PART 3
PROGRAMMING POPULAR MICROPROCESSORS

13

INTRODUCTION TO MICROPROCESSORS

This part of the text is designed to introduce you to some of the more popular microprocessors. The design and operation of a microprocessor are based on the digital circuits which you studied in Part 1.

You will learn the basic principles of microprocessors and how to write simple assembly language programs. In the study of computers, programming, and microprocessors, one fundamental idea emerges:

> If you do correctly a great number of very simple tasks, you will have done something complicated.

If you understand the basic principles and simple programs presented here, you will be on your way to understanding more complicated ideas.

Since the microprocessor is a "computer on a chip," it may help to take a quick look at computers before starting to study microprocessors.

13-1 COMPUTER HARDWARE

The digital circuits you studied in the first part of this text are the building blocks of a computer. In the early days of computers, digital circuits were made by using vacuum tubes and later were built with transistors. Circuits were designed which would act as the "brain" of a computer. These circuits were called the *central processing unit* (CPU). The CPU could perform basic arithmetic operations such as addition and subtraction, logic operations such as ANDing and ORing, and control operations. Thus it could process data.

A CPU cannot be used alone. There are other components which are needed to make a computer. For example, we said that a CPU can process data. Where is this data? We need memory—a place where data can be stored until the CPU needs it. And what if the CPU does a calculation and comes up with an answer? How would we know what the answer is? We need a way for the CPU to communicate with us. We need an output device. Figure 13-1 illustrates what a simple system looks like.

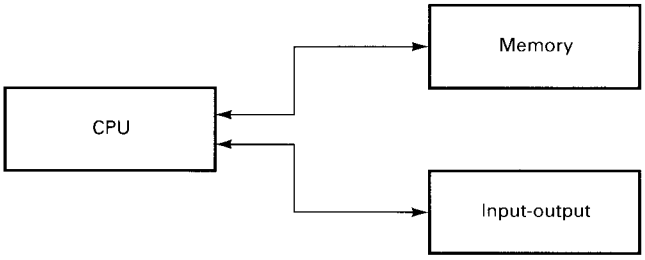


Fig. 13-1 A simplified overview of a microprocessor system.

13-2 DEFINITION OF A MICROPROCESSOR

What exactly is a microprocessor? As the name implies, it must be small (micro-) and it must be able to process data (-processor). A microprocessor is a CPU which is constructed on a single silicon chip. What, then, is a CPU? A *CPU* is an electronic circuit which can interpret and execute instructions and control input and output.

In this text, when reference is made to a microprocessor, only the microprocessor is being referred to. However, if reference is made to a computer, then we are talking about a device which contains a microprocessor and several subsystems. Figure 13-2 serves to illustrate this.

13-3 SOME COMMON USES FOR MICROPROCESSORS

Microprocessors can be found in a variety of products. Some well-known examples are computers and industrial controls. Some not-so-obvious products that use micropro-

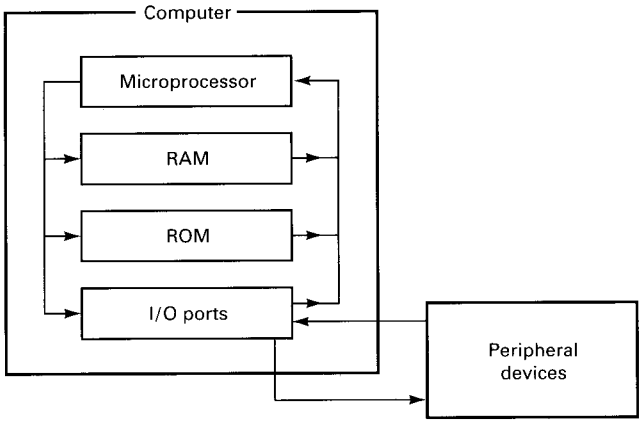


Fig. 13-2 Block diagram of a complete computer and peripherals.

cessors include answering machines, compact disk players, and automobiles.

The microprocessor supplies electronic products with a new dimension. In the past, electronic products have been able to make simple decisions because of certain kinds of circuitry and/or sensors. The microprocessor, however, has multiplied this trait many times: Some devices, most notably computers, now almost appear to think.

13-4 MICROPROCESSORS FEATURED IN THIS TEXT

It is the purpose of this book to examine the most popular 8-bit microprocessor families in addition to the 16-bit Intel 8086-8088 family.

6502 Family

The 6502 family is supported by this text. The 65C02, an advanced version of the 6502 which is used in the Apple IIc, has some additional instructions and enhanced features which can be found in the manufacturer's programming manuals.

6800 Family

The 6800/6808 is supported by this text. The 6809 is an enhanced version of the 6800. It understands all the instructions of the 6800 and includes some other advanced features.

8080/8085/Z80 Family

The 8080, 8085, and Z80 are also supported in this text. The 8080 and 8085 have exactly the same instruction set except for two additional instructions included in the 8085. The Z80 understands all the 8080/8085 instructions and has many other additional instructions.

Only those instructions common to all three microprocessors are discussed in this text. (The extended Z80 instructions are not used in the text.) This has the advantage of making it possible for students to use a mixture of 8085 and Z80 microprocessor trainers in the same class at the same time with all students on equal footing and with a minimum of confusion. Either Z80 or 8085 mnemonics can be used interchangeably for the homework problems and the object code will be the same.

8086/8088 Family

The Intel 8086/8088 is the only 16-bit microprocessor discussed in this text. This microprocessor (in addition to the 80286, 80386, and 80486) is used in the popular IBM PCs, IBM compatibles, and clones. The DOS DEBUG utility is used throughout the text. Assemblers are introduced in later chapters.

13-5 ACCESS TO MICROPROCESSORS

Developing skill in programming and interfacing microprocessors requires access to a microprocessor. Here are some ways to gain access to a microprocessor supported by this text.

Computers

The 6502 or one of its derivatives can be found in the entire line of Commodore computers including the PET, Vic-20, C-64, C-16, Plus-4, and C-128. They can also be found in the Apple II line of computers including the Apple II, II+, IIe, IIc, and IIc+. They are also included in that portion of the Laser line of computers that are Apple-compatible, including the Laser 128, Laser 128 EX, and Laser 128 EX/2. And last of all, some of the older Atari home computers contain this type of microprocessor.

The 8085 and Z80 can be found in some of the older CP/M machines. (CP/M stands for control program for microprocessors.) The Z80 was used in Radio Shack's TRS-80 line of computers and is also found in the Commodore 128 (the Commodore 128 contains two microprocessors). The Commodore 128 will also run CP/M software if that is desired.

The 8086/8088 are found in all of the IBM PCs and XTs, IBM compatibles, and clones. The 80286 is used in AT-class machines, and of course the 80386 is used in the newer 386s. These microprocessors use a superset of the 8086/8088 instructions set and can therefore also be used with this text.

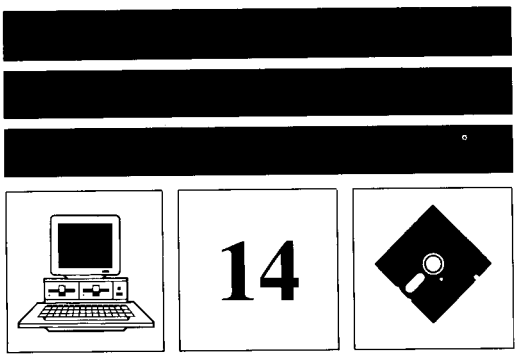
Some IBM compatibles use the NEC-V20 or one of the other NEC microprocessors. These are compatible with the Intel series of microprocessors and will work equally well.

Microprocessor Trainers

Another way to gain access to a microprocessor supported by this text is through the use of a microprocessor trainer. Heathkit's ET-3400-A trainer contains a 6808 chip. E&L Instruments has the "FOX" (MT-80Z) with a Z80 microprocessor. Intel makes the SDK-85, which features the 8085 chip, and the SDK-86, which uses the 8086. Motorola makes the MEK6800D with a 6800 chip.

Software Emulation Programs

Finally, there are software emulation programs that will make a computer act as though it is using another microprocessor.



PROGRAMMING AND LANGUAGES

What is a program and why do we need one? What do we mean by *program design?* What is a *programming language?* Why do we need a language? What is a *flowchart?* How does all of this relate to electronics and digital circuits? These are some of the questions we will try to answer in this chapter.

14-1 RELATIONSHIP BETWEEN ELECTRONICS AND PROGRAMMING

A question sometimes raised by electronics students is, "Why are we learning about programming microprocessors?"

Programming is a topic which is closely related to electronics. Mathematics and physics are topics which support or undergird the subject of electronics. They form a foundation. Programming is not so much a support subject as it is a related subject. Let's take a closer look at this.

Digital Electronics and Microprocessors

What prompted the creation of digital electronics? It was the desire to make a machine without moving parts which could perform mathematical calculations. Such a machine would be much faster than any mechanical calculator. Correctly connecting enough digital logic circuits together created such a machine.

Once the calculating machine had been built, there had to be a way to tell this machine to add, or subtract, or perform some logical operation. Thus programming was born. We simply needed a way to tell the machine what to do. In the beginning, programming was done by connecting wires or patch cords. This was very slow compared to what we do today.

Over the years digital circuits became more complex, the calculating machine grew into far more than just a big calculator, and the need for ways to communicate with the machine grew. Finally, it became possible to put the entire computer "brain" on a single chip.

Until this point an electronics technician might never work on or even see a computer. However, when the "brain" could be put on a chip, and the cost was measured in dollars rather than thousands of dollars, its possibilities became endless.

Designers and engineers realized that these "brains," or microprocessors, could improve the performance of many common electronic products and could make new products economically possible. With microprocessors everywhere, the electronics technician can no longer be unaware of their operation.

The Electronic Technician and Programming

So why should a technician learn about programming? Because the technician will probably eventually work on products with microprocessors, and the microprocessor cannot be separated from its program. A microprocessor without a program would be like a resistor with no resistance or a wire with no conductivity. Without the program, a microprocessor does nothing.

Programming is now part of the overall picture that electronics is concerned with—like mathematics and physics. Some technicians will not need as much knowledge about programming as others: It depends on what your career field is. But everyone should at least be aware of the basics.

The goal of this book is to provide the digital understanding and programming experience which would be appropriate for the "typical" electronics student.

14-2 PROGRAMMING

In everyday language:

> A *program* is a very detailed list of steps which must be followed to accomplish a certain task.

A Familiar Example

We have all used this concept of programming—of following specific steps to accomplish a certain task—but have probably not thought of it in these terms. Let's look at something like taking a city bus downtown. You would be likely to

1. Wear clothes appropriate for the weather that particular day.
2. Take some money or tickets.
3. Go to a nearby bus stop.
4. Wait for the correct bus.
5. Get on.
6. Pay the driver.
7. Sit down if there were empty seats available.
8. Wait until the bus arrived in the area you wished to go to.
9. Alert the driver you wished to get off.
10. Wait for the bus to stop.
11. And finally get off.

Figure 14-1 is a flowchart (we'll talk about flowcharts in just a minute) of this process.

Unless this was your first time riding a bus, you wouldn't think about every detail because much of it is understood and is a natural part of your life. You usually dress for the weather when you go outside, and you usually take money when you go places. With a computer, though, things are different.

Very little is "natural" for a computer. The microprocessor has several temporary storage places where numbers can be kept (called *registers*). The machine can add and subtract, it can AND and OR, it can move numbers from one register to another, and it can do other simple things, but *everything must be specified!* One of the things that often surprises people learning to program microprocessors is the amount of detail which is necessary when writing a program.

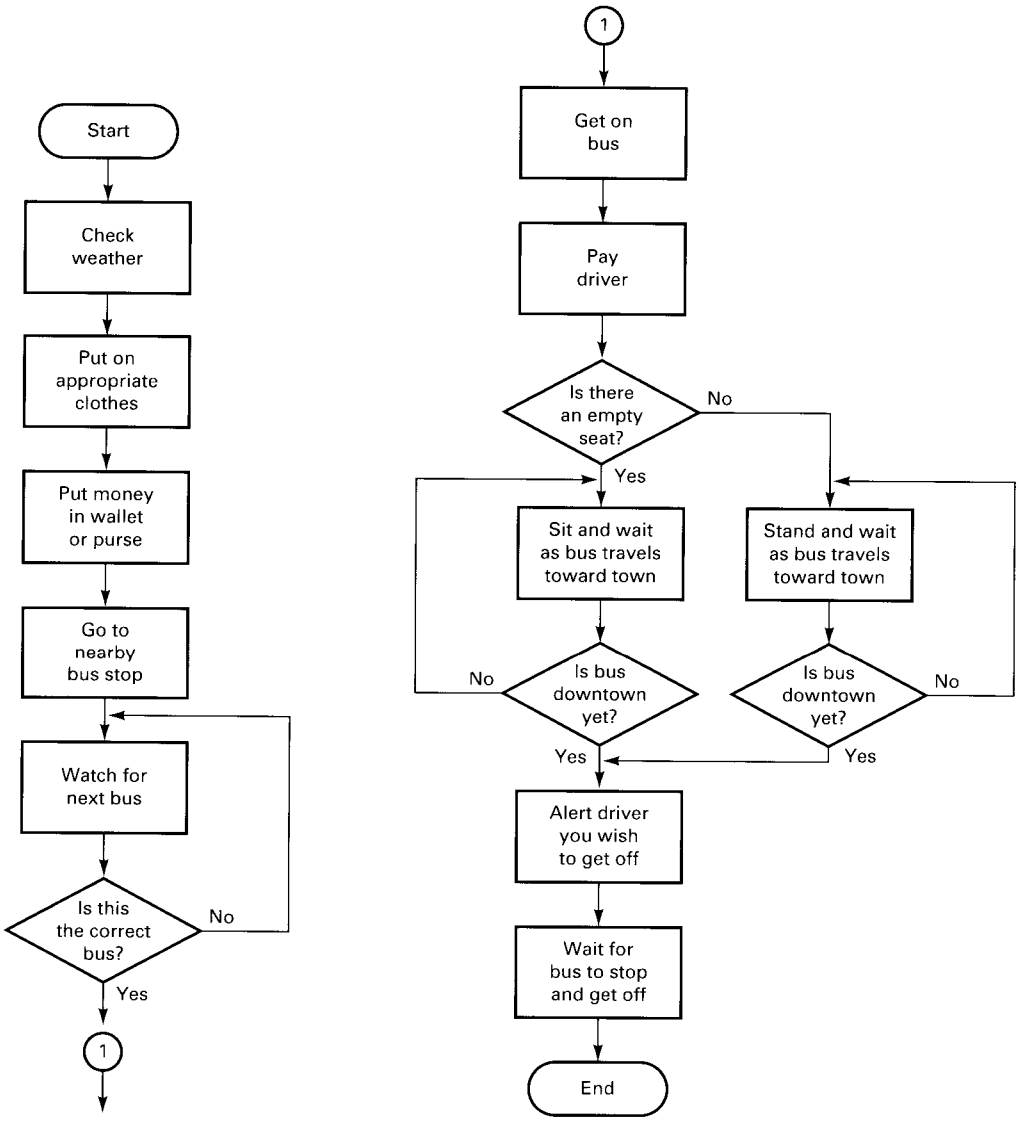


Fig. 14-1 Flowchart of a bus ride.

14-3 FUNDAMENTAL PREMISE

Before we look further at the subject of programming and flowcharts, we need to discuss a fundamental concept of programming. The concept is this:

> You cannot program the computer to do something you don't know how to do.

If you use computers only with application software (spreadsheets, word processors, and so on), this may not always be true, but if you want to program microprocessors, it is. Before you begin to think about how you will program a computer to do something, think about how you would do it yourself without a computer. After you know how you would do it, you can begin to tell the computer how it should do it.

14-4 FLOWCHARTS

When you are writing a program, it helps to have an organized way to write or express the flow of the program's logic. A flowchart is one way to do this.

Flowchart symbols

Figure 14-2 shows some common flowchart symbols. There are others, but we'll need only a few for most of the programs we'll be writing.

Straight-Line Programs

The simplest type of program is the *straight-line program.* In this type of program the steps involved follow each other, one after another, without any alternate routes or paths. Figure 14-3 is an example of a straight-line program.

This program is similar to one that might be used at the cash register of a store. It allows you to enter the price and product code of one item. The program then calculates a 5 percent sales tax, adds the tax to the original price to arrive at a total, and finally displays the total cost. The program will accept only one item, which means that it would have to be "run" again to find the total cost of a second item. Since we often buy more than one item at a time, let's look at another flowchart.

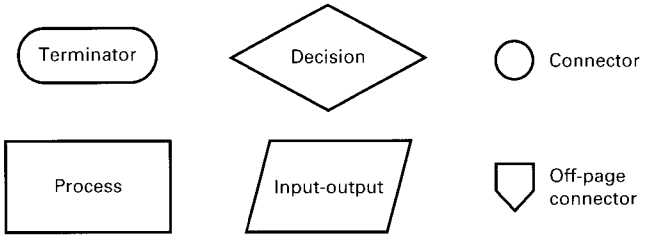


Fig. 14-2 Some flowchart symbols.

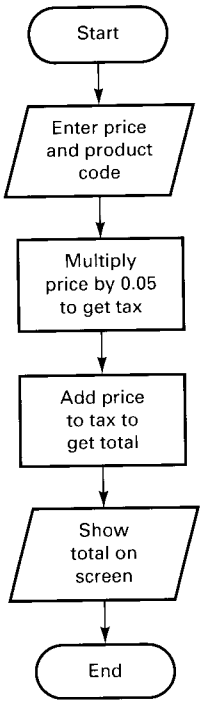


Fig. 14-3 Straight-line program to calculate sales tax and display total cost for one item.

Looping

A *loop* is a section of a program which will repeat over and over again. We can make the loop repeat indefinitely, or make it stop after a certain number of repetitions, or make it stop when some condition is met. Look at Fig. 14-4 and compare it to Fig. 14-3.

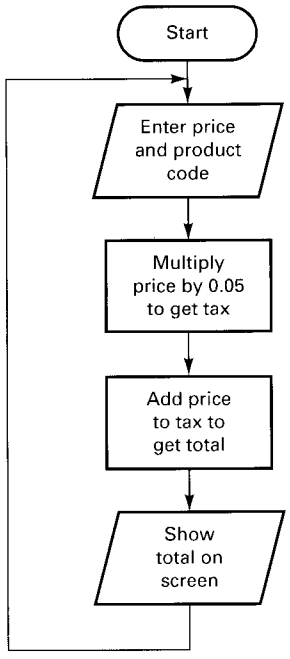


Fig. 14-4 Sales-tax program with loop.

These are almost identical, aren't they? What do you think this program will do that the one in Fig. 14-3 didn't? The answer, of course, is that this program is ready to accept a new number immediately after displaying the previous total. After you enter an item's price, the total cost is shown on the screen and the program then waits for you to enter the price of the next item.

Loops make it easier for programs to perform repetitive tasks. The program that uses loops can do the same calculations or functions over and over again.

Branching

Sometimes we want the computer program to do different things based on the situation at the time or based on the results of certain operations. We need a way to *branch* off from the main program flow. *Branching* allows us to write one program that can do different things at different times. Let's look at the sales-tax situation again. Study Fig. 14-5 at this time. This new version of the sales-tax program has a branch and a decision symbol.

Let's look at the decision symbol (diamond). If the program is to be able to take an alternate path when certain conditions exist, we must give it a chance to check for those conditions. The decision diamond represents that time. If the item is a nonfood item, it will be taxed as usual, and the program flow continues downward. If it is a food item which is not to be taxed, then we take the *branch*. The branch doesn't actually say not to tax the food item. But by making the total cost equal to the original price and bypassing the tax calculation section, we have effectively done the same thing. The total that appears will be the same as the original price, and the program will then loop back to the beginning to wait for the next item.

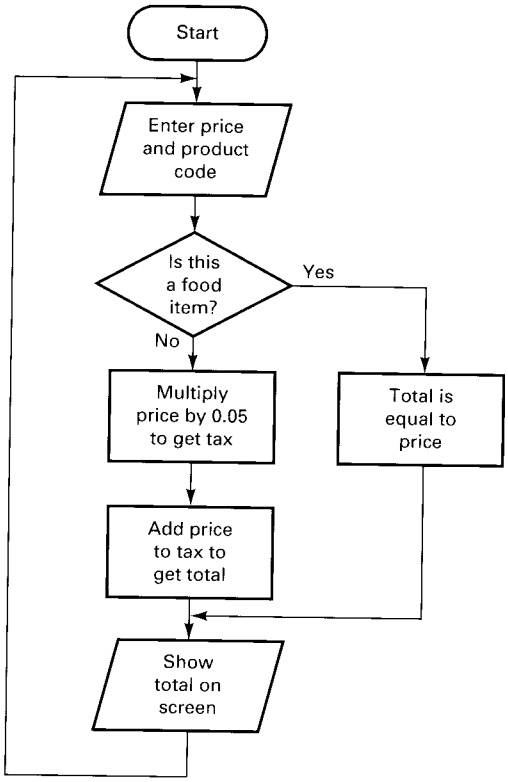


Fig. 14-5 Sales-tax program with loop and branch for nontaxable food items.

Subroutines

Sometimes we need to have the computer program take care of some intermediate task before it can continue with the main job at hand. We don't want it to branch and then end up somewhere else after the branch is finished. Rather, we want it to go to an intermediate task and then come right back to where it was before it left. This is called a *subroutine*. Looking at a subroutine will help clarify this new concept. Figure 14-6 shows our new program.

Everything is the same as in the last (Fig. 14-5) program except that we have added a subroutine which handles inventory. This subroutine is really just another small program that works along with the main one. It reduces the inventory total for this particular item by 1. If this total is less than 10, then it's time to order more. Either way, the subroutine prints a line on a printer in the administrative office with the product code and name of the product. We then "return" from the subroutine to the main program and continue where we left off.

Calling Subroutines

The act of going to a subroutine is often referred to as *calling* a subroutine, at the end of which we *return* to the main program.

The greatest advantage in having subroutines is not in calling or using them once but in using them several times in a program. You write that part of the program only once, but you can use it many times. Figure 14-7 illustrates this.

