

-3 = 1011



In order to find out what is wrong let us look at negative numbers more carefully.

The only way you can tell whether a binary digit is representing the sign of a number or a digit within the number is by looking at the binary digit position. **The left-most bit position must be interpreted as the sign bit:**



How are you going to tell the difference between a signed binary number and an unsigned binary number that is 1 digit longer? The answer is you cannot tell the difference by simple inspection. You simply have to know what you are dealing with. This necessity to interpret numbers is, of course, not new to us. We discussed the interpretation of numbers in Chapter 2, when describing Joe Bitburger's bill paying program. Telling you that you must know in advance whether a binary number is signed or unsigned is much the same as telling you that you must differentiate between a dollar amount and the bank number on a check. This type of number interpretation is a constant necessity when dealing with microcomputers. You have to differentiate between signed and unsigned numeric data, plus many additional ways in which a sequence of bits (binary digits) might have to be interpreted; this gives rise both to the power and the complexity of computer programming.

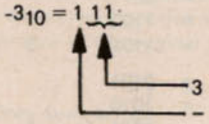
Returning to the signed binary numbers, as we have defined them, let us examine the sign bit critically. By selecting the "off" switch to represent either a plus sign or a 0 bit, positive signed binary numbers have the same numeric interpretation as unsigned binary numbers. This may be illustrated as follows:

Decimal Number		Binary Number
3_{10}	=	11_2
$+3_{10}$	=	011_2

When going from positive unsigned numbers to positive signed numbers we add a leading 0, which in no way affects the magnitude of the number. That means our representation of the sign is satisfactory. What about the numeric bits? We have one of two choices:

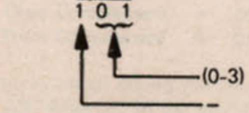
1) We can place a 1, representing the negative sign, to the left of the positive number high order bit:

$-3_{10} = 1\,11$



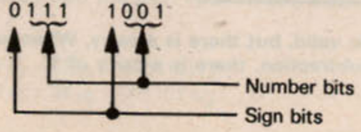
2) Alternatively we can take the binary digit pattern created by subtracting the positive number from 0, as we did when we subtracted 3 from 0, and we can place the one to the left of the most significant numeric bit:

$$\begin{array}{rcr} 0 & = & 00 \\ 3 & = & 11 \\ \hline & & 1\,01 \end{array}$$



It just so happens that method 1), which we automatically selected, does not work (as we demonstrated by trying to subtract 3 from 5). **Method 2) is the simplest way to go.** Using method 2) this is how we would use four binary digits to represent numbers in the range +7 through -7:

Decimal Numbers		Binary Numbers	
Positive	Negative	Positive	Negative
0	0	0000	0000
1	1	0001	1111
2	2	0010	1110
3	3	0011	1101
4	4	0100	1100
5	5	0101	1011
6	6	0110	1010
7	7	0111	1001



TWOS COMPLEMENT

CREATING THE TWOS COMPLEMENT OF A NUMBER

ONES COMPLEMENT

The representation of negative numbers illustrated above is referred to as the twos complement of the number.

In order to generate the twos complement of a number you do not have to subtract the positive number from 0, then add a high order (left most) 1 bit to represent the minus sign; there is a simpler procedure. **First generate the ones complement of the number** by inverting every bit (binary digit), that is to say, replace 0 bits with 1 bits and 1 bits with 0 bits. Here are some examples of binary numbers and their ones complements:

Binary Number:	101	11101	101010
Ones Complement:	010	00010	010101

Add 1 to the ones complement of the number and you have the twos complement of the number. Here are some examples, which you can verify against the negative binary numbers illustrated above.

```
            +3   =  0011
 Ones Complement =  1100
 Twos Complement =  1101   =  -3

            +5   =  0101
 Ones Complement =  1010
 Twos Complement =  1011   =  -5
```

Now let us see if this binary representation of negative numbers is valid. If it is valid, then **we must be able to subtract a number by adding its twos complement, that is, its negative representation.** Consider numbers in the range 1 through 7; if we make sure that we always subtract a smaller number from a larger number, then **here are some examples which verify that we have developed a valid representation for negative numbers:**

Decimal	Binary
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

```
                              7-3    =  4
becomes                      7 +(-3)  =  4
                             0 1 1 1
                            +1 1 0 1
                          = 1 0 1 0 0
                 Carry

                              5-1    =  4
becomes                      5 +(-1)  =  4
This is the binary form:     0 1 0 1
                            +1 1 1 1
                          = 1 0 1 0 0
                 Carry
```

The result is valid, but there is a carry. Whenever a positive difference is generated by a subtraction, there is a carry of 1.

What happens **if you subtract a larger number from a smaller number?** Exactly the same thing that happens with decimal subtraction. **You are left with a negative number.** Here, for example, is what happens when you subtract 5 from 3:

```
                              3-5    =  -2
                becomes      3 +(-5)  =  +(-2)
This is the binary form:     0 0 1 1
                            +1 0 1 1
                           = 1 1 1 0
```

You can tell that the result is negative because the high order digit of the binary result is 1. Remember, when you are dealing with positive and negative binary numbers, the high order bit represents the sign of the number. If the high order bit is 0, the number is positive; if the high order bit is 1, as above, then the number is negative.

It is very easy to create the positive equivalent of a negative number. You simply take the twos complement of the negative number and you get back the positive number. Here are some examples of positive numbers, their twos complement negative equivalents, and regeneration of the original positive number:

```
            +7   =  0111
 Ones Complement =  1000
 Twos Complement =  1001   =  -7
 Ones Complement =  0110
 Twos Complement =  0111   =  +7
            +4   =  0100
 Ones Complement =  1011
 Twos Complement =  1100   =  -4
 Ones Complement =  0011
 Twos Complement =  0100   =  +4
```

There is nothing very tricky about taking the twos complement of a number twice and getting the number back. After all, in the world of decimal arithmetic two negatives make a positive. For example, -(-2) is +2. Thus if the binary representation we have developed for negative numbers is valid then the twos complement of a negative number must give back the positive number, which it does.

We have developed a very elegant method of handling negative numbers and subtraction using binary digits; but remember, computers do not have the intrinsic ability to cope with negative numbers. So long as you are using a computer, it is your responsibility to remember whether a binary digit pattern represents positive numbers only, or positive and negative numbers.

BINARY MULTIPLICATION AND DIVISION

We are not going to discuss binary multiplication or division in much detail. This information will only be of value to you when you are a relatively experienced microcomputer user. There is, however, a very interesting phenomenon associated with multiplying and dividing binary numbers.

In order to multiply a binary number by two you shift each bit one place to the left. Here is an example:

$$27_{10} = 1B_{16} = 00011011_2$$
$$27_{10}x2 = 54_{10} = 36_{16} = 00110110_2$$

Shifting each bit of a binary number one position to the right is equivalent to dividing the number by two. Here is an example:

$$36_{10} = 24_{16} = 00100100_2$$
$$18_{10} = 12_{16} = 00010010_2$$

In reality there is nothing very surprising about shifting binary digits to multiply or divide by two; you can do the same thing with decimal numbers. In order to multiply a decimal number by ten you shift each digit one position to the left:

$$374 \times 10 = 3740$$

Shifting decimal digits one position to the right divides the number by ten:

$$\frac{26730}{10} = 2673$$

There are a variety of methods that people have devised to multiply and divide binary numbers. Some of these methods are described in the "Programming For Logic Design" series of books. If you program a microcomputer in a higher level language you never need to concern yourself with the exact procedure whereby binary numbers are multiplied, the compiler takes care of this chore for you. (Programming language options are discussed in the beginning of Chapter 5.) Why does anyone ever bother with assembly language? The answer once again is to have better control of your program. For example, there are many different programs you can write to perform binary multiplication or division. The various multiplication and division programs will all give you the same answer; but some of them execute very quickly while requiring a lot of memory to store the program, while others take a long time to execute but have relatively short programs. If you use a higher level language, then you take whatever multiplication or division program the language happens to give you. If you write your program you can choose the multiplication or division method that is the fastest to execute, or the method that uses the least memory.

OCTAL AND HEXADECIMAL NUMBERS

It takes a mathematical genius to convert, by inspection, between multidigit decimal and binary numbers. Converting between multidigit decimal and binary numbers is not easy. This is a problem to humans, but it is no problem to a microcomputer. A microcomputer works entirely with binary digits; it does not even know that decimal digits exist. But humans find binary numbers very difficult to work with. **The manipulation of binary numbers is inconvenient, time consuming and prone to error.** Imagine how easy it is to transpose a 0 and a 1 — and how hard it is to spot such an error. **This has resulted in people adopting counting systems which are more compact than binary, yet have a simple relationship with binary digits. The two number systems most commonly used are "octal" and "hexadecimal".**

Every octal digit represents exactly three binary digits, as follows:

Binary:	000	001	010	011	100	101	110	111
Octal:	0	1	2	3	4	5	6	7

The word "octal" is derived from the number 8. Octal numbers are "base 8" numbers. Thus the numeral 10, which is ten in the decimal counting system, is eight in the octal counting system.

You may well look at the 8 octal digits illustrated above and ask what difference there is between octal and decimal digits. There is none in the range 0 through 7 but octal digits have no 8 or 9. Thus, multioctal and multidecimal digits will not have the same values. This may be illustrated as follows:

Decimal	Octal
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14

Octal numbers are identified by a trailing 8 subscript as follows:

$$11_{10} = 13_8$$
$$11\ (\text{Decimal}) = 13\ (\text{Octal})$$

Converting binary numbers into their octal equivalent is very straightforward; you simply partition the binary number into groups of three binary digits and replace each group of three binary digits with its octal digit equivalent. This may be illustrated as follows:

Binary:	110	101	110	111
Octal:	6	5	6	7

Thus $110101110111_2 = 6567_8$

Conversely, if you want to convert an octal number to its binary equivalent, you simply replace each octal digit by its three binary digit equivalent. This may be illustrated as follows:

Octal:	2	5	7	4
Binary:	010	101	111	100

$$2574_8 = 010101111100_2$$

Every hexadecimal digit represents exactly four binary digits as follows:

Binary:	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal:	0	1	2	3	4	5	6	7
Decimal:	0	1	2	3	4	5	6	7

Binary:	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal:	8	9	A	B	C	D	E	F
Decimal:	8	9	10	11	12	13	14	15

The word hexadecimal is derived from the number sixteen. Hexadecimal numbers are base sixteen numbers. Thus the numeral 10, which is ten in a decimal counting system and eight in an octal counting system, is sixteen in a hexadecimal counting system. This poses a novel problem: if 10 represents sixteen in the hexadecimal counting system, then **there must be sixteen single numeric digits in the hexadecimal counting system,** just as there are ten single numeric digits in the decimal counting system. **The six additional hexadecimal single numeric digits are represented by the letters A, B, C, D, E and F,** as illustrated above. Thus you must differentiate between the letters A through F representing hexadecimal digits or letters of the alphabet. While this may seem to make things unnecessarily complicated, in reality you will find it is never a problem. When you look at a piece of data you know automatically whether you are looking at numbers or text; confusion will never arise.

You should understand very clearly that octal and hexadecimal counting systems are useful to humans but computers are unaffected by the counting system you use to write data on a piece of paper; **computers only recognize binary data.**

If microcomputers do not understand octal and hexadecimal number systems, can it really be easier for you to learn counting systems, rather than staying with decimal numbers and coping with decimal-binary conversions? The answer is yes, you are better off learning octal and hexadecimal counting systems. Let us illustrate this point with an example. The decimal number 2735 has the following binary, octal and hexadecimal equivalent:

$$\begin{array}{lccccccccccccccl} & \multicolumn{4}{c}{A} & \multicolumn{4}{c}{A} & \multicolumn{4}{c}{F} & & \text{Hexadecimal}\\ \text{Decimal } 2735 = & \multicolumn{4}{c}{\overbrace{1010}} & \multicolumn{4}{c}{\overbrace{1010}} & \multicolumn{4}{c}{\overbrace{1111}} & & \text{Binary}\\ & \multicolumn{3}{c}{\underbrace{101}_{5}} & \multicolumn{3}{c}{\underbrace{010}_{2}} & \multicolumn{3}{c}{\underbrace{101}_{5}} & \multicolumn{3}{c}{\underbrace{111}_{7}} & & \text{Octal}\end{array}$$

We have described the standard techniques you can use to create binary digits out of decimal digits and vice versa. But these techniques are time consuming, clumsy and never easy to work with, no matter how well you understand binary and decimal numbers. On the other hand, you can convert between octal or hexadecimal numbers, and their binary equivalents by inspection. Once you learn to think in octal and hexadecimal, and that is really quite easy, you will have the advantage of a short notation for writing data, and a trivially simple conversion process for going to and from binary equivalents. This argument is much the same as the argument which is leading the entire world to metric measuring systems. There is no inherent difference between measuring distance in kilometers and meters, as against miles and yards; similarly there is no inherent difference between decimal or hexadecimal counting. But in one case conversions are clumsy, while in the other case they are straightforward.

Table 4-3 summarizes numbers in the range zero through sixteen, showing their binary, decimal, octal and hexadecimal representations.

Table 4-3. Number Systems

Hexadecimal	Decimal	Octal	Binary
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	1C	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111
10	16	20	10000

The only aspect of octal and hexadecimal numbers with which you need to concern yourself is the conversion between these numbers and binary or decimal numbers. Addition and subtraction using octal and hexadecimal numbers is identical to addition and subtraction using decimal numbers — bearing in mind, of course, that each numbering system has its own set of numeric digits.

OCTAL-HEXADECIMAL CONVERSIONS

If you wish to convert from octal to hexadecimal or from hexadecimal to octal, the simplest way of doing it is via a binary intermediate step, since binary numbers have a digit-by-digit correlation with both octal and hexadecimal numbers. For example, consider the hexadecimal number $3C2F_{16}$; we may create its octal equivalent, using Table 4-3, as follows:

$$\begin{array}{rcccc} \text{Hexadecimal:} & 3 & C & 2 & F\\ \text{Binary:} & \overbrace{0011} & \overbrace{1100} & \overbrace{0010} & \overbrace{1111}\\ \text{Octal:} & 0\ \ 3 & 6 & 0 & 5\ \ 7\end{array}$$

Thus $3C2F_{16} = 36057_8$.

You may convert the octal number 23754 to its hexadecimal equivalent as follows:

$$\begin{array}{rccccc} \text{Octal:} & 2 & 3 & 7 & 5 & 4\\ \text{Binary:} & \overbrace{010} & \overbrace{011} & \overbrace{111} & \overbrace{101} & \overbrace{100}\\ \text{Hexadecimal:} & 2 & 7 & & E & C\end{array}$$

Thus $23754_8 = 27EC_{16}$.

DECIMAL-OCTAL AND DECIMAL-HEXADECIMAL CONVERSIONS

Converting a decimal number to its octal or hexadecimal equivalent follows the logic which we have already described for generating the binary equivalent of a decimal number.

Consider first the conversion of a decimal number to its hexadecimal equivalent. We make the conversion beginning with the least significant (that means the right most) digit. If our hexadecimal number is going to be a two-digit number, then it would consist of sixteen (the base) multiplied by some fixed digit (R) with a remainder of S:

$$RS_{16} = R_{10} \times 16_{10} + S_{10}$$

In order to find out what this remainder S is, we **divide the decimal number by the base (sixteen) as follows:**

$$\begin{array}{l} 16\,\underline{\lfloor\ NNN}\\ \qquad R \text{ remainder } S\end{array}$$

NNN is a decimal number; it is equal to the hexadecimal number RS_{16}. Here is a real numeric example:

$$\begin{array}{l} 16\,\underline{\lfloor\ 124}\\ \qquad 7 \text{ remainder } 12\end{array}$$

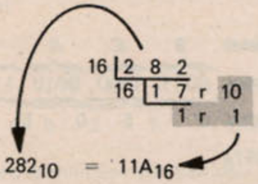
$$\begin{array}{rcl} \text{Therefore } 124 & = & 7 \times 16 + 12\\ \text{so } 124_{10} & = & 7C_{16}\end{array}$$

Thus we are able to generate the digits R and S and we have a complete decimal-to-hexadecimal conversion. Decimal 124 equals hexadecimal 7C.

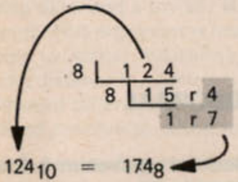
Now consider the larger decimal number 282_{10}. When we divide 282_{10} by 16_{10} this is what we get:

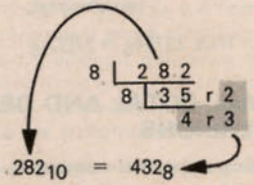
$$16 \,\underline{\,|\; 282\,}$$
$$17 \text{ remainder } 10$$

The remainder (S) is 10 decimal which you will see from Table 4-3 has the hexadecimal equivalent A_{16}. So far so good. But R, the multiplier for the base, has the decimal equivalent 17. You will see from Table 4-3 that there is no single hexadecimal digit representing the decimal value 17. In order to generate the hexadecimal equivalent, therefore, we must go the next step and divide 17_{10} by 16_{10}:



Let us convert the same two decimal numbers, 124_{10} and 282_{10}, to their octal equivalent. Since octal numbers are base eight, we must repeatedly divide the decimal number by 8_{10} in order to generate the octal equivalent. This may be illustrated as follows:





Now we can make sure that our conversions are correct by checking the octal and hexadecimal equivalents of 124_{10} and 282_{10} via their binary intermediate:

$$\overbrace{001}^{1}\,\overbrace{111}^{7}\,\overbrace{100}^{4} \qquad\qquad \overbrace{100}^{4}\,\overbrace{011}^{3}\,\overbrace{010}^{2}$$
$$\underbrace{0}_{0}\,\underbrace{0111}_{7}\,\underbrace{1100}_{C} \qquad\qquad \underbrace{1}_{1}\,\underbrace{0001}_{1}\,\underbrace{1010}_{A}$$

$124_{10} = 174_8 = 7C_{16}$ $\qquad\qquad$ $282_{10} = 432_8 = 11A_{16}$

The octal and hexadecimal conversions are indeed correct.

Converting octal and hexadecimal numbers to their decimal equivalent is easy enough if you remember what the various digits of an octal or hexadecimal number represent. In order to keep things simple, consider four-digit numbers. The decimal representation of any four-digit number may be defined by the following equation:

$$\text{Decimal value} = P \times (\text{Base})^3 + Q \times (\text{Base})^2 + R \times \text{Base} + S$$

A number with more than four digits would simply have terms to the left with higher powers of the base. Now the general four-digit equation can be rewritten for the specific cases of octal and hexadecimal numbers as follows:

$$\text{Decimal value} = P \times 8^3 + Q \times 8^2 + R \times 8 + S \text{ for octal numbers}$$
$$\text{Decimal value} = P \times 16^3 + Q \times 16^2 + R \times 16 + S \text{ for hexadecimal numbers}$$

In order to convert an octal or hexadecimal number to its decimal equivalent, you multiply each digit of the octal or hexadecimal number by the appropriate base multiplier. Here are some examples:

$$\begin{aligned}
2473_8 &= (2\times 8^3 + 4\times 8^2 + 7\times 8 + 3)_{10}\\
&= (2\times 512 + 4\times 64 + 7\times 8 + 3)_{10}\\
&= (1024 + 256 + 56 + 3)_{10}\\
&= 1339_{10}\\
149A_{16} &= (1\times 16^3 + 4\times 16^2 + 9\times 16 + 10)_{10}\\
&= (1\times 4096 + 4\times 256 + 9\times 16 + 10)_{10}\\
&= (4096 + 1024 + 144 + 10)\\
&= 5274_{10}
\end{aligned}$$

CHARACTER CODES

The two-state switches which microcomputer logic uses to generate binary digits must also be used to represent letters of the alphabet and any character capable of being displayed, printed or otherwise handled. If, as we stated at the beginning of this chapter, computer logic consists of nothing more than an array of two-state switches, then **in order to represent characters, we** have no option but to **use switch (and therefore binary digit) patterns** to also represent characters.

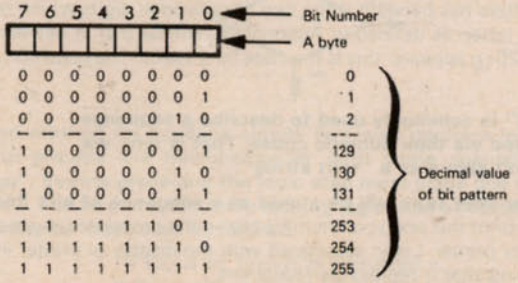
In order to come up with some reasonable character coding technique we must explore two problems:

1) **Is there any "natural" method of representing characters, as there is for representing binary data?**

2) **How will we distinguish between a binary digit pattern representing a character, as against the binary digit pattern representing numbers or any other information?**

There is no "natural" method of representing characters using binary digit codes and any binary digit code which we generate could also be interpreted as a binary number. Once again we encounter the need for you, as a programmer, to know in advance what a binary numeric digit sequence represents. And once again you can rest assured that this multiple use of binary digits never creates problems.

The various codes used to represent characters all use a byte (8 binary digits) to represent a single character. A byte has 256 different possible combinations of 8 binary digits:



7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
—	—	—	—	—	—	—	—	---
1	0	0	0	0	0	0	1	129
1	0	0	0	0	0	1	0	130
1	0	0	0	0	0	1	1	131
—	—	—	—	—	—	—	—	---
1	1	1	1	1	1	0	1	253
1	1	1	1	1	1	1	0	254
1	1	1	1	1	1	1	1	255

Thus a byte may be interpreted as:

1) A positive number with a decimal value in the range 0 through +255.
2) A signed number with a decimal value in the range -128 through +127.
3) One byte of a multibyte signed or unsigned number.
4) A character code.

ASCII

The most popular coding scheme used to represent characters is known as the American Standard Code for Information Interchange, generally referred to by the letters ASCII. The complete ASCII code set for all printable characters is given in Appendix A.

Only the ASCII character codes for the numeric digits 0 through 9 have a logical basis for their selection. If you look at these character codes you will see that the four low order binary digits exactly equal the numeric value associated with the character:

Binary Code	Hexadecimal Equivalent	ASCII Character
00110000	30	0
00110001	31	1
00110010	32	2
00110011	33	3
00110100	34	4
00110101	35	5
00110110	36	6
00110111	37	7
00111000	38	8
00111001	39	9

Notice that in Appendix A we show the binary digit pattern representing every character code using its hexadecimal equivalent. This makes the character codes far easier to read. **We can show the numeric equivalent of a text string using hexadecimal digits as follows:**

```
T  h  i  s     i  s     t  h  e     n  u  m  e  r  i  c
54 68 69 73 20 69 73 20 74 68 65 20 6E 75 6D 65 72 69 63 0D

e  q  u  i  v  a  l  e  n  t     o  f     a
65 71 75 69 76 61 6C 65 6E 74 20 6F 66 20 61 0D

t  e  x  t     s  t  r  i  n  g
24 65 78 74 20 73 74 72 69 6E 67 2E
```

Each of the text letters has beneath it the two hexadecimal digits which represent the ASCII code for the letter, as defined in Appendix A. Notice that in between words the hexadecimal code 20_{16} appears; this is the code for a space. The code $0D_{16}$ represents a carriage return.

STRING

The word "string" is commonly used to describe a sequence of characters stored via their numeric codes. That is why we referred to the text above as a "text string".

Within a computer your text will be stored as a sequence of bits (binary digits). When you want to print this text you fetch the bits in the proper sequence and transmit them to a display or printer. Logic associated with the display or printer interprets the binary data assuming that it represents characters.

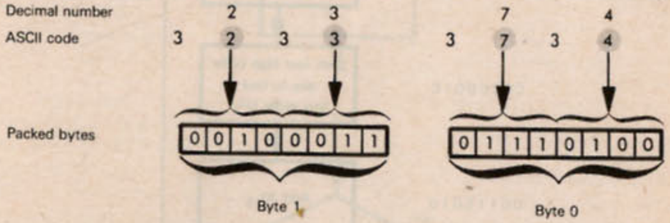
If you enter text via a keyboard, then each time you depress a key **the binary digit code associated with the key you depress is transmitted to the microcomputer —** which stores the code in appropriate memory.

You can modify character codes by treating them as binary data. Suppose, for example, that you have a large amount of numeric data which you wish to store in memory, and you have no alphabetic data. If you look again at the ASCII table in Appendix A you will see that all decimal digits have the same four high order bits:

ASCII code for decimal digit N is $3N_{16}$.

PACKED BYTES

You could save a lot of memory by packing decimal digits, two per byte, as follows:



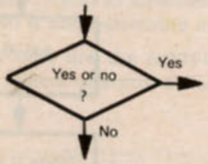
By performing the operations illustrated in Figure 4-1 on your character codes, you can re-create the ASCII characters.

COMPUTER LOGIC AND BOOLEAN OPERATIONS

A microcomputer will spend very little of its time doing arithmetic. In fact there are many programs that contain no arithmetic whatsoever. A computer will spend most of its time performing "logical" operations.

STATUS FLAGS

If you examine the flowcharts for program logic given in Chapter 2, you will frequently see the following type of decision step:



One common method for handling simple two-way decision logic, as illustrated above, is to provide the microcomputer with some special switches called "status flags". Events preceding the logic step must place one of these switches "off" or "on". The two-way decision making logic then becomes a single instruction which may be illustrated as follows:

"If flag is on, branch to instruction x.
If flag is off, continue with the next instruction."

Status flags represent one of the simplest forms of microcomputer logic. Different microcomputers have different numbers and types of status flags, but identifying them

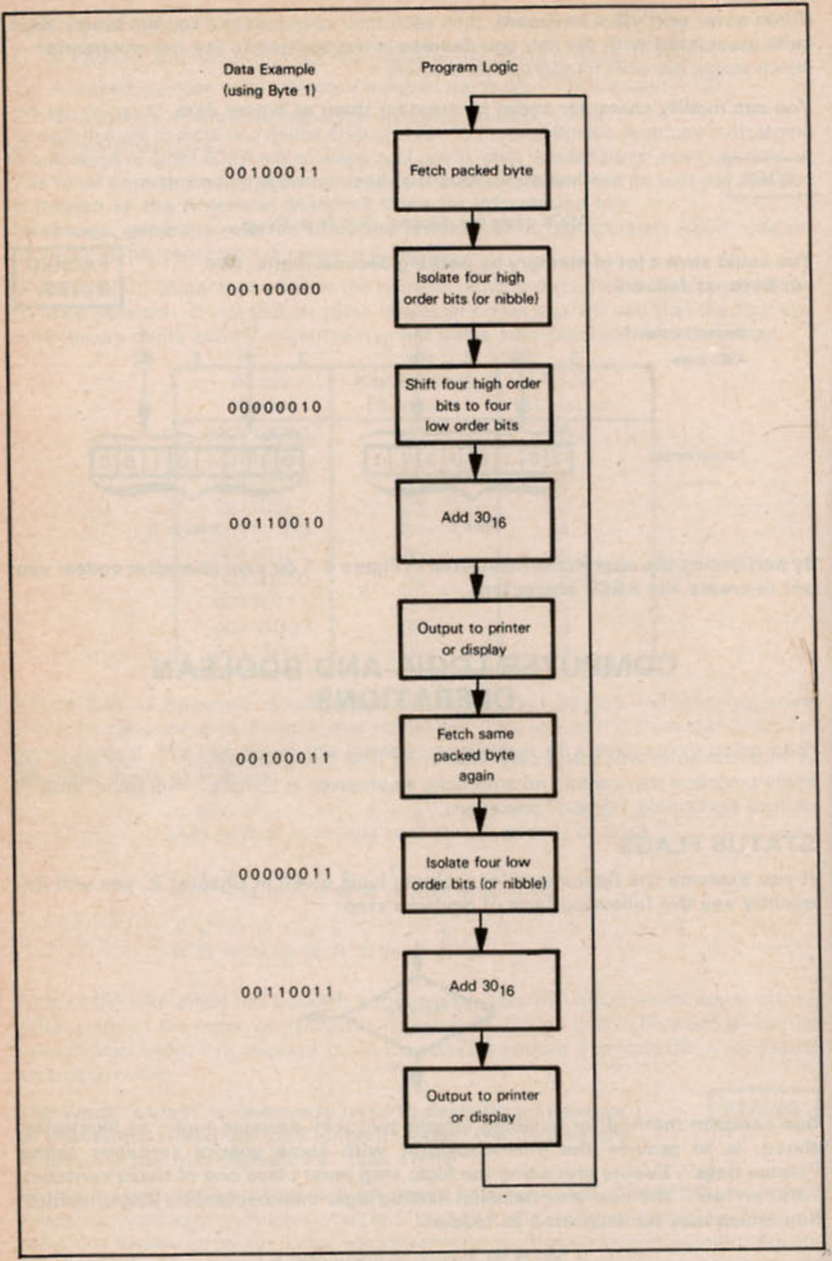
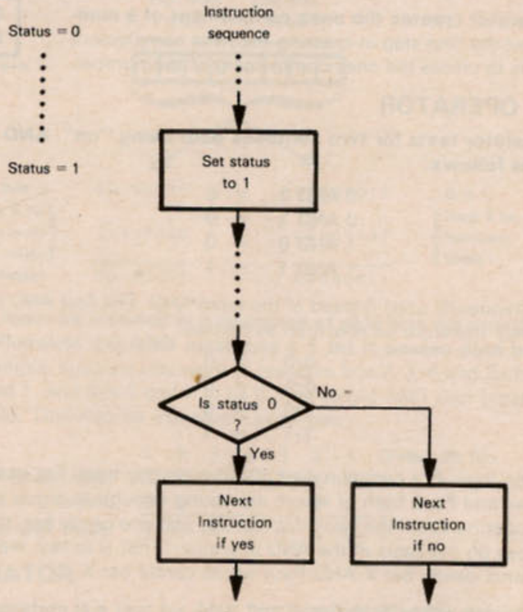

Data Example
(using Byte 1)
Program Logic
00100011
Fetch packed byte
00100000
Isolate four high order bits (or nibble)
00000010
Shift four high order bits to four low order bits
00110010
Add 30₁₆
Output to printer or display
00100011
Fetch same packed byte again
00000011
Isolate four low order bits (or nibble)
00110011
Add 30₁₆
Output to printer or display

Figure 4-1. Packed Byte Disassembly And ASCII Code Creation Logic

individually is unnecessary at this point. **In order to understand the concept of a status flag, all you need to think of is a two-way switch which instructions in your program can turn "on" or "off"; subsequent instructions in your program test the switch in order to determine which of two paths your program logic will take.** This may be illustrated as follows:



LOGICAL OPERATORS

LOGICAL OPERATORS

Most computer logic is generated by four "logical operators". There is nothing very mysterious about "logical operators"; addition, subtraction, multiplication and division are "arithmetic operators". A logical operator takes data input, does something non-arithmetic to it, and creates a result — just as an arithmetic operator takes data input, does something arithmetic to it and generates a result.

There are four logical operators; they are the NOT, AND, OR and Exclusive-OR operators. We will discuss each logical operator in turn.

THE NOT OPERATOR

The NOT operator is the simplest to understand. This operator simply says: move a switch to its opposite setting: that is, if it is "on", turn it "off" and if it is "off", turn it "on". Looking at the effect of **a NOT operator** on a bit (binary digit), it **converts a 1 to a 0 or a 0 to a 1.** This may be illustrated as follows:

```
      NOT 0  =  1
      NOT 1  =  0
 NOT 101101  =  010010
```

Frequently a bar over a number is used instead of the NOT. This may be illustrated as follows:

$$\overline{0} = 1$$
$$\overline{1} = 0$$
$$\overline{101101} = 010010$$

ONES COMPLEMENT

The NOT operator creates the ones complement of a number. Remember the first step in creating the twos complement of a number is to create the ones complement of the number.

THE AND OPERATOR

The AND operator tests for two switches both being "on". AND operations may be defined as follows:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

A dot (**.**) is frequently used instead of the word AND. The four AND operations illustrated above may therefore be rewritten as follows:

$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

AND operation logic is a common part of our everyday lives. For example, I have two small sons, Ian and Paul, both of whom are young enough to argue a lot. When shopping at the supermarket, if the two boys can buy just one candy bar, then the candy bar will be selected on the basis of the AND operation. That is to say, we buy candy bar X only if Ian wants candy bar X AND Paul wants candy bar X:

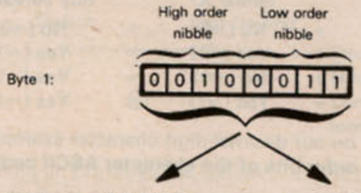
Ian wants candy bar A and
Paul wants candy bar X. $A \cdot X = 0$
No candy bar selected.

Ian wants candy bar X and
Paul wants candy bar X. $X \cdot X = 1$
Buy candy bar X.

If either Ian or Paul rejects a candy bar, then based on AND logic, the candy bar will not be selected.

BIT MASK

We have also seen a microcomputer application for the AND operation. Recall that we explained how numeric digit character codes can be stored two per byte. In Figure 4-1 **you could use an 8-bit mask and an AND operation in order to isolate one or the other numeric nibble.** This may be illustrated for byte 1 of Figure 4-1 as follows:



Byte 1:	00100011	00100011	Byte 1
Mask A for high order nibble	11110000	00001111	Mask B for low order nibble
(Byte1).(MaskA)	00100000	00000011	

Wherever a 0 bit must be inserted in a sequence of data bits we provide a 0 bit in an AND mask. Wherever the AND mask has a 1 bit it passes data bits through unaltered. For example, suppose you want to preserve bits 2, 3, 4 and 5 of a byte, but; you want bits 0 and 1, and bits 6 and 7 to be 0. You would AND your binary data with the mask 00111100. This may be illustrated as follows:

	7	6	5	4	3	2	1	0 ◄—Bit No.
Binary data:	X	X	X	X	X	X	X	X
mask:	0	0	1	1	1	1	0	0
Data AND mask:	0	0	X	X	X	X	0	0

THE OR OPERATOR

The OR logical operation is a test for ANY "on" switch. The OR operation may be defined as follows:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

The plus sign (+) is frequently used instead of OR. The four OR operations illustrated above may therefore be rewritten as follows:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 1$

The fact that a plus sign may be used to represent the OR logical operator may confuse you, but it is irrelevant to a computer. There will be distinct instructions, with their own independent instruction codes, representing an addition operation or a logical OR operation. The plus sign being used to represent addition or a logical OR operation will occur only in printed or written material.

The OR operation once again is a familiar part of our daily lives. For example, when stocking up on breakfast cereals at the supermarket, I will buy cereal X if either Ian or Paul wants it:

Does Ian want cereal X?		Does Paul want cereal X?		Buy cereal X?
No(=0)	+	No (=0)	=	No (=0)
No (=0)	+	Yes (=1)	=	Yes (=1)
Yes (=1)	+	No (=0)	=	Yes (=1)
Yes (=1)	+	Yes (=1)	=	Yes (=1)

We can use the OR operator on our decimal digit character example (Figure 4-1) in order **to provide the four high order bits of the character ASCII code.** This may be illustrated as follows:

	7	6	5	4	3	2	1	0 ← Bit No.
Isolated nibble:	0	0	0	0	0	0	1	1
OR mask for high order four bits:	0	0	1	1	0	0	0	0
	0	0	1	1	0	0	1	1

Wherever a 1 bit must be inserted in a sequence of data bits we provide a 1 bit in an OR mask. Wherever the OR mask has a 0 bit it passes data bits through unaltered. This may be illustrated as follows:

	7	6	5	4	3	2	1	0 ← Bit No.
Binary data:	X	X	X	X	X	X	X	X
Arbitrary mask:	0	0	1	1	1	1	0	0
Data OR mask:	X	X	1	1	1	1	X	X

Thus the AND operator may be used as a "clearing" mask while the OR operator may be used as an "inserting" mask.

THE XOR OPERATOR

The last logical operator we will describe is the Exclusive-OR, or XOR. The Exclusive-OR tests for differences and changes; it may be defined as follows:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

The ⊕ symbol is frequently used instead of the letters XOR. Thus the four XOR operations illustrated above may be rewritten as follows:

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

The Exclusive-OR is also a part of our daily logical lives; it identifies differences of opinion. For example, a fight results when Ian says yes and Paul says no:

Ian's opinion		Paul's opinion		Fight?
No (=0)	⊕	No (=0)	=	No (=0)
No (=0)	⊕	Yes (=1)	=	Yes (=1)
Yes (=1)	⊕	No (=0)	=	Yes (=1)
Yes (=1)	⊕	Yes (=1)	=	No (=0)

In computer logic you will use the Exclusive-OR operation to check for changes in state. Suppose, for example, knowing that a switch is "on" or "off" is insufficient; you also need to know whether the switch has been switched "on" or "off" since you last tested it. You can save the condition of any switch each time you test it, then compare the switch position with the saved condition as follows:

Switch		Last Setting	
X	⟶	X	Save switch setting
Y			New switch setting
Y	⊕	X	Compare settings and look for change

Thus by performing an Exclusive-OR operation on the switch condition and its previous setting you can find out whether the switch changed setting since you last tested it.

Chapter 5
INSIDE A MICROCOMPUTER

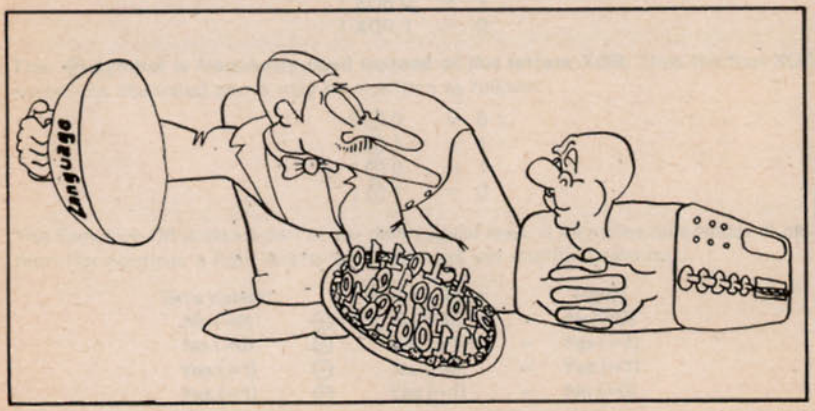
We have now described overall microcomputer concepts in Chapters 1, 2 and 3; then in Chapter 4 we went to the other extreme, defining the basic concepts out of which any computer function can be created. It is now time to bridge the gap between the fundamental concepts and the end product — the microcomputer system. We are going to bridge this gap in the next two chapters. In this chapter we will look at the microcomputer itself, separating and exploring its various components. In Chapter 6 we will look at the way in which the basic digital logic concepts of Chapter 4 can be used to create the components of the microcomputer system described in this chapter.

In order to examine the functional components of a microcomputer let us begin by looking at the way in which a microcomputer may be programmed using a programming language.

ABOUT PROGRAMMING LANGUAGES

There are so-called "higher level" languages, such as BASIC, FORTRAN and COBOL. There are also "fundamental" programming languages referred to as "assembly language".

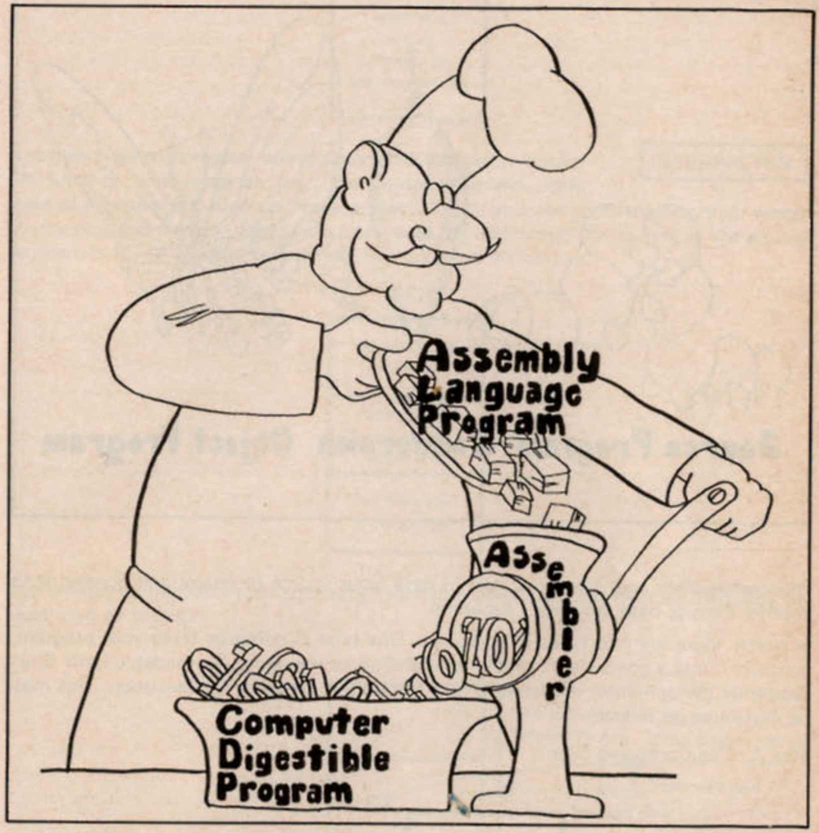
Regarding any programming language, the most important point to understand is that **a programming language is a programmer's convenience.** A programming language is an artificial creation, designed to make your life as a programmer easier. Whatever language you decide is best for you, **the computer still demands that it receive the program as a sequence of numbers:**



Now the computer will, itself, take care of converting the program from the form in which you, the programmer, write it, to the form in which it, the computer, can understand and execute it. In order to make this conversion, the computer executes another program — a program which someone else wrote for you.

ASSEMBLER

A program called **an "assembler" converts programs which you write in assembly language** into programs which the computer can understand and execute:



COMPILER

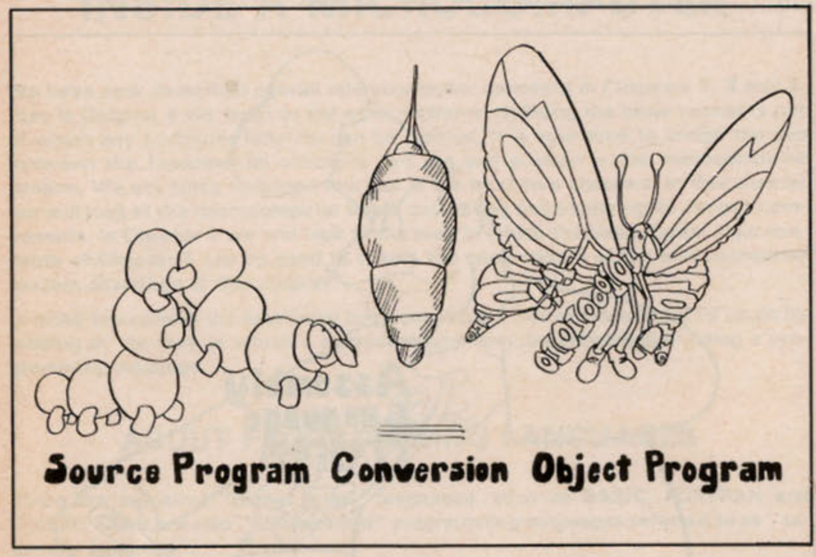
A program referred to as **a "compiler" accomplishes the same conversion task for programs which you write using a higher level language.**

Assemblers and compilers treat your program as data; they read in data (your program) and convert it to another form of data (the computer executable version of your program).

SOURCE PROGRAM

OBJECT PROGRAM

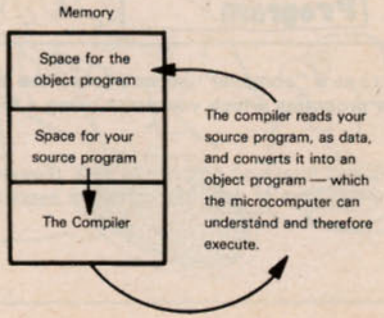
We refer to a program in human readable form as a "source program". That is to say, a source program is a program written in a programming language. **Once the program has been converted into its computer readable form, it is called an "object program".** An object program is nothing but a sequence of numbers. This may be illustrated as follows:



Thus assemblers and compilers read in data (your source program) and convert it to another form of data (an object program).

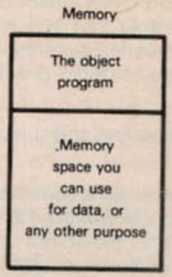
In reality there are two types of compilers. One type of compiler takes your program, converts it into a computer readable form and saves the computer readable form. Subsequently the computer readable form is loaded into memory for execution. This may be illustrated as follows:

Step 1 - The Compiling Step



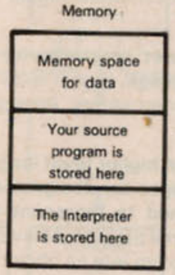
The compiler reads your source program, as data, and converts it into an object program — which the microcomputer can understand and therefore execute.

Step 2 - The Execution Step

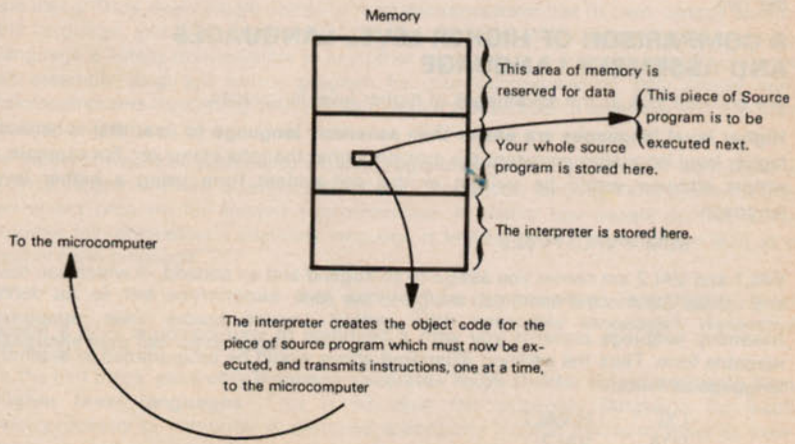


INTERPRETER

Another type of compiler never saves the computer readable form of your program (i.e., the object program). This type of compiler is called an "Interpreter". When you use an interpreter, your whole source program resides in memory, along with the interpreter, for as long as the source program is being executed. This may be illustrated as follows:


Memory
Memory space for data
Your source program is stored here
The Interpreter is stored here

The interpreter converts your source program into object code as needed. This may be illustrated as follows:


Memory
This area of memory is reserved for data
This piece of Source program is to be executed next.
Your whole source program is stored here.
The interpreter is stored here.
To the microcomputer

The interpreter creates the object code for the piece of source program which must now be executed, and transmits instructions, one at a time, to the microcomputer

The illustration above shows an area of memory being set aside for your whole source program. You might be misled into thinking that the amount of memory set aside for your source program puts an upper bound on the size of source program which you can execute. In fact, you can execute much larger programs so long as the larger program can be broken into blocks, where no one block overflows the available source program memory space.

Compilers and Interpreters are themselves object programs which someone else wrote for you.

We can explain the difference between a compiler and an interpreter in non-technical terms by thinking of the ways in which an actor may learn to deliver lines in a play. Think of the source program as the actor's script; object program instructions going to the microcomputer are equivalent to the actor delivering his lines to an audience. If the actor learns his entire part, then throws away the script and delivers his lines, what he has done is equivalent to compiling a source program. But suppose the actor does not learn his entire part; suppose the actor keeps the script and has a prompter display his lines one at a time, using prompting boards. He is now delivering his lines in the fashion of an interpreter.

BASIC is the most popular microcomputer higher level language; it is also an interpreter language.

In summary, we can divide most programming languages into "higher level" languages and "assembly" language. Higher level languages are converted into object programs by compilers and interpreters. Assembly languages are converted into object code by an assembler.

The principal difference between higher level languages and assembly language is the fact that higher level languages are designed to represent problems, whereas assembly languages are designed to represent the computer. Thus a computer views a higher level language source program as a very alien thing and a compiler has a big job converting the source program into an object program. In contrast, an assembly language source program can be converted into an object program quite easily; an assembler is therefore a relatively simple program. Let us now compare higher level languages and assembly language in order to more clearly identify differences between the two.

A COMPARISON OF HIGHER LEVEL LANGUAGES AND ASSEMBLY LANGUAGE

We will first look at the advantages of higher level languages.

Higher level languages are easier than assembly language to use; that is because higher level languages represent the problem rather than the computer. For example, a simple addition would be written in this self-evident form using a higher level language:

```
        SUM = VAL1+VAL2
```

VAL1 and VAL2 are names you assign to an augend and an addend — which can have any values. SUM is the name you assign to the sum.

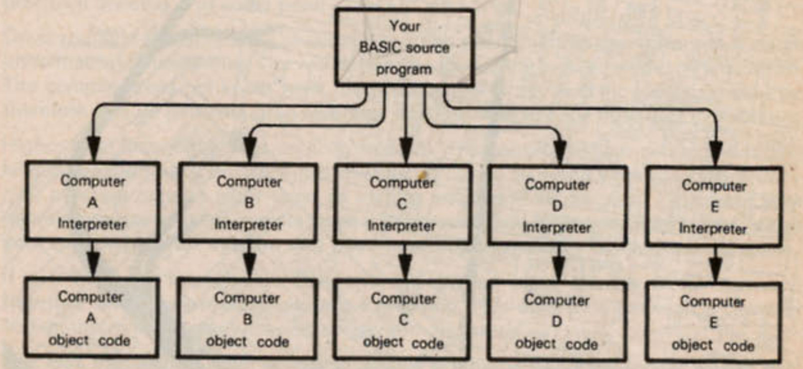
Assembly language presents you with a definition of your computer — in a human readable form. Thus the addition illustrated above would be programmed in assembly language as follows:

```
        LXI     H,VAL1
        LDA     VAL2
        ADD     A,M
        STA     SUM
```

VAL1 and VAL2 are no longer names you assign to the augend and addend; VAL1 and VAL2 are now addresses — they identify memory locations in which the augend and addend are stored. Thus the augend and addend must each be small enough to fit within one memory location. SUM, likewise, is the address of the memory location where the sum will be stored — providing it will fit into one memory word.

The assembly language definition of the addition is by no means self-evident.

There is another important advantage associated with the fact that higher level languages are "problem-oriented". What we mean by "problem-oriented" is that the language is not designed with any computer in mind. Therefore **if you write a program in a higher level language, you can convert this higher level language source program into an object program that will run on any computer — providing the computer has a compiler (or interpreter) for your higher level language.** Suppose, for example, you write a program in BASIC. You can execute this BASIC program on your computer; and all of your friends can execute your program on their totally different computers — providing their computers also have BASIC interpreters. This may be illustrated as follows:



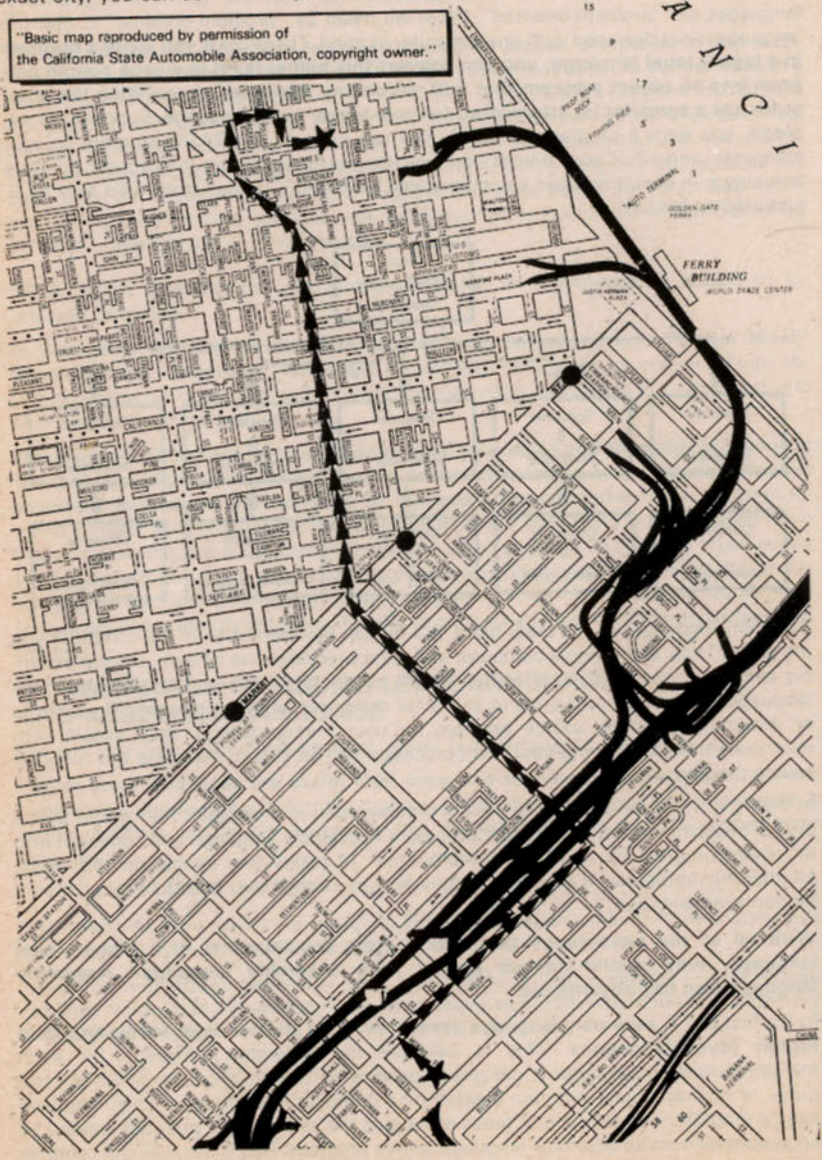
Assembly language, on the other hand, is a human representation of the computer you are using. Thus, every single computer and microprocessor has its own, unique assembly language; and a program written in one computer or microprocessor's assembly language is totally unintelligible to any other computer or microprocessor. If you write an assembly language source program for your microprocessor, only people with microcomputers containing your microprocessor will be able to assemble and run your source program.

In theory it would be possible to write a program akin to a compiler that would take a source program written in one microprocessor's assembly language and convert it into an object program for another microprocessor. In reality few people do this, since another microprocessor's assembly language is as strange and hard to deal with as a higher level language.

With all of the advantages that accrue from programming in a higher level language, why would anyone bother with assembly language? Assembly language also has advantages.

In the first place, **assembly language generates much shorter object programs than higher level languages.** This is because the assembly language for each microprocessor or computer is designed specifically for that microprocessor or computer. In fact, an object program created by a compiler from a higher level language source program is usually 2 to 4 times as long as the same object program created by an assembler from an assembly language source program. This is because the compiler must, in reality, write an assembly language program to represent the problem, as defined in the higher level language. But whereas a human programmer can write an assembly language program using human judgement, a compiler must do the job by fixed rules.

Consider an everyday analogy: you must give someone directions to drive from one point to another in a city. If you know the exact source and destination, and the exact city, you can define a very direct route:



"Basic map reproduced by permission of the California State Automobile Association, copyright owner."