In Fig. 14-7 the boxes are not process boxes but rather representations of certain parts or modules of the whole computer program.

In this hypothetical situation there may be times when merchandise needs to be ordered other than when inventory drops below 10. For example, if a clerk finds a piece of merchandise damaged too badly to sell at a reduced price, it may simply be disposed of; however, it must be replaced to keep inventory up. The "damaged merchandise" part of the program can then *call* the "inventory-ordering subroutine" at some point.

Likewise, the store might sometimes give food or clothing to charity. This part of the program might also call the inventory-ordering subroutine to replace that merchandise.

This store's computer program uses the same subroutine in three different situations, but the programmer had to write the subroutine only once.

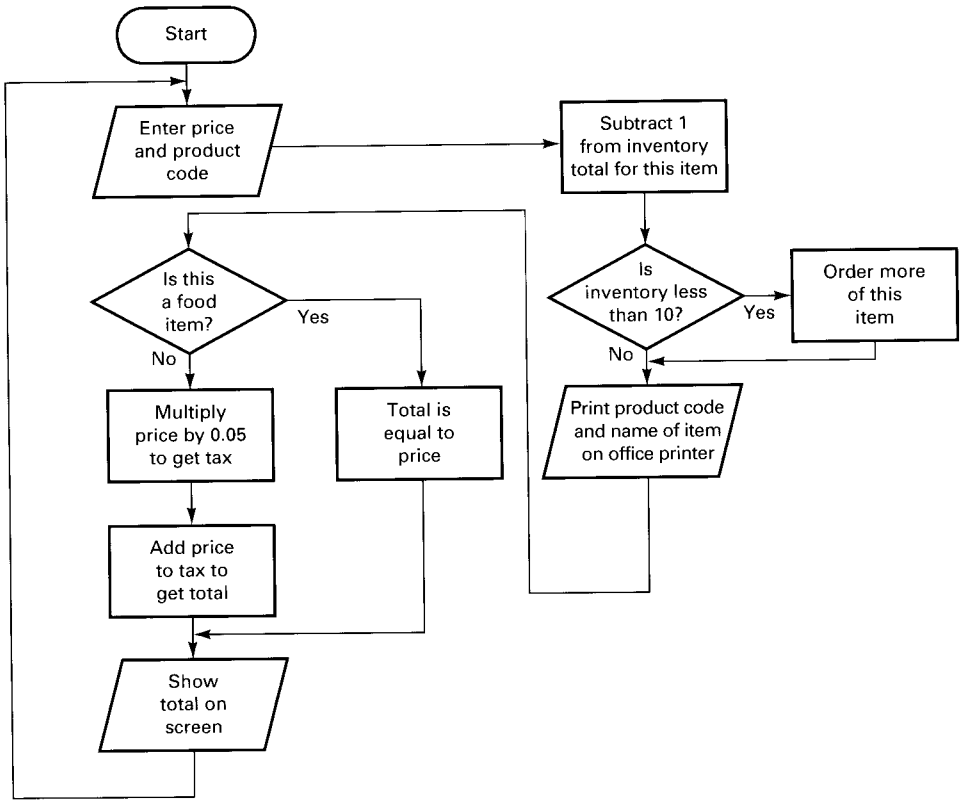


Fig. 14-6 Sales-tax program with inventory control reordering subroutine.

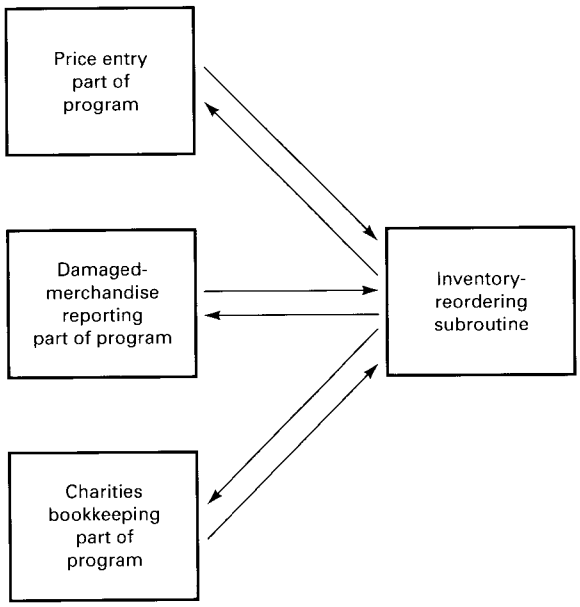


Fig. 14-7 Repetitive calling of inventory-reordering subroutine.

14-5 PROGRAMMING LANGUAGES

Now that we can define and flowchart the desired process, we need to be able to communicate this process to the computer. We need a language which the computer understands. Many languages have been developed for use with computers.

Machine Language

There is only one language the computer actually understands, and that is machine language, which consists of 1s and 0s. This binary language is fine for the computer but not for people. To have to communicate with the computer in binary, you would place in its memory a series of numbers that might look like this:

10010100
01001010
11101110
00101001

It would be nearly impossible to remember what the many different patterns of 1s and 0s meant, and the probability of making a mistake would be very high. Something better is needed.

Assembly Language

The first step toward a language that is easier for people to work with uses abbreviations to stand for different operations. For example, the instruction which tells a 6800 microprocessor to add numbers is the ADDA instruction, which stands for ADD accumulator A to a memory location.

This "language" of abbreviations is called *assembly language*. The "abbreviations" are called *mnemonics*. A mnemonic (pronounced ne-'män-ik) is something that aids the memory. Mnemonics are designed to be easy to remember and are a significant improvement over binary digits.

Machine language and assembly language are the subjects of this book. We refer to them as *low-level languages* because only very simple instructions exist.

High-Level Languages

Over the course of time, people working with computers felt it would be helpful to create languages that were more like English, so that it would not be so difficult to communicate with the computer and so that more advanced commands could be created. We call these *high-level languages*.

For example, many microprocessors do not have the ability to multiply or divide. It is obvious, however, that these are common mathematical functions that must be available to a computer programmer. In machine or assembly language one can use repeated additions to multiply or repeated subtractions to divide. This is not necessarily the best way to multiply or divide, but it is one way. In a high-level language there are "multiply" and "divide" commands. The language knows how to create the multiply and divide functions even though the microprocessor does not have these functions built in. In fact, these languages can understand English commands like *print, run, do, next,* and *end*. The microprocessor does not understand these English words, but the language changes (interprets or compiles) them into machine language before sending them to the microprocessor.

Many high-level languages have been created over the years. FORTRAN (formula translation) is a language that handles high-level mathematics very well and is designed for scientists and engineers. COBOL, which stands for common business-oriented language, is tailored to the needs of business. BASIC, which stands for beginner's all-purpose symbolic instruction code, was designed to be easy for nonprofessional programmers to learn and use. Pascal, named for the French mathematician Blaise Pascal, is designed to encourage the programmer to adhere to what are considered "correct" programming practices.

There are some languages that are somewhat "in between" the high-level and low-level languages, most notably C and FORTH. Figure 14-8 illustrates this.

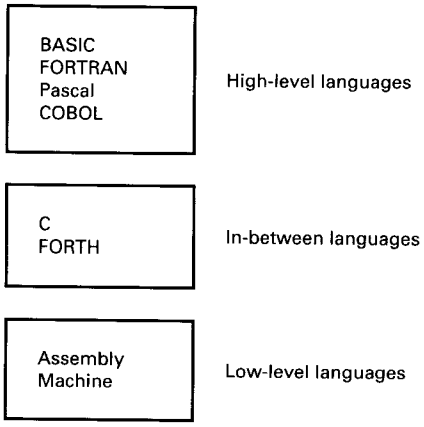


Fig. 14-8 Some examples of high-level, low-level, and in-between languages.

14-6 ASSEMBLY LANGUAGE

Let's look at the subject of assembly-language programming in a little more detail.

Machine language is the language the computer understands, but it is difficult for people to work with. Assembly language gives us the advantages of machine language without the disadvantage of doing something that seems so unnatural.

When we write in assembly language, we use abbreviations called mnemonics for certain operations or functions. The assembly language is called *source code*. It is more like English than machine language. The microprocessor, however, cannot act upon or execute mnemonics. It doesn't understand mnemonics. We need to convert the assembly language or source code into machine language or *object code*. There are a couple of ways to do this.

Manual Assembly

Let's look at manual assembly first. When using this technique, you write your program on paper using mnemonics. Then you look up each mnemonic on a chart. On the chart there will be a number which is the machine-language code for the assembly-language mnemonic. You then write down this object code so that you can later key it into the microprocessor trainer or computer. This is called *manual assembly* because you must look up the codes yourself.

Assembly with an Assembler or Monitor

The other way to create machine-language object code from assembly-language source code is through the use of a monitor or assembler. Since manual assembly involves simply looking up mnemonics on a chart, it seems reasonable that the chart could be stored in a computer and the computer

could look up the mnemonics and find their corresponding object code. Though there is much more to a fairly sophisticated assembler or monitor, this is the basic idea.

A *monitor* is a program that is normally stored in ROM and gives you access to the microprocessor's various registers. It sometimes has in it a simple *assembler* to change mnemonics into machine code and a *disassembler* to change machine code back into mnemonics.

An assembler program is usually more sophisticated than a monitor and has features that are difficult to explain at this point, but suffice it to say they are for more serious programming than the monitor. A longer period of time is required to become skilled in the use of an assembler, but it is a more powerful tool.

14-7 WORKSHEETS

During the remainder of this book you will be writing assembly-language programs. In addition to the flowchart, the *worksheet* is a tool which helps you stay organized as you write programs. The worksheet is simply a form on which you can write your program. It is laid out in such a way that it's a little easier to stay neat. Figure 14-9 is a portion of such a worksheet.

Name _____ Date _____

Program name _____ Sheet _____ of _____

Address	Obj code	Label	Mnemonic	Operand/Addr	Comment

Fig. 14-9 Example of a portion of a worksheet.

GLOSSARY

assembler A program which translates assembly language mnemonics into binary patterns (machine language).

assembly language A low-level language which uses mnemonics in place of binary patterns (machine language).

branch A section of a program which causes different actions to be taken based on conditions.

disassembler A program which translates binary patterns (machine language) into assembly language mnemonics.

loop A section of a program which will repeat over and over again.

mnemonic Something that aids the memory. Assembly language uses mnemonics, which are abbreviations for machine-language instructions.

monitor A program (usually stored in ROM) which gives the programmer access to the microprocessor's stack, accumulator, registers, and so forth. It sometimes contains a simple assembler.

straight-line program A program in which each step is followed by the next without any alternate routes or paths.

subroutine A portion of the program which is called upon to perform a specific task. When the task is finished, the main part of the program is returned to.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. Without a _____, a microprocessor does nothing.
2. (*program*) A _____ is a very detailed list of steps which must be followed to accomplish a certain task.
3. (*program*) What is the shape of the decision symbol?

4. (*Diamond*) _____ make programs more practical for doing repetitive tasks.
5. (*Loops*) The only language a computer actually understands is _____ language.
6. (*machine*) What does COBOL stand for?
7. (*Common business-oriented language*) A program in which the steps involved occur one after the other without any alternate paths is called a _____ program.
8. (*straight-line*) A section of a program which repeats indefinitely, a certain number of times, or while or until a certain condition exists is called a _____. (*loop*)

PROBLEMS

14-1. If you want to write a program to do something, what should you think about before you try to figure out what computer instructions to use?

14-2. What is the shape of the process symbol?

14.3. What provides an alternate path for program flow based on certain conditions?

14-4. What allows program execution to go to an intermediate task and then return to the place where it was before it started the intermediate task?

14-5. What is one of the advantages of using subroutines?

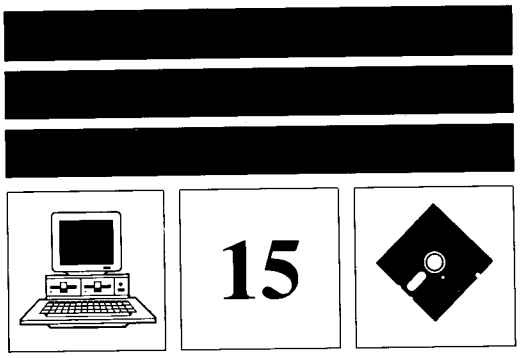
14-6. What is assembly language?

14-7. What does FORTRAN stand for?

14-8. What does BASIC stand for?

14-9. What was one of the goals of the creator of the Pascal language?

14-10. What does an assembler translate source code (mnemonics) into?



15

SYSTEM OVERVIEW

New Concepts

We'll begin this chapter by reviewing computer architecture. Then we'll spend the greater part of the chapter looking at microprocessor architecture in general and at the architecture of the microprocessor families supported by this text in particular.

15-1 COMPUTER ARCHITECTURE

Let's review computer architecture a little. Refer to Fig. 15-1.

Memory

We said that memory was needed so that there would be a place for data and instructions to be stored. Data and instructions which can be lost after power is removed are stored in RAM (random-access memory). Data and instructions which must never be lost, even after the power is turned off, are stored in ROM (read-only memory). Remember that ROM is a type of memory which cannot have its contents changed once the ROM chip is manufactured. PROM and EPROM are used in much the same way as ROM but can be programmed after being manufactured (PROM) or even programmed more than once (EPROM). PROM and EPROM differ from RAM in that they require special equipment to program them.

When we refer to memory in this text, we will usually be referring to RAM.

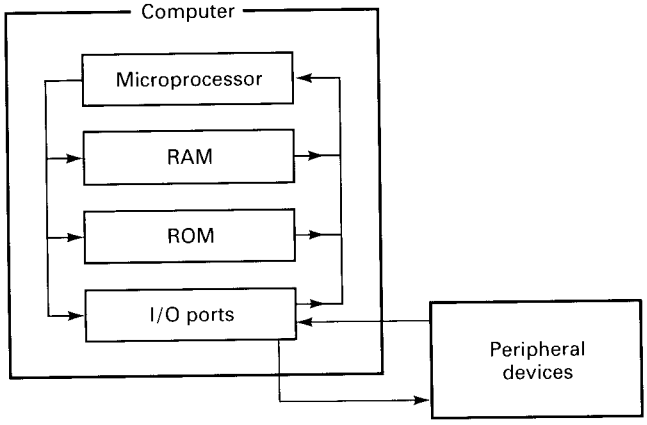


Fig. 15-1 Block diagram of a complete computer with peripheral devices. (Arrows indicate data flow.)

Addressing

Since there are many memory locations, it is necessary to have a means of referring to specific locations. This is done through addressing. Typically, memory locations are numbered from 0000 (in hexadecimal numbering) to the highest location used by that particular trainer or computer. This sequential number which is assigned to each location is its *address*. See Fig. 15-2.

A memory address is similar to the address of your home. Your house has a number or *address* assigned to it, and no other house on your street can have the same address. Inside your house are its *contents;* chairs, beds, and so on. Notice

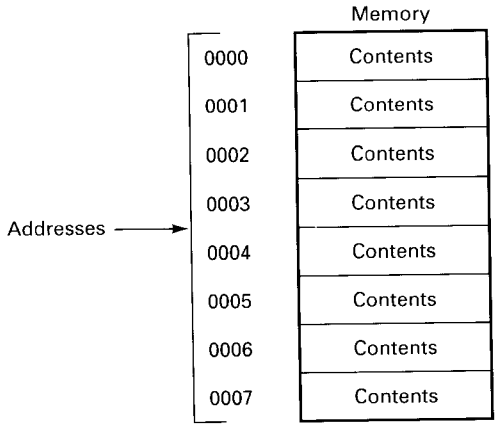


Fig. 15-2 Memory addressing.

that your home's address and your home's contents are not the same.

Each memory location has an address and contents. The address is necessary to specify which memory location to *read* information from or *write* information into. The contents is the information itself.

Address Bus

Most microprocessors can store information and instructions in a wide range of memory locations. Usually the memory locations are in a memory chip rather than in the microprocessor. The microprocessor needs a way to tell the memory chip which memory location it wants to put data into or take data from. It does this through the *address bus*. See Fig. 15-3.

The address bus is a communications link between the microprocessor and the memory chips. Physically, it is simply a group of electrical paths which are connected to RAM, ROM, and the I/O chips. Through this bus the microprocessor can specify the address of any memory location in any chip or device. Notice in Fig. 15-3 that information travels on the address bus in only one direction, from the microprocessor to memory and I/O. There are more details involved, but this is the basic idea.

Data Bus

Once the microprocessor has specified which memory location or device it wants to put data into or take data from, it then needs a set of electrical paths for this information to travel on. This set of paths is called the *data bus*.

It is this set of electrical paths that allows data to flow from one chip to the next. Notice in Fig. 15-3 that information on the data bus travels both to and from the microprocessor, memory, and I/O devices. Eight-bit microprocessors have a data bus that is 8 bits wide; 16-bit microprocessors have a data bus that is 16 bits wide. That is, the bus consists of 8 or 16 parallel connecting paths.

Addressing Range

Let's look at the normal range of addresses possible with 8-bit computers at this time.

In earlier chapters you studied the binary number system and learned that each position represents a certain power of 2. This is similar to the way each position in our decimal number system represents a certain power of 10. This is illustrated below.

Decimal	10^3	10^2	10^1	10^0
	1,000's	100's	10s	1s
Binary	2^3	2^2	2^1	2^0
	8s	4s	2s	1s

If we look at a decimal number like $9{,}999_{10}$ (the subscript 10 means that we are using a number in base 10), it not only tells us about a quantity of items, such as apples, but also tells us about possible combinations.

The number 9,999 is a *four*-digit number. Using the 10 different decimal digits from 0 through 9, and using no more than *four* digits at a time, there would be 9,999 + 1, or 10,000, possible numbers you could create. (You add the 1 because the number 0000 or simply 0 must also be included.) This can also be calculated as $10^4 = 10{,}000$.

If you were interested in giving unique addresses to 10,000 homes on the same street (quite a long street), it would be possible to do so by using only four digits. The first house would have the address 0, and then you would just continue numbering up to 9,999.

EXAMPLE 15-1

Using only three digits, how many unique addresses could you give to homes on a single street (a decimal number)?

SOLUTION

Since $10^3 = 1{,}000$, this is the number of unique addresses that are possible.

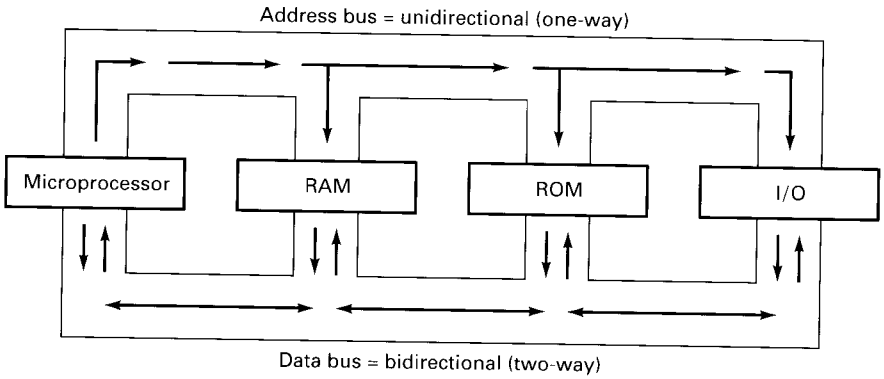


Fig. 15-3 Data bus and address bus.

Now, let's try the same problem in binary: 1111_2 is a binary number. (The subscript 2 tells us we are using base 2 or binary numbers.) The size of this number is shown below.

Binary	2^3	2^2	2^1	2^0
	8s	4s	2s	1s
	1	1	1	1

We have one 8. We have one 4. We have one 2. And we have one 1. That is, we have an 8, a 4, a 2, and a 1. If we add this up, we get

$$8 + 4 + 2 + 1 = 15$$

The number 1111_2 is the same as 15_{10} (decimal 15). This means that using only 4 binary digits or bits, there are a total of 15 + 1, or 16 unique numbers possible. This can be calculated by using $2^4 = 16$.

If you wanted to give unique binary addresses to 16 houses on the same street (not such a long street), it would be possible to do so with only 4 bits. The first house would be 0000 or simply 0, the next would be 0001, the next 0010, and so on up to 1111.

EXAMPLE 15-2

Using 12 binary digits, how many unique house addresses would be possible?

SOLUTION

$2^{12} = 4{,}096$ unique addresses

This is essentially what is necessary in the matter of addressing memory locations. The highest number that exists in binary using only 4 bits is 1111_2 (15_{10}). That means that if we had only four address lines—that is, an address bus with only four lines—we would be able to have only a maximum of 16_{10} different addresses. (0000 counts as one address.) Obviously, this is not enough. Look at Fig. 15-4. This illustrates the number of unique addresses possible with different numbers of address lines.

As can be seen in Fig. 15-4, if we decide to use only eight address lines, since we are studying 8-bit chips, we then limit ourselves to 256 memory locations. (Add the values of the first eight positions starting from the far right + 1.) This is not nearly enough. Most 8-bit chips use 2 bytes for addressing purposes, which then allows 65,536 different memory locations. (One byte is 8 bits; 2 bytes is 16 bits, which then allows 2^{16} combinations.) This is often adequate. If not, there are ways to increase this number by using a method known as bank switching.

EXAMPLE 15-3

How many memory locations could be addressed by a 10-line address bus?

SOLUTION

$2^{10} = 1{,}024$ memory locations can be addressed.

15-2 MICROPROCESSOR ARCHITECTURE

We now need to look more closely at the actual microprocessor, which is the "brain" of our computer. First, we will study those features which most microprocessors have in common. Then we will look at each of the microprocessor families and study their specific features.

Accumulator

One of the most often used parts of a microprocessor is the accumulator. The *accumulator* is a storage place or register which often has its contents altered in some way. For example, we can add the contents of the accumulator to the contents of a memory location. Usually the result of an operation is also placed in the accumulator. This action is illustrated in Fig. 15-5.

The microprocessor can take the contents of the accumulator and the data coming in, perform some operation on the two, and place the result back in the accumulator. There are times when no data is coming in but some operation is being performed on the contents of the accumulator only. For example, the microprocessor might find the 1's complement of the contents of the accumulator and place the result in the accumulator in place of the original number.

Some microprocessors have only one accumulator; others have more than one.

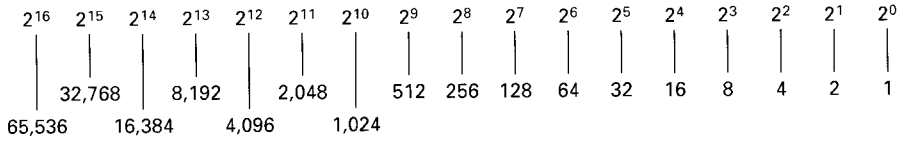


Fig. 15-4 Powers of 2. Also the number of memory addresses available with varying numbers of address lines.

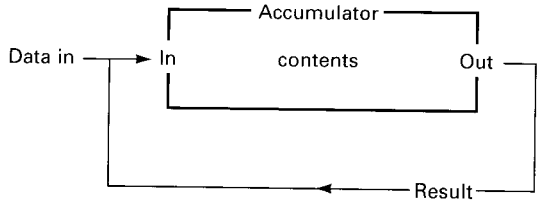


Fig. 15-5 Accumulator operation.

General-Purpose Registers

General-purpose registers are similar to the accumulator. In fact, the accumulator is a special type of register. *General-purpose registers* are temporary storage locations. They differ from the accumulator in that operations involving two pieces of data are usually not performed in them with the result going back into the register itself, as in the case of the accumulator. The microprocessor will often alter the contents of a register, however. Figure 15-6 shows the operation of a general-purpose register.

One might wonder why a microprocessor needs general-purpose registers when it has RAM to temporarily store information. The answer is speed. Data in registers can be accessed and moved much more quickly than data in RAM.

Program Counter/Instruction Pointer

We mentioned earlier that instructions are stored in memory. Considering the fact that there can be tens of thousands, hundreds of thousands, or even millions of memory locations, it's obvious that the microprocessor must keep track of the location from which it will be getting its next instruction. This is the job of the program counter.

The *program counter* is a very special register whose only job is to keep track of the location of the next instruction which the microprocessor will use. Figure 15-7 illustrates its operation.

The program counter "points" to the address of the next instruction to be retrieved and used by the microprocessor.

The act of "getting" an instruction is usually referred to as *fetching* the instruction. The period of time needed for this is often called the *fetch cycle*.

Index Registers

Another type of register is the index register. In the same way that the index of a book helps a person locate information, the index register can be used to help locate data. The *index register* is normally used as an aid in

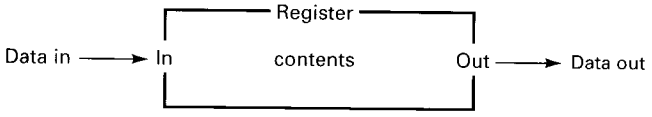


Fig. 15-6 General-purpose register operation.

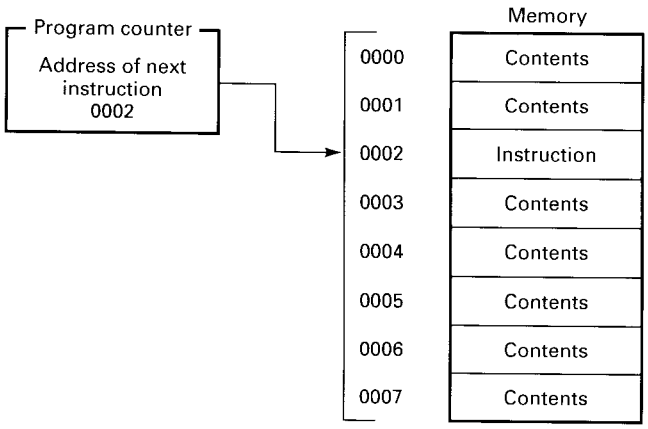


Fig. 15-7 Program counter operation.

accessing data in tables stored in memory. The index register(s) can be incremented (increased by 1) or decremented (decreased by 1) but normally does not have other arithmetic or logical capabilities.

We will look at the index register(s) more completely in later chapters.

Status Register

The *status register,* sometimes called the *condition code register,* or *flag register,* is a special register which keeps track of certain facts about the outcome of arithmetic, logical, and other operations. This register makes it possible for the microprocessor to be able to test for certain conditions and then to perform alternate functions based on those conditions. This is done through the use of *flags.*

We will now take an overall look at flags. Don't be concerned if these next few paragraphs are not completely clear at this point. They can serve as a refresher for those who may have had some experience with microprocessors in the past. And for those who are new to this subject, reading about them now will at least give you some idea of what flags are and how they are used. These concepts will be covered again in greater detail as they arise in later chapters.

The status register is divided into individual bits which have their own unique functions. Each bit is called a *flag.* Each flag keeps track of, or "flags," us concerning certain conditions. Not every operation or instruction affects every flag. Some instructions affect many flags, and some don't affect any at all. Figure 15-8 shows a model of a typical status register.

When referring to flags, the following logic is used. If some condition has come to be, or is true, the flag uses a 1 to say, "Yes, this is true or has happened." If that condition has not occurred, the flag uses a 0 to say, "No, this is not true or has not happened." Causing a flag to become 1 is called *setting* a flag. Causing a flag to become 0 is called *clearing* a flag.

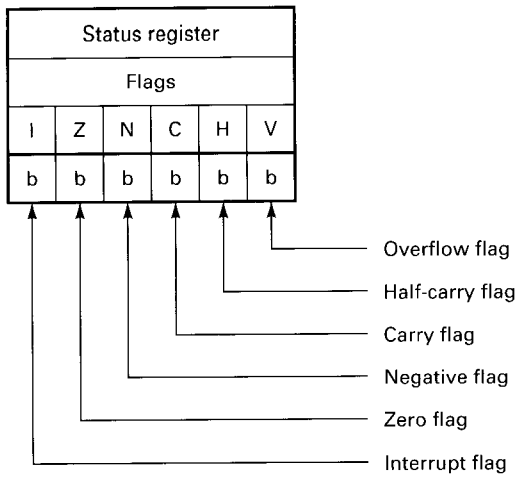


Fig. 15-8 Model of a typical status register. (b's represent bits.)

The zero flag keeps track of whether the last operation which affects this flag produced an answer of zero. This flag is set or 1 if a zero result has been produced and is cleared or 0 if a nonzero result has been produced.

The negative flag tells us if the last operation which affects this flag produced a negative number. When 8-bit signed binary numbers are used, if bit 7 (the eighth bit) of the number is 1, then the number is negative and the N flag will be set; if bit 7 of the number is 0, then the number is positive and the N flag will be cleared or 0. (This negative flag is sometimes called a sign flag and is indicated with an "S.")

The carry flag tells us if the last operation which affects this flag produced a carry from bit 7 (in 8-bit systems) of the accumulator (bit 7 is the left-most or most significant bit) into the carry bit. The carry flag also tells us if, during subtraction, a borrow into bit 7 was needed. How a borrow is indicated depends on which microprocessor is being used. See Fig. 15-9.

The half-carry flag tells us if the last operation which affects this flag was an arithmetic operation which produced a carry from bit 3 to bit 4. This feature is primarily used with BCD (binary-coded-demical) numbers.

The overflow flag tells us if the last operation which affects this flag caused a result that is outside the range of signed binary numbers for the word size being used at the time. In the case of 8-bit microprocessors, this is +127 or −128. If this range is exceeded, the overflow flag is set (1) to warn the programmer.

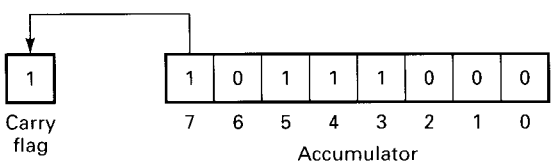


Fig. 15-9 A "carry" from bit 7 into the carry flag.

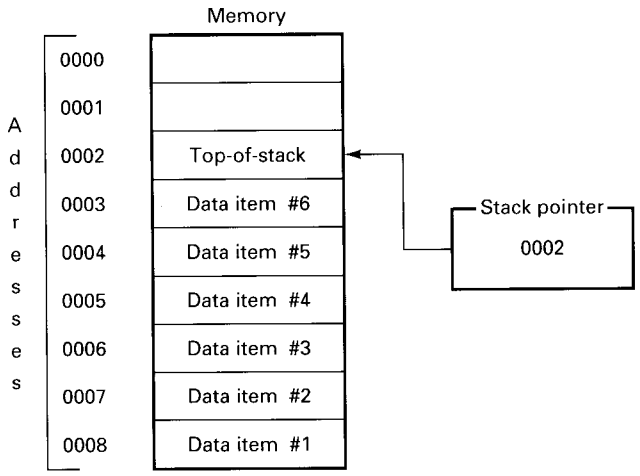


Fig. 15-10 Typical stack and stack pointer.

The interrupt (interrupt mask, interrupt flag, interrupt enable bit) prevents maskable interrupts from occurring when it is set and allows them when cleared.

Stack and Stack Pointer

The stack is a special place in memory. The *stack* is most often used to store certain critical pieces of data during subroutines and interrupts. You'll learn more about these later, but let's look at the structure of a stack at this time. Refer to Fig. 15-10.

The structure of the stack is a *first-in-last-out* (FILO) type of structure. Unlike main memory, where you can access any data item in any order, the stack is designed so that you can access only the top of the stack. If you want to place data in the stack, it must go on top; if you wish to remove data from the stack, it must be on top before it can be removed.

Let's see how the situation in Fig. 15-10 has come to be. To do that, refer to Fig. 15-11. Data item #1 is the first item we wish to place on the stack.

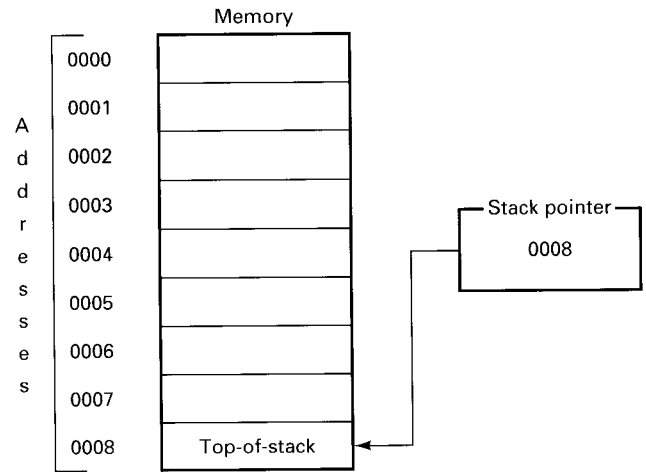


Fig. 15-11 Typical stack and stack pointer.

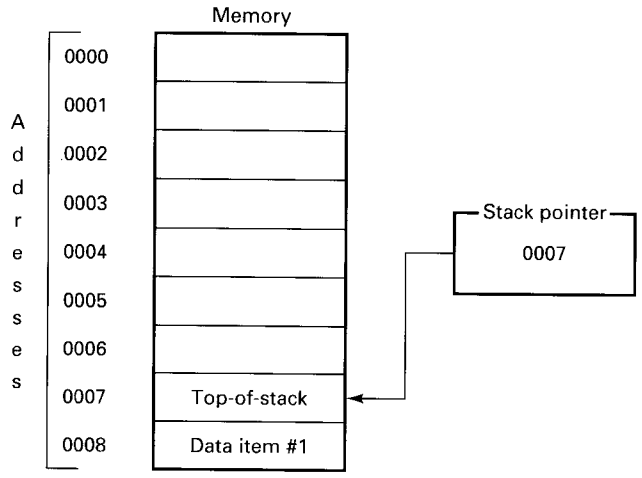


Fig. 15-12 Typical stack and stack pointer.

At this time the stack pointer is "pointing" to memory location 0008; therefore data item #1 will be placed in the stack at that memory location. The act of putting a piece of data in the stack is called *pushing* data onto the stack. It is as though the data is being pushed in from the top. Now look at Fig. 15-12.

We have pushed data item #1 onto the stack and the stack pointer has been decremented or decreased by one, which means that it is now pointing to memory location 0007. Location 0007 is the top-of-the-stack now. Now let's push data item #2 onto the stack. The stack will appear as it does in Fig. 15-13.

When data item #2 was *pushed* onto the stack, it went into the location the stack pointer was pointing to—which was 0007. The stack pointer was then decremented to 0006. This process will be repeated until it appears as it did in Fig. 15-10.

At some point we will need this data in the stack, so we will remove it from the top-of-the-stack. This is called *popping* or *pulling* the data from the stack. We simply reverse the whole process. As each data item is removed, the stack pointer will drop, which in this case means that it will point to the next-greater memory address.

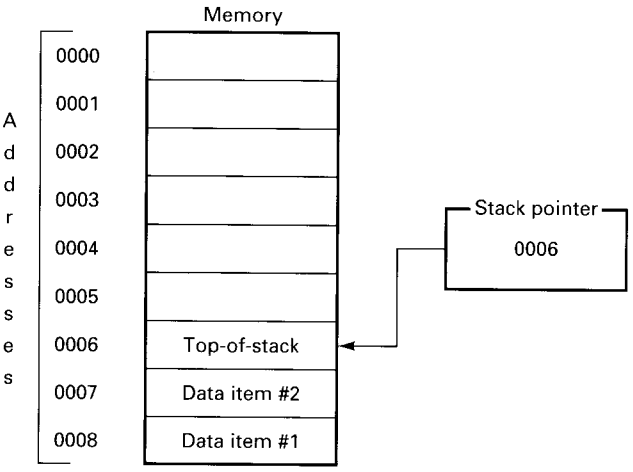


Fig. 15-13 Typical stack and stack pointer.

EXAMPLE 15-4

Refer to Fig. 15-13. If we *pull* data item #2 from the stack, will the stack pointer increment or decrement? What hexadecimal value will appear in the stack pointer?

SOLUTION

The stack pointer will be incremented as data item #2 is pulled from the stack. The hexadecimal value 0007 will appear in the stack pointer. In fact, the stack will appear as it did in Fig. 15-12.

Width of Registers

All registers have a maximum capacity. That is, they will only hold a certain number of bits. The width is generally 8, 16, or 32 bits.

8-Bit Registers

An 8-bit register is one that is *8 bits wide*. This means it can hold 1 byte as shown in Fig. 15-14. Most computers and trainers you will be using will not display an 8-bit register in binary. Instead, they will have a hexademical display. If you have forgotten how to convert binary to hexadecimal and hexadecimal to binary, review that section in Chap. 1.

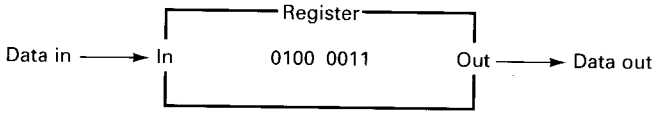


Fig. 15-14 Eight-bit register model.

It is often useful to separate the 8 bits into two groups of 4. The left group of 4 is called the upper nibble, and the right group of 4 is called the lower nibble. This is illustrated in Fig. 15-15.

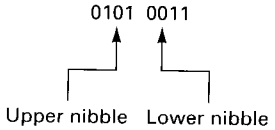


Fig. 15-15 Upper- and lower-nibble positions.

EXAMPLE 15-5

If a register contained the binary number shown in Fig. 15-16, what would appear in the hexadecimal display for that register?

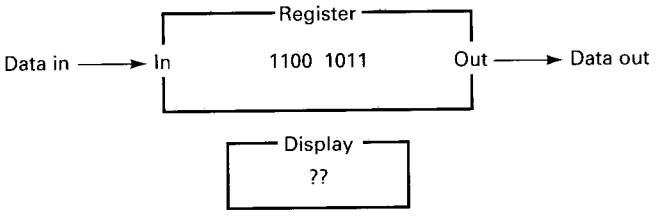


Fig. 15-16 Example A.

SOLUTION

The upper nibble, 1100, is the same as the hexadecimal digit C. The lower nibble, 1011, is the same as the hexadecimal digit B. Therefore, the hexadecimal display will show CB.

16-Bit Registers

A 16-bit register of course is *16 bits wide*. This is illustrated in Fig. 15-17. As you can see, the 16 bits are again separated into groups of 4. Each nibble, or group of 4, will be represented in the display as 1 hexadecimal digit.

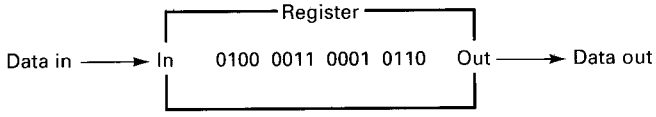


Fig. 15-17 Sixteen-bit register model.

EXAMPLE 15-6

In Fig. 15-18, what are the binary contents of the register when the display is as shown?

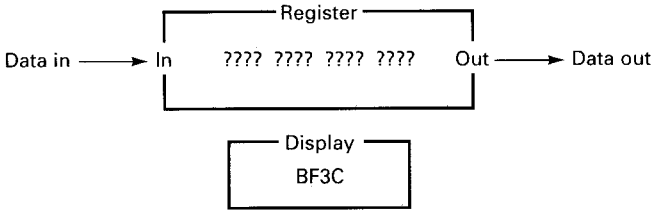


Fig. 15-18 Example B.

SOLUTION

The far left digit (also called the *most significant digit*), the B, has a binary equivalent of 1011. The F would be 1111. The 3 would be 0011. And the hexadecimal digit C would be represented by 1100 in binary. Putting the four nibbles together produces 1011 1111 0011 1100, which constitutes the binary contents of this register.

Specific Microprocessor Families

The rest of this chapter is divided into sections, each of which is devoted to one particular microprocessor family. Go to the section which discusses the microprocessor family you are using.

15-3 6502 FAMILY

Let's look at specific characteristics of the 6502 family of microprocessors.

Accumulator

The accumulator in the 6502 family of microprocessors is 8 bits wide. The 6502 has only one accumulator, unlike others which have more than one. Figure 15-19 shows what it looks like.

General-Purpose Registers

The 6502 has no general-purpose registers. The functions they perform must be accomplished in the 6502 by using the accumulator, index registers, and memory.

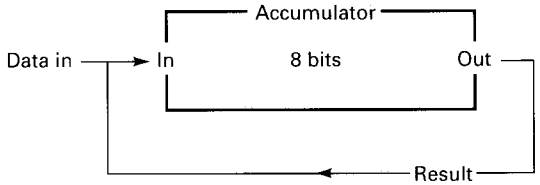


Fig. 15-19 6502 accumulator model.

Program Counter

The 6502 family program counter, as shown in Fig. 15-20, is 16 bits wide and is divided into an upper half which we have labeled PC_H (*program counter high*) and a lower half which we have labeled PC_L (*program counter low*).

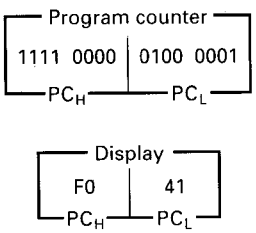


Fig. 15-20 Sixteen-bit 6502 program counter and display.

Most of the time it operates as one 16-bit counter, but there are times, particularly when subroutines are involved, when the division into 2 bytes is necessary. The display for the program counter will appear as four hexadecimal digits as shown in the figure.

Index Registers

The 6502 has two index registers. They are each 8 bits wide. One is the X index register, and the other is the Y index register.

Status Register

The 6502 status register contains 8 bits, but only 7 are actually used. The layout of this register is shown in Fig. 15-21.

The 6502 has several flags in addition to those mentioned in the New Concepts section of this chapter.

The break flag keeps track of what are called "software interrupts." When the programmer puts a BRK (**BR**ea**K**) instruction in the program telling the microprocessor to stop, the programmer "interrupts" the program in progress. If this occurs, the break flag is set.

The decimal mode flag, when set, tells the microprocessor to assume that any numbers which it is instructed to add or subtract are BCD (binary-coded decimal) numbers instead of regular binary numbers. This will result in a BCD answer.

During addition the carry flag in the 6502 is used as described in the New Concepts section of this chapter. When a carry goes out from bit 7 of the accumulator, it goes into the carry bit. During subtraction, however, if a borrow is needed from the carry bit by bit 7, then the carry flag is cleared (0). If you think of it as though the 1 that was needed during the borrow actually came from the carry bit, it will be easier to remember. Please note that other microprocessors handle this situation with the carry flag and subtraction in just the opposite manner.

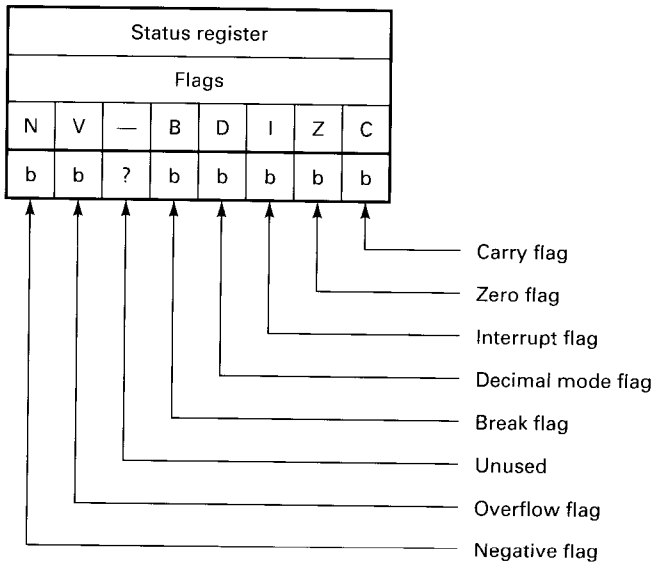


Fig. 15-21 6502 family status register. (b's represent bits.)

Stack and Stack Pointer

The 6502 has a stack with a maximum size of 256 bytes or memory locations. The stack pointer is 8 bits wide with a 9th bit that is always set. Figure 15-22 shows it in more detail.

The greatest memory address (lowest position) which can be designated as the top-of-the-stack is 1 1111 1111_2, which is $01FF_{16}$. Each time another number is pushed onto the stack, the top-of-the-stack rises, which means that the stack pointer is decremented by one (since smaller-numbered memory addresses are toward the top). The smallest address which can be designated as the top-of-the-stack is 1 0000 0000_2, which is 0100_{16}. This is not always the top; it is simply the highest position (smallest memory address) at which the top can exist.

We will look at the stack and its uses in later chapters.