Now try to create a set of general-purpose instructions which you can string together in order to define the route to be driven between any two points in any city. These instructions, if they are to be interpreted by a machine, can leave nothing to the imagination. Thus there must be some fixed number of instructions such as:

Turn left
Turn right
Test for a one way street
Test for a dead end road
Test for a 45° turn
etc.

You cannot include instructions that assume you know whether or not a street is one way, since one way streets are subject to change. You cannot include instructions that simply define the number of blocks to travel in a straight line, since there may be barriers in the road preventing such travel; or in cities with steep hills such as San Francisco, a road which appears to be continuous in reality has a 100 foot (i.e., 30 meter) precipice dividing it at some point.

Once you start devising a set of general purpose direction rules that take into account undefinable contingencies, you will have some idea of the problem faced by a compiler. The compiler does not know what the peculiarities of any specific computer may be, therefore it must generate programs that take into account the strangest possibilities.

Higher level languages have another problem. The compiler which converts a higher level language source program into an object program is itself a large program. A compiler program may be eight times as long as an assembler program. Thus **until your microcomputer system is quite large you cannot use a higher level language, since your microcomputer system will have insufficient memory to hold the compiler.**

If you have an interpreter, then the interpreter must always be in memory, together with the program you are executing. This difference between a compiler and an interpreter was illustrated earlier in the chapter.

The fact that higher level language source programs generate longer object programs also means that the object program will take longer to execute, since there are more instructions to be executed. If your application is running into speed problems, you can speed things up by a factor of 2, or more, by simply re-writing your program into assembly language.

Even some of the advantages associated with higher level languages are not all they appear to be. For example, **higher level languages are supposed to be portable;** that is to say, one higher level language source program can be compiled and executed by many different microprocessors. This is not always true. **Frequently you will find that there are minor differences in the way one computer's compiler expects the source program to appear, as compared to the next.** However, even in the worst case, the changes you would have to make to a higher-level language source program, when going to a new microprocessor or computer, are tiny compared to the problems associated with completely re-writing the program in the new microprocessor or computer's assembly language.

What then is our conclusion?

If you are going to use a microcomputer simply as a vehicle for executing programs, you should go to higher level languages as quickly as you can. If, on the other hand, you plan to get inside the microcomputer itself, building your own, changing it, extending it, or otherwise playing with its components, then you should learn assembly language as quickly as possible, and you will probably stay with assembly language.

MICROCOMPUTER FUNCTIONAL LOGIC

The object program you create determines the functions that will be performed by the logic of your microcomputer.

Functionally Figure 5-1 illustrates the logic of a microcomputer; this is the logic which we are now going to discuss.

It does not matter what the microcomputer is going to do — ultimately the task consists of these three steps:

1) **Bringing data into the microcomputer.**
2) **Modifying the data.**
3) **Transmitting the modified data back out from the microcomputer.**

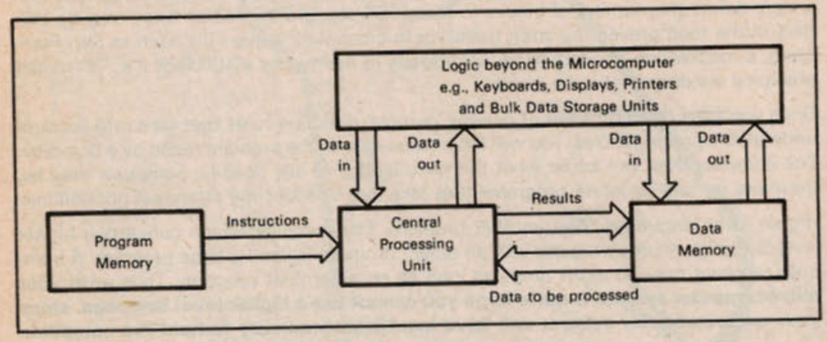
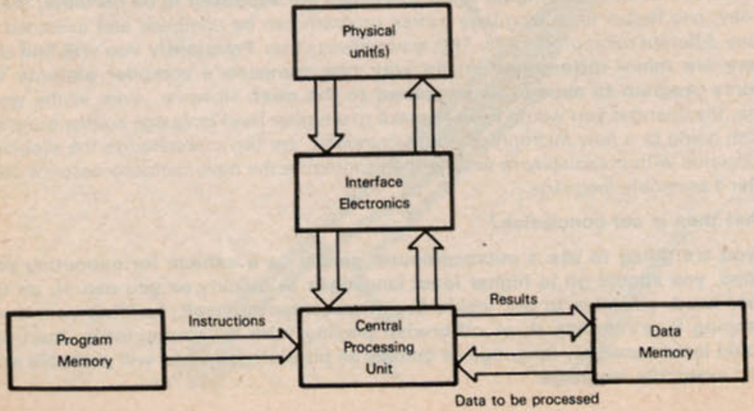


Figure 5-1. Microcomputer Functional Logic

Logic beyond the microcomputer (which consists of physical units we described earlier in this book) is used to enter information, receive results and store large quantities of data. Data that is in the process of being operated on is stored in data memory, which you will recall (from Chapter 2) is fast access, read/write memory. Therefore, **steps 1) and 3) above are handled by the shaded microcomputer logic shown in Figure 5-2.**

Physical units, you will recall, transmit information to and from the microcomputer via appropriate interface logic. With reference to Figure 5-1 this may be illustrated as follows:



CENTRAL PROCESSING UNIT

Operations which are actually performed on data are performed by logic within the Central Processing Unit. These operations are defined by a sequence of instructions which, taken together, constitute a program. The program is stored in program memory. Thus **step 2) of the above three steps is handled by the shaded microcomputer logic shown in Figure 5-3.**

PROGRAM MEMORY

Program memory can be read only memory, or it can be read/write memory. Program memory can be read only memory because instructions are transmitted from the program stored in program memory, to the Central Processing Unit; but instructions are usually not transmitted from the Central Processing Unit to program memory. Program memory does not have to be read only memory. It is common practice in microcomputer systems to separate programs from data, as shown in Figure 5-1, and in many industrial microcomputer applications, programs are held in read only memory to ensure that the program is never accidentally changed or lost.

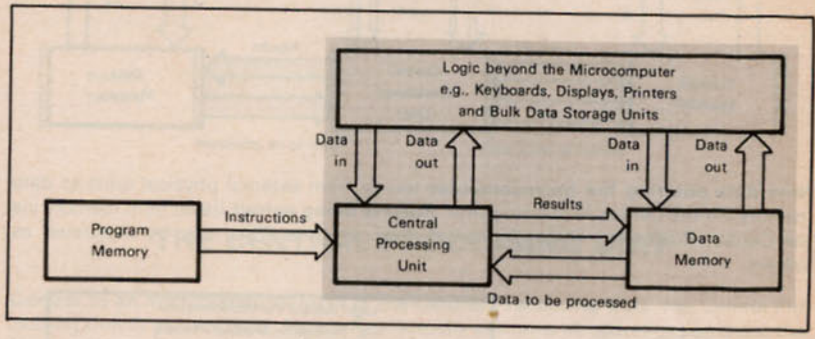


Figure 5-2. Microcomputer Functional Logic Involved In Data Movement And Storage

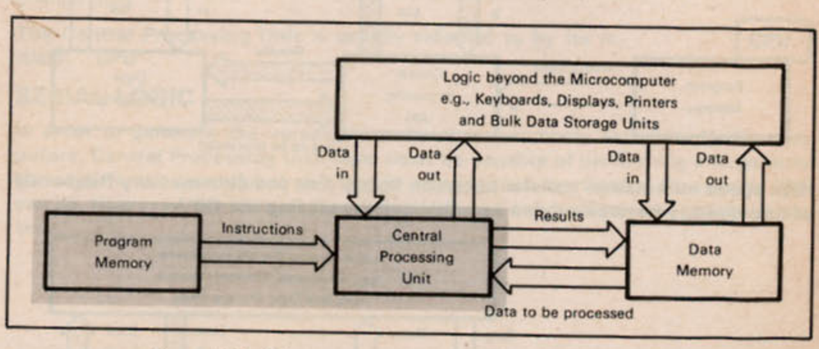


Figure 5-3. Microcomputer Functional Logic Involved In Data Modification

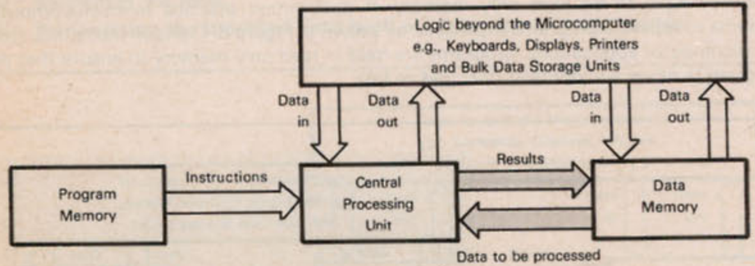
But **program memory and data memory could be one and the same memory;** moreover, it is possible for one part of a program to treat another part of the program as data, in which case the program changes itself. As you might expect, programs which change themselves can become very complex; so at least while you are a beginner, it is wise to think of program memory and data memory as separate and distinct entities.

The fact that you do not have a good understanding yet of how program and data memories work is unimportant. Photographs of program and data memory chips are shown in Chapters 1 and 2. These chips can store information in a computer-readable form. For now that is all you need to know about program and data memory.

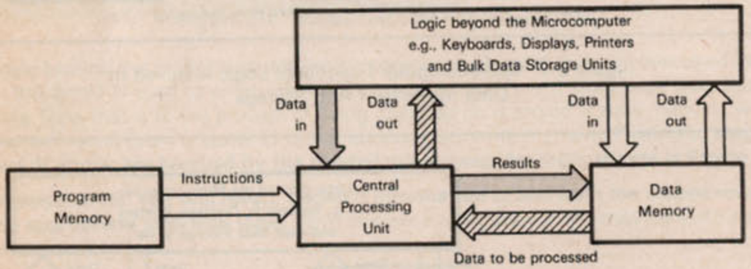
INFORMATION PATHS

Let us now consider the various information paths shown in Figure 5-1.

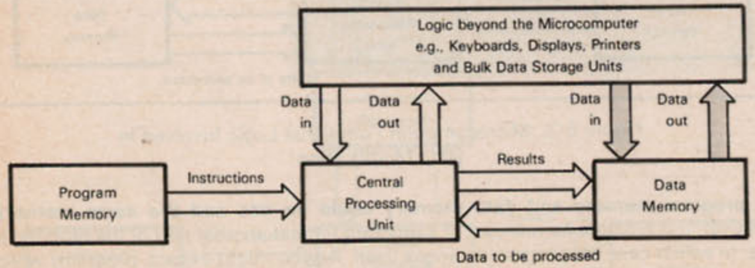
When the Central Processing Unit is modifying data, it usually fetches the data to be modified from data memory, and it usually returns the results to data memory. **Therefore there are paths in both directions between data memory and the Central Processing Unit:**



New data entering the microcomputer travels from external physical units to data memory via the Central Processing Unit. **Results being output** travel from memory via the Central Processing Unit to external physical units. This may be illustrated as follows:



High speed information transfer between floppy disk and data memory frequently occurs directly between these two devices, bypassing the CPU:

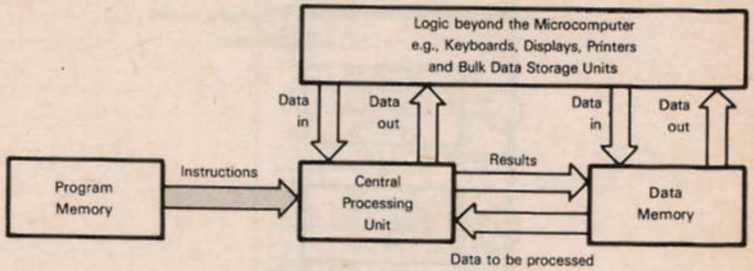


DIRECT MEMORY ACCESS

DMA

The data path illustrated above is referred to as Direct Memory Access. Direct Memory Access is usually referred to by its initials: DMA. While memory has to be at one end of the DMA data transfer, a floppy disk need not be at the other end, even though it frequently is. Any external logic may provide the other end of the DMA data transfer.

Whenever the Central Processing Unit is doing something — moving data or modifying data — a stream of instructions transmitted from program memory to the Central Processing Unit controls Central Processing Unit operations. Thus **there must be a unidirectional path for information to flow from program memory to the Central Processing Unit:**



THE CENTRAL PROCESSING UNIT

Central to all microcomputer logic is the Central Processing Unit. The Central Processing Unit is the electronic logic which actually performs all operations on data; that is to say, in various other parts of the microcomputer system you can move data from one location to another, but only within the Central Processing Unit can you actually change data.

CPU

The Central Processing Unit is usually referred to by its initials: "CPU".

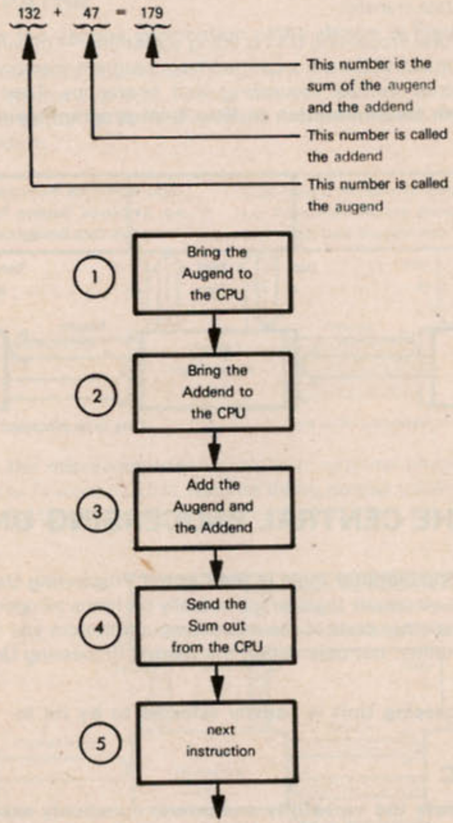
SERIAL LOGIC

In order to generate the versatility and power commonly associated with computers, Central Processing Unit logic must be capable of performing a large number of different operations; and that is indeed what the Central Processing Unit can do. However, the Central Processing Unit can only perform one operation at a time.

AUGEND

ADDEND

Consider the addition of two numbers; when two numbers are added, they are called the augend and the addend. The augend and the addend are summed via the following serial sequence of events:


132 + 47 = 179
This number is the sum of the augend and the addend
This number is called the addend
This number is called the augend
1
Bring the Augend to the CPU
2
Bring the Addend to the CPU
3
Add the Augend and the Addend
4
Send the Sum out from the CPU
5
next instruction

Each event is identified by a number①.②.③.④. etc. The CPU performs each event as a single operation. Therefore, in order to perform the addition illustrated above, the **CPU performs event①, then event②, then event③, then event④.**

During the first step the augend is brought to the CPU.

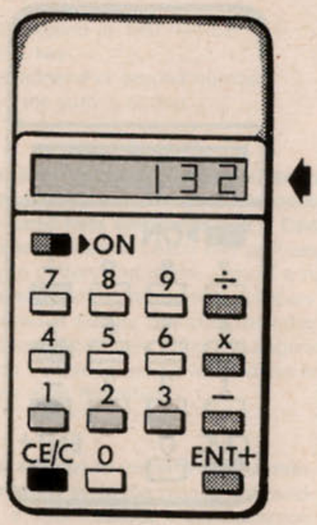
During the second step the addend is brought to the CPU.

During the third step the augend and addend are summed by electronic logic within the CPU.

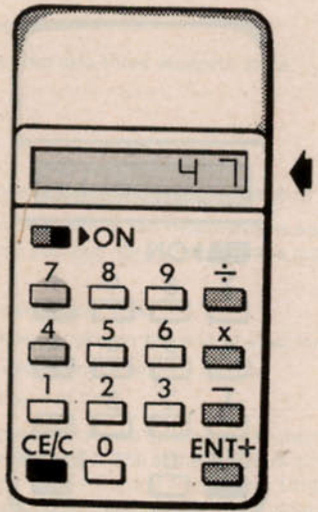
During the fourth step the sum is transmitted out from the CPU.

These four steps are essentially identical to the four steps via which you will add two numbers using some types of hand-held calculators.

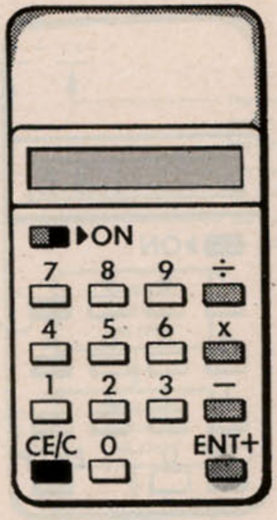
During step one you will key in the augend:



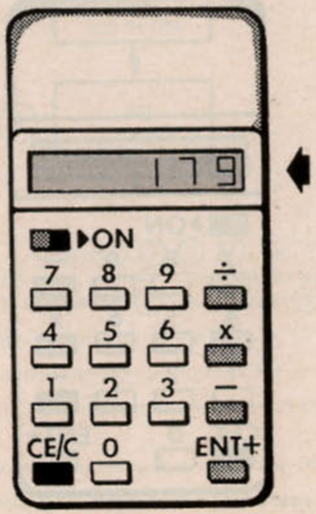
During step two you will key in the addend:



During step three you will press the + key:



Step four occurs automatically: the sum is output, from logic of the calculator, to a display where you can read it:



Now you know why some calculators make you do things awkwardly; they force you to use computer logic sequences.

Many calculators use a more complex logic which lets you work in algebraic sequence, which is the way we learned arithmetic.

During step 1 you key in the augend or first number.
During step 2 you press the + key.
During step 3 you key in the addend or second number.
Step 4 occurs automatically: the sum is output.

SERIAL DEVICES

We can use the four hand-held calculator steps (either version) via which you add two numbers to illustrate the concept of a serial device, since a hand-held calculator and a Central Processing Unit are both serial devices; each can perform just one operation at a time. This is simple enough to understand in the case of a hand-held calculator; you cannot, for example, simultaneously key in the two numbers which are to be added. The two numbers must be keyed in serially, one after the other. In the case of a Central Processing Unit, you cannot simultaneously bring the augend and the addend to the Central Processing Unit; each number must be fetched via an independent step, and the two steps must occur one after the other.

SERIAL LOGIC STEP

INSTRUCTION STEP

The next problem that we are going to run into is determining what a single "step" consists of. In the case of the hand-held calculator, this is not a very important consideration. When you enter the number 132 via the keys, does entry of the entire number constitute one "step"? Or does each keystroke constitute an individual "step"? Frankly, for a hand-held calculator, this question is inconsequential. But what if you have to write down a sequence of instructions which someone else must follow? You could write down the following single step:

1) Enter 132 at the keyboard

You could break up the one step into three separate steps:

1) Press the 1 key
2) Press the 3 key
3) Press the 2 key

Consider an even more mundane example: eating a piece of cake.

Suppose a piece of cake can be eaten in ten mouthsful; is eating this piece of cake a ten-step process? Perhaps, but perhaps not. Eating a single mouthful of cake may itself consist of these four steps:

1) Separate a piece of cake with your fork.
2) Impale the separated piece of cake on the end of your fork.
3) Transfer the separated piece of cake to your mouth.
4) Chew and swallow the piece of cake.

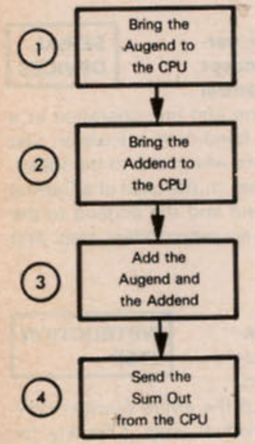
INSTRUCTIONS

It would be easy to nitpick these four cake eating steps, creating any number of additional smaller steps. The same is true of single Central Processing Unit steps. Some Central Processing Units perform operations in relatively big steps; others sequence events as a series of relatively small steps. But **for every Central Processing Unit, every step is clearly and unambiguously defined as an "instruction".** There is nothing vague about an individual instruction, or step, that can be executed by any Central Processing Unit.

INSTRUCTION SET

Every Central Processing Unit responds to a fixed number of instructions. These instructions, taken together, are referred to as an instruction set. Typically a Central Processing Unit will have from 40 to 200 different instructions in its instruction set.

Every instruction is represented by a unique number, which when transmitted to the Central Processing Unit at the proper time, causes the Central Processing Unit to execute the operations associated with the instruction. For example, our addition sequence may be illustrated as follows:


1
Bring the Augend to the CPU
2
Bring the Addend to the CPU
3
Add the Augend and the Addend
4
Send the Sum Out from the CPU

This becomes a single instruction represented by the number 156010048

This becomes a single instruction represented by the number 156010049

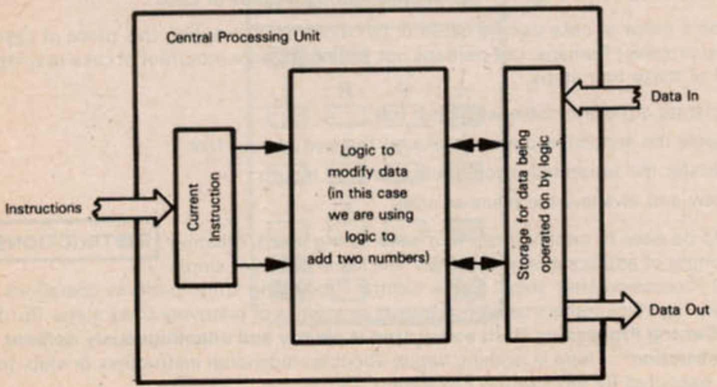
This becomes a single instruction represented by the number 128

This becomes a single instruction represented by the number 096010049

CENTRAL PROCESSING UNIT LOCAL DATA STORAGE

The four instructions shown above illustrate a logistic problem associated with the CPU.

The CPU has storage space to hold the data that it is about to operate on, and that is all. This may be illustrated as follows:

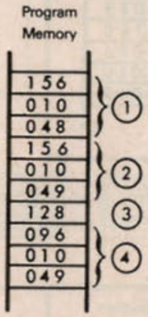

Central Processing Unit
Data In
Instructions
Current Instruction
Logic to modify data (in this case we are using logic to add two numbers)
Storage for data being operated on by logic
Data Out

You cannot expect to leave the augend, addend and sum in the Central Processing Unit data storage space, because you will almost certainly need this space for the very next operation which the Central Processing Unit performs. The augend, the addend and the sum must therefore have permanent storage locations somewhere beyond the Central Processing Unit — for example, in external read/write memory. **That is why steps ①, ②, ④ are present.**

PROGRAM MEMORY

PROGRAM

In order to perform any operation, such as the illustrated addition, you must create a sequence of instructions, which taken together constitute a program. The program is a sequence of numbers. This sequence of numbers is stored in a fast access memory, which we call program memory. Using arbitrarily assigned number codes for the addition instructions, **the addition program may be represented conceptually as follows:**



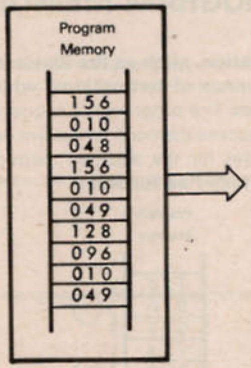
The method used above to illustrate memory contents is one which you are going to see frequently in this book, and in other books of this series. Memory is being likened to a ladder of "pigeon holes"; each "pigeon hole" represents an individually identifiable and addressable location.

Whenever a number is transferred from the CPU to memory, one "pigeon hole" will be filled. When a number is transferred from memory to the CPU, the CPU receives the contents of one "pigeon hole".

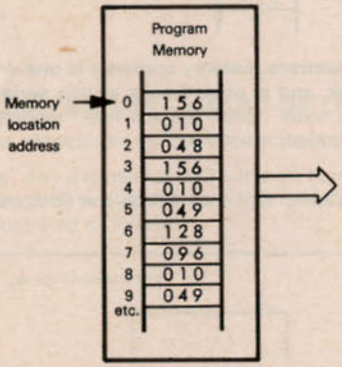
MEMORY LOCATIONS AND ADDRESSES

Each "pigeon hole" is called a "memory location". Every memory location is individually identifiable via a unique memory address.

We are not going to concern ourselves with how you create the memory address which identifies any individual addressable location within memory; therefore the addition program instruction sequence illustrated above will be represented occupying an undefined sequence of program memory locations as follows:



Without discussing memory addressing at all, we could illustrate the addition program instruction sequence occurring in the first ten addressable locations of program memory as follows:



It takes no understanding of computer logic in order to see how the first ten addressable locations of program memory may be filled with numbers as illustrated above. It is going to take some understanding of computer logic in order to explain how we identify any one of these, or any other memory location. This subject is discussed in Chapter 6.