Complete Model

Let's look at a complete model of the 6502 family of microprocessors. Refer to Fig. 15-23.

In our model we do not show the binary numbers that are actually in each register or location but, rather, the hexadecimal numbers which appear in the display of microprocessor trainers. The exception is the status register, in which both binary and hexadecimal are shown. The small h's and b's represent the data that would be in each register or memory location. Each "h" stands for one hexadecimal digit or nibble—which is to say, 4 bits. Each "b" stands for 1 bit. When we use this model in later chapters, we will place actual values in place of the h's and b's.

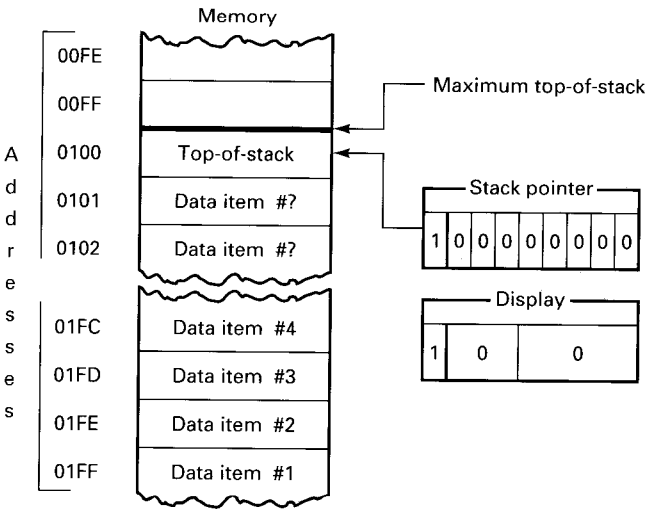


Fig. 15-22 6502 family stack and stack pointer.

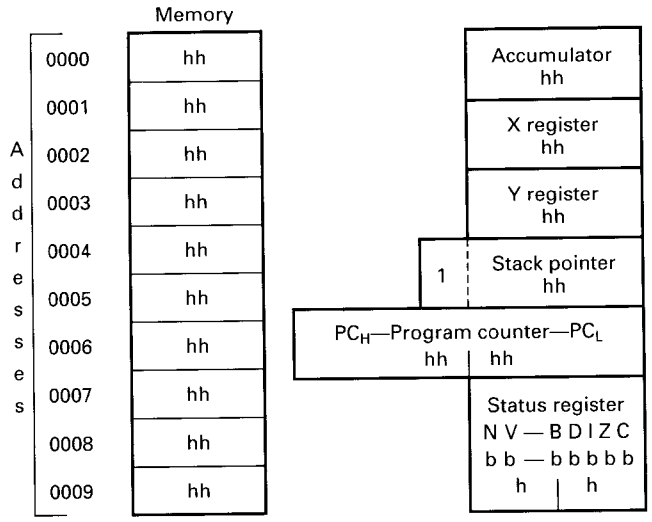


Fig. 15-23 Complete 6502 programming model.

15-4 6800/6808 FAMILY

This section covers the Motorola 6800 and 6808 microprocessors. The 6809 is an enhanced version of the 6800/6808, but most of this section can be applied to the 6809 as well. The 6809 has all of the features of the 6800 plus additional ones. The 6800 and 6808 are the primary subjects of this section, but some differences in the 6809 are mentioned.

Accumulators

The 6800/6808 microprocessors have two 8-bit accumulators. Each has the same capabilities; that is, neither is a general-purpose register. Both are true accumulators. (General-purpose registers do not have all of the features of an accumulator.) Figure 15-24 illustrates their functions.

The operation of these accumulators is the same as that described in the New Concepts section of this chapter. One note of interest concerning the 6809. It has the same 8-bit accumulators; however, it has the additional ability to treat the two as a single 16-bit accumulator known as *accumulator D* and has special instructions for such operation.

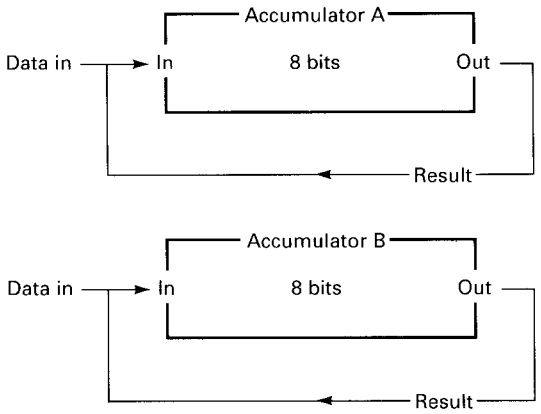


Fig. 15-24 Models of the 6800/6808 family accumulators.

General-Purpose Registers

The 6800/6808, like the 6502, has no general-purpose registers. Their functions must be performed by using the accumulators, index register, and memory.

Program Counter

The 6800, 6808, and 6809 each have 16-bit program counters. The 6800 family program counter, as shown in Fig. 15-25, is 16 bits wide but is divided into an upper half which we have labeled PC_H (for *program counter high*) and a lower half we have labeled PC_L (for *program counter low*). Most of the time it operates as one 16-bit counter, but there are times, particularly when subroutines are involved, when the division into 2 bytes is necessary. The display for the program counter will appear as four hexadecimal digits as shown in the figure.

Index Register

The 6800 and 6808 microprocessors each have one 16-bit index register called the *X index register*. The 6809 has two 16-bit registers named the *X index register* and the *Y index register*.

The 6800 family's index registers operate as described in the New Concepts section of this chapter and will be discussed in more detail in later chapters.

Condition Code Register

The 6800/6808 condition code register (called a *status register* in other microprocessors), which is shown in Fig. 15-26, is composed of 6 flags or bits in an 8-bit register. The 2 most significant bits are not used and are always set (1).

In the 6809 the 2 bits that are unused on the 6800/6808 have functions and are called the *E flag* and the *F flag*. They will not be discussed in this text.

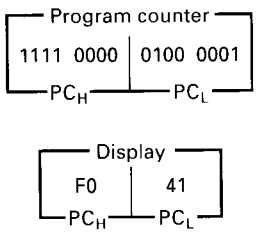


Fig. 15-25 Sixteen-bit 6800 family program counter and display.

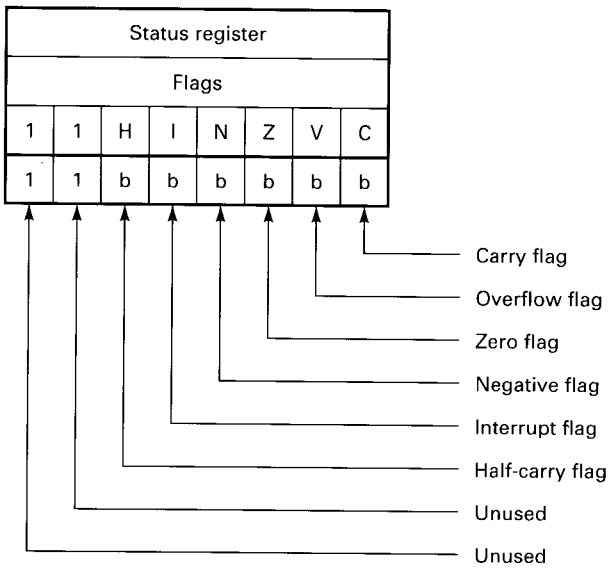


Fig. 15-26 6800/6808 status register. (b's represent bits.)

The carry flag in the 6800 family is set (1) when either a *carry or borrow* from bit 7 occurs. (The 6502 by contrast sets the flag for a carry but clears it for a borrow.)

All flags used in the 6800/6808 operate as described in the New Concepts section of this chapter.

Stack and Stack Pointer

The 6800/6808 has a 16-bit stack pointer which uses RAM for the stack itself. It operates as described in the New Concepts section of this chapter.

The 6809 has a second stack called the *user stack* which operates in a fashion similar to the first stack, which is called the *hardware stack*. The user stack is not used for interrupts and subroutines but is left free for the programmer to use.

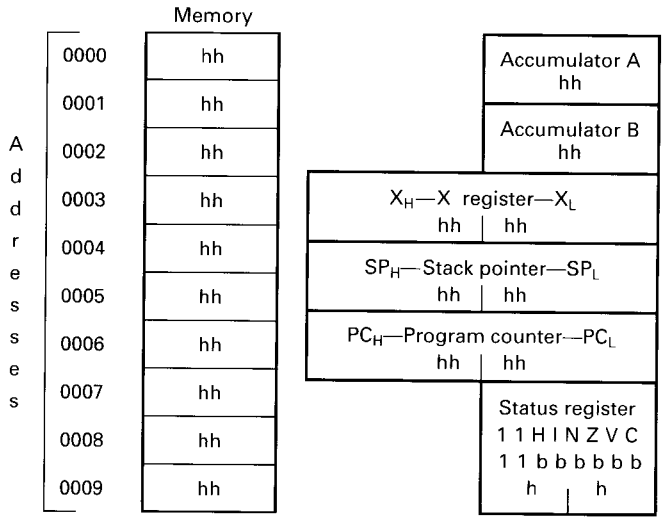


Fig. 15-27 Complete 6800/6808 programming model.

Complete Model

Let's look at a complete model of the 6800 family of microprocessors. Refer to Fig. 15-27.

In our model we do not show the binary numbers that are actually in each register or location but, rather, the hexadecimal numbers which appear in the display of microprocessor trainers. The exception is the status register in which both binary and hexadecimal are shown. The small h's and b's represent the data that would be in each register or memory location. Each "h" stands for one hexadecimal digit or nibble—which is to say, 4 bits. Each "b" stands for 1 bit. When we use this model in later chapters, we will place actual values in place of the h's and b's.

15-5 8080/8085/Z80 FAMILY

This section deals with the 8080 and 8085 microprocessors from Intel and the Z80 microprocessor manufactured by the Zilog Corp.

The 8080 and 8085 are nearly identical, the 8085 being a slightly improved version of the 8080. Except for two instructions, the instruction sets for the two chips are identical.

The Z80 is a considerably enhanced version of the 8080. It understands all the instructions of the 8080 and many more. It has all the registers of the 8080 plus a number of additional registers. We will cover only those aspects of the Z80 that are found in the 8080 and 8085 at this time.

Accumulator

The 8080/8085/Z80 chips have one 8-bit accumulator. It operates as described in the New Concepts section of this chapter. Its operation is shown in Fig. 15-28. The Z80 also has a second *alternate* accumulator.

General-Purpose Registers

The 8080/8085/Z80 chips have an abundance of general-purpose registers. These registers are arranged in pairs. Notice the arrangement of one of these pairs in Fig. 15-29.

In this figure, 8 bits of data can go into and out of either register B or C. Or, 16 bits can go into and out of the pair, at which point they act as one 16-bit register.

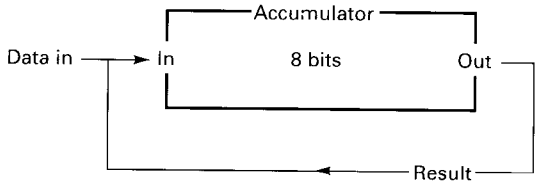


Fig. 15-28 8080/8085/Z80 accumulator model.

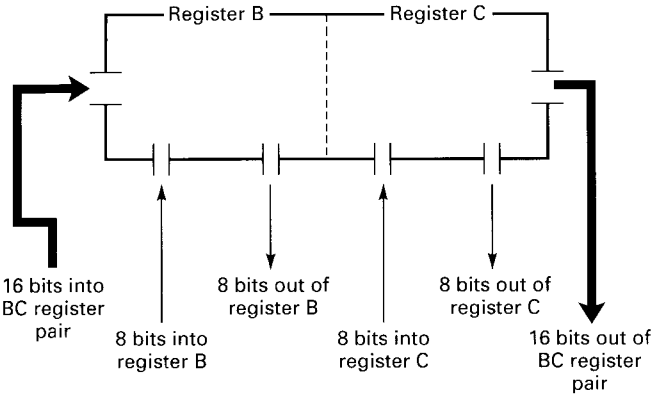


Fig. 15-29 Model of 8080/8085/Z80 general-purpose registers.

There are three sets of these general-purpose register pairs. They are the BC pair, the DE pair, and the HL pair. The letters B, C, D, and E are assigned to stand for each register. The letters H and L stand for high and low. The HL register pair is usually used for a different purpose than the other two pairs. We will discuss that purpose more in a later chapter.

Each of these registers has a mate, or "alternate," register in the Z80.

Program Counter

The 8080/8085/Z80 chips each have a 16-bit program counter which operates as described in the New Concepts section of this chapter. This program counter, as is the case with the 6502 family and the 6800 family, is divided into two halves for some operations. The upper byte or 8 bits are called the PC_H (for *program counter high*), and the lower byte is called the PC_L (for *program counter low*). See Fig. 15-30.

Most of the time the program counter operates as one 16-bit counter, but there are times, particularly when subroutines are involved, when division into 2 bytes is necessary. The display for the program counter will appear as four hexadecimal digits as shown in the figure.

Index Register(s)

The 8080 and 8085 have no index registers. The Z80 has two—an X index register and a Y index register. The index registers in the Z80 are each 16 bits wide.

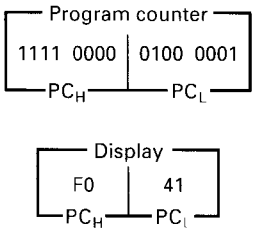


Fig. 15-30 Sixteen-bit 8080/8085/Z80 program counter and display.

Status Register

The status register in the 8080 and 8085 contains five flags in an 8-bit register. See Fig. 15-31.

The parity flag involves a topic which has not been discussed yet. *Parity* refers to the number of 1s in a binary number. *Even parity* exists when there is an even number of 1s. For example, the binary number 0110 000 has even parity because it has two 1s, and 2 is an even number. *Odd parity* exists when there is an odd number of 1s. For example, the binary number 0111 0000 has odd parity because there are three 1s, and 3 is an odd number. It is sometimes useful to keep track of parity for error-checking routines and in data communications. If the parity is even, the parity flag becomes set (1); if parity is odd, it clears (0).

The Z80 has the same five flags as the 8080 and 8085, and in the same positions, plus one additional flag. See Fig. 15-32.

The half-carry flag in the Z80 has exactly the same function as the auxiliary carry in the 8085/8080.

The parity flag in the Z80 has a dual role—that of parity checking and that of warning the programmer of 2's-complement overflow. Also, the Z80 has a negative or sign flag (the 8080 and 8085 do not have one) which operates as described in the New Concepts section of this chapter.

Stack and Stack Pointer

The 8080, 8085, and Z80 each have a stack with a 16-bit stack pointer which operates as described in the New Concepts section of this chapter.

Complete Model

Let's look at a complete model of the 8080/8085/Z80 family of microprocessors. Refer to Fig. 15-33 at this time.

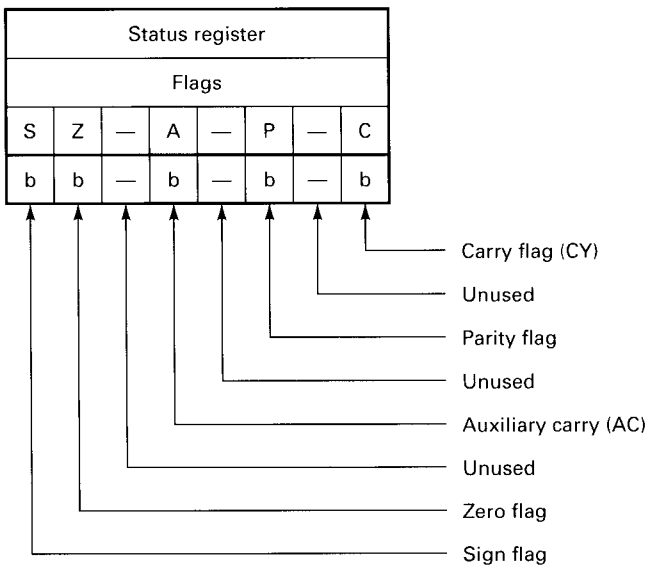


Fig. 15-31 8080/8085 status register. (b's represent bits.)

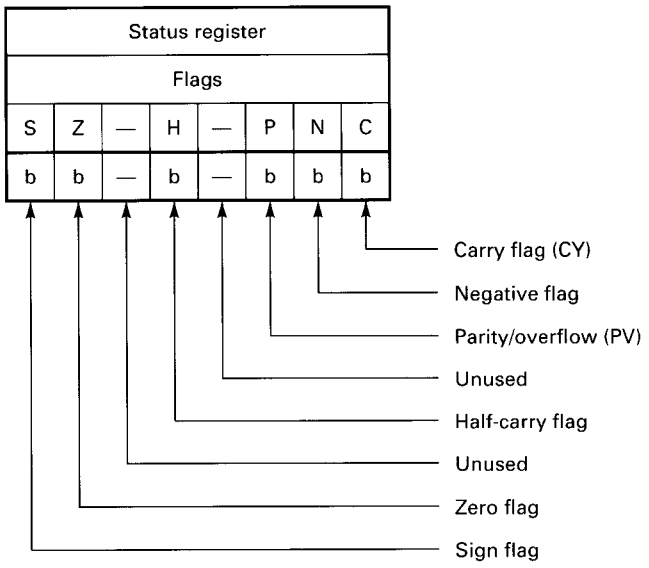


Fig. 15-32 Z80 status register. (b's represent bits.)

A couple of points concerning differences between the 8080/8085 and the Z80 should be noted. Figure 15-33 is a model of the 8080/8085. The Z80 has an additional set of alternate registers and two index registers which are not shown in the model. The status register in the Z80 has an additional flag called the *negative flag*. And the auxiliary carry flag in the 8080/8085 is usually called the half-carry flag in the Z80.

In our model we will not show the binary numbers that are actually in each register or location but rather the hexadecimal numbers which appear in the display of microprocessor trainers. The exception is the status register in which both binary and hexadecimal are shown. The small

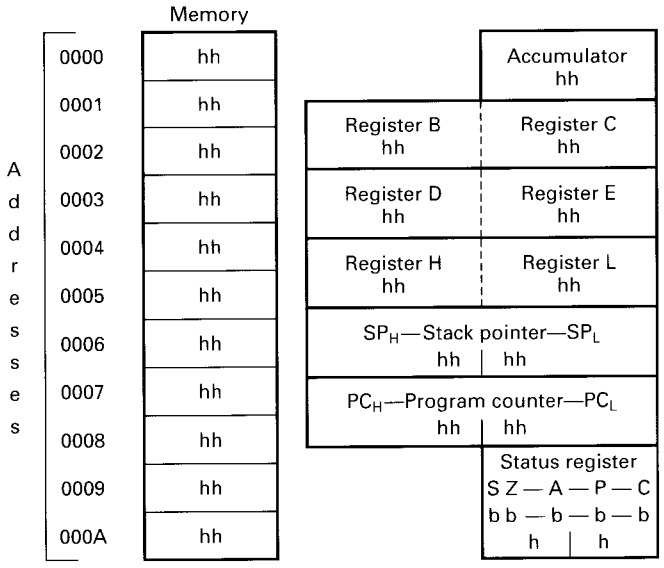


Fig. 15-33 Complete 8080/8085 and Z80 (8080 subset) programming model.

h's and b's represent the data that would be in each register or memory location. Each "h" stands for one hexademical digit or nibble, which is to say 4 bits. Each "b" stands for 1 bit. When we use this model in later chapters, we will place actual values in place of the h's and b's.

There is one point of significant difference between the 8080/8085/Z80 family and the 6502 or 6800 family. In the case of the 6502 and 6800 microprocessors, the registers and accumulators are completely independent of one another. In the 8080/8085/Z80 family, the six registers, namely B and C, D and E, and H and L, can operate as six independent 8-bit registers or as three 16-bit register pairs. This allows single operations to be performed on 16-bit data words.

15-6 8086/8088 FAMILY

In this section we will examine the 8086 and 8088 microprocessors from Intel. The 8088 is the microprocessor used in the popular IBM PCs, XTs, and compatibles. The 80286 used in ATs and the 80386 can also be used with this text.

Since the 8086/8088 chips are the successors of the 8085, they are similar to it but have many additional registers and capabilities.

Accumulator(s)

The 8086/8088 has an accumulator (shown in Fig. 15-34) which is 16 bits wide and is called AX. The upper 8 bits is called AH (*accumulator high*), and the lower 8 bits is called AL (*accumulator low*).

General-Purpose Registers

The 8086/8088 has three 16-bit or six 8-bit general-purpose registers (besides the accumulator). These are shown in Fig. 15-34 and are called the BX, CX, and DX registers. Each can be divided into an upper and lower byte called BH, BL, CH, CL, DH, and DL, respectively. Also note in the figure that A stands for accumulator, B for base, C

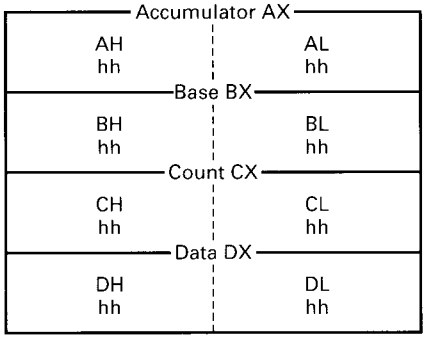


Fig. 15-34 8086/8088 accumulator and general-purpose registers.

for count, and D for data. This can help you remember the main functions of each register.

Instruction Pointer

Instead of a program counter, the 8086/8088 has an *instruction pointer* which does what the program counter does in the 8-bit microprocessors. The instruction pointer is 16 bits wide.

Index Registers

The 8086/8088 has several index registers and pointers including the base pointer, source index, and destination index. All are 16 bits wide. These are not used alone, as with the 8-bit chips, but are used in combination with registers called *segment registers*. Figure 15-35 is a model of the 8086/8088 pointers and index registers.

Stack and Stack Pointer

The 8086/8088 stack is a standard memory stack (as are all the 8-bit microprocessors we've covered). The 8086/8088, however, can have a *very* large stack, up to 64K (65,536 bytes). The location of the top-of-the-stack is calculated by using both the stack pointer and the stack segment.

Status Register

The status register containing the 8086/8088 flags is 16 bits wide, although not all 16 bits are used. This register, shown in Fig. 15-36, has a lower byte (8 bits) which is exactly the same as the 8-bit 8085 microprocessor's status register. It has the same flags in the same positions. The upper byte has four flags which the 8085 does not have.

The first flag is the *trap* flag, which controls a single-step mode of operation.

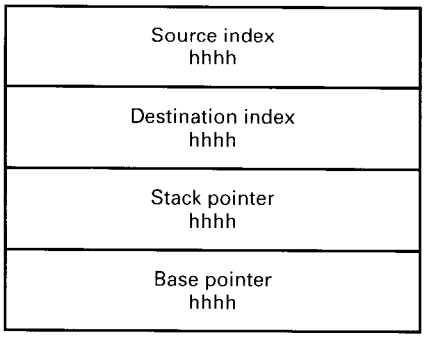


Fig. 15-35 8086/8088 index registers and pointers.

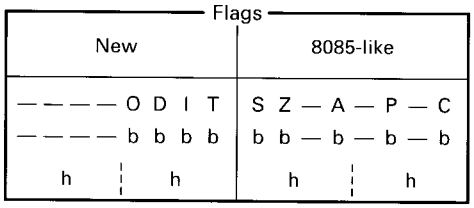


Fig. 15-36 8086/8088 flag register. (b's represent bits; h's represent hex digits.)

The *interrupt enable* flag controls the interrupt request pin on the microprocessor chip.

The *direction* flag controls whether the source index and destination index increment or decrement during string operations.

Finally, the *overflow* flag alerts the programmer to the existence of an arithmetic overflow when set. This is a condition in which the legal range for signed binary numbers of a particular word size has been exceeded.

Segment Registers

The 8086/8088 microprocessor has several other registers which do not exist on the 8-bit chips. These are the *segment registers*. We'll explain very briefly how they are used at this time.

All the pointers and index registers in the 8086/8088 chips are 16 bits wide; 2^{16} is 65,536 (64K) bytes. The address bus, however, is 20 bits wide. We can have memory locations extending up to 2^{20} or 1,048,576 (1 mega-) bytes. None of the pointers, including the instruction pointer, would be able to point to this wide of a range of addresses. To solve this problem, segment registers are used. Their contents are combined with the contents of the various pointers and index registers to form an address which is 20 bits wide. Exactly how this is done will be explained in a later chapter.

Complete Model

Figure 15-37 is a complete model of the 8086/8088 microprocessors.

In the model shown in Fig. 15-37 the placeholders for each binary digit are not shown. Rather, the hexadecimal digits that would be seen on a computer or trainer are indicated. The exception is the status register, in which both binary and hexadecimal placeholders are shown. The small h's and b's represent the data that would be in each register or memory location. Each "h" stands for one hexadecimal digit or nibble, which is 4 bits. Each "b" stands for 1 bit. When we use this model in later chapters, we will place actual values in place of the h's and b's.

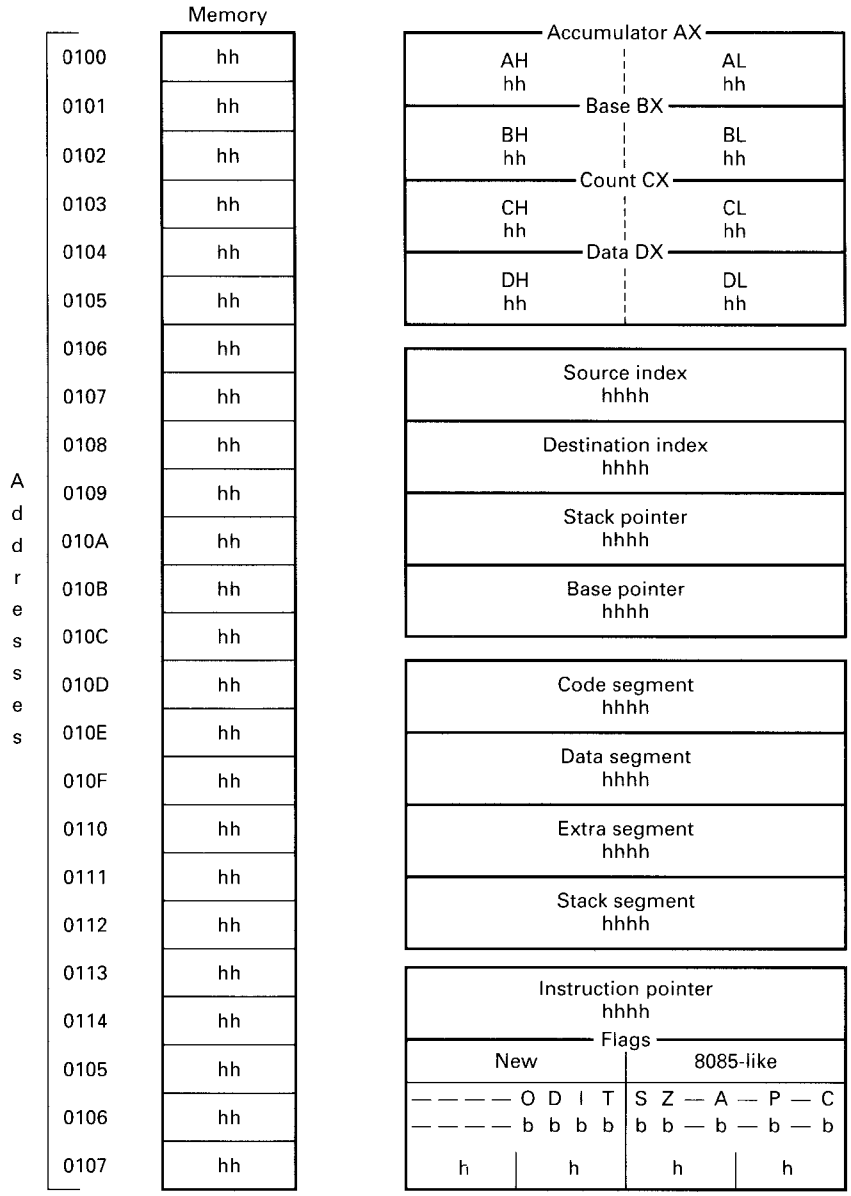


Fig. 15-37 Complete 8086/8088 microprocessor programming model.

GLOSSARY

accumulator A register in a microprocessor which can not only store a byte or word of data but can have its contents operated on, with the result of that operation going back into the accumulator, replacing the previous value.

address Binary numbers which are assigned to consecutive memory locations. Specific memory locations are accessed through their addresses.

address bus A set of conductors upon which binary addresses travel to memory chips.

data bus A set of conductors which carry binary data to and from the microprocessor, memory, and I/O devices.

fetching The act of going to memory to get an instruction which is to be decoded and executed.

flag One of the bits in the status register. (See status register.)

general-purpose registers Locations which can store a byte or word of data similar to RAM but which are inside the microprocessor itself. Certain operations can usually be performed on the contents of registers.

index register A register which can be incremented and decremented and whose primary function is to point to data (often used in tables).

program counter A special-purpose register whose purpose is to keep track of the next instruction to be fetched from memory.

RAM An acronym for random-access memory. This type of memory loses its data when power is removed.

ROM An acronym for read-only memory. This type of memory does not lose its data when power is removed.

stack An area (usually in RAM) which holds vital information during subroutines and interrupts. It can also be used by the programmer as a LIFO (last-in-first-out) data storage area.

status register (condition code register) A special register whose individual bits show the status of certain conditions or the results of certain operations.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. _____ is the type of memory which can have its contents changed thousands of times per second.
2. (*RAM*) The _____ of a memory location is similar to the address of your home and the _____ inside the memory location is similar to the beds, chairs, dishes, and so on, in your home.
3. (*address, data*) The _____ of a memory location is necessary to specify which of many locations is to be written to or read from.
4. (*address*) The address bus is usually _____ (unidirectional, bidirectional).
5. (*unidirectional*) The data bus is usually _____.
6. (*bidirectional*) Each different bit position in binary numbers represents a certain power of _____.
7. (*2*) Probably the most used register in a microprocessor is the _____.
8. (*accumulator*) A register which helps microprocessors to work with tables of data is the _____.
9. (*index register*) When a flag has a _____ in it, this indicates that the condition which the flag tests has *not* come true.
10. (*0*) When a flag has a _____ in it, this indicates that the condition which the flag tests *has* come true. (*1*)

PROBLEMS

General

15-1. By what means is one memory location differentiated from another?

15-2. Using decimal numbers, how many combinations can be represented by using only five digits?

15-3. Using binary numbers, how many combinations can be represented by using only 20 bits?

15-4. If we had $20{,}000_{10}$ memory locations, what would be the least number of address lines needed to describe each location? (Hint: Change 20,000 to binary or hex and determine the number of bits needed.)

15-5. What register can have its contents altered in the greatest variety of ways and is the real "workhorse" in the microprocessor?

15-6. In simplest terms, what are general-purpose registers?

15-7. What advantage do registers have over RAM?

15-8. What has the sole purpose of keeping track of the next instruction to be fetched?

15-9. In what register are the flags located?

15-10. What has happened if the zero flag has a 1 in it?

15-11. Which flag will be set if a carry from bit 7 of the accumulator is produced during an arithmetic operation?

15-12. Which flag is primarily used with binary-coded decimal numbers?

15-13. When normal stack instructions are used, can a number be pulled from somewhere in the middle of the stack?

15-14. What is taking a number from the top of the stack called?

15-15. If an 8-bit register contained the binary number 1101 1110, what hexadecimal number would appear as the display or readout for that register?

15-16. What are the binary contents of a register whose hexadecimal display reads 2A?

15-17. What would the hexadecimal display of a 16-bit register with $1100\ 0101\ 1000\ 0001_2$ as its contents read?

6502 Family

15-18. How many general-purpose registers does the 6502 have?

15-19. How wide are the index registers in the 6502?

15-20. What flag, when set, tells the 6502 to assume that binary-coded decimal (BCD) numbers are being used?

15-21. What is the maximum size of the 6502 stack?

6800 Family

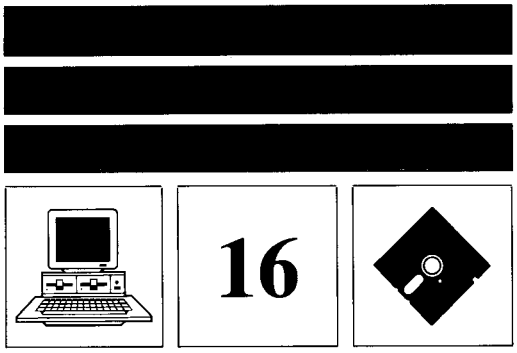
15-22. How many accumulators does the 6800 have?
15-23. How wide is the 6800 program counter?
15-24. How many memory locations can the 6800 program counter reference or point to?
15-25. What are the 2 most significant bits in the 6800 condition code register used for?

8080/8085/Z80 Family

15-26. How many 8-bit general-purpose registers does the 8085 have?
15-27. How many index registers does the 8085 have?
15-28. How wide is the 8085 stack pointer?

8086/8088 Family

15-29. Describe how the 8088 accumulator is labeled and arranged.
15-30. How many 8-bit general-purpose registers does the 8088 have?
15-31. In the 8088 what has the same function as the program counter in the 8-bit microprocessors?
15-32. What 8-bit microprocessor is the lower byte of the 8088 flag register patterned after?
15-33. How large can the 8088 stack be?



DATA TRANSFER INSTRUCTIONS

New Concepts

So far we've been able to get an overview of computers, computer architecture, microprocessor architecture, programming, languages, flowcharting, and hardware. Now let's take a closer look at some of these areas.

Instruction Sets

The commands that microprocessors understand are called *instructions*, and the complete "vocabulary" of each chip is called its *instruction set*.

We will be studying the 6502, 6800/6808, 8080/8085/Z80, and 8086/8088 microprocessor families and each family's instruction set. We will deviate from this plan in two respects.

Rather than study the entire Z80 instruction set, we will study only those instructions which are common to the 8080 and 8085. (The Z80 has many instructions which neither the 8080 nor the 8085 understands. However, the Z80 understands *all* the instructions of the other two chips with only two exceptions.)

Also, we will not study the entire 8086/8088 instruction set but will omit the loop and string instructions since they have no counterpart in the 8-bit microprocessors.

Organization of This Text

You may find it helpful to know how this programming portion of the text was developed.

We are ready to begin learning about microprocessor instructions. The instructions being discussed in each chapter, the sequence in which the instructions are being presented, the sequence of the chapters, and the instruction categories have all been carefully planned.

As mentioned before, this text centers around the most popular general-purpose 8-bit microprocessors (the 6502 family, the 6800/6808 family, and the 8080/8085/Z80 family) and the 16-bit 8086/8088 family. During the preparation of this text, the instruction sets of each of these microprocessors were carefully analyzed, and it was found that each chip's instructions fell into natural groups. After each instruction was placed into its natural category, it was possible to identify those categories which were common to every microprocessor family. Those instructions which did not fall naturally into one of these common groups were placed in the group in which they most nearly fit. In short, a consistent and uniform method of classifying instructions was applied to each microprocessor family. In the tables section of this book (Part 4) you will find the complete instruction set of each chip broken down into these groups or categories.

Next, the chapters were planned to reflect these same groups. Thus, rather than trying to make the microprocessors fit the scheme of this text, the text was designed around the natural characteristics of the microprocessors. Each chip's instruction set has been broken down into the same categories as the others, and the appendixes and chapters treat each chip family equally.

Organization within Each Chapter

Most chapters start with a New Concepts section (which is where we are now). The discussion here is general—that is, it can be applied equally well to all microprocessor families and does not focus on any one family. Then, after this general discussion, the remainder of the chapter is divided into family-specific sections.

For example, if you are using the 6808 microprocessor, you would read the New Concepts section and then go immediately to the 6800/6808 Family section. There, specific information will be given to help you apply the principles discussed in the New Concepts section to the 6800/6808 microprocessors.

Now let's look at our first instruction category.

16-1 CPU CONTROL INSTRUCTIONS

The easiest instruction to learn about is an instruction which does nothing, and surprisingly, there is such an instruction. Let's look at it.

The No Operation Instruction

The *no operation* instruction does exactly that: It does nothing. This is a waste of time, and wasting time is what this instruction does best.

A microprocessor is quite fast, in some situations too fast. We can give it a certain number of these no operation instructions to stall it until a certain amount of time passes.

The no operation instruction has another use—that of filling space in the program. When writing programs, we must sometimes insert additional instructions into the middle of a program to alter the way it works or to fix a problem.

If you use one of the simpler monitors (instead of an assembler, or a monitor with an insert feature), it may not have a feature which will let you insert instructions into the middle of a program you have entered. When this happens, you must rewrite every part of the program beginning from the point at which the inserted instruction must be placed, to the end. By adding some no operation instructions at various locations in the program when you first write it, some spaces will have been created where new instructions can go. The new instructions can simply take the place of the no operation instructions.

The Halt Instruction

Called *wait*, *halt*, or *break* (depending on the microprocessor), this instruction has the obvious purpose of stopping the microprocessor. There is no *go* instruction—we'll see how that is done shortly—but there must be a way to stop the program. In some microprocessor families this is not the *only* function of this instruction, but this is all we need to be concerned with at this time.

16-2 DATA TRANSFER INSTRUCTIONS

This category of instructions has the job of transferring or moving data from one place to another. Before studying these instructions, we need to consider a basic concept.

Physical Places

Sometimes people think that when we speak of moving data from one place to another within a microprocessor, we are referring only to the "net effect" of the transfer, and that nothing actually moved.

If this were so, the operation of a microprocessor would resemble what happens when you go to the bank and transfer money from your savings account into your checking account. Though the net effect of the transfer is to decrease the amount of money in the savings account and to increase the amount in the checking account, you know that no one in the bank actually picked up the money in the savings account and placed it in another spot where your checking account was. It all happened "on paper."

This is not the case with microprocessors. The accumulators, general-purpose registers, program counter, index registers, and so on, are all real places. While it is true that tiny numbers don't move around inside the chip, the voltages representing these numbers can be made to appear in various places, so for all practical purposes the numbers themselves move.

If you experience difficulty visualizing what a program does, it may help to write down the contents of each register and/or memory location. Then as each location is changed by the program, change it on your paper. We will use this technique in many of the figures.

Where Data Is Transferred

Data is moved between registers or between registers and memory. The number of possible combinations depends on the microprocessor and how many registers it has. Figure 16-1 shows some typical possibilities.

How Data Is Transferred

Different microprocessor instruction sets use different terms to represent the act of transferring data. "Move," "load," "store," and "transfer" are all common terms.

Though we will use the term "moving," and even though thinking of it in that way will work as you become proficient, in the beginning a distinction has to be made. *When a*

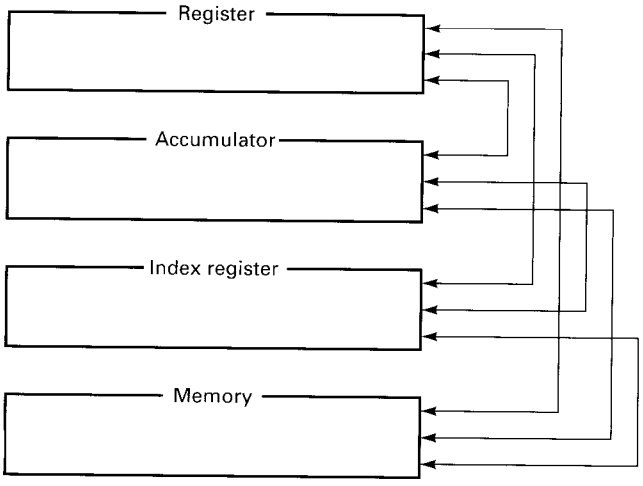


Fig. 16-1 Some of the possible data transfer combinations.

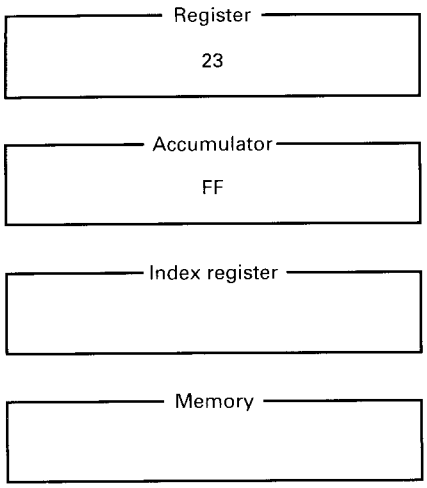


Fig. 16-2 An example of a transfer instruction.

move, load, transfer, or store instruction is executed, a duplicate of the data is actually being placed in the target register or destination.

If you were to move your car from one parking spot to another in a parking lot, your car would no longer be in its original place. This is true moving. This is *not* what happens in a microprocessor. If, however, you photocopy an important document, place the copy in a filing cabinet, and keep the original, you have not actually moved the *document* to the filing cabinet, but rather you have moved a *copy* of the document. This *is* what happens in a microprocessor.

An Example of a Transfer Instruction

Look at Fig. 16-2.

Suppose we wanted to transfer the FF in the accumulator to the register, which now contains 23. We would write a program which instructs the microprocessor to transfer the contents of the accumulator to the register. The result of this action is shown in Fig. 16-3.

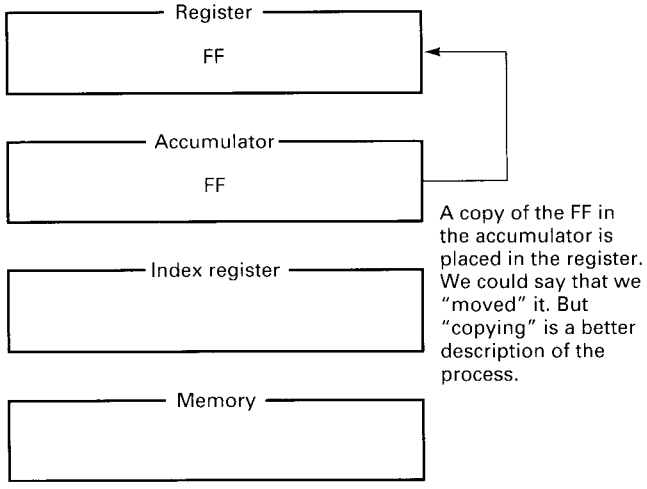


Fig. 16-3 An example of a transfer instruction.

Notice that the original FF in the accumulator is still there. We simply made a copy of it and placed the copy in the register. The original contents of the register are lost.

Now go to the section of this chapter which discusses your particular microprocessor family.

Specific Microprocessor Families

16-3 6502 FAMILY

Let's see how the ideas which were introduced in the New Concepts section apply to the 6502 microprocessor family.

CPU Control Instructions

The 6502 family has a *no operation* instruction which uses the mnemonic NOP. Refer to the Expanded Table of 6502 Instructions Listed by Category in Part 4 of this text.

Look at the NOP instruction, which is the very first instruction in this table. In the third column, the Boolean/Arithmetic Operation column, we see that this instruction does "nothing," just as we said it would. Also notice the hexadecimal number under the Op (op code) column, in this case EA. This is the actual hexadecimal code for NOP. Don't worry about the rest of the NOP information at this time.

The 6502 family doesn't have an actual halt instruction, but the instruction which serves its purpose is the **BReaK** instruction. Refer to the table again. Notice that the **BReaK** instruction uses the mnemonic BRK and has an op code of 00.

Data Transfer Instructions

Look under the **BReaK** instruction and you will see the beginning of the Data Transfer Instructions section of the table. In this section you will see a list of all of the different types of data transfer instructions available in the 6502 family. (To those with previous microprocessor experience: You may notice that we have excluded transfer instructions involving the stack. This is intentional. They have been included in the Stack Instructions category.)

Direction of Data Transfer

Let's look at the data transfer instructions more closely. The first instruction listed is the **LoaD** Accumulator instruction. The boldfaced letters show where the LDA mnemonic came from. The third column shows the Boolean/Arithmetic Operation. This is a concise and graphic way to state exactly what this instruction does. It shows M, which stands for memory, moving toward A, which stands for the accu-

mulator. To put it another way, the contents of a certain memory location are being transferred into the accumulator.