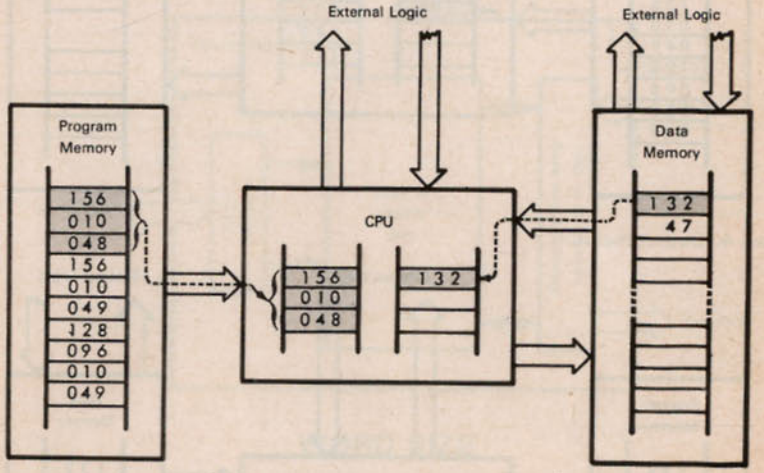
DATA MEMORY

The information which is used by a program while it executes is referred to as data. **In our simple addition example we are going to handle three pieces of data: the augend and the addend which are to be added, and the sum. These three pieces of data will likely be stored in local, fast access data memory.**

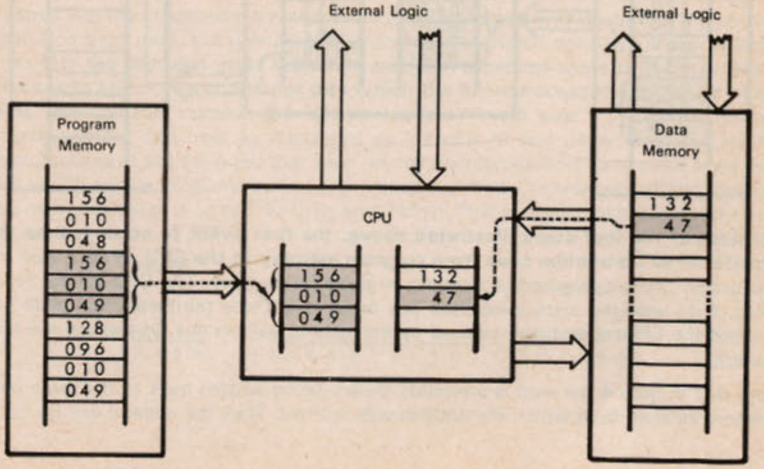
ADDITION PROGRAM EVENT SEQUENCE

The process of adding two numbers may now be illustrated conceptually as follows:

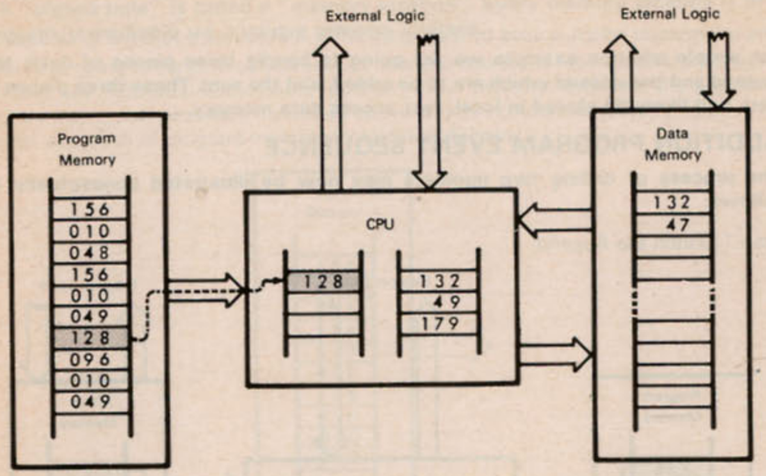
Step 1: Fetch the Augend



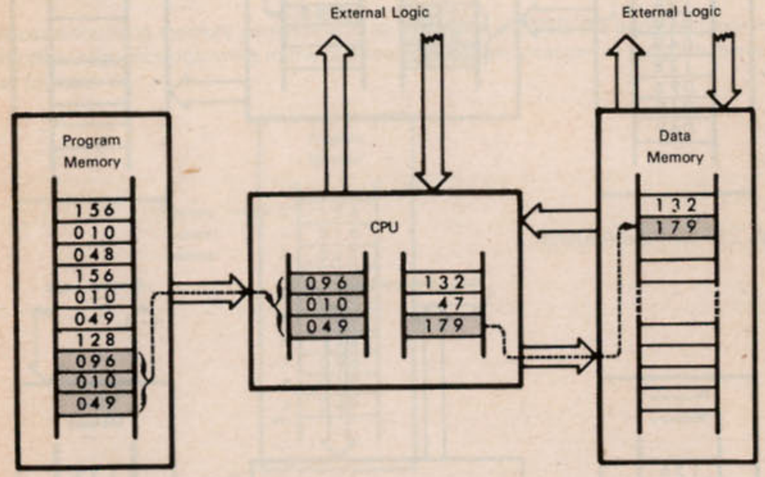
Step 2: Fetch the Addend:



Step 3: Generate the Sum:



Step 4: Output the Sum:



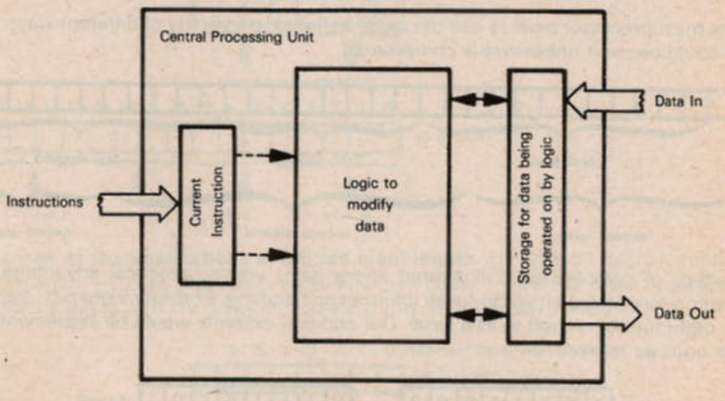
For each of the four steps illustrated above, the first event to occur will be the transfer of an instruction code from program memory to the CPU. In each step the instruction code is the shaded numbers in program memory. The CPU cannot know what to do until the instruction code has reached it. Once the instruction code has reached the CPU, operations required by the step actually occur. Operations are self-evident.

Note that in Step 4 the sum is arbitrarily shown being written back to the same data memory location from which the addend was fetched. Thus the addend will be lost.

Chapter 6
PUTTING IT ALL TOGETHER

We will now look at the logic of the Central Processing Unit itself.

Chapter 5 superficially illustrated a Central Processing Unit as follows:

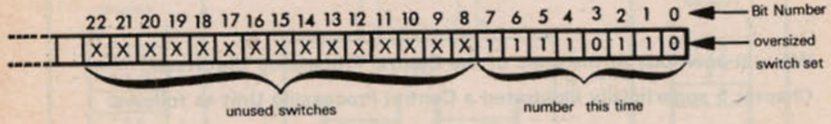


WORD SIZE

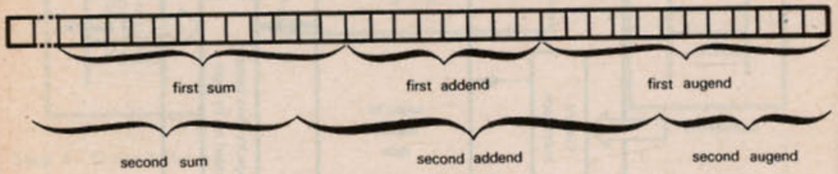
In Chapter 4 we saw how instruction codes and data of various types all eventually become identical-looking bit patterns. Possibly the most important conceptual result of Chapter 4 is the conclusion we must draw that two-state switches, or bits, are not very useful on their own; but groups of bits can be interpreted variously and powerfully. Therefore **the first and most important decision a microprocessor designer has to make is to select the number of bits which the microprocessor will handle at one time. We call this number the microprocessor "word size".** You could design a microprocessor that uses as many bits as it needs at any given time; but no real microprocessors are designed that way. All real microprocessors have some fixed word size which applies to every type of information handled. Consider arithmetic; suppose the microprocessor is to add 4213_{10} and 246_{10}. This may be illustrated as follows:

```
  Decimal                      Binary
               12 11 10 9  8  7  6  5  4  3  2  1  0 ◄— Bit No.
  4 2 1 3       1  0  0  0  0  0  1  1  1  0  1  0  1    Augend
+   2 4 6   +                     1  1  1  1  0  1  1  0    Addend
  -------      ---------------------------------------
= 4 4 5 9   =   1  0  0  0  1  0  1  1  0  1  0  1  1    Sum
```

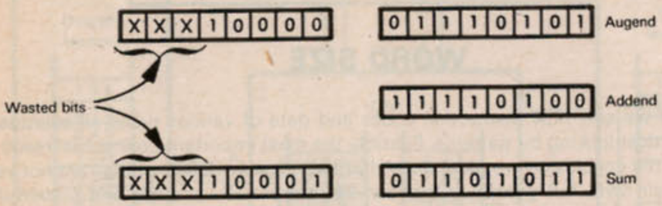
The two numbers which are to be added require thirteen bits for the augend, eight bits for the addend and thirteen bits for the sum. A microprocessor could conceivably be designed so that for the addition example illustrated above it assigns thirteen switches for the augend and the sum, and eight switches for the addend; but what happens if the very next addition requires 7 switches for the augend, 15 switches for the addend' and 16 switches for the sum? Either the microprocessor must have some oversized switch set and waste a lot of switches most of the time:



or the microprocessor must re-use the same switches in a variety of different ways; and that could become unbelievably complicated:

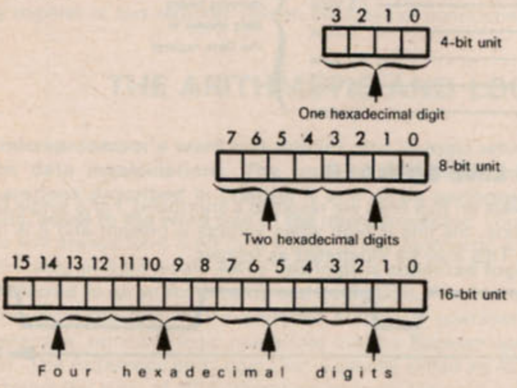


The type of complex logic illustrated above gains you no practical advantage over adopting some fixed bit width for all information handling. Consider eight bits; (remember, eight bits is referred to as a byte). Our addition example would be implemented in byte units as follows:

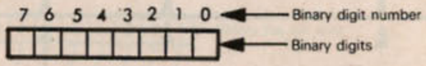


Yes indeed, there are some wasted bits, but the simplified logic is more than enough compensation.

Most of the popular microprocessors on the market today handle information in 8-bit units. Consequently these microprocessors are referred to as "8-bit" machines. **A few obsolete microprocessors handle information in 4-bit units, while some of the newer more powerful microprocessors handle information in 16-bit units.** Notice that these bit lengths — 4, 8 and 16, are all designed to allow easy counting within a binary system:

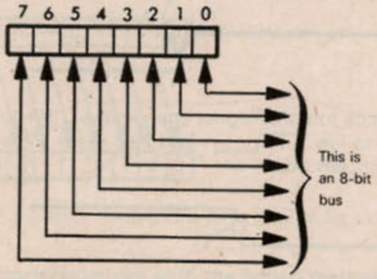


Let us look at the implications of a fixed word length. For an 8-bit microcomputer, all information — data, characters and instruction codes — must be stored in units of eight binary digits. We are going to represent this 8-bit unit as follows:

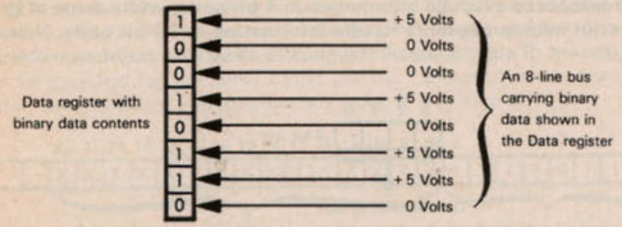


BUSSES

Information must also be transmitted from one part of a microcomputer system to another. Since all information is handled as 8-bit units, all information transfers must occur eight bits at a time; therefore these transfers occur via eight parallel conductors:

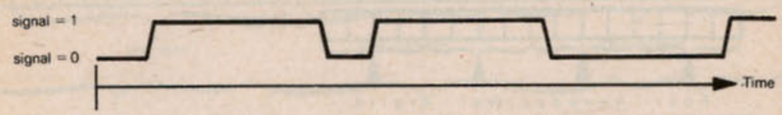


We call the conductors "busses". A "bus" is nothing more than a collection of electrical conducting lines over which binary data is transferred. A 1 bit or "on" switch is represented by the presence of voltage on the conductor; a 0 bit or "off" switch is represented by the absence of any voltage on the conductor. This may be illustrated as follows:

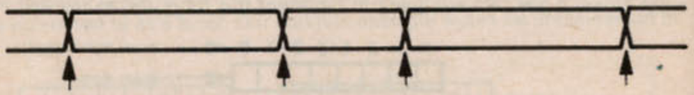

Data register with binary data contents
1
0
0
1
0
1
1
0
+5 Volts
0 Volts
0 Volts
+5 Volts
0 Volts
+5 Volts
+5 Volts
0 Volts
An 8-line bus carrying binary data shown in the Data register

REPRESENTING BUS LINE SIGNALS

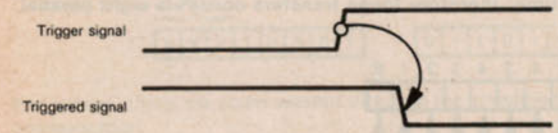
In order to identify signal levels on bus lines we resort to a shorthand. For a single bus line we draw a continuous line; the line is high when voltage is present and it is low when no voltage is present. This may be illustrated as follows:



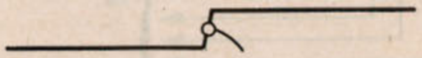
When a bus has more than one line some lines may have voltage present while others do not. We identify the instant at which one or more bus lines change state as follows:


one or more bus lines may change state at these points

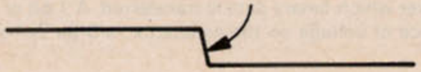
Sometimes one signal's change in state triggers another signal's change in state. This "triggering" action is represented as follows:



There is no significance to either signal change level: low-to-high ___/‾‾‾ could be substituted for high-to-low ‾‾‾\___, and vice versa. What is important is that one signal level change shown thus:



is identified as causing another signal to change state, shown thus:



REGISTERS

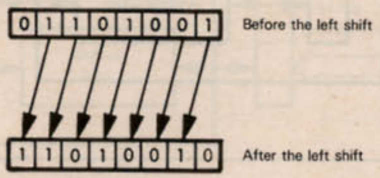
The switches which hold information are called "registers". Information is transmitted to and from registers via busses. Now we can immediately see that a 4-bit microprocessor will be simpler than an 8-bit microprocessor, since every register or bus requires only four switches or conducting lines, rather than eight. Similarly, a 16-bit microprocessor has more complicated logic than an 8-bit microprocessor, because every register or bus requires sixteen switches or conducting lines, rather than eight.

THE ARITHMETIC AND LOGIC UNIT

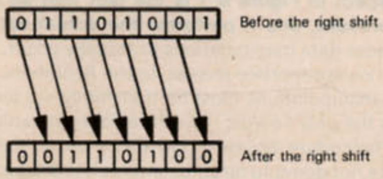
ALU

A microprocessor's word size applies also to logic which handles data manipulations. The various arithmetic and logic operations described in Chapter 4 will all be performed on fixed bit length data units, equal to the word size of the microprocessor. Thus for an 8-bit microprocessor you will always have 8-bit data inputs to any arithmetic or Boolean operation, and you will create an 8-bit result. **All of this logic is collected together in one location which is referred to as an Arithmetic and Logic Unit.** The Arithmetic and Logic Unit gets its name from the fact that it performs arithmetic operations (add, subtract) and logic operations; principal logic operations are the Boolean operations AND, OR, XOR and NOT. The Arithmetic and Logic Unit is usually called an ALU. Figure 6-1 functionally illustrates the logic of an Arithmetic and Logic Unit.

In addition to performing the arithmetic and Boolean operations described in Chapter 4, an Arithmetic and Logic Unit (ALU) will probably be able to shift data left:



and/or it will be able to shift data right:



The 0 and 1 bits shown in the shift illustrations have been selected arbitrarily and have no particular significance.

We are not going to concern ourselves with the actual method used to create addition logic — or any other logic within the ALU. Only someone who is actually designing microprocessors needs this type of detail. What is worth noting is that the Arithmetic and Logic Unit (ALU) has one or more busses bringing it data input, and one or more busses via which data is transmitted out. (The ALU will probably have one or more internal busses, as shown in Figure 6-1, but these are of no concern to you.) Busses become 4-bit busses for a 4-bit microprocessor, 8-bit busses for an 8-bit microprocessor, and 16-bit busses for a 16-bit microprocessor.

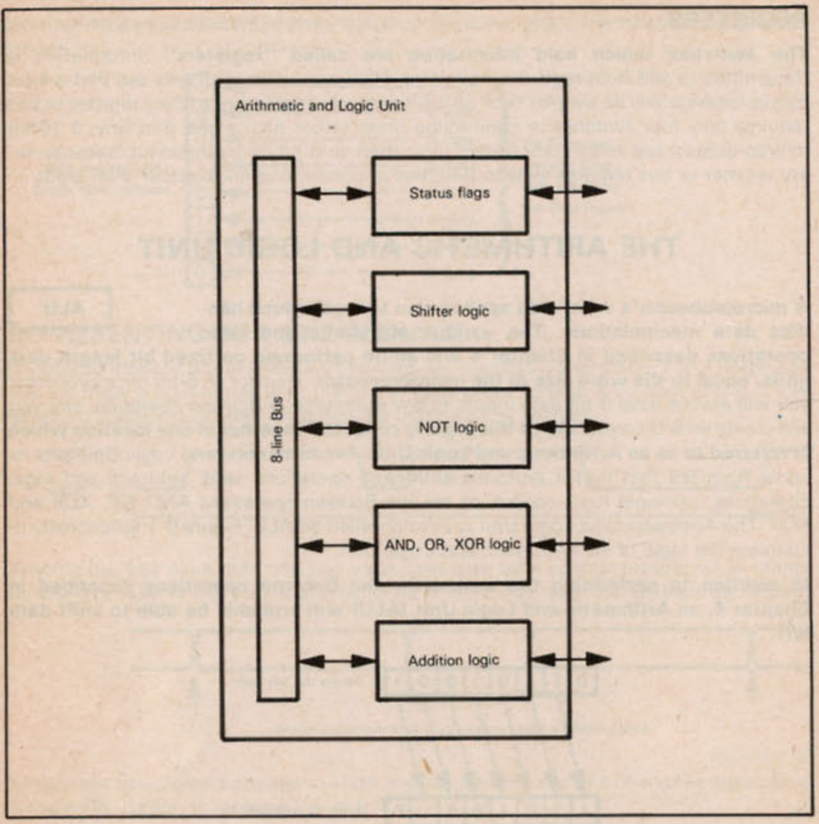
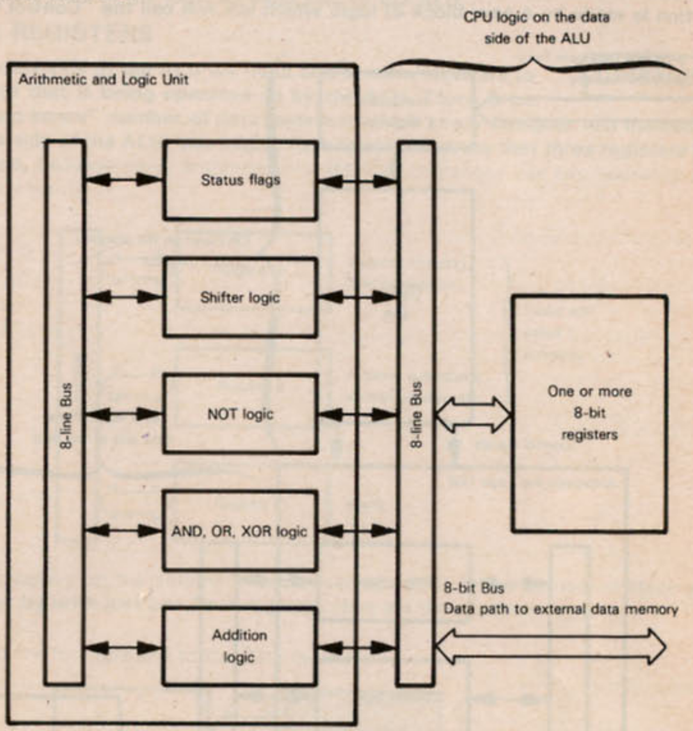


Figure 6-1. The Arithmetic and Logic Unit

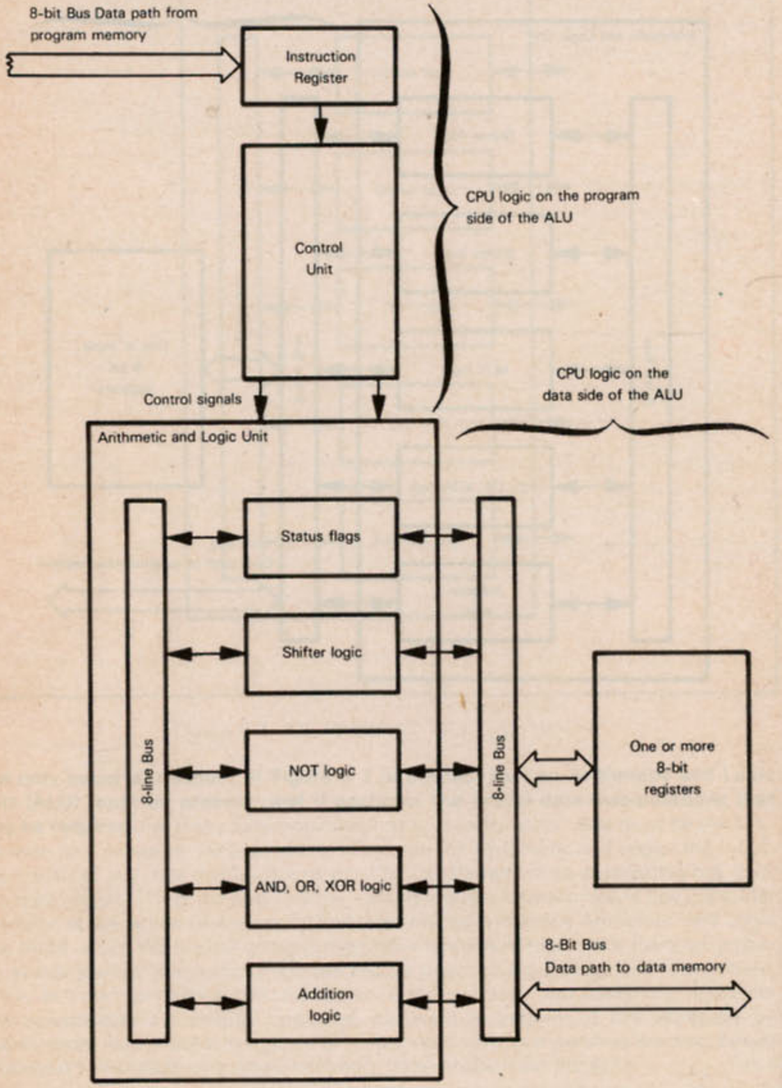
The only important aspect of Figure 6-1 is the fact that an Arithmetic and Logic Unit (ALU) must be present, and it performs the actual data manipulations that may be required. For these data manipulations to actually occur, data must be transferred from data registers, via appropriate busses, to the Arithmetic and Logic Unit (ALU). The results of any data manipulations must be transferred via an appropriate bus back to a data register. Thus the only "magic" which remains unexplained in any computer operation is the actual procedure whereby operations within the Arithmetic and Logic Unit (ALU) occur. We are not going to describe internal ALU logic since there are probably as many ways of accomplishing the task as there are logic designers; furthermore, to you as a microprocessor user, the subject is of no interest. But remember, no matter how complicated a computer operation you want to perform, it will ultimately be broken down into a serial sequence of steps, each of which involves sending binary data input to the ALU and receiving binary data results from the ALU.

Since the Arithmetic and Logic Unit (ALU) receives operands from data registers and returns results to data registers, we must add appropriate data registers on one side of the ALU:


CPU logic on the data side of the ALU
Arithmetic and Logic Unit
8-line Bus
Status flags
Shifter logic
NOT logic
AND, OR, XOR logic
Addition logic
8-line Bus
One or more 8-bit registers
8-bit Bus
Data path to external data memory

CONTROL UNIT

On the other side of the ALU we need a register which holds the instruction that determines which part of the ALU will be used, and how it will be used. The "which" and "how" determination is made by a new block of logic which we will call the "Control Unit":


8-bit Bus Data path from program memory
Instruction Register
Control Unit
CPU logic on the program side of the ALU
Control signals
CPU logic on the data side of the ALU
Arithmetic and Logic Unit
8-line Bus
Status flags
Shifter logic
NOT logic
AND, OR, XOR logic
Addition logic
8-line Bus
One or more 8-bit registers
8-Bit Bus Data path to data memory

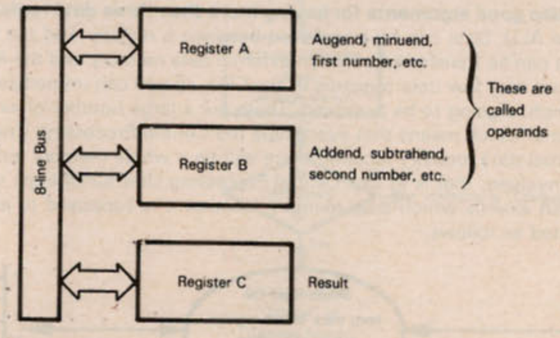
ADDITIONAL CPU LOGIC

Let us now look at the logic on each side of the ALU.

DATA REGISTERS

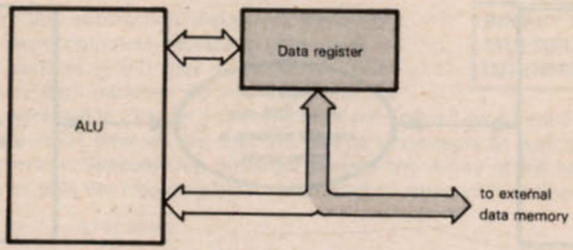
OPERANDS

On the data side of the ALU we need one or more registers to hold data that is being operated on by the ALU. There is no "common sense" number of data registers which every designer will include on the data side of the ALU; you might, for example, assume that three registers is a good idea, because many arithmetic and Boolean operations use two operands and create one result:

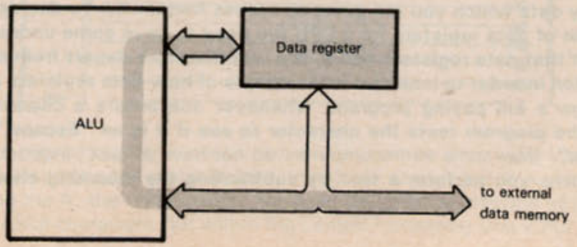

8-line Bus
Register A
Register B
Register C
Augend, minuend, first number, etc.
Addend, subtrahend, second number, etc.
Result
These are called operands

But by breaking up the arithmetic or Boolean operation into a number of steps you could get by with just one Data register. Here are the necessary steps:

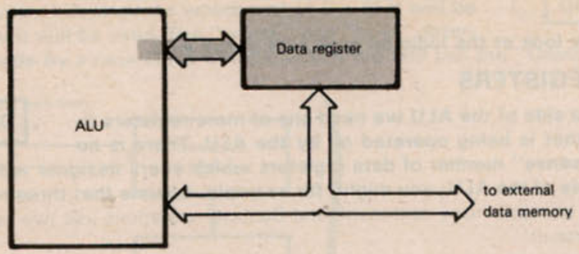
1) Bring the first operand to the Data register:



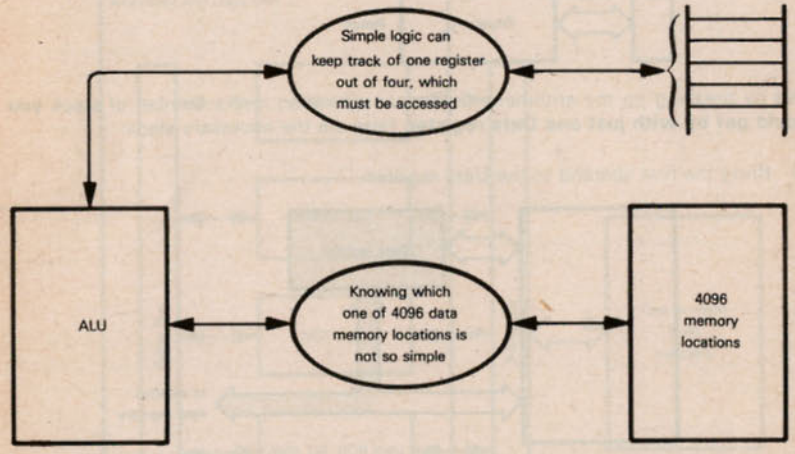
2) Perform any ALU operations, fetching one operand from the Data register, the other operand directly from external memory:



3) Return the result to the Data register:

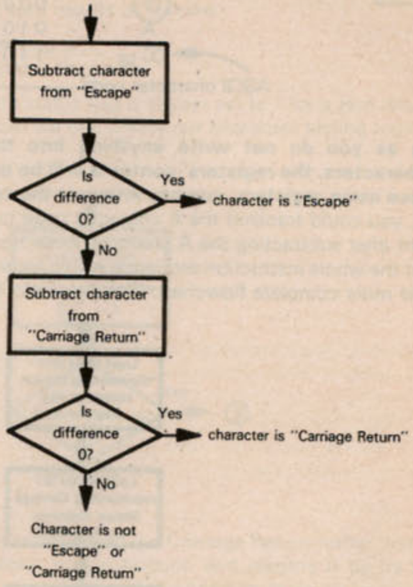


But **there are also good arguments for having more than three data registers** on the data side of the ALU. Data can be transferred between a register and the ALU much faster than data can be transferred between external data memory and the ALU. This is because there are very few data registers in the CPU, so you can immediately identify the register which is going to be accessed. There are a large number of external data memory locations, which means that every time the Central Processing Unit (CPU) accesses an external data memory location there will be a whole memory location identification step involved: that is to say, Central Processing Unit (CPU) logic must spend time figuring out exactly which data memory location it is supposed to access. This may be illustrated as follows:


Simple logic can keep track of one register out of four, which must be accessed
ALU
Knowing which one of 4096 data memory locations is not so simple
4096 memory locations

USING DATA REGISTERS

By having many data registers, a microprocessor allows you to maintain within the CPU any data which you are going to access frequently. To decide on the optimal number of data registers for a CPU we need to have some understanding of the function that data registers serve. We will therefore depart from our immediate discussion in order to look at a real example of how data registers are used. In Joe Bitburger's bill paying program, whenever Joe enters a character via the keyboard, the program tests the character to see if it is an "Escape" or a "Carriage Return". Now some microcomputers do not have a "test" instruction; lacking test instructions **you perform a test by subtracting the incoming character code from the "Escape" character, and then from the "Carriage Return" character.** Logic may be illustrated as follows:


Subtract character from "Escape"
Is difference 0?
Yes
character is "Escape"
No
Subtract character from "Carriage Return"
Is difference 0?
Yes
character is "Carriage Return"
No
Character is not "Escape" or "Carriage Return"

We will represent a "Carriage Return" keyboard character using the symbol (Cr). Does it make sense to subtract a letter of the alphabet from a Carriage Return?