Recall from the New Concepts section that moving or transferring is actually more like making a copy of what's in a particular location and placing the copy in the destination.

Referring to the Expanded Table of 6502 Instructions, notice that the second and third instructions, LDX and LDY, are similar to the LDA. The difference is that they copy the contents of a particular memory location and place it in either the X register or the Y register instead of the accumulator.

It may help to have a mental picture of our programming model of the 6502, shown in Fig. 16-4, as we discuss these instructions.

We have talked about moving or copying the contents of some particular memory location to the accumulator, the X register, or the Y register. Now let's consider doing the reverse.

Look at the fourth, fifth, and sixth instructions in the table. They are STA, STX, and STY, that is, *Store the contents of the accumulator in a memory location, store the contents of the X register in a memory location*, and *store the contents of the Y register in a memory location*, respectively. The *store* instructions are just the reverse of the *load* instructions. (See the Boolean/Arithmetic Operation column.)

Now, continue referring to both the table and Fig. 16-4. The next two instructions (TAX and TXA) allow you to transfer the contents of the accumulator and X register between each other. The last two instructions (TAY and TYA) allow you to transfer the contents of the accumulator and the Y register between each other.

Op Codes

Does your computer or microprocessor trainer understand the words "load accumulator"? No. Does it understand the mnemonic LDA? No, but if you are using an assembler, the assembler translates the mnemonics into binary numbers which it does understand. (If you use a hexadecimal keypad or type in hex numbers, you do not have an assembler.) The point here is that the microprocessor inside your computer does not understand English words like "load" or mnemonics like LDA.

If you are using an assembler, the assembler program is translating the mnemonics, which the microprocessor does not understand, into something it does understand. What does the microprocessor understand? Binary numbers. In our case we will enter them as their equivalent hexadecimal value and let the monitor or assembler translate that into binary. For our purposes, at least at this point, we'll say that the microprocessor understands hexadecimal. (The monitor is part of the *firmware* built into your microprocessor trainer.)

Refer to the Expanded Table of 6502 Instructions. If we wanted to tell the microprocessor to load the accumulator from memory (the first data transfer instruction, LDA) the microprocessor chip would actually need the hex code in the seventh column over, the Op code column (Op for short). We would place the hex number A9, AD, A5, A1, B1, B5, BD, or B9, depending on which variation of the instruction we wanted to use, in the computer's memory as the first instruction to execute.

Let's try another example. What if you wanted to have the microprocessor store the contents of the Y register in memory? What would be the hex number the microprocessor would need to understand what you wanted to do? (You should have said either 8C or 84 or 94 from the STY instruction.)

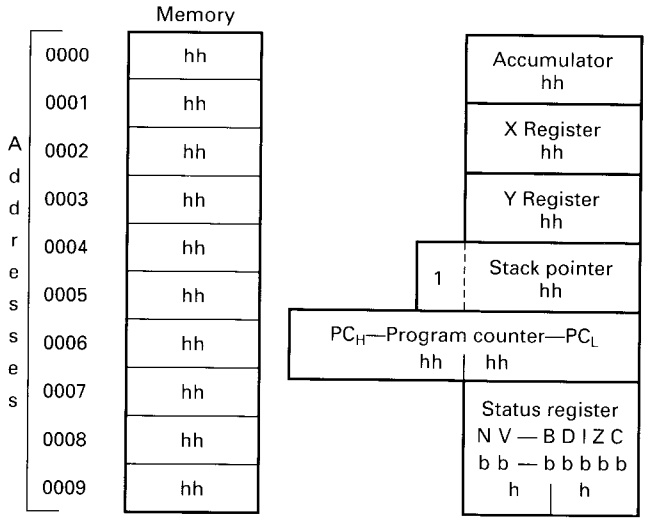


Fig. 16-4 Complete 6502 programming model.

Sample 6502 Program

Program Objective

Let's create a program which will

1. Place the number 11 in the accumulator.
2. Stop.

Creating the Program

Refer to the Data Transfer Instructions section of the Expanded Table. Do you see an instruction which could be used to place a number in the accumulator? Look in the Boolean/Arithmetic Operation column. You need an instruction which has an arrow pointing to the accumulator. There are three such instructions—LDA, TXA, and TYA. Since we don't want to involve the X register or Y register, LDA will be our choice.

The next step is to determine which of the LDA instructions to use. There are eight. The key to this decision is in the Address Mode column. The LDA instruction which has Immediate in the address column is the one we want.

Addr	Obj	Assembler	Comment
0000	A9	LDA #$11	Load the accumulator with the number (11) immediately following the LDA# op code (A9)
0001	11		
0002	00	BRK	Halt

Fig. 16-5 Sample program. (*Note:* The addresses should be an area where user programs can be placed. If 0000 is not such a place on your system, then you will need to change these addresses.)

Immediate addressing tells the microprocessor that the data it needs will be coming *immediately* after the op code. We will learn more about addressing modes in the next chapter.

Finally, you want the program to stop. The instruction which does this is in the CPU Control Instructions section of the Expanded Table. The BRK instruction is the obvious choice.

Entering the Program

The completed program is shown in Fig. 16-5. We'll see how to enter it into your microprocessor first by using an assembler and then without an assembler.

Note that the column labeled Obj contains the actual 6502 op codes while the Assembler column contains the mnemonic and data in a format similar to that which is used by an assembler.

Refer to the LDA instruction in the Expanded Table. To the right of the word Immediate, you see LDA #$dd. This is in the Assembler Notation column and describes how many assemblers require that you type this instruction. With eight different **LoaD** Accumulator instructions, the assembler must know which one you want. The format of the information after the LDA is how the various forms of the command are differentiated. The # means that the data to be used is coming *immediately* after the command itself. The $ indicates that it is a hexadecimal number. The dd simply stands for two hexadecimal digits of data. (Each d stands for one nibble or 4 bits.)

It is important to remember that we are talking about a typical assembler format; however, there is no absolute standard that must be followed. Refer to the manual which came with your assembler, or ask your instructor for information about your assembler's format.

We are going to enter this program into memory starting at location 0000 (hexadecimal). If the trainer you are using does not allow programs to be placed in these memory locations, refer to your manual and substitute addresses which are valid for your trainer or computer for those shown in Fig. 16-5.

If you are using an assembler, please enter the program at this time. It will look similar to what is shown in Fig. 16-6.

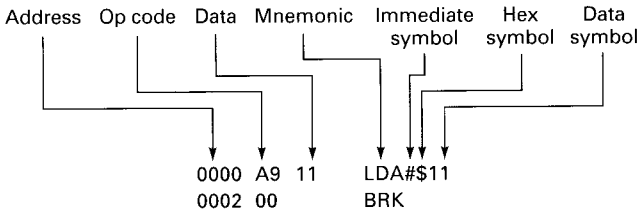


Fig. 16-6 Disassembly of the sample program. (The mnemonic and the data to the right of the mnemonic are all that's typed in during assembly.)

Now place 0s in the accumulator, the X register, and the Y register so that you will know what numbers are in each register before the program is run. Refer to Fig. 16-7 to see what memory and the registers should look like.

If you are *not* using an assembler, you must look up the op codes by hand in the Expanded Table. This is called *hand-assembly*. Let's go through the necessary steps for hand-assembly.

To the right of the LDA #$dd, in the op code (op for short) column you will see the hexadecimal number A9. This is the 6502 op code, which stands for *Load the accumulator with the number immediately following this op code*. Set your trainer so that the memory address at which the next instruction will be loaded is someplace within the area allowed for user programs. We chose 0000, but you

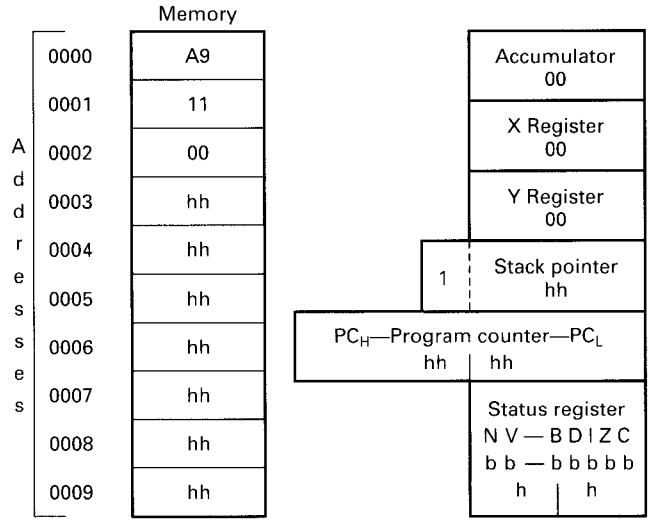


Fig. 16-7 6502 sample program.

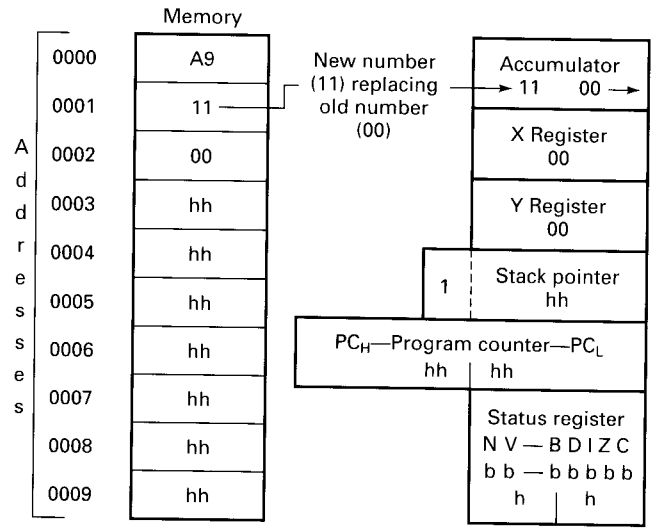


Fig. 16-8 6502 sample program.

may need to use another location. Enter the number A9 into the first available memory location. Since this was a *load accumulator immediate* instruction, the microprocessor will expect the next address, which *immediately* follows the op code, to contain the number which is to be placed in the accumulator. Therefore enter 11 next. In the third address enter 00, which is the op code for the BRK instruction.

Enter 0s into the accumulator, X register, and Y register at this time so that you will know the condition of these registers before the program is run.

If you check your registers and memory, you should see what is shown in Fig. 16-7 (although you may have placed the program at a different memory location). The h's and b's represent hex and binary digits which we are not concerned with at this time.

Running the Program

Let's use Fig. 16-8 during our analysis of program operation. The first op code is A9, which means *Load the accumulator with the contents of the next memory location*, or more properly, *Place a copy of the contents of the next memory location in the accumulator*. As you see, the number 11 is replacing 00 in the accumulator. The program then continues to the next instruction op code, 00, which stands for BREAK, and stops.

Checking the Results of Program (Analysis)

After running the program, you should have 00 in the X register, 00 in the Y register, and 11 in the accumulator. The program does what we designed it to do.

Here's one for you to try.

EXAMPLE 16-1

Manually place 00s in the accumulator, the X register, and the Y register. Next, write a program which will

1. Place the hex number EE in the accumulator.
2. Transfer (copy) the contents of the accumulator (A) into the X register (X).
3. Transfer (copy) the contents of the accumulator (A) into the Y register (Y).
4. Stop.

SOLUTION

Figure 16-9 shows the completed program. Figure 16-10 shows memory and the registers and what happens during program execution.

16-4 6800/6808 FAMILY

Let's see how the ideas which were introduced in the New Concepts section apply to the 6800/6808 microprocessor family.

CPU Control Instructions

The 6800/6808 family has a *no operation* instruction which uses the mnemonic NOP. Refer to the Expanded Table of 6800 Instructions Listed by Category in Part 4 of this text.

In the third column, called the Boolean/Arithmetic Operation column, we see that this instruction does "nothing," just as we said it would. Also notice the hexadecimal number under the op (op code) column, in this case 01. This is the actual hex code for NOP.

The 6800 family doesn't have an actual halt instruction, but the instruction which serves its purpose is the **WAIt** for Interrupt instruction. (Bold type and capital letters

Addr	Obj	Assembler	Comment
0000	A9	LDA #$EE	Copy the hex number EE into the accumulator (A)
0001	EE		
0002	AA	TAX	Transfer the contents of A into X
0003	A8	TAY	Transfer the contents of A into Y
0004	00	BRK	Stop

Fig. 16-9 Example 16-1 program listing.

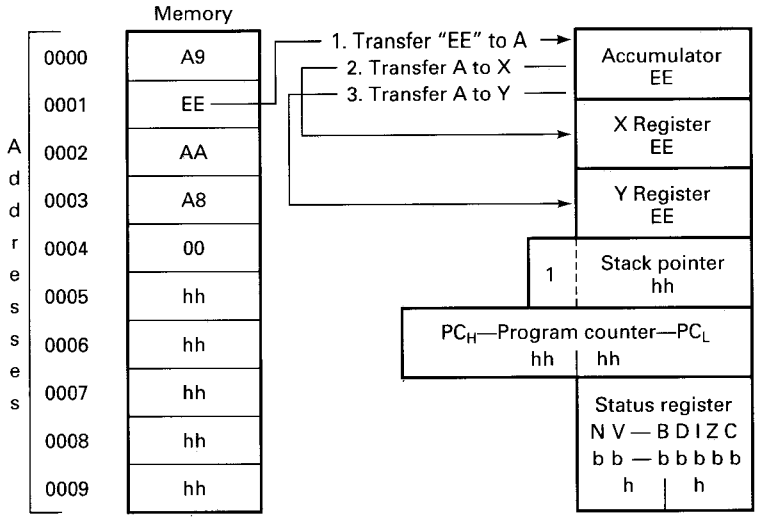


Fig. 16-10 Example 16-1 program analysis.

identify the mnemonic.) Refer to the Expanded Table of 6800 Instructions. Notice that the wait for interrupt instruction uses the mnemonic WAI and has an op code of 3E.

Data Transfer Instructions

Look in the Expanded Table at the next entry underneath the WAI instruction. This is the first entry in the Data Transfer Instructions section, which is a list of all of the different types of data transfer instructions available in the 6800/6808 family. (To those with previous microprocessor experience: You may notice that we have excluded transfer instructions involving the stack. This is intentional. They have been included in the Stack Instructions category.)

Direction of Data Transfer

Let's look at this Data Transfer section a little more closely. The first instruction listed is the **LoaD Accumulator A** instruction. The boldfaced letters show where the LDAA mnemonic came from. The third column shows the Boolean/Arithmetic Operation. This is a concise and graphic way to state exactly what this instruction does. It shows M, which stands for memory, moving toward A, which stands for the accumulator. To put it another way, the contents of a certain memory location are being transferred into the accumulator.

Recall from the New Concepts section that moving or transferring is actually more like making a copy of what's in a particular location and placing the copy in the destination.

Referring to the table, notice that the second (**LoaD Accumulator B**) and seventh (**LoaD X** register) instructions are similar to the first (LDAA). The difference is that they copy the contents of a particular memory location and place it either in accumulator B or in the X register instead of accumulator A.

It may help to have a mental picture of our programming model of the 6800, shown in Fig. 16-11, as we discuss these instructions.

We have talked about moving or copying the contents of some particular memory location to accumulator A, accumulator B, or the X register. Now let's consider doing the reverse.

Look at the third, fourth, and eighth instructions in the Expanded Table. They are STAA, STAB, and STX, which is to say, *store the contents of accumulator A in a memory location, store the contents of accumulator B in a memory location*, and *store the contents of the X register in a memory location*, respectively. The STORE instructions are just the reverse of the LOAD instructions. (Note the Boolean/Arithmetic Operation column.)

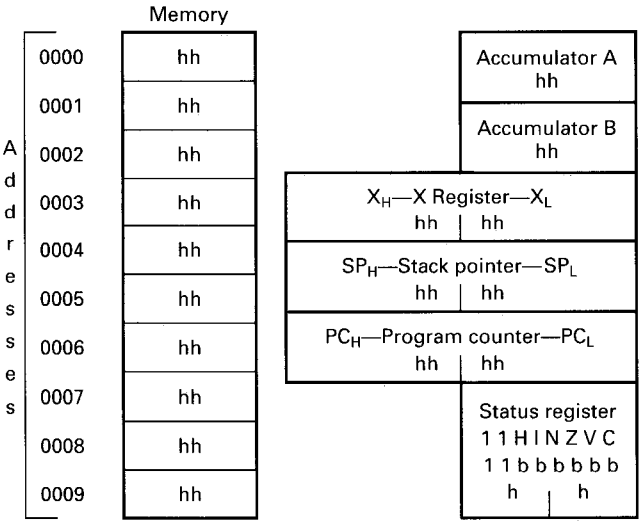


Fig. 16-11 Complete 6800/6808 programming model.

Continue referring to both the Expanded Table and Fig. 16-11. Instructions 5 and 6 in the Expanded Table (TAB and TBA) allow you to transfer the contents of accumulator A and accumulator B between each other.

The last three instructions (CLR, CLRA, and CLRB) simply transfer or place the number zero in accumulator A or B or in a memory location.

Op Codes

Does your computer or microprocessor trainer understand the words "load accumulator A"? No. Does it understand the mnemonic LDAA? If you are using an assembler, the assembler translates the mnemonic into binary numbers, which it does understand. (If you can type the mnemonic LDAA into your computer or trainer, you have an assembler. If instead you must use a hexadecimal keypad or type in hex numbers, you do not have an assembler.) The point here is that the microprocessor inside your computer does not understand English words like "load" or mnemonics like LDAA.

If you are using an assembler, the assembler program is translating the mnemonics, which the microprocessor does not understand, into something it does understand. What does the microprocessor understand? Binary numbers. In our case we will enter them as their equivalent hexadecimal value and let the monitor or assembler translate that into binary. For our purposes, at least at this point, we'll say that the microprocessor understands hexadecimal. (The monitor is part of the *firmware* built into your microprocessor trainer.)

Look again at the Expanded Table. If we wanted to tell the microprocessor to load the accumulator from memory (the first data transfer instruction, LDAA), the microprocessor chip would actually need the hex code in the seventh column over, the op code column (op for short). We would place the hex number 86, 96, A6, or B6 (depending on which variation of the instruction we wanted to use) in the computer's memory as the first instruction to execute. (We'll talk more about these variations later.)

Let's look at another example. What if you wanted to have the microprocessor store the contents of the X register in memory? What would be the hex number the microprocessor would need to understand what you wanted to do? You should have said either DF or EF or FF from the STX instruction.

Sample 6800/6808 Program

Program Objective

Let's create a program which will

1. Place the number 11 in the accumulator.
2. Stop.

Creating the Program

Refer to the Data Transfer Instructions section of the Expanded Table. Do you see an instruction which could be used to place a number in the accumulator? Look in the Boolean/Arithmetic Operation column. You need an instruction which has an arrow pointing to the accumulator. There are three such instructions—LDAA, TBA, and CLRA. Since we don't want to involve accumulator B, and since we don't want to clear accumulator A, LDAA will be our choice.

The next step is to determine which LDAA instruction to use. There are four. The key to this decision is in the Address Mode column. The LDAA instruction which has Immediate in the address column is the one we want. *Immediate addressing* tells the microprocessor that the data it needs will be coming *immediately* after the op code. We will learn more about addressing modes in the next chapter.

Finally, you want the program to stop. The instruction which does this is in the CPU Control Instructions section of the Expanded Table. The WAI instruction is the correct choice.

Entering the Program

The completed program is shown in Fig. 16-12. We'll see how to enter it into your microprocessor first by using an assembler and then without an assembler.

Note that the column labeled Obj contains the actual 6800 op codes, and the Assembler column contains the mnemonic and data in a format similar to that used by an assembler.

Addr	Obj	Assembler	Comment
0000	86	LDAA #$11	Load the accumulator with the number (11) immediately following the LDAA# op code (86)
0001	11		
0002	3E	WAI	Halt

Fig. 16-12 Sample program. (*Note:* The addresses should be an area where user programs can be placed. If 0000 is not such a place on your system, then you will need to change these addresses.)

Refer to the LDAA instruction in the Expanded Table. To the right of the word Immediate you see LDAA #$dd. This is in the Assembler Notation column and describes how many assemblers require that you type this instruction. With four different **LoaD** Accumulator **A** instructions, the assembler must know which one you want. The format of the information after the LDAA is how the different forms of the command are differentiated. The # means that the data to be used is coming *immediately* after the command itself. The $ indicates that it is a hexadecimal number. The dd simply stands for two hexadecimal digits of data. (Each d stands for one nibble or 4 bits.)

It is important to remember that we are talking about a typical assembler format; however, there is no absolute standard that must be followed. Refer to the manual which came with your assembler or ask your instructor for information about your assembler's format.

We are going to enter this program into memory starting at location 0000 (hexadecimal). If the trainer you are using does not allow programs to be placed in these memory locations, refer to your manual and substitute valid addresses in place of those shown in Fig. 16-12.

If you are using an assembler, please enter the program now. It will look similar to what is shown in Fig. 16-13.

Also place 0s in accumulator A, accumulator B, and the X (index) register so that you will know what numbers are in each register before you run the program. Refer to Fig. 16-14 to see what the memory and registers should look like.

If you are *not* using an assembler, you must look up the op codes by hand in the Expanded Table. This is called *hand-assembly*. Let's go through the necessary steps for hand-assembly.

To the right of the LDAA #$dd, in the op code (op for short) column you will see the hexadecimal number 86. This is the 6800/6808 op code, which stands for *Load accumulator A with the number immediately following this op code*. Set your trainer so that the memory address where the next instruction will be loaded is someplace within the area allowed for user programs. We chose 0000, but you may need to use another location. Enter the number 86 into the first available memory location. Since this was a *Load Accumulator A Immediate* instruction, the microprocessor will expect the next address, which *immediately* follows the op code, to contain the number which is to be placed in accumulator A. Therefore enter 11 next. In the third

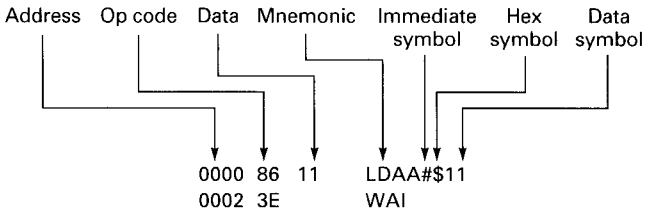


Fig. 16-13 Disassembly of the sample program.

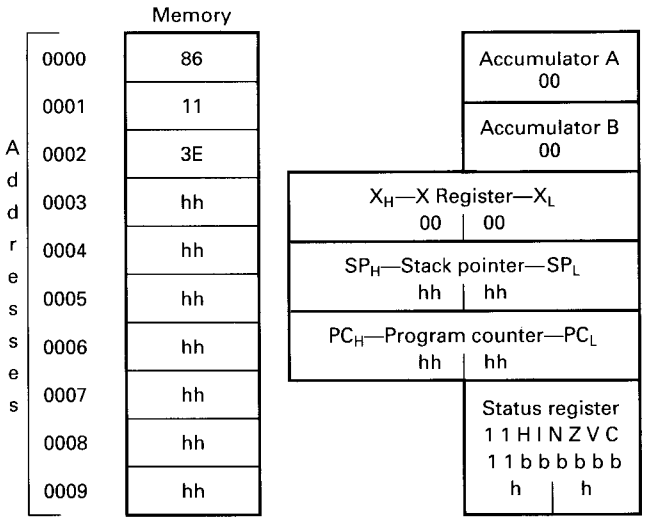


Fig. 16-14 6800/6808 sample program.

address enter 3E, which is the op code for the WAI instruction.

Enter 0s into accumulator A, accumulator B, and the X (index) register now so that you will know the condition of these registers before the program is run.

If you check your registers and memory, you should see what is shown in Fig. 16-14 (although you may have placed the program at a different memory location). The h's and b's represent hex and binary digits which we are not concerned with now.

Running the Program

Let's use Fig. 16-15 during our analysis of program operation.

The first op code is 86, which means, *Load accumulator A with the contents of the next memory location*, or more properly, *Place a copy of the contents of the next memory location in accumulator A*. As you see, the number 11 is

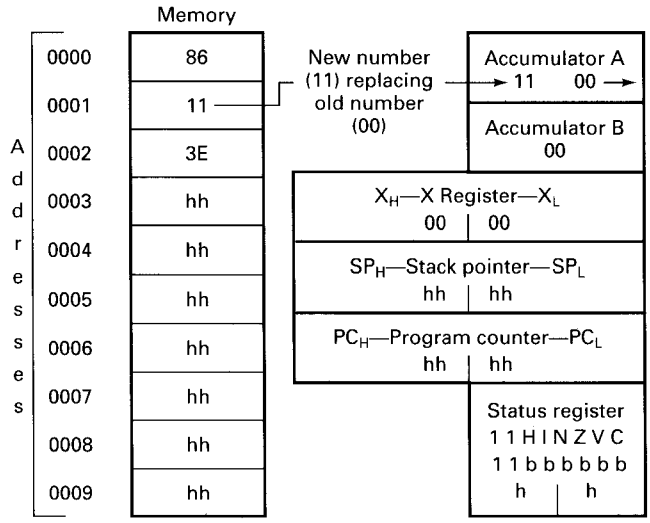


Fig. 16-15 6800/6808 sample program.

Addr	Obj	Assembler	Comment
0000	86	LDAA #$EE	Load accumulator A with the hex number immediately following the LDAA# op code (86)
0001	EE		
0002	16	TAB	Transfer the contents of A into B
0003	3E	WAI	Stop

Fig. 16-16 Example 16-2 program.

replacing 00 in the accumulator. The program then continues to the next instruction op code, 3E, which stands for **WAI**, and stops.

Checking the Results of Program (Analysis)

After running the program, you should have 00 in accumulator B and the X (index) register and 11 in accumulator A. The program does what we designed it to do.

Here's one for you to try.

EXAMPLE 16-2

First manually place 00s in accumulator A, accumulator B, and the X register. Then write a program which will

1. Load accumulator A with the hex number EE.
2. Transfer a copy of the contents of the accumulator A into accumulator B.
3. Stop.

SOLUTION

Figure 16-16 shows the completed program. Figure 16-17 shows the memory and registers and what happens during program execution.

16-5 8080/8085/Z80 FAMILY

Let's see how the ideas which were introduced in the New Concepts section apply to the 8080/8085/Z80 microprocessor family.

CPU Control Instructions

The 8080/8085/Z80 family has a *no operation* instruction which uses the mnemonic NOP. Refer to the Expanded Table of 8085/8080 and Z80 (8080 Subset) Instructions Listed by Category in Part 4 of this text.

In the ninth column, called the Boolean/Arithmetic Operation column, we see that this instruction does "nothing," as we said it would. Also notice the hexadecimal number under the op (op code) column, in this case 00. This is the actual hex code for NOP.

The 8080/8085/Z80 family has an actual halt instruction. Refer to the Expanded Table again. Notice that the halt instruction uses the mnemonic HLT [Z80 = HALT] and has an op code of 76.

Data Transfer Instructions

Refer to the Expanded Table. Underneath the halt instruction you will see the MOV A,A [Z80 = LD A,A] instruction

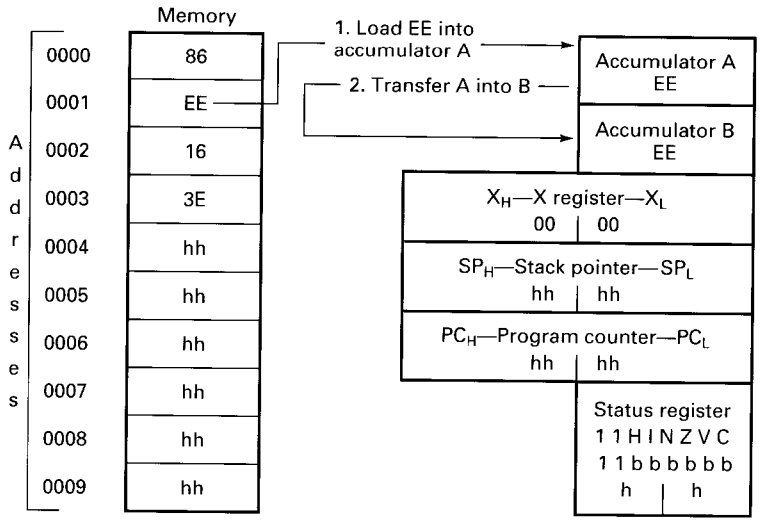


Fig. 16-17 Example 16-2 program analysis.

at the beginning of the Data Transfer Instructions section. This section is a list of all of the different types of data transfer instructions available in the 8080/8085/Z80 family. (To those with previous microprocessor experience: You may notice that we have excluded transfer instructions involving the stack. This is intentional. They have been included in the Stack Instructions category.)

Direction of Data Transfer

Let's look at the data transfer section a little more closely. The second instruction listed is the **MOV**e data to **A** from **B** instruction. The boldfaced letters help show where the MOV A,B mnemonic came from. (If you are using the Z80 microprocessor, it is the **LoaD** data into **A** from **B** instruction. The boldfaced letters show where the LD A,B mnemonic came from.) The ninth column shows the Boolean/Arithmetic Operation. This is a concise and graphic way to state exactly what this instruction does. It shows B, which stands for register B, moving toward A, which stands for the accumulator. To put it another way, the contents of register B are being transferred into the accumulator.

Recall from the New Concepts section that moving or transferring is actually more like making a copy of what's in a particular location and placing the copy in the destination.

It may help to have a mental picture of our programming model of the 8085/8080/Z80, shown in Fig. 16-18, as we discuss these instructions.

There are many directions in which data could be transferred with an accumulator, six registers, and memory. This can be seen in the Expanded Table. The first eight instructions transfer the contents of a register or memory location into the accumulator. (This can be seen in the Operation column and the Boolean/Arithmetic Operation column.) The second group of eight instructions copy the contents of the accumulator, one of the registers, or memory into register B. The third group of eight transfer data into register C. The fourth group into D. The fifth into E. The sixth into H. The seventh into L. And the eighth into a memory location. This makes 8 × 8 or 64 instructions just to do simple data transfers between registers.

The next group of eight instructions consists of the *Move Immediate* instructions. They move a specified number directly into a register or memory.

We will leave it to you to glance at the rest of the data transfer instructions in the Expanded Table.

If you have used the 6502 family or 6800/6808 family chips before (especially the 6502 family) and are now studying the 8085/Z80 family for the first time, you may be surprised by the great number of different instructions this family has. This is offset, however, by the relatively few addressing modes available and the simplicity this can offer the programmer. (The 6502 family, by contrast, has very few different instructions but has a large number of addressing modes for an 8-bit chip from its era.)

Op Codes

Does your computer or microprocessor trainer understand the statement "Move data to A from B"? No. Does it understand the mnemonic MOV A,B? If you are using an assembler, the assembler translates the mnemonic into binary numbers, which it does understand. (If you can type the mnemonic MOV A,B into your computer or trainer, you have an assembler. If instead you must see a hexadecimal keypad or type in hex numbers, you do not have an assembler.) The point here is that the microprocessor inside your computer does not understand English words like "Move" or mnemonics like MOV A,B.

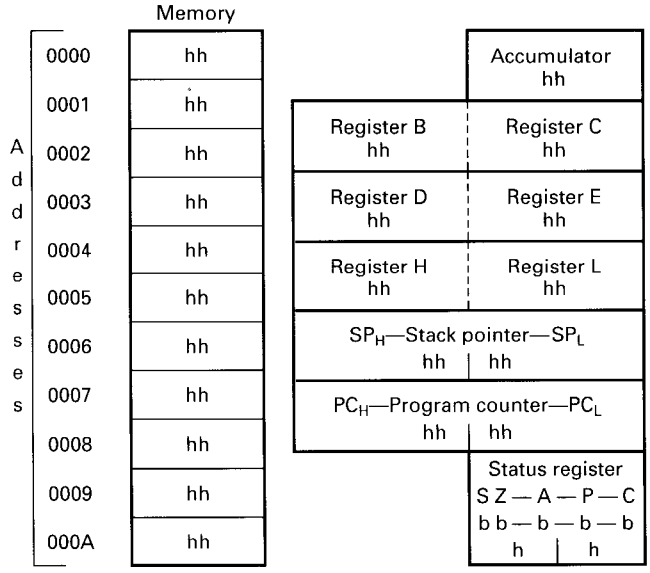


Fig. 16-18 Complete 8080/8085 and Z80 (8080 subset) programming model.

If you are using an assembler, the assembler program is translating the mnemonics, which the microprocessor does not understand, into something it does understand. What does the microprocessor understand? Binary numbers. In our case we will enter them as their equivalent hexadecimal value and let the monitor or assembler translate that into binary. For our purposes, at least at this point, we'll say that the microprocessor understands hexadecimal. (The monitor is part of the *firmware* built into your microprocessor trainer.)

Look again at the Data Transfer section of the table. If we wanted to tell the microprocessor to load the accumulator from register B (the second data transfer instruction, MOV A,B *[LD A,B]*, the microprocessor chip would actually need the hex code in the eighth column over, the op code column (op for short). We would place the hex number 78 in the computer's memory as the first instruction to execute.

Let's look at another example. What if you wanted to have the microprocessor copy the contents of the C register into the accumulator? What would be the hex number the microprocessor would need to understand what you wanted to do? You should have said 79 from the MOV A,C *[LD A,C]* instruction.

Sample 8085/Z80 Program

(*Note:* Since we are simultaneously covering the 8085 and Z80 microprocessors, we will give the 8085 mnemonic first, followed by the Z80 mnemonic in *italic* print and enclosed by square brackets, for example, MVI A,dd *[LD A,dd]*.)

Program Objective

Let's create a program which will

1. Place the number 11 in the accumulator.
2. Stop.

Creating the Program

Refer to the Data Transfer Instructions section of the Expanded Table. Do you see an instruction which could be used to place a number in the accumulator?

[*Note:* You may want to use the Mini Table of 8085/Z80 (8080 Subset) Instructions listed by Category at this time. There are so many 8085/Z80 data transfer instructions that it may prove to be a bit time-consuming to page through the Expanded Table.]

Look in the Boolean/Arithmetic Operation column (simply labeled Operation in the Mini Table). You need an instruction which has an arrow pointing to the accumulator (indicated by an A). There are 12 such instructions; using 8085 mnemonics, they are MOV A,A; MOV A,B; MOV A,C; MOV A,D; MOV A,E: MOV A,H; MOV A,L; MOV A,M; MVI A,dd; LDAX B; LDAX D; and LDA aaaa. *[Using Z80 mnemonics, they are LD A,A; LD A,B; LD A,C; LD A,D; LD A,E; LD A,H; LD A,L; LD A, (HL); LD A,dd; LD A, (BC); LD A, (DE); and LD A, (aaaa).]*

The next step is to determine which one of these instructions to use. The key to this decision can be found in the Operation or (Boolean/Arithmetic Operation) column. The data transfer instruction we want is one which will take a number (which we will place *immediately* after the instruction op code) and will transfer it into the accumulator.

The first eight instructions mentioned above take a number which is already in one of the seven 8085/Z80 registers or memory and place it in the accumulator. This is not what we want. The last three instructions take a number or data byte from a memory location and place it in the accumulator. This is not what we want either. The MVI A,dd (**MoVe Immediate dd to A**) *[Z80 = LD A, dd (**LoaD dd into A**)]* instruction takes the number *immediately* following the move instruction and places it in the accumulator. This is what we want since it allows us to specify the number 11 right after the op code for the move instruction.

Finally, you want the program to stop. The instruction which does this is in the CPU Control Instructions section. The halt instruction is the obvious choice.

Entering the Program

The completed program is shown in Fig. 16-19. We'll see how to enter it into your microprocessor first using an assembler and then without an assembler.

Note that the column labeled Obj contains the actual 8085 and Z80 op codes, and the Assembler column contains the mnemonic and data in a format similar to that used by an assembler.

Refer to the MVI A,dd *[LD A,dd]* instruction in the Mini Table. These mnemonics are used by assemblers, which means that you must type the instruction using this format. To the right of the mnemonic, in the Op column, is the op code for that particular instruction. The 8085 and Z80 microprocessors use the same op codes: Only the mnemonics are different. The dd simply stands for two hexadecimal digits of data. (Each d stands for one nibble or 4 bits.)

We are going to enter this program into memory starting at location 0000 (hexadecimal). If the trainer you are using does not allow programs to be placed in these memory locations, refer to your manual to determine where programs can be placed in memory and substitute those addresses.

If you are using an assembler, please enter the program now. It will look similar to what is shown in Fig. 16-20.

Also place 0s in the accumulator and all the general-purpose registers (registers B, C, D, E, H, and L) so that you will know what numbers are in each register before you run the program. Refer to Fig. 16-21 to see what the memory and registers should look like.

If you are not using an assembler, you must look up the op codes by hand in either the Expanded Table or the Mini Table. This is called *hand assembly*. Let's go through the necessary steps for hand assembly.

8085 mnemonics

Addr	Obj	Assembler	Comment
0000	3E	MVI A, 11	Load the accumulator with the number (11) immediately following the MVI op code (3E)
0001	11		
0002	76	HALT	Halt

Z80 mnemonics

Addr	Obj	Assembler	Comment
0000	3E	LD A, 11	Load the accumulator with the number (11) immediately following the LD A,dd op code (3E)
0001	11		
0002	76	HALT	Halt

Fig. 16-19 Sample program. (*Note:* The addresses should be an area where user programs can be placed. If 0000 is not such a place on your system, then you will need to change these addresses.)

If you look up the MVI A,dd *[LD A,dd]* mnemonic in either the Expanded Table or the Mini Table (for the 8080/8085/Z80), you will see the hex number 3E in the Op column. This is the op code which stands for, "**M**o**V**e the number **I**mmediately following this op code into the **A**ccumulator." *["LoaD the number following this op code into the Accumulator."]* Set your trainer so that the memory address where the next instruction will be loaded is someplace within the area allowed for user programs. We chose 0000, but you may need to use another location. Enter the hex number 3E into the first available memory location. Since this was a **M**o**V**e **I**mmediate to **A**ccumulator *[LoaD Accumulator]* instruction, the microprocessor will expect the next address, which *immediately* follows the op code, to contain the number which is to be placed in the accumulator. Therefore enter 11 next. In the third address enter 76, which is the op code for the halt instruction.

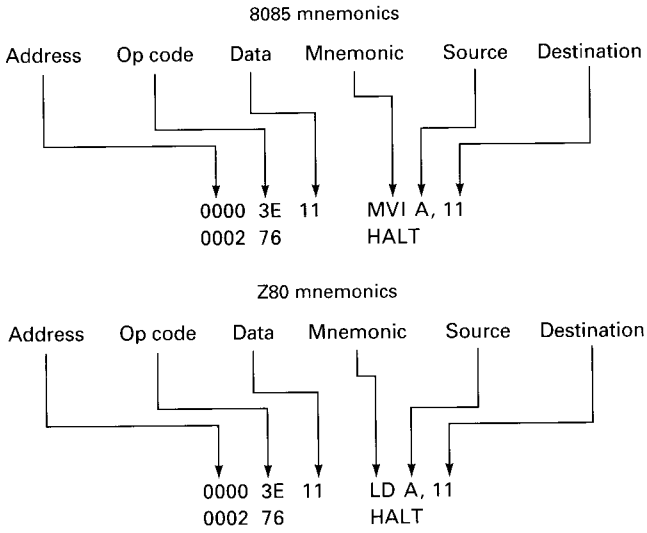


Fig. 16-20 Disassembly of the sample program.

Enter 0s into the accumulator and all the general-purpose registers at this time so that you will know the conditions of these registers before the program is run.

If you check your registers and memory, you should see what is shown in Fig. 16-21 (although you may have placed the program at a different memory location). The h's and b's represent hex and binary digits which we are not concerned with at this time.

Running the Program

Let's use Fig. 16-22 during our analysis of program operation.

The first op code is 3E, which means, *Load the accumulator with the contents of the next memory location*, or

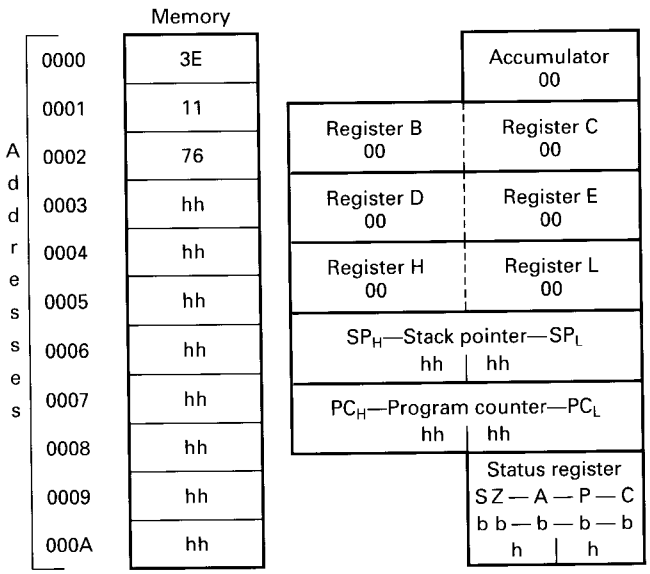


Fig. 16-21 8085/Z80 sample program.

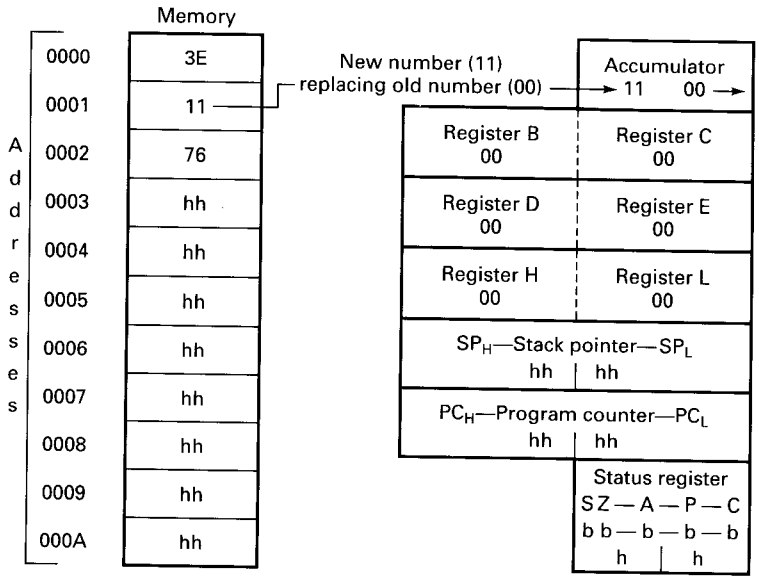


Fig. 16-22 8085/Z80 sample program.

more properly, *Place a* copy *of the contents of the next memory location in the accumulator*. As you see, the number 11 is replacing 00 in the accumulator. The program then continues to the next instruction op code, 76, which stands for halt, and stops.

Checking the Results of Program (Analysis)

After running the program, you should have 00 in all the general-purpose registers and 11 in the accumulator. The program does what we designed it to do.

Here's one for you to try.

EXAMPLE 16-3

First manually place 00s in the accumulator and all general-purpose registers. Then write a program which will

1. Place the hex number EE in the accumulator.
2. Move (copy) the contents of the accumulator (A) into register B.
3. Move (copy) the contents of the accumulator (A) into register C.
4. Stop.

SOLUTION

Figure 16-23 shows the completed program in both 8085 and Z80 mnemonics. Figure 16-24 shows the memory and registers and what happens during program execution.

16-6 8086/8088 FAMILY

We will approach the 16-bit 8086/8088 microprocessor a little differently than we did the 8-bit microprocessors. The 8-bit sections are designed to fit the needs of a person using op code charts and hand assembly in the earlier chapters and an assembler in the later chapters.

In the 16-bit section we assume that you are using the DOS DEBUG utility in the earlier chapters. DEBUG is readily available to all who use MS-DOS–type machines, and it is less sophisticated than assemblers, which keeps you closer to the hardware during the early part of the learning process.

In later chapters we will use both an assembler and DEBUG in figures and in answers to chapter questions. This will allow you to explore the advantages of a full-featured assembler and to continue to use DEBUG if you wish.