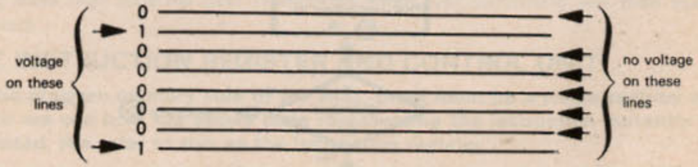
(Cr) - A = ?

BINARY DATA MULTIPLE INTERPRETATIONS

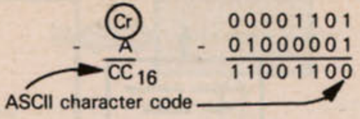
To a human this subtraction makes no sense; but remember, microcomputers represent characters via binary digit patterns — and they also represent numbers via binary digit patterns. Do you recall how frequently you were told in Chapter 4 that you must remember how a binary digit pattern is being interpreted? Now we see that this has its advantages as well as giving you more to remember. Suppose Joe Bitburger presses the A key at his keyboard. The microcomputer sees the following binary digit pattern arrive from the keyboard:

01000001

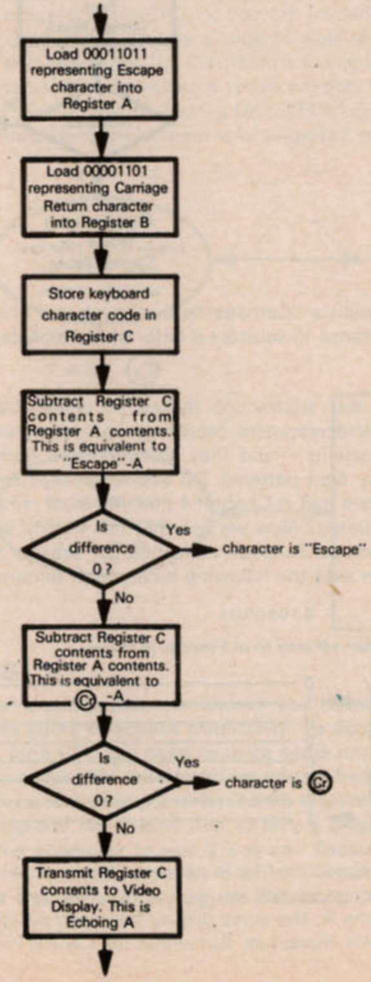
This bit pattern will arrive on an 8-line bus, as follows:



So far as the microcomputer is concerned, this is a simple binary digit pattern. If you preserve it in memory and re-use it later, you could send it out to a video display in order to echo the A; the video display has been designed to interpret all incoming bit patterns as ASCII characters. But within the Central Processing Unit (CPU) a bit pattern is simply a bit pattern. Thus you can subtract the A bit pattern from the Carriage Return bit pattern, treating each as pure binary data. This may be illustrated as follows:



So long as you do not write anything into the registers holding the A or (Cr) characters, the registers' contents will be preserved. The next time you access these same registers, you can interpret the contents as ASCII characters. For example, you could transmit the A character code to the video display to generate an echo right after subtracting the A character code from a Carriage return. We can now represent the whole instruction sequence which receives and tests incoming characters using this more complete flowchart: (See Appendix B)

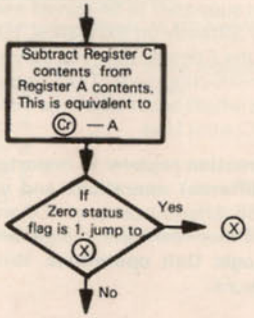

Load 00011011 representing Escape character into Register A
Load 00001101 representing Carriage Return character into Register B
Store keyboard character code in Register C
Subtract Register C contents from Register A contents. This is equivalent to "Escape"-A
Is difference 0?
Yes
character is "Escape"
No
Subtract Register C contents from Register A contents. This is equivalent to (Cr) -A
Is difference 0?
Yes
character is (Cr)
No
Transmit Register C contents to Video Display. This is Echoing A

ZERO STATUS FLAG

Following each subtraction, you will use a status flag in order to determine if the character is a Carriage Return or an Escape code. Nearly all microprocessors have a status flag called the Zero status. This flag is used to detect zero results as follows:

If result is 0, Zero status flag is 1.
If result is not 0, Zero status flag is 0.

By a perverse twist of logic the Zero status flag is always set to 1 for a zero result and it is set to 0 for a nonzero result. Now we can create our character testing logic via the following instruction sequence:


Subtract Register C contents from Register A contents. This is equivalent to (Cr) — A
If Zero status flag is 1, jump to Ⓧ
Yes
Ⓧ
No

Ⓧ is the instruction which is to be executed if a Carriage Return rather than an A is entered via the keyboard. You select this instruction and identify it by its program memory address — that is to say, the address of the program memory location where the instruction code is stored.

All of this has been something of a departure from our discussion at hand — the correct number of registers to have on the data side of the ALU. But now that we understand how Data registers are used, can we justify some "correct" number of Data registers, which every microprocessor should have? Unfortunately there is no "correct" number, rather the microprocessor designer has to make tradeoffs. How much of the limited logic is he going to set aside for registers and how much for other things? Remember it takes eight switches for each register, plus connections from the register to busses within the microprocessor. The same switches and bus space could be used in a variety of other ways.

CPU MEMORY

Commonly you will find between 4 and 8 registers on the data side of the ALU; however you may have as few as 1 register and as many as 32. There are also some microprocessors that have a small part of data memory within the CPU. There may be 64 or more data memory locations within the Central Processing Unit. These storage locations are something more than external data memory locations, yet something less than true data registers.

THE INSTRUCTION REGISTER AND CONTROL UNIT

On the program memory side of the ALU, there must be a single register within which we can hold the binary code representing the instruction currently being executed. We refer to this as the Instruction register.

Suppose the instruction currently being executed adds the contents of Data registers A and B, returning the sum to Data Register A. (We assume for the moment that Data Registers A and B exist on the data side of the CPU.) The appropriate instruction binary code will be brought (as data) to the Central Processing Unit where it will be stored in the Instruction register.

The contents of the Instruction register act as one of 256 different "triggers" to a block of logic called the Control Unit; there are 256 different "triggers" because there are 256 combinations of 0 and 1 bits that you can generate out of eight bits. A 16-bit microprocessor would have a 16-bit Instruction register — with 65,536 different 0 and 1 bit combinations.

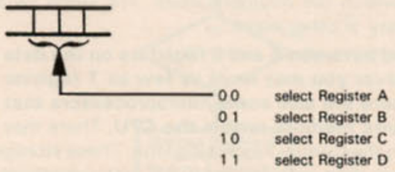
The Instruction register "trigger" initiates a sequence of signals output by the Control Unit to enable the data transfers and ALU operations required by the instruction being executed.

There is very little you need to know about the Control Unit, the Instruction register and the way in which they interact; the Control Unit creates control signals that move data where it is supposed to be moved and exercise ALU logic at the proper time. For microcomputers currently on the market, there is nothing you, as a microcomputer user, can do about the Control Unit; you cannot, for example, modify the way in which it responds to an instruction code. The Control Unit acts as the link between an instruction and the events which occur when the instruction is executed. That is all you need to know about the Control Unit.

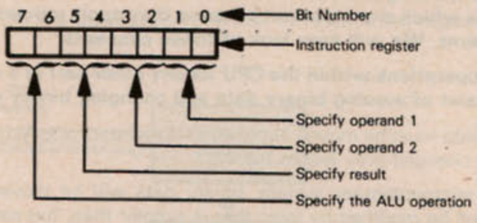
But the size of the Instruction register is important. An 8-bit Instruction register can only specify 256 different operations and variations on operations. That is because there are only 256 different patterns of 0 and 1 digits that you can create out of eight binary digits. 256 may seem like a large number, but in reality it is not. **Consider Arithmetic and Logic Unit operations.** Within this block of logic we have defined just seven operations:

Shift left
Shift right
Complement
AND
OR
Exclusive-OR
Add

Even though there are only seven ALU operations, the instruction has much more to define; it must define the sources for the operands, and the destination for the result. Suppose you have four Data registers (Registers A, B, C and D). You will need two of the eight instruction object code bits each time you have to identify one of the four registers.



In other words, an instruction code must be absolutely specific. If you have four register options, you will need four separate and distinct binary digit codes to specify the selected option. Thus **the 8-bit instruction code may be illustrated as follows:**



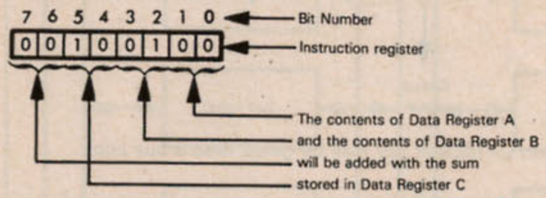
We might arbitrarily assume that for each of the three register specifications the Control Unit will interpret instruction code bits as follows:

00 - Select Data Register A
01 - Select Data Register B
10 - Select Data Register C
11 - Select Data Register D

Instruction register bits 6 and 7 might be interpreted by the Control Unit as follows:

Bits 7 6	
0 0	Add
0 1	AND
1 0	OR
1 1	Exclusive-OR

Here is one example of how an instruction code would be interpreted:


7 6 5 4 3 2 1 0
Bit Number
0 0 1 0 0 1 0 0
Instruction register
The contents of Data Register A
and the contents of Data Register B
will be added with the sum
stored in Data Register C

We have accurately illustrated the way in which the Control Unit decodes instruction register contents, but the illustration, while conceptually accurate, is impractical.

The bit patterns illustrated above apply only to four of the seven ALU operations — the four that require two input operands. But these four operations, together with their options, use up all 256 instruction codes. There are no codes left for any other ALU operations, nor for any of the instructions that do not use the ALU. **Clearly 256 possible instruction combinations is not very many. In Volume I we will explore the way in which microprocessor designers spread the limited number of instruction code options among all the types of instructions which must be present, giving a limited capability within each class of instructions. This is a process which is easily understood, since it is about the same as getting by on a limited budget. You do not have enough money to do everything you would like to do, therefore you limit yourself in all areas, striking the best balance between lifestyle and income.**

LOGIC CONCEPTS AND TIMING

Even though the exact workings of a Control Unit are unimportant to you, there are some logic concepts which are important because they apply universally within microcomputer systems. We will now look at these concepts.

Any sequence of logic operations within the CPU (or any other part of a microcomputer system) will consist of moving binary data and changing binary data.

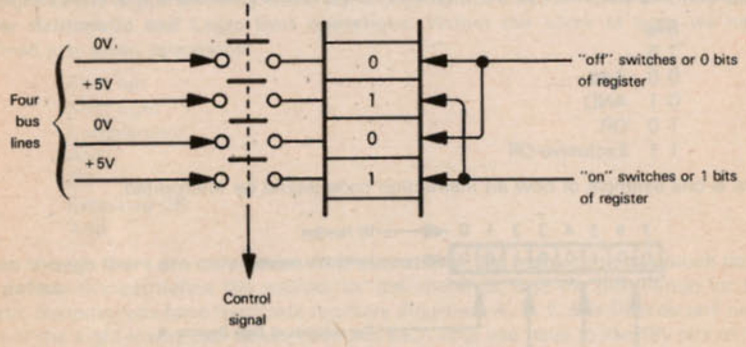
Within the CPU, binary data must be moved between data registers, the ALU and external logic. Binary data is changed only within the ALU.

In any other part of a microcomputer system, binary data will be moved between registers and the data will be modified by logic that is simpler than, but similar to the ALU.

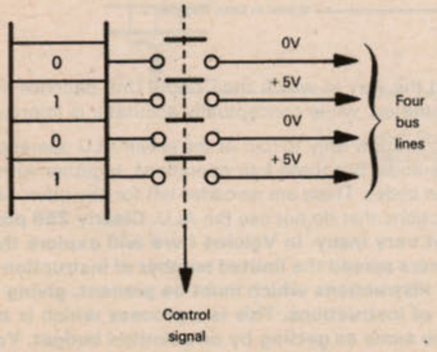
LOGIC TO MOVE BINARY DATA

GATE SIGNAL

In order to move binary data, a control signal must be created to act as a connector between a register switch and a bus line. This control signal is sometimes referred to as a gate. You may gate data from bus lines into a register:


0V
+5V
0V
+5V
Four bus lines
0
1
0
1
"off" switches or 0 bits of register
"on" switches or 1 bits of register
Control signal

The switch position (or register bit) may be gated onto a bus line:

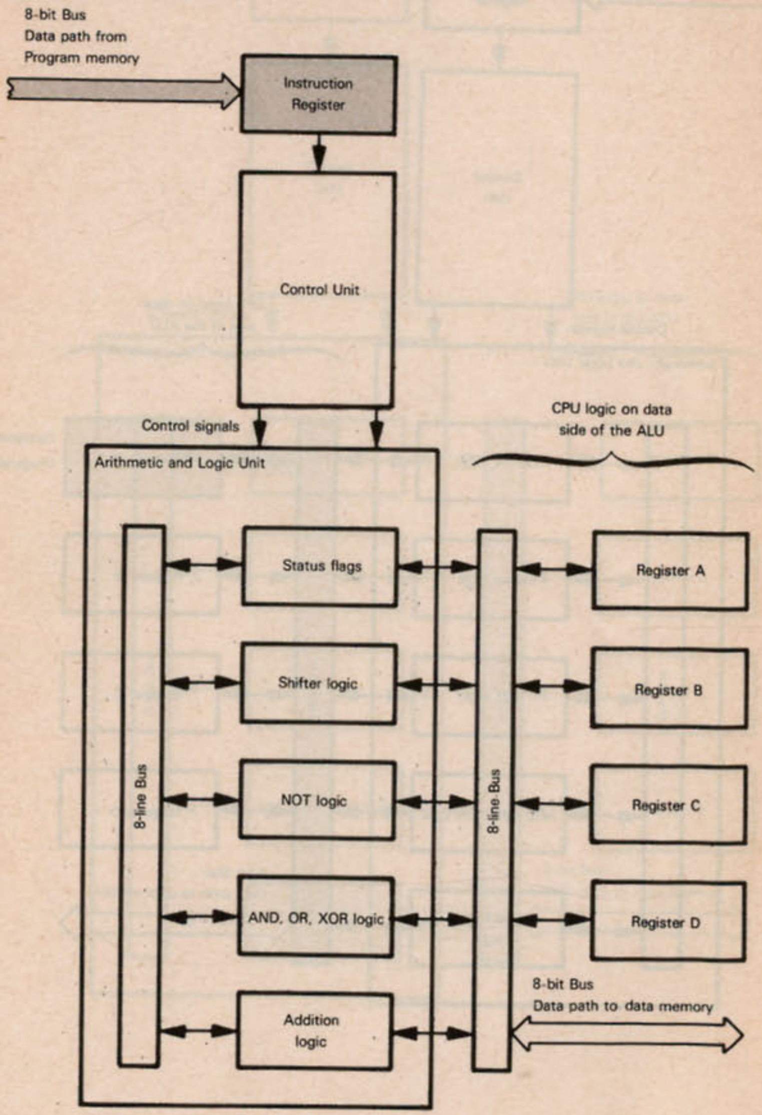


INSTRUCTION TIMING

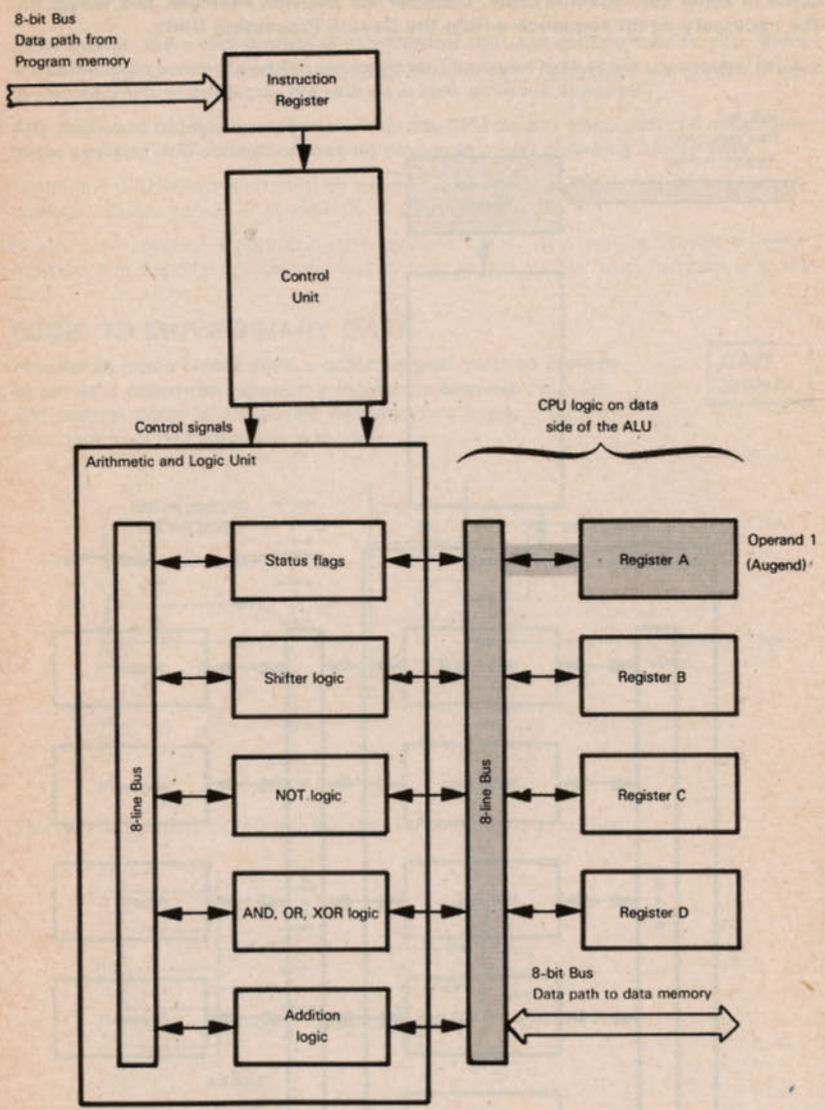
Timing is extremely important during these data transfer operations because it takes a finite amount of time for voltage levels to appear or disappear on bus lines; and until a steady state exists on the bus line, logic cannot connect the bus lines to any new switch.

Also, the sequence of events corresponding to any instruction's execution must occur in some very specific order. Consider our addition example; this would be the necessary event sequence within the Central Processing Unit:

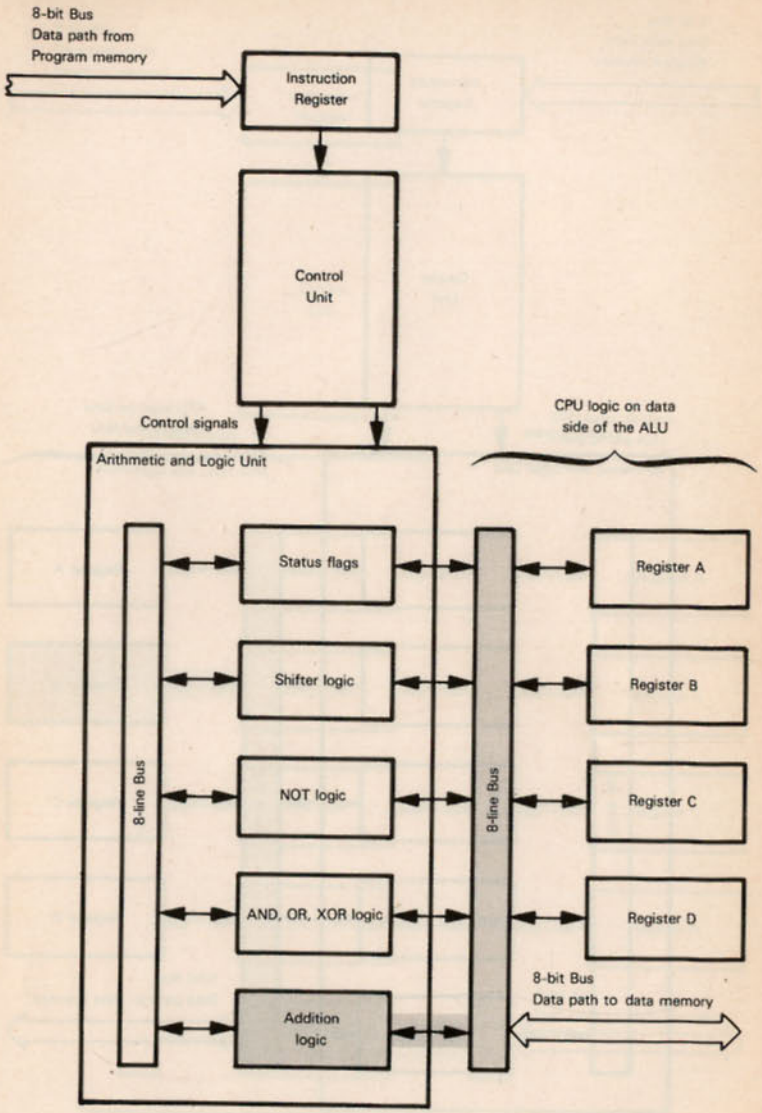
Step 1) Bring the instruction code into the Instruction register



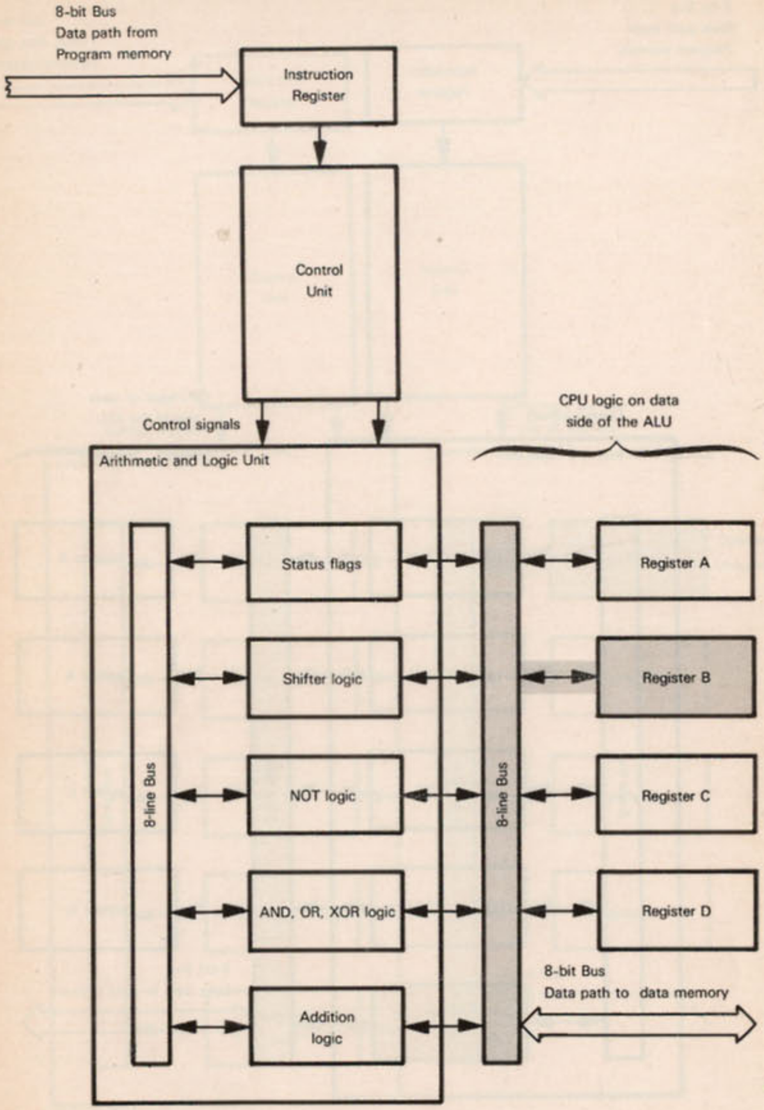
Step 2) Gate Register A contents onto the Bus



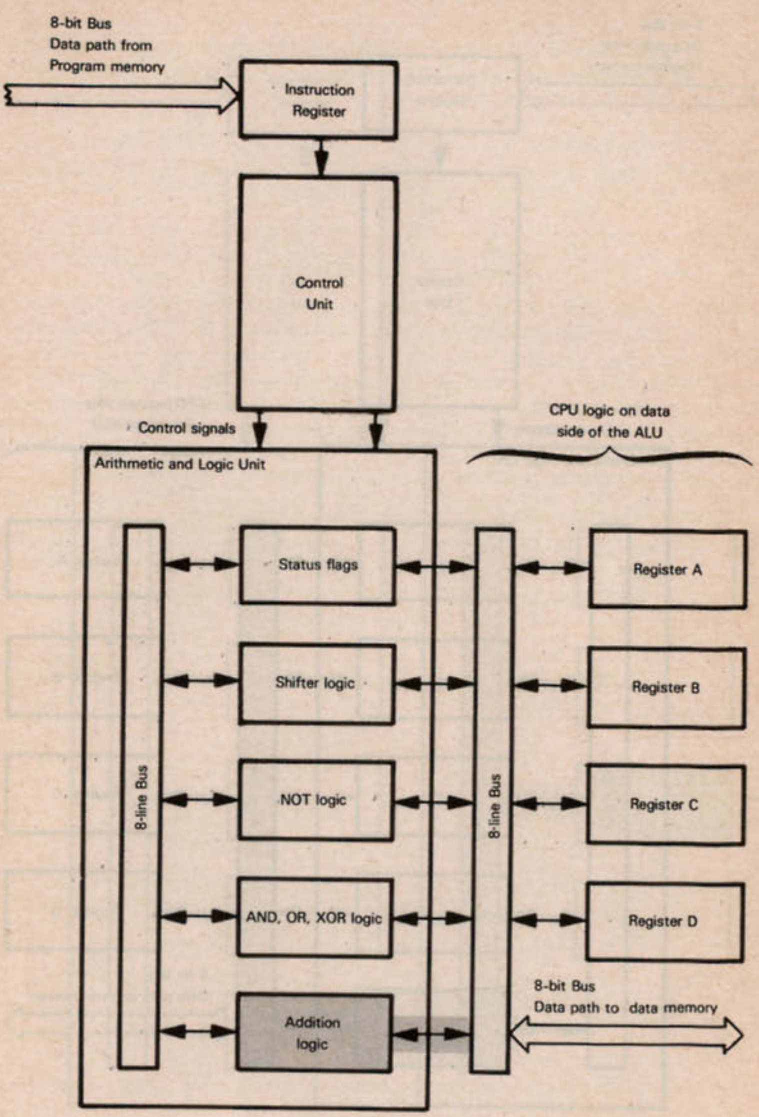
Step 3) Gate Bus to Addition logic



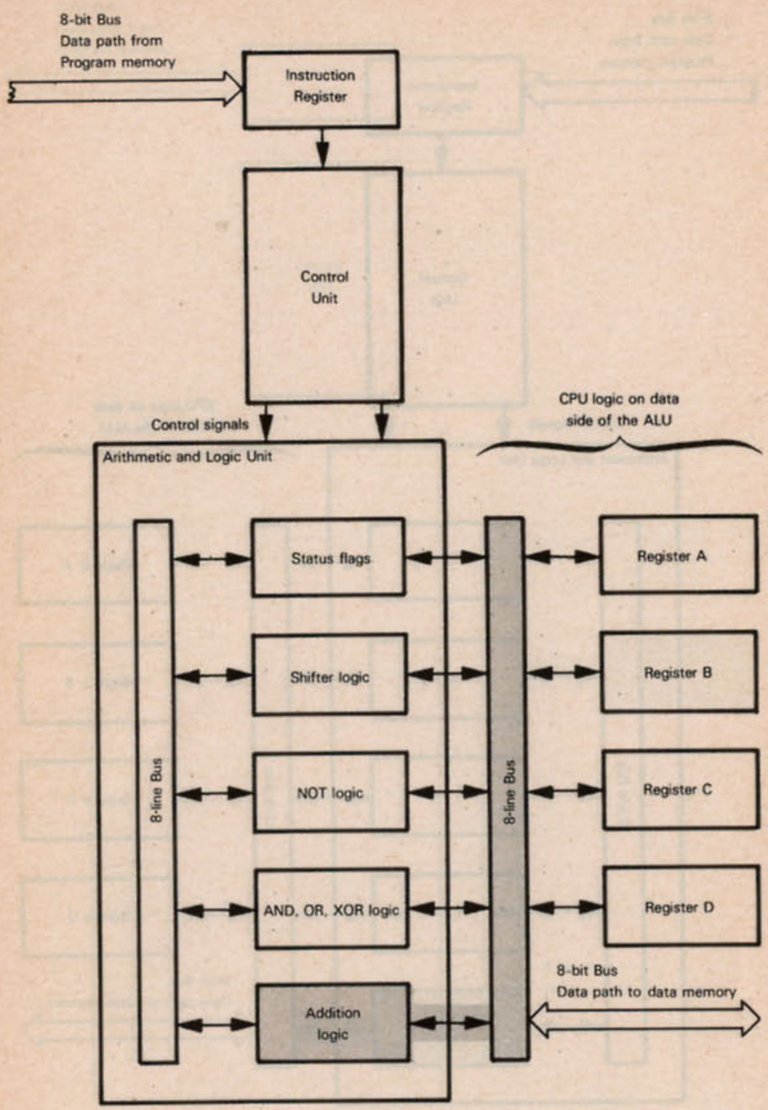
Step 4) Gate Register B contents onto the Bus



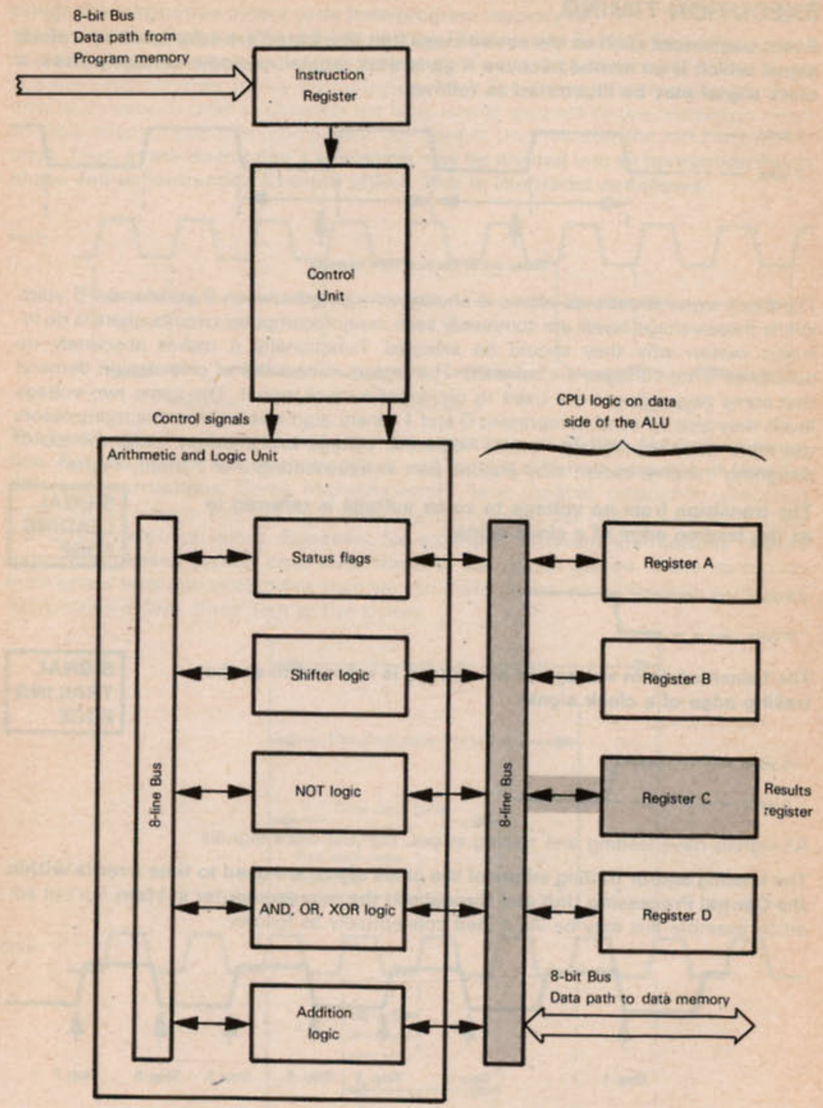
Step 5) Add



Step 6) Gate sum onto the Bus

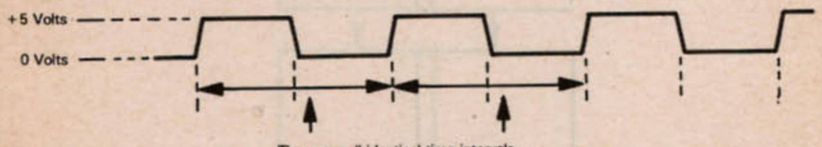


Step 7) Gate sum into Register C



THE CLOCK SIGNAL AND INSTRUCTION EXECUTION TIMING

Event sequences such as the seven steps just illustrated are scheduled by a clock signal which is so named because it generates regular, periodic voltage pulses; a clock signal may be illustrated as follows:

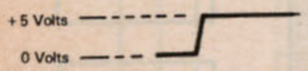


These are all identical time intervals

The clock signal illustrated above is shown switching between 0 volts and +5 volts. While these voltage levels are commonly seen in microcomputer circuits, there is no intrinsic reason why they should be selected. Functionally it makes absolutely no difference what voltages are selected. The logical necessities of chip design demand that some two voltages are used to represent a clock signal. The same two voltage levels may also be used to represent 0 and 1 binary digit levels. Older microprocessors use more than two voltage levels. Additional voltage levels simply make the job of designing the chip easier; they play no part in representing 0 or 1 binary digits.

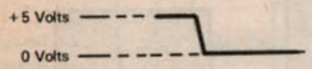
SIGNAL LEADING EDGE

The transition from no voltage to some voltage is referred to as the leading edge of a clock signal:



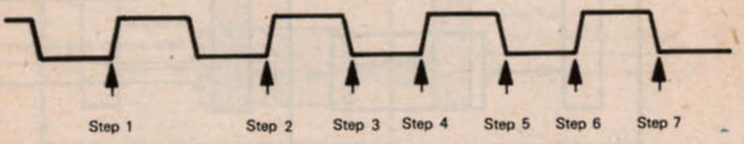
SIGNAL TRAILING EDGE

The transition from voltage to no voltage is referred to as the trailing edge of a clock signal:



All signals have leading and trailing edges, not just clock signals.

The leading and/or trailing edges of the clock signal are used to time events within the Central Processing Unit and throughout the microcomputer system. For our addition example this may be illustrated conceptually as follows:

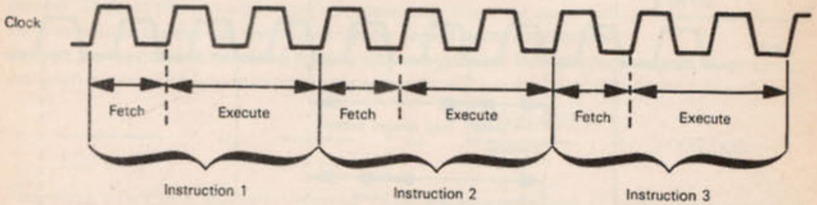


More time follows Step 1 than any other step since the Control Unit must be given time to decode the Instruction register contents.

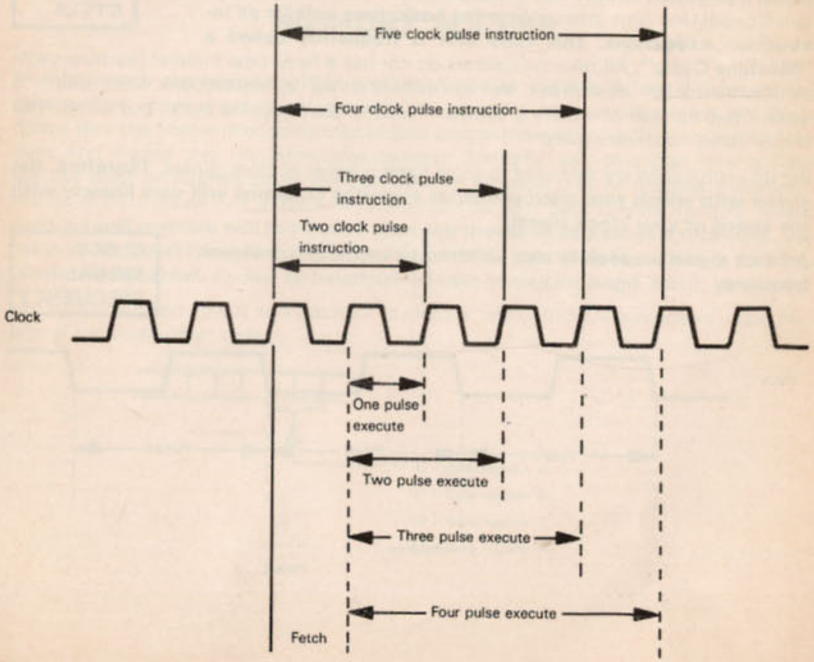
Different microcomputers have different types of clock signals, ranging from the very simple to the very complex. There may also be more than one clock signal, in which case the many clock signals will have some definite relationship to each other. But **in every case it is the clock signal which acts as the master event sequencer for all logic.**

INSTRUCTION FETCH

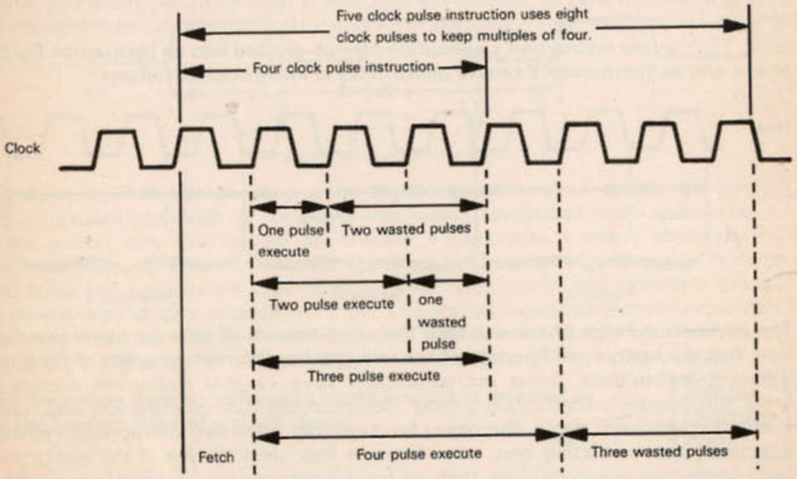
Let us look again at an instruction execution's event sequence. **Every instruction's execution must begin with a step that brings the instruction binary code from program memory to the Instruction register. We call this step an "Instruction Fetch".** Once the instruction binary code is in the Instruction register the Control Unit takes over — after being triggered appropriately by the Instruction register. The Control Unit acquires as much time as it needs in order to generate the logic events required by the instruction. After all logic events have been completed, the Control Unit initiates the next instruction fetch. Thus, **every instruction's execution may be divided into an Instruction Fetch phase and an Instruction Execute phase. This is illustrated as follows:**



The Instruction Fetch phase will take the same amount of time for every instruction. But the Instruction Execute phase will require different amounts of time for different instructions. Some microprocessors have variable instruction execution times; but in order to simplify logic, other microprocessors use one, or a very few, fixed instruction execution times. **Suppose, for example, different instructions require execution phases lasting one, two, three or four clock pulses.** If the instruction fetch phase lasts one clock pulse, then we could design the microprocessor to execute instructions in two, three, four or five pulses:


Five clock pulse instruction
Four clock pulse instruction
Three clock pulse instruction
Two clock pulse instruction
Clock
One pulse execute
Two pulse execute
Three pulse execute
Four pulse execute
Fetch

Given the above situation, **a microprocessor designer may decide to make all instructions last five clock pulses,** wasting the unused clock pulses for the shorter instructions. If, on the other hand, there were very few long instructions, **the microprocessor designer may decide to standardize on a four clock pulse instruction time, using two instruction times for the few five clock pulse instructions.** This may be illustrated as follows:



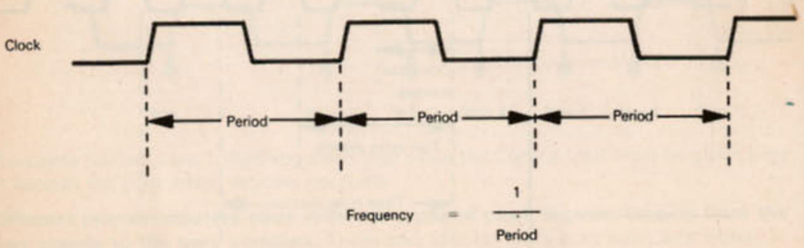
MACHINE CYCLE

In the illustration above, four clock pulses constitute an important Control Unit time interval; it is **the basic time unit for all instruction executions. This time unit is frequently called a "Machine Cycle".** All microprocessors do not use a fixed time interval machine cycle to time instruction executions; but for those that do, all instructions will execute in some fixed number of machine cycles — usually one machine cycle, but sometimes two or three machine cycles.

But all instructions are executed in some fixed number of clock pulses. **Therefore, the speed with which your microcomputer executes programs will vary linearly with the speed of your clock signal.**

CLOCK SIGNAL FREQUENCY

A clock signal's speed is also referred to as the clock signal frequency. Clock signal frequency may be illustrated as follows:



CLOCK PERIOD
NANOSECOND
MICROSECOND

The clock period is the time that elapses between repeated identical clock wave shapes occurring. Nowadays clock periods are measured in nanoseconds. One nanosecond is equal to one thousandth of one millionth of one second (1×10^{-9} seconds). **The period of a clock signal will typically vary between one hundred nanoseconds and five hundred nanoseconds.** A few microprocessors use a clock signal of one microsecond. One microsecond is equal to one millionth of a second.

MEGAHERTZ
MHz

The frequency of a clock signal is the reciprocal of the clock period; that is to say, it is equal to one second divided by the clock period. Thus, if the period is one microsecond, the frequency will be one million pulses per second. **A clock signal with one million pulses per second is referred to as a one megahertz (MHz) clock signal.** A clock signal with a period of five hundred nanoseconds will have two million pulses per second:

$$\frac{1}{500 \times 10^{-9}} = \frac{1}{5 \times 10^{-7}} = \frac{10\ 000\ 000}{5} = 2\ 000\ 000$$

Therefore this clock signal is referred to as a two megahertz (2 MHz) clock signal. A clock signal with a period of one hundred nanoseconds will generate ten million pulses per second:

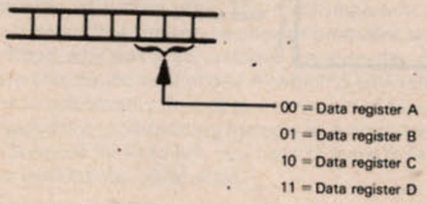
$$\frac{1}{100 \times 10^{-9}} = \frac{1}{1 \times 10^{-7}} = 10\ 000\ 000$$

Therefore this clock signal is said to have a frequency of ten megahertz (10 MHz).

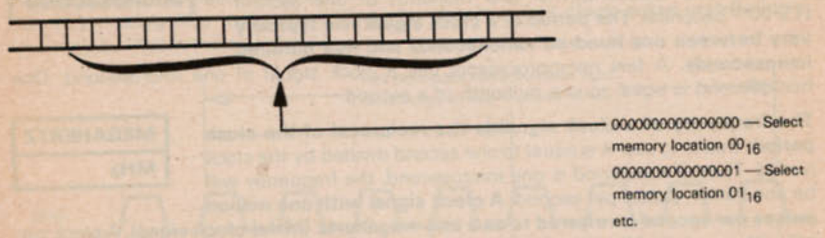
MEMORY ACCESS

Nothing much can happen within a microcomputer system before a memory access occurs. For example, every instruction begins with an instruction fetch, which requires that the contents of some identifiable program memory location be fetched as data, and loaded into the Instruction register. Similarly, any data that is in a Data register must be fetched either from an external physical unit, or from data memory; results of ALU operations in Data registers must be sent back to data memory. **Since even a small memory will have more than one thousand addressable locations, the method for identifying the single location which you wish to access is not immediately apparent.**

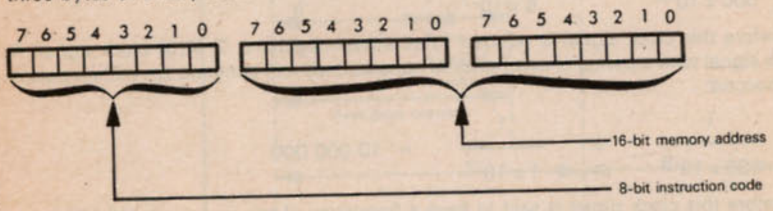
We earlier showed how it was possible to identify one of four Data registers using two bits of the instruction code:



This easy method of addressing Data registers, if applied to external memory, may have to address up to 65,536 memory locations; it takes a 16 binary digit number to select one addressable location out of 65,536:

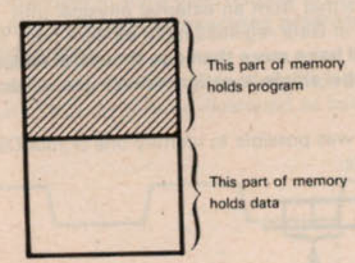

0000000000000000 — Select memory location 00_{16}
0000000000000001 — Select memory location 01_{16}
etc.

Now many microcomputers do allow you to directly address a memory location by supplying a 16-bit memory address; but this means the instruction is represented by three bytes of binary data, not one:

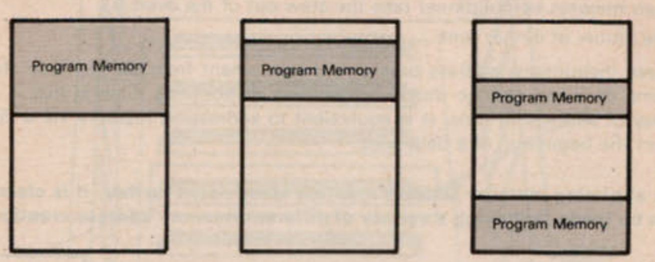


But there are some problems associated with this approach to creating memory addresses.

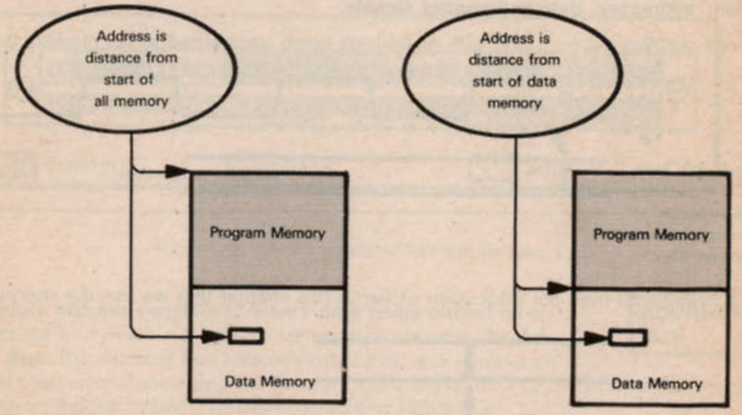
In the first place **it is** rather **wasteful of memory.** Given the frequency with which memory reference instructions occur, and the fact that most microcomputers do not have anywhere close to 65,536 bytes of memory actually present, some alternative, more economical memory addressing procedures would be desirable. Furthermore, in most microcomputer systems **even though we speak of "program" memory and "data" memory, they are usually one and the same thing:**


This part of memory holds program
This part of memory holds data

When you write a program for a microcomputer, you decide which part of memory becomes program memory; the remainder of memory is data memory. But the next time you write a program, or if someone else writes a program for the same microcomputer, you may find memory allocated between the program and data areas in completely different ways:



Frequently you will write simple programs (for example, to perform arithmetic on standard dollar amount numbers); the **programs will be much more useful if you can reuse them in many applications. To do this you will need some memory addressing technique that identifies a data memory location in terms of its displacement (or distance) from the beginning of the data area of memory, rather than from the beginning of memory:**


Address is distance from start of all memory
Program Memory
Data Memory
Address is distance from start of data memory
Program Memory
Data Memory

Let us look at the advantages of identifying a memory location by its displacement from the beginning of a data area. The most important advantage is that you can move the data area without having to change programs that access the data area. In order to understand why this is so, consider the everyday problem of identifying time. You can identify time by the time of day — which is equivalent to absolute memory addressing — or you can identify time as some number of hours before or after a movable event; this is equivalent to addressing memory as a displacement from the beginning of the data area. Suppose for example, you have to leave instructions for the preparation of dinner. Your instructions could state:

1) Turn the oven on at 6:00 p.m.
2) At 6:15 p.m. place the stew in the oven.
3) Remove the stew at 8:15 p.m.
4) Serve at 8:30 p.m.

If you change dinner time, you must change all of your dinner preparation instructions. On the other hand, you could rewrite the dinner preparation instructions as follows:

1) Turn the oven on 2-1/2 hours before dinner.
2) At 2-1/4 hours before dinner put the stew in the oven.
3) Fifteen minutes before dinner take the stew out of the oven.
4) Serve dinner at dinner time.

Since these instructions address time as a displacement from dinner time, changing dinner time does not change dinner preparation instructions. Clearly this is a more useful way of addressing time; it is equivalent to addressing memory via a displacement from the beginning of a data area.

Without exploring memory address creation ideas much further, it is clear that a case can be made for having a variety of different memory address creation techniques.

MEMORY ADDRESSING

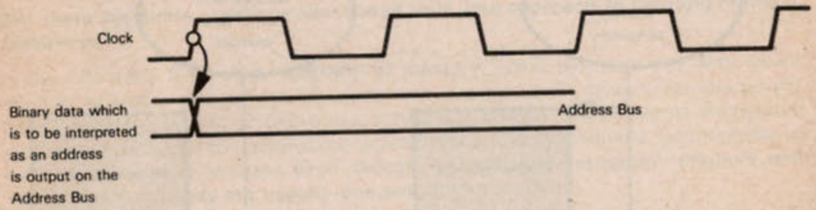
Memory addressing is the term used to describe in general the techniques which a microprocessor allows you to use in order to identify memory locations.

Let us look at exactly how memory addresses work.

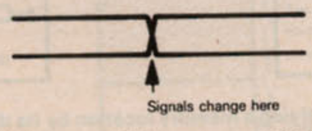
MEMORY ADDRESSING LOGIC

Memory itself will consist of logic devices that handle three types of information: addresses, data and control signals.

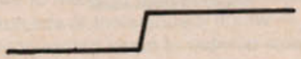
Every memory access starts with an address being transmitted to the memory device:


Clock
Binary data which is to be interpreted as an address is output on the Address Bus
Address Bus

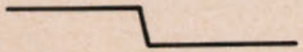
You will recall from our discussion earlier in this chapter that we use the shorthand:



to identify the instant at which one or more signals on a multiline bus may change their level. For a single signal we can show the actual change — from low to high:



or from high to low:



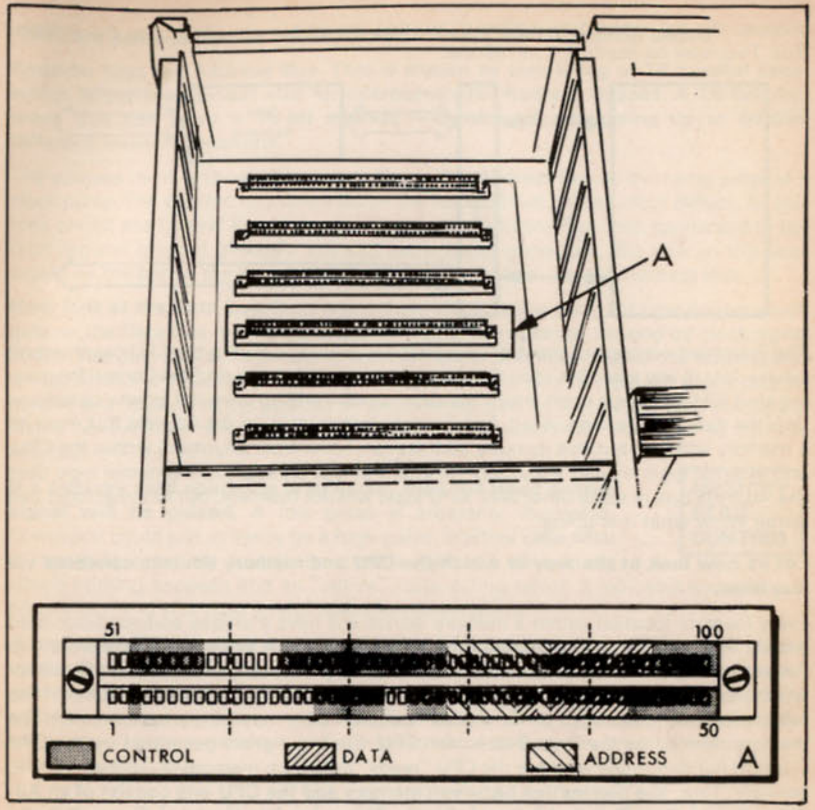


Figure 6-2. Microcomputer System Busses

ADDRESS BUS

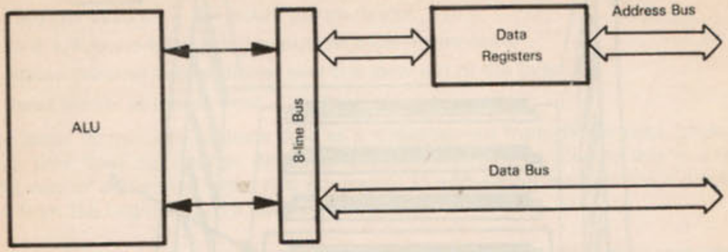
A memory address consists of binary data being output on an appropriate bus which we call (for self-evident reasons) **the Address Bus.** The Address Bus, like any other bus, is a number of parallel conductors connecting the Central Processing Unit (CPU) to memory devices of the microcomputer system. This is illustrated in Figure 6-2.

DATA BUS

There will be a separate Data Bus connecting the Central Processing Unit to memory devices; the Data Bus is also illustrated in Figure 6-2. Once the Central Processing Unit (CPU) places binary data on a bus, there is no longer any possibility for errors in interpretation. Binary data appearing on the Address Bus must be interpreted as an address, while binary data appearing on the Data Bus must be interpreted as data.

Timing illustrations in this chapter show the address appearing on the Address Bus at the rising edge of a clock pulse. Logic within the Central Processing Unit (CPU) will use an appropriate rising clock pulse edge as the trigger to create control signals that connect Address Bus lines to a Data register within the CPU. It is this connection which causes the data to become an address.

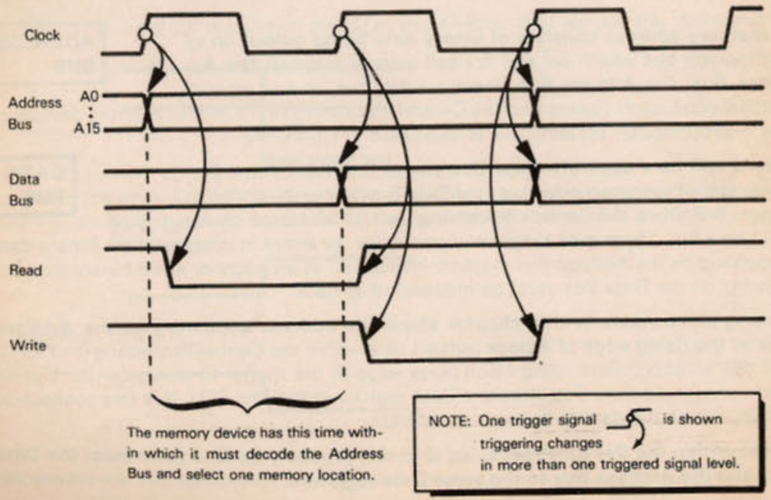
Logic within the Central Processing Unit will probably be able to connect the Data Bus and the Address Bus to the same Data registers. This allows you to compute addresses, like any other data, before outputting the data as an address on the Address Bus. This may be illustrated as follows:



The important conceptual point to understand is that logic within the CPU can interpret binary data in any way. But once the data appears on a bus beyond the Central Processing Unit (CPU), the bus upon which the data appears will, to a limited extent, determine how the data is to be interpreted. For example, information on the Address Bus must be a memory address, but this memory address can come from anywhere within the CPU, and external logic is only "supposed" to interpret it as an address. If external logic uses the Address Bus in some other way, CPU logic will not interfere, but external logic had better know what it is doing.

Let us now look at the way in which the CPU and memory devices converse via bus lines.

Every memory location within a memory device will have a unique address associated with it. The memory device decodes the binary data on the Address Bus and uses it to "select" a single memory location. Subsequently data may be transmitted to the memory device, via the Data Bus, in which case for a write operation it will be stored in the selected memory location; or for a read operation data may be transmitted from the memory device, via the Data Bus, to the CPU. Control signals generated by the CPU Control Unit determine whether the CPU "reads" data from memory or "writes" data to memory. Thus, **the connection between memory and the CPU will consist of an Address Bus, a Data Bus and control signals. These are illustrated in Figure 6-2 and represented functionally as follows:**


Clock
Address Bus
A0
:
A15
Data Bus
Read
Write
The memory device has this time within which it must decode the Address Bus and select one memory location.
NOTE: One trigger signal
is shown
triggering changes
in more than one triggered signal level.

There are some important concepts embodied in the illustration above.

Consider first the Address Bus. This is shown as consisting of 16 parallel lines, which is the commonest size for microcomputer Address Busses. A 16-line Address Bus can carry a 16-bit address — capable of addressing up to 65,536 different memory locations.

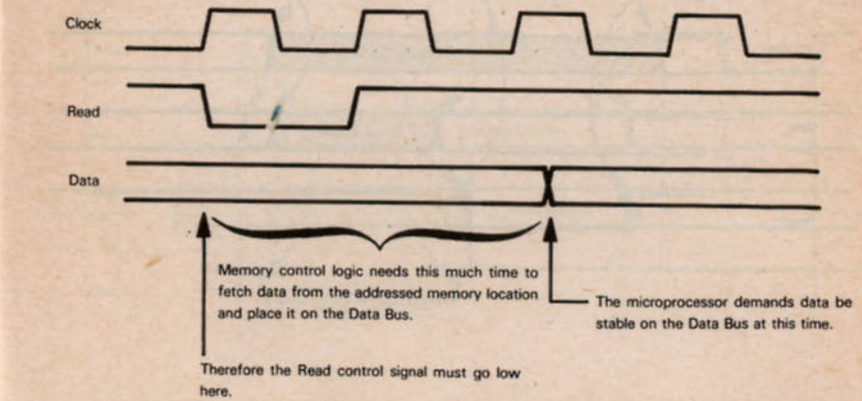
The address itself is shown being gated onto the Address Bus at the rising edge of a clock pulse. The address is maintained on the Address Bus for two clock pulses; for this time period the Central Processing Unit keeps the Address Bus lines connected to the Data register bits out of which the address is being generated. We refer to this time period as the period during which the address is "stable" on the Address Bus.

Now look at the Data Bus. Once again there will be some period of time during which data on the Data Bus must be stable; this period will again be defined by clock signal edges. The Data Bus will start being stable at some time interval after the Address Bus is stable; this is necessary since logic at the memory device must be given time to receive and respond to the address on the Address Bus, so that by the time the data is stable on the Data Bus, the addressed memory location has had time to become selected.

MEMORY READ CONTROL

If a memory read operation is to occur, then the Read control signal will be pulsed. A low pulse is arbitrarily illustrated; however it could just as easily be a high pulse. In either case what is important is that the control signal has a "passive" state during which nothing happens and an "active" state during which it indicates something is happening — in this case a read operation is occurring.

A read operation requires that the addressed memory location contents be placed on the Data Bus. Once again it is going to take some amount of time for logic at the memory device to sense the low Read pulse and respond to it by placing data from the addressed memory location on the Data Bus. The low Read pulse must therefore occur early enough to give memory logic time to transfer data from the selected memory location to the Data Bus:

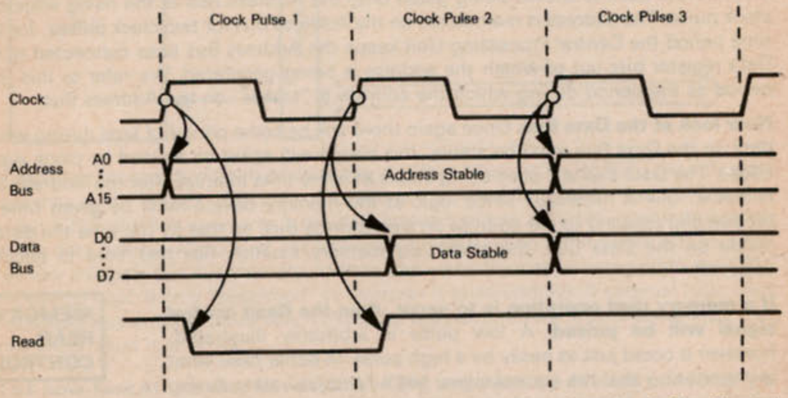

Clock
Read
Data
Memory control logic needs this much time to fetch data from the addressed memory location and place it on the Data Bus.
The microprocessor demands data be stable on the Data Bus at this time.
Therefore the Read control signal must go low here.

MEMORY WRITE CONTROL

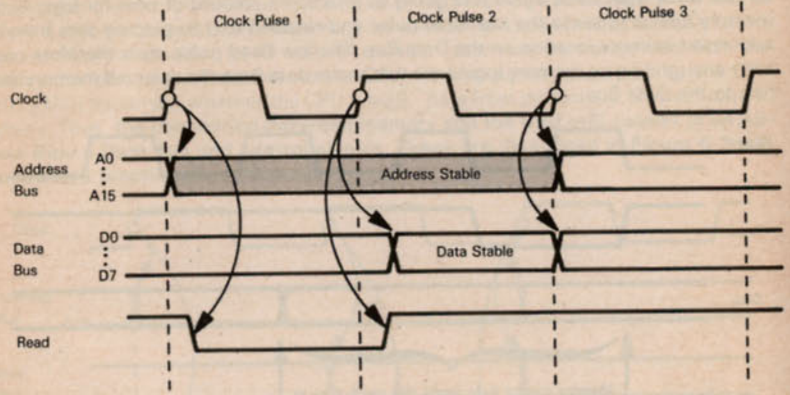
The Write control signal, which again is shown as a low pulse signal, indicates that data on the Data Bus is to be written into the addressed memory word. The Write control signal acts as a strobe; it cannot occur "true" (a low pulse is "true") until valid data is stable on the Data Bus. This is because the memory device will use the low Write strobe in order to connect the Data Bus to the addressed memory location. Were the low Write strobe to occur in advance of stable data on the Data Bus, erroneous data might get written into the memory device.

MEMORY READ OPERATION

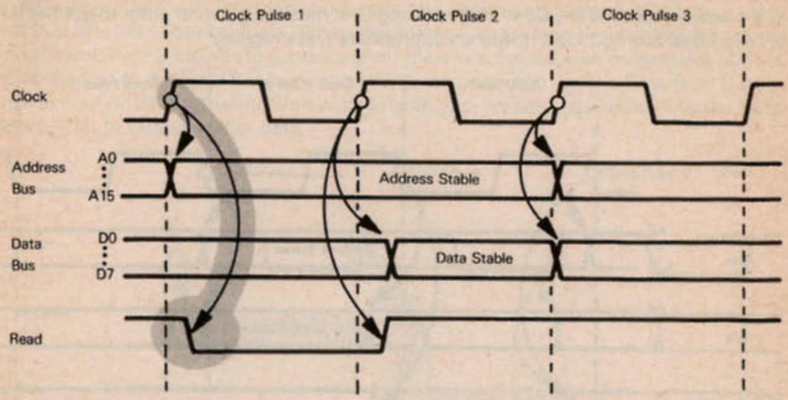
Let us look at memory read and write operations individually. First of all, here is timing associated with a read operation:



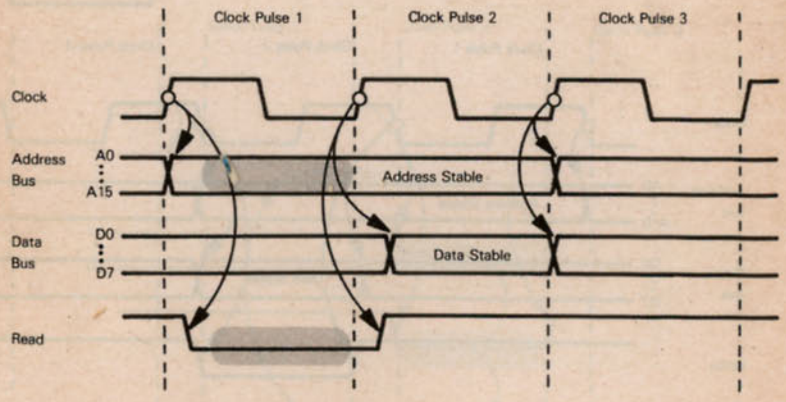
The memory read operation begins on the leading edge of clock pulse 1. The Central Processing Unit uses this clock pulse to connect appropriate Data registers to the Address Bus and this initiates a stable address appearing on the Address Bus:



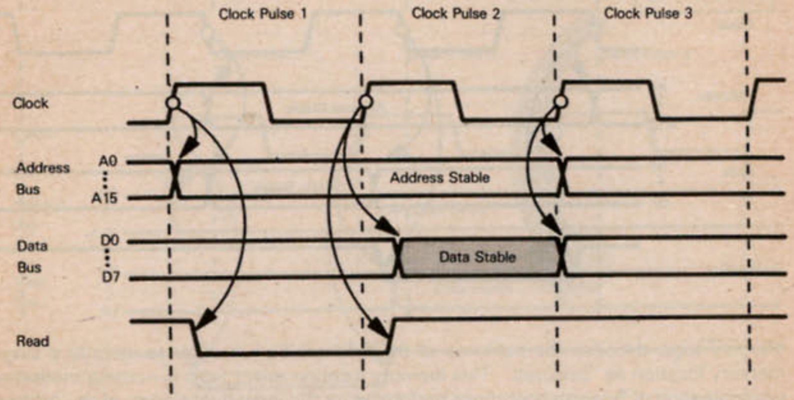
Very shortly thereafter the Read control signal goes low:



Memory logic decodes the contents of the Address Bus in order to identify a single memory location as "selected". This memory location select logic is nothing more than a combination of Boolean operations performed on the individual signals of the Address Bus. However, for now, the actual method used by a memory device to decode the contents of the Address Bus is not important. So long as you realize that logic exists to select one single memory location out of 65,536 possibilities, that is all that matters. The accompanying Read control signal causes the memory device to connect the bits of the selected memory location to the Data Bus. The process of identifying a single memory location, and then connecting it with a Data Bus, must occur before the leading edge of clock pulse 2:



At this time the contents of the selected memory location must be stable on the Data Bus. This data must be held stable on the Data Bus for some finite amount of time: this is the amount of time the Central Processing Unit must be given in order to get the data off the Data Bus and load it into an appropriate Data register:

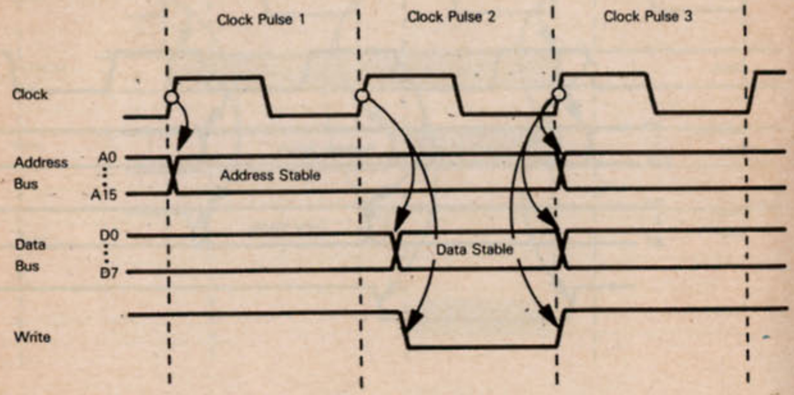


The CPU connects the appropriate Data register to the Data Bus and thus the read is completed.

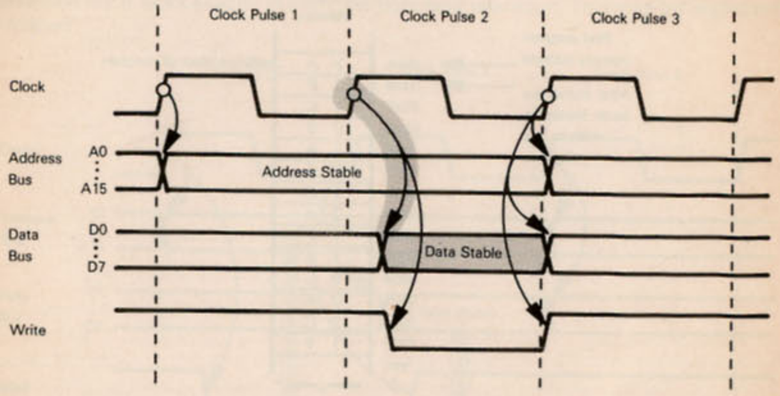
In the illustrations above, one clock pulse is shown as the time duration during which the data on the Data Bus is held stable.

MEMORY WRITE OPERATION

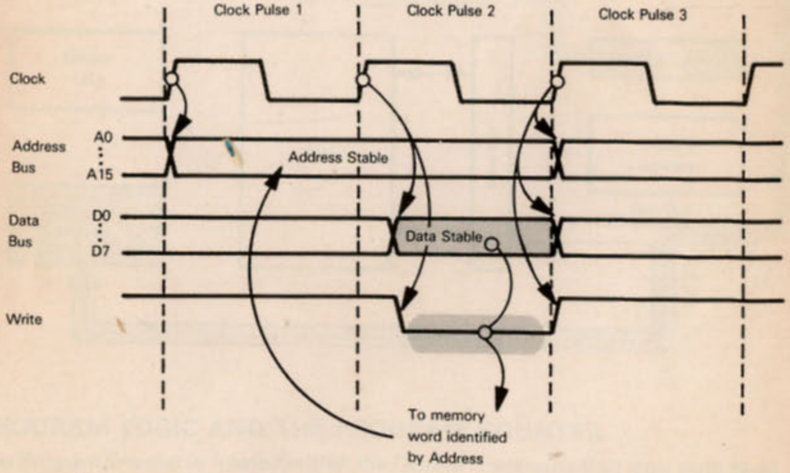
Now consider a memory write operation. Its timing may be illustrated as follows:



Timing associated with a memory write operation does not differ significantly from timing associated with a memory read operation. The logic which outputs a stable address on the Address Bus is in fact identical for the two operations. Differences occur on the Data Bus and the control signals. When the address appears on the Address Bus there is no accompanying low Write control signal; therefore the selected memory location is not connected to the Data Bus and its contents are not output on the Data Bus. Subsequently the Central Processing Unit connects an appropriate Data register to the Data Bus in order to output stable data:



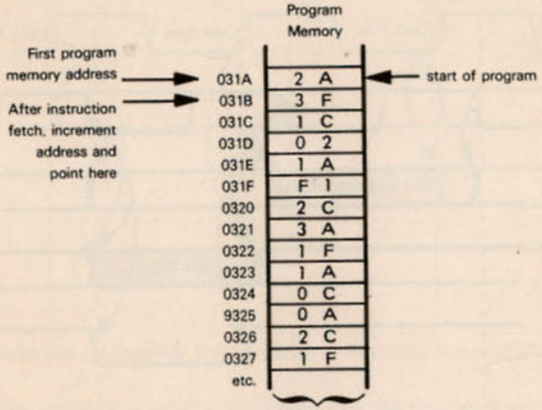
At this time the Write control signal is pulsed low. The Write control signal will be received by the memory device and used to connect the selected memory location to the Data Bus so that the contents of the Data Bus are written into the selected memory location and permanently stored:



PROGRAM MEMORY ADDRESSING AND THE PROGRAM COUNTER

Let us now look at the CPU registers that are required in order to create memory addresses. We will begin by looking at program memory addressing.

Logic required to address program memory is quite straightforward. A program is nothing more than a sequence of instruction codes which will normally be stored in memory locations with incrementing addresses:

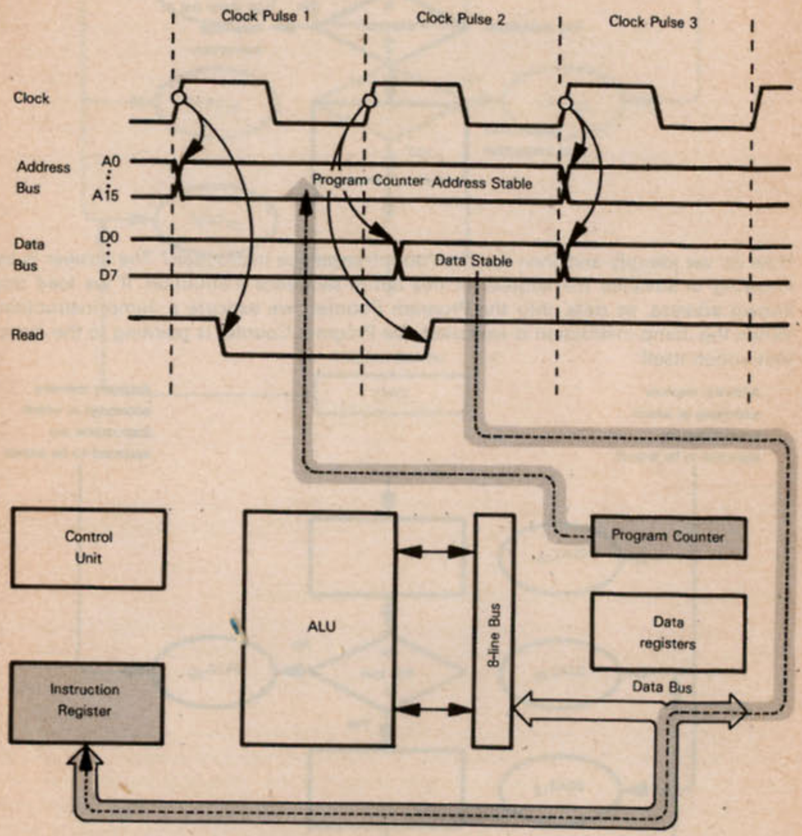

Program Memory
First program memory address
After instruction fetch, increment address and point here
start of program

031A	2 A
031B	3 F
031C	1 C
031D	0 2
031E	1 A
031F	F 1
0320	2 C
0321	3 A
0322	1 F
0323	1 A
0324	0 C
9325	0 A
0326	2 C
0327	1 F
etc.	

These hexadecimal numbers have been arbitrarily selected to represent instruction codes.

PROGRAM COUNTER

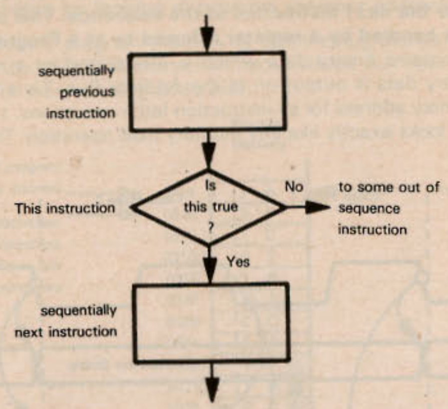
This being the case, all we need to do is identify the address of the first instruction within the sequence. After every instruction fetch, if we increment this address, then it will accurately point to the next instruction in the sequence. This program memory addressing logic is handled by a register referred to as a Program Counter. The Program Counter contains binary data which is interpreted as a memory address only because the binary data is output on to the Address Bus. Thus the Program Counter provides the memory address for all instruction fetch operations; to external memory an instruction fetch looks exactly like any memory read operation. This may be illustrated as follows:



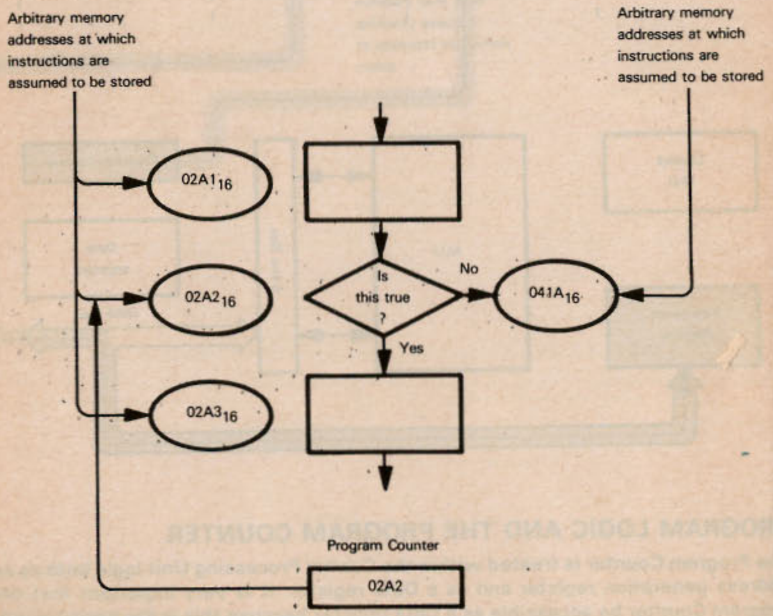
PROGRAM LOGIC AND THE PROGRAM COUNTER

The Program Counter is treated within the Central Processing Unit logic both as an address generation register and as a Data register. It is very important that the Program Counter be accessible as a Data register because this is the basis for program logic.

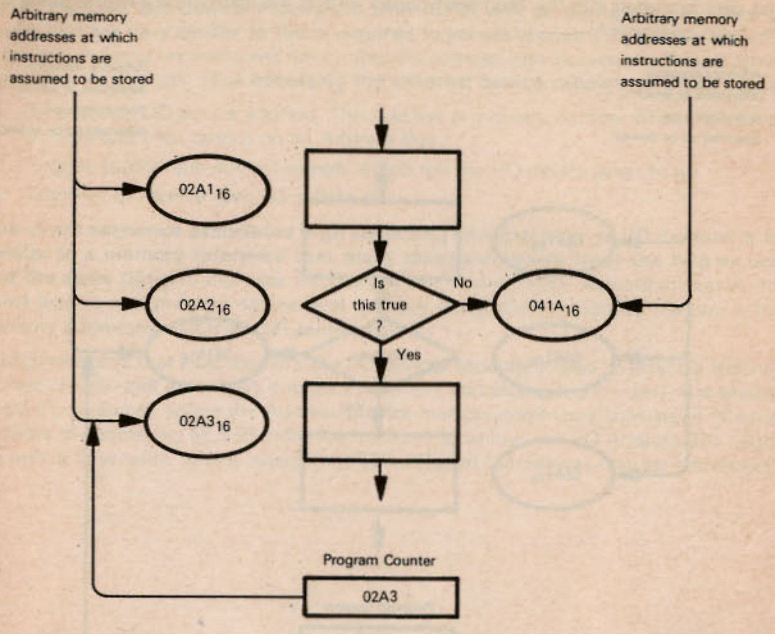
We have already seen in program flowcharts decision-making steps which allow the program to continue by executing the next sequential instruction, or some out-of-sequence instruction:



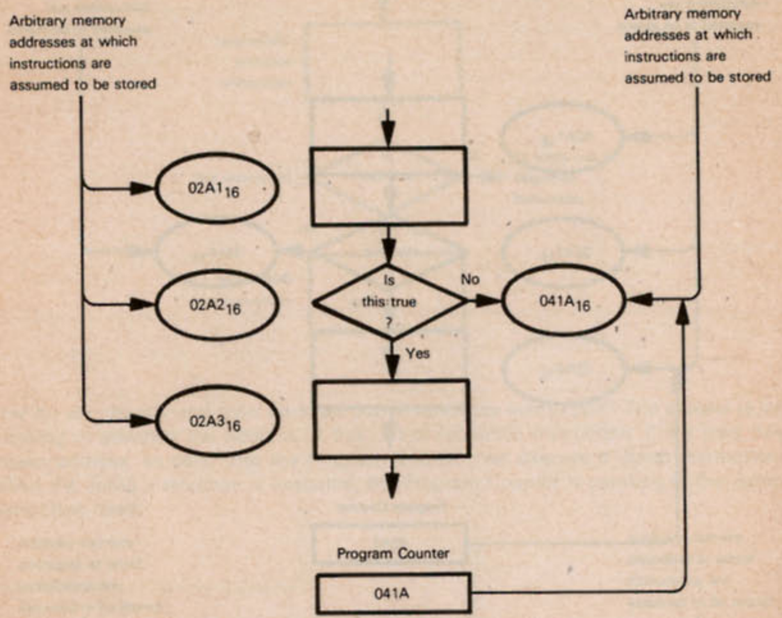
How do we identify and then fetch an out-of-sequence instruction? The answer is by knowing in advance the address of this out-of-sequence instruction. If we load this known address, as data, into the Program Counter, we execute a Jump instruction. When the Jump instruction is executed, the Program Counter is pointing to the Jump instruction itself:



After the Jump instruction has been fetched, the Program Counter will be left pointing to the next sequential instruction:



If the Jump is to occur, then the Central Processing Unit will make the Jump happen by loading the out-of-sequence instruction's address, as data, into the Program Counter. This data is loaded into the Program Counter during the instruction execute phase of the Jump instruction:


Arbitrary memory addresses at which instructions are assumed to be stored
Arbitrary memory addresses at which instructions are assumed to be stored
$02A1_{16}$
$02A2_{16}$
$02A3_{16}$
Is this true ?
No
Yes
$041A_{16}$
Program Counter
041A

Now when the Jump instruction has finished executing, and the next instruction starts executing, an instruction fetch will occur; but instead of getting the next sequential instruction, you get the new out-of-sequence instruction.

DATA MEMORY ADDRESSING REGISTERS

As we saw earlier in this chapter, addressing data memory is not as straightforward as addressing program memory, because there is no"usual" sequence in which data will be stored. Thus a variety of ingenious techniques are used to create addresses for data memory.

Data memory address computation becomes a simple extension of logic on the data side of the Arithmetic and Logic Unit (ALU). Given Data registers in which we can store binary data, plus an Arithmetic and Logic Unit which we can use to perform computations, we have all the prerequisites to calculate binary data which will subsequently become data memory addresses. Thus data memory address computation is easily handled by logic already present in a Central Processing Unit.

In this book we are not going to discuss the various methods used to create data memory addresses. This is information which only becomes necessary when you start learning how to write programs in assembly language, and that is material for Volume I. A data memory address is the result of data computations that preceded a data access; that is all you need to understand for now.

EXTERNAL LOGIC ADDRESSING

In order to access some external logic or peripheral device (such as a keyboard or video display), the Central Processing Unit (CPU) will go through a sequence of steps that are very similar to those required to access memory. The actual external logic device being accessed will have a specific address, just as every memory location has a specific address. Thus **accessing the external device consists of these steps:**

1) Generate an I/O device address. This address is a binary number which is created as data and then output on an Address Bus.
2) Trigger appropriate control signals which tell the I/O device what to do.
3) Transmit or receive data via a Data Bus.

The event sequence associated with accessing external logic or I/O devices is so similar to a memory reference that many microprocessors treat the two as one and the same thing. In this case, I/O devices and external logic respond to exactly the same signals as a memory device; and the only thing which separates the two is the memory addresses which are set aside for each.

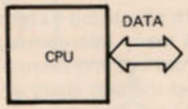
Microprocessors that treat memory and I/O devices separately have, in effect, a secondary set of logic that essentially duplicates memory addressing logic — but on a smaller scale. For example, where the Address Bus for memory commonly consists of 16 lines capable of addressing 65,536 individual memory locations, the I/O Address Bus might be only 8 lines wide, which means that 256 different I/O devices may be addressed.

INSTRUCTION SETS AND PROGRAMMING

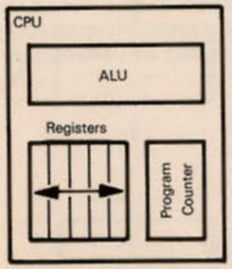
The instructions of a microprocessor, you will recall, identify the individual operations which can be performed as single entities by logic within the microprocessor. There are five types of events which may be specified by individual instructions; they may be illustrated as follows:

Microprocessor Instructions

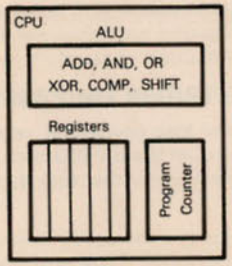
① Transferring data between the microprocessor and logic beyond the microprocessor



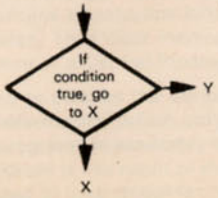
② Moving data from one register to another within the microprocessor



③ Specifying an arithmetic or logic unit operation



④ Modifying Program Counter (PC) contents and thus enabling programmed logic



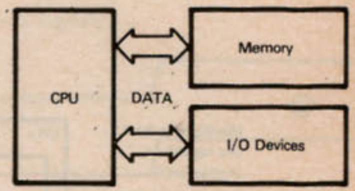
⑤ Status flag manipulation "condition" in type ④ instruction is one example of status.

Let us consider these instruction types one at a time.

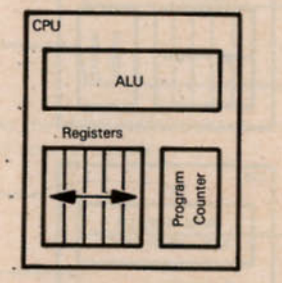
Instructions that move data between the microprocessor and logic beyond the microprocessor may access memory or physical devices. Physical devices beyond the microcomputer system are referred to generally as input/output (or I/O) devices. Instructions that access I/O devices are called I/O instructions. **Some microprocessors have separate instructions to transfer data between the microprocessor and I/O devices or memory:**

Microprocessor Instructions

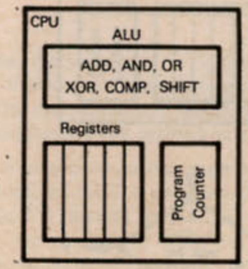
① Transferring data between the microprocessor and logic beyond the microprocessor



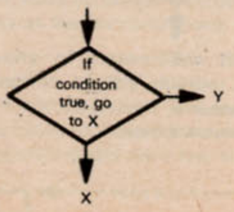
② Moving data from one register to another within the microprocessor



③ Specifying an arithmetic or logic unit operation



④ Modifying Program Counter (PC) contents and thus enabling programmed logic

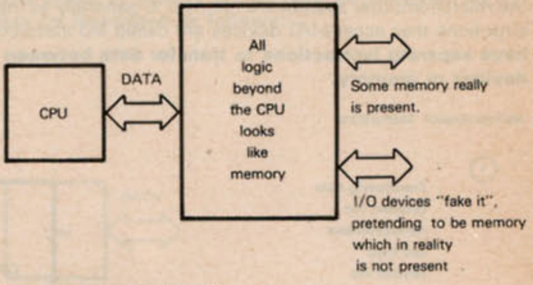


⑤ Status flag manipulation "condition" in type ④ instruction is one example of status.

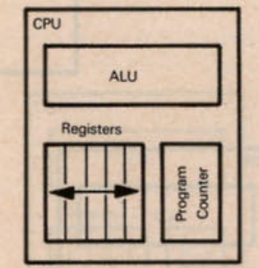
Other microprocessors use the same instructions to transfer data between the microprocessor and memory or I/O devices:

Microprocessor Instructions

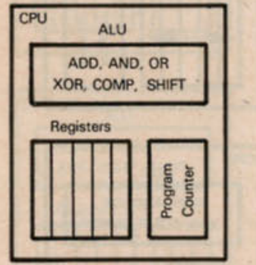
① Transferring data between the microprocessor and logic beyond the microprocessor


CPU
DATA
All logic beyond the CPU looks like memory
Some memory really is present.
I/O devices "fake it", pretending to be memory which in reality is not present

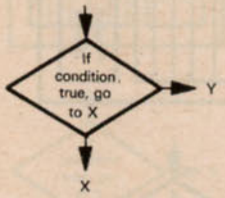
② Moving data from one register to another within the microprocessor



③ Specifying an arithmetic or logic unit operation

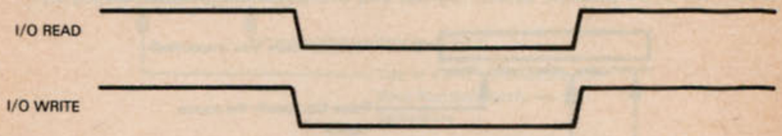


④ Modifying Program Counter (PC) contents and thus enabling programmed logic

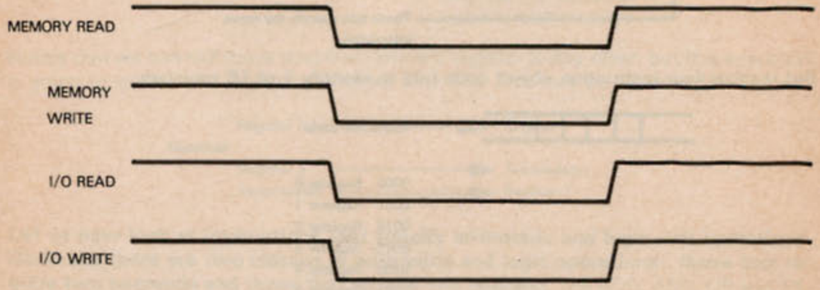


⑤ Status flag manipulation "condition" in type ④ instruction is one example of status.

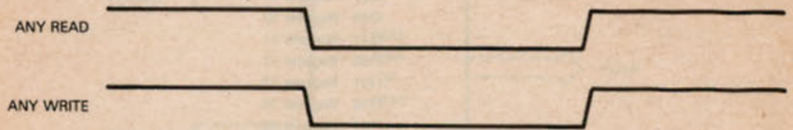
You will recall from our earlier discussion of signals and busses that **there is really very little difference between I/O and memory reference instructions.** I/O reference instructions generate control signals identifying transfer of data to or from an I/O device:



Memory reference instructions generate equivalent signals:



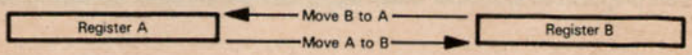
A microprocessor that uses the same instructions to access memory or an I/O device will simply have one set of control signals:



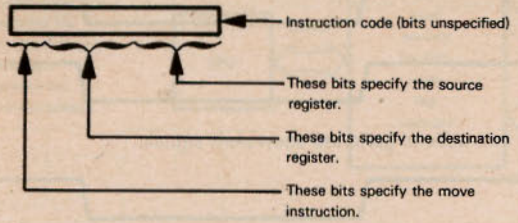
When a microprocessor uses the same control signals to access memory or I/O devices, logic associated with the memory address distinguishes between memory or an I/O device. Whoever designs the microcomputer decides which addresses will, indeed, access memory, and which addresses will instead select an I/O device.

Note that just because a microprocessor has separate memory and I/O instructions a microcomputer designer does not have to use the I/O instructions. The microcomputer designer could still address I/O devices as though they were memory locations. The converse is not true. A microprocessor that has no I/O instructions forces the microcomputer designer to address I/O devices as memory locations. In every case, you, as a microcomputer user, have no choice. You must use memory reference and I/O reference instructions exactly as the microcomputer designer tells you to.

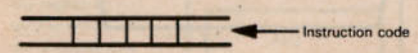
The number and complexity of instructions that move data between CPU registers is strictly a function of the number of registers provided by the microprocessor. Obviously a microprocessor that only has one addressable register is not going to have any instructions to move data between registers. A microprocessor that has two addressable registers could have two such instructions:



As the number of registers within the CPU increases, the microprocessor will run out of possibilities. Suppose, for example, a microprocessor has 16 addressable registers. In order to move data from any one of the 16 registers (as the source) to any one of the 16 registers (as the destination), the microprocessor instruction code must have some bits to identify every possible source register and every possible destination register.

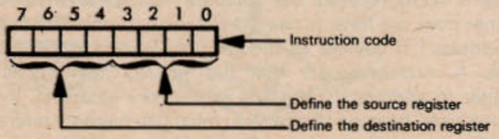

Instruction code (bits unspecified)
These bits specify the source register.
These bits specify the destination register.
These bits specify the move instruction.

But it takes four instruction object code bits to identify 1 of 16 registers:



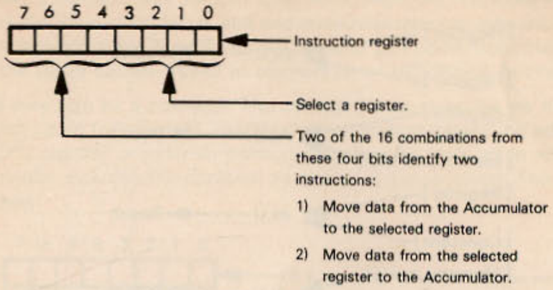
0000	Register 0
0001	Register 1
0010	Register 2
0011	Register 3
0100	Register 4
0101	Register 5
0110	Register 6
0111	Register 7
1000	Register 8
1001	Register 9
1010	Register 10
1011	Register 11
1100	Register 12
1101	Register 13
1110	Register 14
1111	Register 15

If four instruction object code bits are needed to specify the source register, and another four instruction object code bits are needed to specify the destination register, then for an 8-bit microprocessor, all eight instruction object code bits will be used up simply specifying instructions that move data between a source register and a destination register:



No codes left !!

One solution adopted by many microprocessor designers is to have a "primary" register, often referred to as an Accumulator; the Accumulator must be the source or the destination for any data movement within the CPU. Now instructions that move data from one location to another within a CPU must always move the data through the Accumulator. For the case of 16 registers we now need just four bits to define the selected register:



Two of the 16 combinations from these four bits identify two instructions:

1) Move data from the Accumulator to the selected register.

2) Move data from the selected register to the Accumulator.

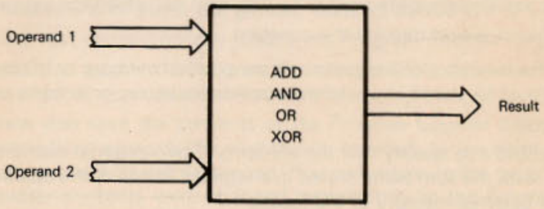
Notice that we can still move data from any one register to any other, but this operation is going to require two instructions rather than one. This may be illustrated as follows:

Register 3 → Register 7

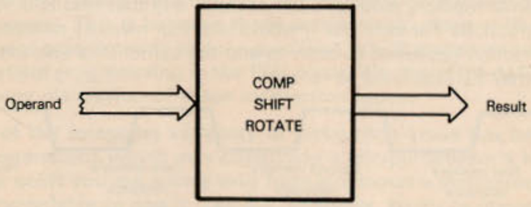
becomes

Register 3 → Accumulator
Accumulator → Register 7

Let us now look at instructions that specify arithmetic and logic unit operations. Generally there are two classes of arithmetic and logic operations: those that require two operands and those that require one operand. The ADD, AND, OR and EXCLUSIVE OR operations require two operands:



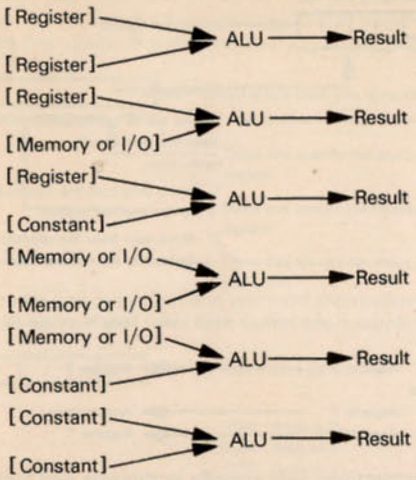
COMPLEMENT, SHIFT and ROTATE operations require one operand:



Operands can come from one of three locations:

- From a register within the microprocessor
- From an external memory or I/O location
- As a constant provided by the instruction itself

Consider first two operand instructions. You have six possibilities, just for the operands. Using the notation [ ] to signify "contents of", these possibilities may be illustrated as follows:



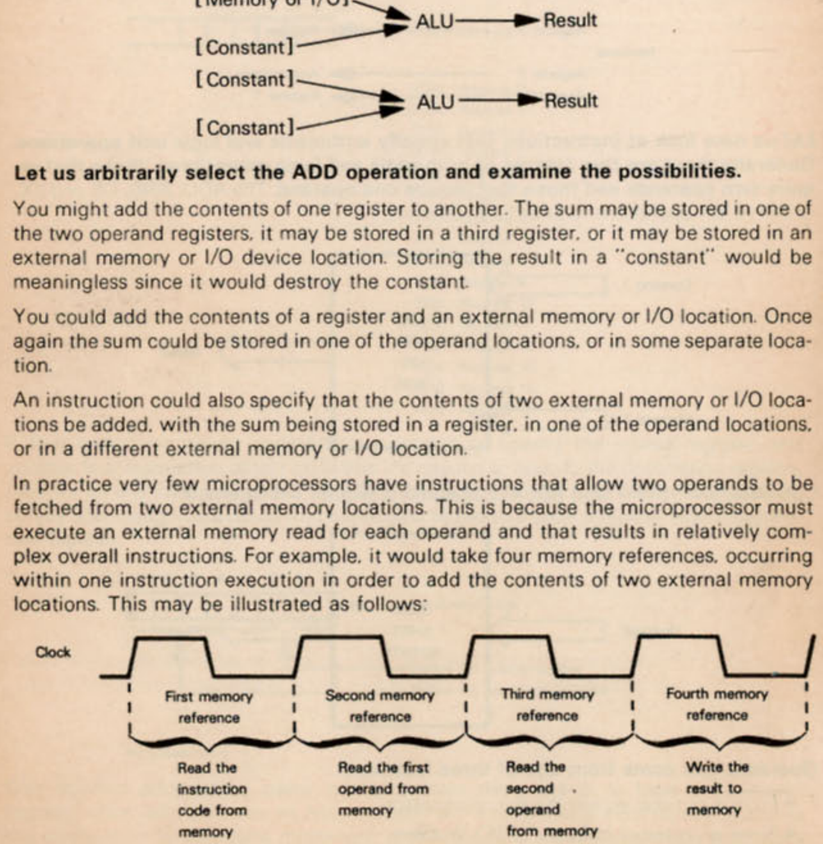
Let us arbitrarily select the ADD operation and examine the possibilities.

You might add the contents of one register to another. The sum may be stored in one of the two operand registers, it may be stored in a third register, or it may be stored in an external memory or I/O device location. Storing the result in a "constant" would be meaningless since it would destroy the constant.

You could add the contents of a register and an external memory or I/O location. Once again the sum could be stored in one of the operand locations, or in some separate location.

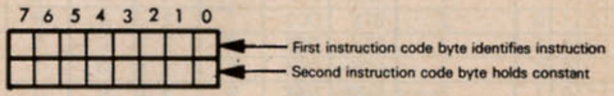
An instruction could also specify that the contents of two external memory or I/O locations be added, with the sum being stored in a register, in one of the operand locations, or in a different external memory or I/O location.

In practice very few microprocessors have instructions that allow two operands to be fetched from two external memory locations. This is because the microprocessor must execute an external memory read for each operand and that results in relatively complex overall instructions. For example, it would take four memory references, occurring within one instruction execution in order to add the contents of two external memory locations. This may be illustrated as follows:


Clock
First memory reference
Second memory reference
Third memory reference
Fourth memory reference
Read the instruction code from memory
Read the first operand from memory
Read the second operand from memory
Write the result to memory

There is nothing intrinsically impossible about having a complex instruction, as illustrated above, and in fact many minicomputers have such instructions. But microprocessors are generally simpler than minicomputers. Typically microprocessors are limited to the instruction fetch, and one additional memory reference during one instruction's execution time. Therefore one operand can come from memory (or an I/O device), or the result can be stored in memory or an I/O device, but not both.

An operand may also be a constant. The constant is provided by the instruction code. An instruction could for example, specify that a constant value 3 be added to the contents of a CPU register or external memory location. An instruction which specifies a constant actually includes the constant as part of the instruction. This may be illustrated as follows:


7 6 5 4 3 2 1 0
First instruction code byte identifies instruction
Second instruction code byte holds constant

In reality a constant is nothing more than the contents of a location in program memory. But since program memory frequently becomes Read Only Memory, the constant does indeed become a constant — because it resides in a part of memory that can never be modified.

ALU operations that require a single operand may specify the contents of a register, memory location or I/O device as the operand. Microprocessor instructions will not allow you to specify a constant as the input for a single operand ALU instruction because that would make no sense. For example, an instruction to complement 3 would be an unnecessary instruction; you know what the complement of 3 is, so you might as well store this value in the first place. **Instructions that modify the Program Counter contents fall into one of three categories.** These are:

1) Instructions that unconditionally modify the Program Counter contents
2) Instructions that modify the Program Counter contents only when specific conditions identified by appropriate status are met
3) Instructions that save the contents of the Program Counter before modifying it. These instructions give you the opportunity to return to the point where the Program Counter contents were changed. You can return to the point where the Program Counter contents were changed by restoring to the Program Counter the value which you saved before changing it.

While instructions that move data and manipulate ALU logic are self-evident, instructions that manipulate the Program Counter contents are logically elusive to a novice programmer. This is because these instructions, along with status instructions, implement programming logic — a subject which makes no sense to you until you understand programming in the first place. We therefore defer until Volume 1 any reasonable discussion of these instruction types.

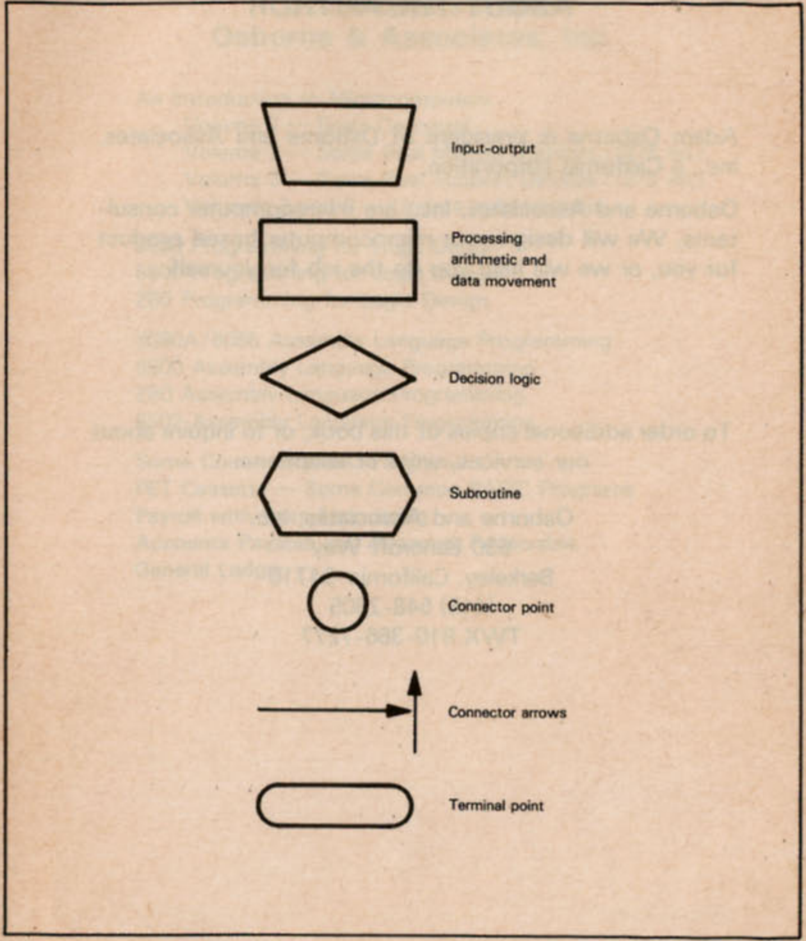
The purpose of the foregoing summary of instruction types has been to identify the types of operations which may constitute a microprocessor's instruction set. Clearly at this point you are a long way from looking at a microprocessor instruction set and being able to use it. You are, however, ready to move on to "An Introduction To Microcomputers: Volume 1 — Basic Concepts" which covers essentially the same material as Chapters 4, 5 and 6 of this book, but in much more detail. By the time you have finished reading "Volume 1 — Basic Concepts", you will be in a position to start using microprocessors.

APPENDIX A
ASCII CHARACTER CODES

b7 →						0	0	0	0	1	1	1	1
b6 →						0	0	1	1	0	0	1	1
b5 →						0	1	0	1	0	1	0	1
BITS	b4	b3	b2	b1	Column Row	0	1	2	3	4	5	6	7
	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
	0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
	0	0	1	0	2	STX	DC2	"	2	B	R	b	r
	0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
	0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
	0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
	0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
	0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
	1	0	0	0	8	BS	CAN	(	8	H	X	h	x
	1	0	0	1	9	HT	EM	)	9	I	Y	i	y
	1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
	1	0	1	1	11	VT	ESC	+	;	K	[	k	{
	1	1	0	0	12	FF	FS	,	<	L	\	l	|
	1	1	0	1	13	CR	GS	-	=	M	]	m	}
	1	1	1	0	14	SO	RS	.	>	N	Λ	n	~
	1	1	1	1	15	SI	US	/	?	O	_	o	DEL

NUL	Null	DC1	Device control 1
SOH	Start of heading	DC2	Device control 2
STX	Start of text	DC3	Device control 3
ETX	End of text	DC4	Device control 4
EOT	End of transmission	NAK	Negative acknowledge
ENQ	Enquiry	STN	Synchronous idle
ACK	Acknowledge	ETB	End of transmission block
BEL	Bell, or alarm	CAN	Cancel
BS	Backspace	EM	End of medium
HT	Horizontal tabulation	SUB	Substitute
LF	Line feed	ESC	Escape
VT	Vertical tabulation	FS	File separator
FF	Form feed	GS	Group separator
CR	Carriage return	RS	Record separator
SO	Shift out	US	Unit separator
SI	Shift in	SP	Space
DLE	Data link escape	DEL	Delete

APPENDIX B
STANDARD FLOWCHART SYMBOLS



Flowcharts are a pictorial representation of computer program organization. They show the sequence of operations and are therefore helpful to the programmer in locating errors in program design. Flowcharts use the above standard symbols which are comprehensible to a non-programmer. The symbols used represent operations, data, flow, and equipment.

ABOUT THE AUTHOR

Adam Osborne is president of Osborne and Associates, Inc., a California corporation.

Osborne and Associates, Inc., are microcomputer consultants. We will design your microcomputer based product for you, or we will help you do the job for yourself.

To order additional copies of this book, or to inquire about our services, write or telephone:

Osborne and Associates, Inc.
630 Bancroft Way
Berkeley, California 94710
(415) 548-2805
TWX 910-366-7277

Other books from Osborne & Associates, Inc.

An Introduction to Microcomputers
  Volume 1 — Basic Concepts
  Volume 2 — Some Real Microprocessors (1978 ed.)
  Volume 3 — Some Real Support Devices (1978 ed.)
  Volumes 2 and 3 — Update subscriptions

8080 Programming for Logic Design
6800 Programming for Logic Design
Z80 Programming for Logic Design

8080A/8085 Assembly Language Programming
6800 Assembly Language Programming
Z80 Assembly Language Programming
6502 Assembly Language Programming

Some Common BASIC Programs
PET Cassette — Some Common BASIC Programs
Payroll with Cost Accounting
Accounts Payable and Accounts Receivable
General Ledger

"The latest volume in the classic series **An Introduction to Microcomputers, The Beginner's Book** is probably the most realistic and thorough orientation guide for the new computer enthusiast available . . . Osborne concentrates on educating the reader about the functional capabilities that the computer and peripherals he buys should have."

-Computer Dealer, March 1978

". . . it presents general concepts in a way that gives the reader confidence and a sense of achievement . . . All in all **Volume 0** is an excellent introduction and one I highly recommend."

-Tom Williams, People's Computers
March 1978

Created out of a demand for a primer to **Volume 1, Volume 0 — The Beginner's Book** introduces the novice to the world of microcomputers. While clearly defining technical terms, it provides an overview of microcomputer components and how these components relate to each other within a microcomputer system. The book serves as a basic introductory text on programming languages, binary code and arithmetic, operators, logic, timing and memory, or as a refresher for those already familiar with these concepts.

AN INTRODUCTION TO MICROCOMPUTERS

Volume 0 — The Beginner's Book
Volume 1 — Basic Concepts
Volume 2 — Some Real Microprocessors
Volume 3 — Some Real Support Devices

ISBN 0-931988-26-8

AN INTRODUCTION TO MICROCOMPUTERS VOLUME

931988-26-8