One final point should be kept in mind. This text is designed to make the learning process as simple as possible for the beginner. A 16-bit chip like the 8086/8088 is quite complex for the beginner. Therefore we do not attempt to cover every aspect of this chip.

CPU Control Instructions

The 8086/8088 has a no operation (NOP) instruction which works as described in the New Concepts section of this chapter. A brief description of the NOP instruction can be found in the CPU Control Instructions section of the Expanded Table of 8086/8088 Instruction Listed by Category in Part 4 of this text. The NOP has an op code of 90 and affects no flags.

The 8086/8088 has a halt instruction which functions as described in the New Concepts section. A description of this instruction appears in the CPU Control Instructions section of the 8086/8088 instruction set. Its mnemonic is HLT, and its op code is F4.

8085 mnemonics

Addr	Obj	Assembler	Comment
0000	3E	MVI A,EE	Place the hex number EE in the accumulator (A)
0001	EE		
0002	47	MOV B,A	Copy into register B the contents of A
0003	4F	MOV C,A	Copy into register C the contents of A
0004	76	HALT	Stop

Z80 mnemonics

Addr	Obj	Assembler	Comment
0000	3E	LD A,EE	Place the hex number EE in the accumulator (A)
0001	EE		
0002	47	LD B,A	Copy into register B the contents of A
0003	4F	LD C,A	Copy into register C the contents of A
0004	76	HALT	Stop

Fig. 16-23 Example 16-3 program.

Data Transfer Instructions

The 8086/8088 has eight instructions which we have placed in the Data Transfer Instructions section. While the Expanded Table of 8086/8088 Instructions Listed by Category lists all eight of these instructions, the most versatile and by far the most useful for the beginner is the **MOV**e instruction.

A copy of our programming model for the 8086/8088 appears in Fig. 16-25.

Direction of Data Transfer

A move can be *from* (source) a register, memory, or an immediate number *to* (destination) a register or memory. While *either* the source or the destination can be a memory location, *both* cannot be memory locations in the same instruction. The source and destination must both be either 8 bits wide or 16 bits wide; you can't mix data widths in the same instruction. And finally, you can't move from one segment register to another.

As you have seen from the programming model, the 8086/8088 has several 8-bit and 16-bit registers. This causes the number of move combinations between registers alone to number in the hundreds. A few examples are

```
MOV   AL,DL      AL ← DL
MOV   BH,BL      BH ← BL
MOV   AX,DX      AX ← DX
MOV   SP,BP      SP ← BP
MOV   SI,DI      SI ← DI
MOV   BX,DS      BX ← DS
MOV   AL,76      AL ← 76
```

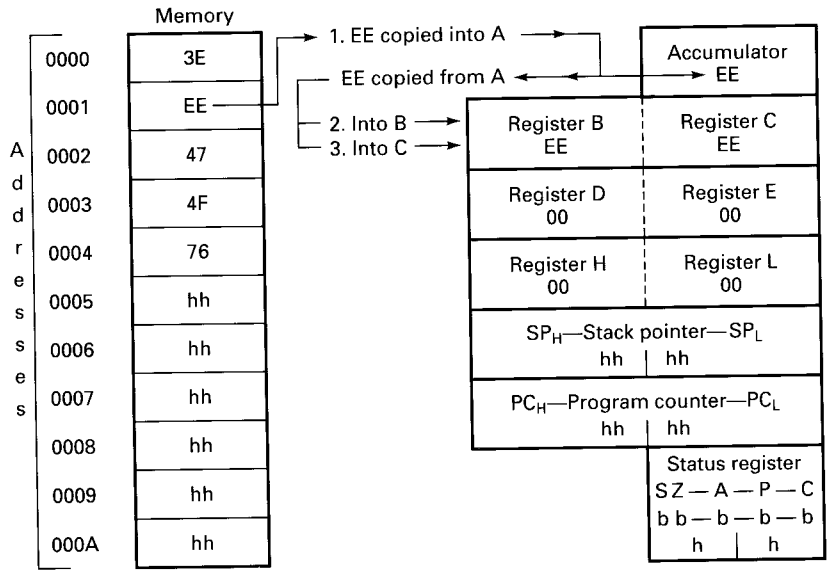


Fig. 16-24 Example 16-3 program analysis.

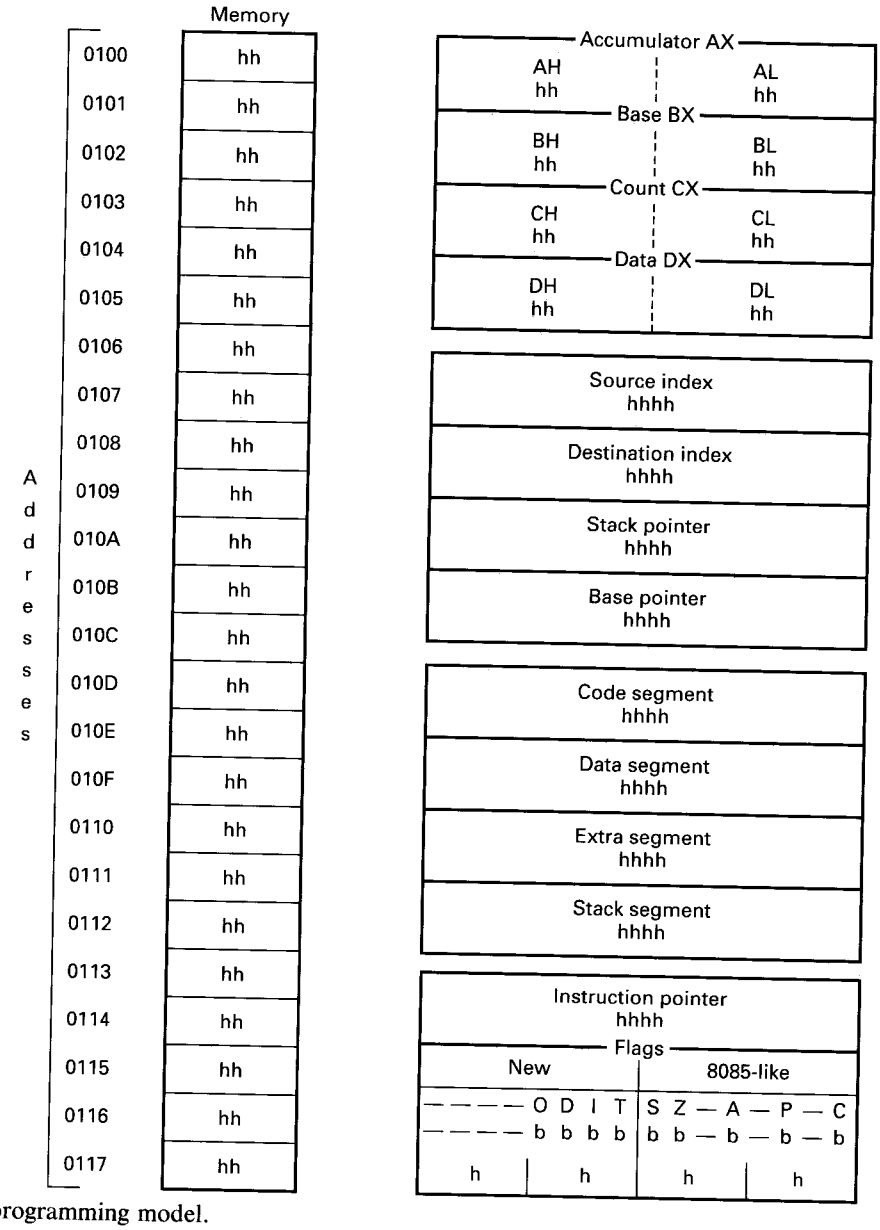


Fig. 16-25 8080/8086 programming model.

MOV	AX,89E3	AX ← 89E3
MOV	[1234],AX	memory location 1234 ← AX
MOV	BL,[4456]	BL ← memory location 4456
MOV	DX,[BX + DI]	DX ← memory location found by adding the contents of BX and DI
MOV	AX,[BX + DI + 0200]	AX ← memory location pointed to by the sum of the contents of BX, the contents of DI, and the hex number 200_{16}

The left column shows the instruction exactly as it appears when disassembled by DEBUG. The right column indicates where the data comes from and where it goes.

Sample 8086/8088 Program

Figure 16-26 shows a sequence of commands that will demonstrate a simple **MOV**e instruction and give you practice entering programs into DEBUG.

First, we started DEBUG by typing

```
C>debug
```

at the DOS prompt as shown. DEBUG responded with a

```
-
```

which indicates it is waiting for a command.

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0100   NV UP EI PL NZ NA PO NC
1A2A:0100 7420          JZ      0122
-a
1A2A:0100 mov al,dl
1A2A:0102
-u 100 101
1A2A:0100 88D0          MOV     AL,DL
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0100   NV UP EI PL NZ NA PO NC
1A2A:0100 88D0          MOV     AL,DL
-rdx
DX 0000
:00f3
-r
AX=0000  BX=0000  CX=0000  DX=00F3  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0100   NV UP EI PL NZ NA PO NC
1A2A:0100 88D0          MOV     AL,DL
-t
AX=00F3  BX=0000  CX=0000  DX=00F3  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0102   NV UP EI PL NZ NA PO NC
1A2A:0102 65            DB      65
-q

C>
```

Fig. 16-26 MOVe instruction (DEBUG screens).

Next we typed an "r," which stands for register. This causes DEBUG to display the values of all registers as shown in Fig. 16-27.

We will now duplicate (several times) that portion of Fig. 16-26 (in bold type) which shows the values in various registers. You should compare these sections (as we progress through each figure) to our 8086/8088 programming model in Fig. 16-25.

The current values of the general-purpose registers are shown in bold type in Fig. 16-28.

The values of the stack pointer, base pointer, source index, and destination index are shown in bold type in Fig. 16-29.

The values in the segment registers are shown in bold in Fig. 16-30.

The value of the instruction pointer and the current status of the flags are shown in bold in Fig. 16-31.

Finally, the address, op code, and assembler notation for the next instruction which is to be executed are shown in bold type in Fig. 16-32.

The area shown in bold type in Fig. 16-33 illustrates how we then typed an "a," which is the DEBUG assemble command, at the DEBUG prompt.

-a <ENTER>

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0100   NV UP EI PL NZ NA PO NC
1A2A:0100 7420          JZ      0122
-
```

Fig. 16-27 DEBUG screens (cont.).

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0100   NV UP EI PL NZ NA PO NC
1A2A:0100 7420          JZ      0122
-
```

Fig. 16-28 DEBUG screens (cont.).

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A2A  ES=1A2A  SS=1A2A  CS=1A2A  IP=0100   NV UP EI PL NZ NA PO NC
1A2A:0100 7420          JZ      0122
-
```

Fig. 16-29 DEBUG screens (cont.).

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 7420          JZ      0122
-
```

Fig. 16-30 DEBUG screens (cont.).

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 7420          JZ      0122
```

Fig. 16-31 DEBUG screens (cont.).

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 7420          JZ      0122
-
```

Fig. 16-32 DEBUG screens (cont.).

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 7420          JZ      0122
-a
992A:0100 mov al,dl
992A:0102
-u 100 101
992A:0100 88D0          MOV     AL,DL
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 88D0          MOV     AL,DL
-rdx
DX 0000
:00f3
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 88D0          MOV     AL,DL
-t

AX=00F3  BX=0000  CX=0000  DX=00F3  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0102   NV UP EI PL NZ NA PO NC
992A:0102 65            DB      65
-q

C>
```

Fig. 16-33 DEBUG screens (cont.).

DEBUG then responded with

992A:0100

which is the address at which our program will start. The 992A is the memory segment, and 0100 is the memory location within that segment. If you try this program on your computer, your segment will probably not be the same as ours. This is normal and will not affect the results of the program.

We then typed

mov al,dl <ENTER>

and DEBUG responded with

992A:0102

which is the address of the next available memory location. We then pressed <ENTER> to terminate assembly, and DEBUG waited for our next command.

We told DEBUG to create or *assemble* the machine code for the MOV AL,DL instruction. Then we wanted to check to see that this is what DEBUG did. We wanted to *disassemble* the machine code. The DEBUG command for this is "u," which stands for **u**nassemble (DEBUG's name for disassemble). The next command in our program is

-u 100 101

which tells DEBUG to unassemble memory locations 100_{16}–101_{16} within the current code segment. DEBUG responded with

992A:0100 88D0 MOV AL,DL

992A is the current code segment. 0100 is the memory location of the first byte of this instruction. 88D0 is the machine code for MOV AL,DL, which was the assembly-language instruction we typed in.

We typed the register command, and DEBUG again displayed the current status of all registers. DEBUG's response is shown in Fig. 16-34.

When DEBUG displays the registers, it also displays the instruction which it finds at the memory location pointed to by the instruction pointer in the current code segment. These appear in bold type in Fig. 16-34. Our MOV AL,DL instruction appears in the assembly-language section.

Since our instruction said to move the contents of register DL to register AL, we needed to place some value in register DL. Notice that at this point AX, BX, CX, and DX all contained 0000. Even if the contents of DL were copied to AL, we wouldn't see any difference. We needed to place some value in DL which we could observe.

The area in bold type in Fig. 16-35 shows our next command

-rdx

which told DEBUG we wanted to change the value in register DX. DEBUG responded with

DX 0000
:

which was the current contents of register DX. The cursor waited after the colon. If we had typed in a value, that value would have been placed in the DX register. If we had pressed the <ENTER> key, the value in DX would not have changed.

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 88D0          MOV     AL,DL
```

Fig. 16-34 DEBUG screens (cont.).

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 7420          JZ      0122
-a
992A:0100 mov al,dl
992A:0102
-u 100 101
992A:0100 88D0          MOV     AL,DL
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 88D0          MOV     AL,DL
-rdx
DX 0000
:00f3
-r
AX=0000  BX=0000  CX=0000  DX=00F3  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 88D0          MOV     AL,DL
-t

AX=00F3  BX=0000  CX=0000  DX=00F3  SP=6D5E  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0102   NV UP EI PL NZ NA PO NC
992A:0102 65            DB      65
-q

C>
```

Fig. 16-35 DEBUG screens (cont.).

We wanted to place a new number in DL. However, we could not single out the low byte of the DX register, so we simply placed 0s in the high byte and our number in the low byte. We typed that number (00f3) and pressed <ENTER>.

:00f3 <ENTER>

Figure 16-36 shows how we again used the register command ("r").

Notice that the value in register DX has been changed to the value we typed in.

Running the Program

Next we wanted the computer to execute the MOV AL,DL instruction. However, we did not want it to continue any further than that. Even though we had not entered any other instruction into the computer, there were others. When we turned the computer on, each unused memory location contained some number, even if it was 00_{16}. Most of these "random" numbers were actually the op code for some instruction. We didn't want these "random" instructions to execute.

DEBUG has a command called trace which executes the next instruction (the one displayed at the bottom of the register display) and then stops and automatically displays the contents of the registers for viewing. This is what we did in Fig. 16-37.

Notice that the value in DX has been *copied* into register AX. Notice also that the **I**nstruction **P**ointer has been incremented to the position of the next instruction in memory which is displayed at the bottom of the register display (in bold type).

Checking the Results

Figure 16-38 shows the operation of the program by using our programming model to illustrate the movement of F3 from one register to the other.

Our program worked. In the future we will not discuss each 8086/8088 program in such detail, but we have done so here to give you an idea of how to monitor the execution of a program. We have also introduced you to some DEBUG commands. Remember that the DEBUG commands—*assemble, unassemble, trace, register*, and *quit*—are not assembly-language instructions but are commands to the DEBUG utility, which helps you to enter, modify, and execute assembly-language instructions.

Finally, you may want to exit from the DEBUG program. That command is simply the **q**uit command, which is entered with the letter q. You will then be returned to the DOS prompt.

EXAMPLE 16-4

Place the number FE in register DH. Place the number 12 in DL. Then write a program that will

1. Copy DH to AH.
2. Copy DL to BH.

Use the trace command to execute the program and follow its operation.

```
-r
AX=0000  BX=0000  CX=0000  DX=00F3  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0100   NV UP EI PL NZ NA PO NC
992A:0100 88D0          MOV     AL,DL
```

Fig. 16-36 DEBUG screens (cont.).

```
-t

AX=00F3  BX=0000  CX=0000  DX=00F3  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=992A  ES=992A  SS=992A  CS=992A  IP=0102   NV UP EI PL NZ NA PO NC
992A:0102 65            DB      65
```

Fig. 16-37 DEBUG screens (cont.).

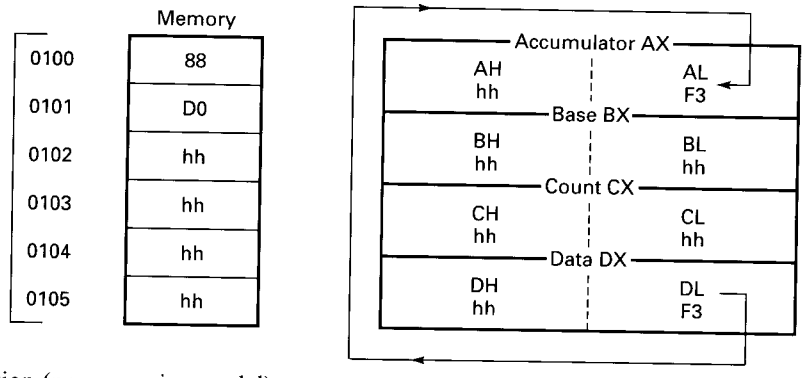


Fig. 16-38 MOVe instruction (programming model).

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=404E  BP=0000  SI=0000  DI=0000
DS=9BFB  ES=9BFB  SS=9BFB  CS=9BFB  IP=0100   NV UP EI PL NZ NA PO NC
9BFB:0100 A3FF72        MOV     [72FF],AX                       DS:72FF=FF1F
-rdx
DX 0000
:fe12
-r
AX=0000  BX=0000  CX=0000  DX=FE12  SP=404E  BP=0000  SI=0000  DI=0000
DS=9BFB  ES=9BFB  SS=9BFB  CS=9BFB  IP=0100   NV UP EI PL NZ NA PO NC
9BFB:0100 A3FF72        MOV     [72FF],AX                       DS:72FF=FF1F
-a
9BFB:0100 mov ah,dh
9BFB:0102 mov bh,dl
9BFB:0104
-r
AX=0000  BX=0000  CX=0000  DX=FE12  SP=404E  BP=0000  SI=0000  DI=0000
DS=9BFB  ES=9BFB  SS=9BFB  CS=9BFB  IP=0100   NV UP EI PL NZ NA PO NC
9BFB:0100 88F4          MOV     AH,DH
-t

AX=FE00  BX=0000  CX=0000  DX=FE12  SP=404E  BP=0000  SI=0000  DI=0000
DS=9BFB  ES=9BFB  SS=9BFB  CS=9BFB  IP=0102   NV UP EI PL NZ NA PO NC
9BFB:0102 88D7          MOV     BH,DL
-t

AX=FE00  BX=1200  CX=0000  DX=FE12  SP=404E  BP=0000  SI=0000  DI=0000
DS=9BFB  ES=9BFB  SS=9BFB  CS=9BFB  IP=0104   NV UP EI PL NZ NA PO NC
9BFB:0104 3C3A          CMP     AL,3A
-
```

Fig. 16-39 Example 16-4 (DEBUG screens).

SOLUTION

Figure 16-39 shows the process of changing the contents of the registers, entering the assembly-language instructions, and tracing program execution. Especially notice the areas in bold type. (They, of course, will not appear in bold on your computer screen.)

Figure 16-40 shows the same program, illustrating the movement of the data with our programming model.

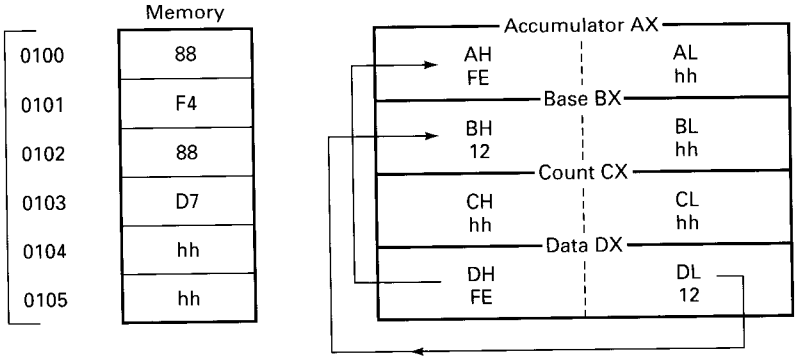


Fig. 16-40 Example 16-4 (programming model).

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The various instructions which form the instruction set of most microprocessors fall into natural _____ or groups.
2. (*categories*) The _____ and _____ are the microprocessor chips found in IBM PC compatibles.
3. (*8086, 8088*) A technique which is sometimes helpful when analyzing a program involves _____ the contents of each register or memory location and updating each as it changes in the program.
4. (*writing*) When we talk of moving, loading, transferring, or storing data, while working with the microprocessors in this text, are we referring to *moving* in the sense that the data no longer exists in its original location?
5. (*No*) When we talk about moving, loading, transferring, or storing data, we are actually _____ the data.
6. (*copying*) If you can type mnemonics into your computer or trainer, it must have an _____.
7. (*assembler*) What do microprocessors understand?
8. (*binary numbers*) An assembler translates mnemonics into _____.
(*binary numbers*)

PROBLEMS

General

16-1. What does the NOP (no operation) instruction do?
16-2. What are two purposes of the NOP instruction?
16-3. If you move, load, or transfer the contents of the accumulator to a general-purpose register, what is left in the accumulator?

6502 Family

16-4. What is the op code for the NOP instruction?
16-5. What is the op code for the **BReaK** instruction?
16-6. What is the op code for the TAX (**T**ransfer **A**ccumulator to **X** register) instruction?
16-7. What does the TYA instruction do?
16-8. What does the mnemonic STX stand for?
16-9. Which instruction would you use to copy the contents of the Y register into a memory location?
16-10. Write a program which will
a. Place the number 45_{16} in the accumulator.
b. Transfer the contents of the accumulator to the X register.
c. Stop.

6800/6808 Family

16-11. What is the op code for the NOP instruction?
16-12. What is the op code for the WAI instruction?
16-13. What is the op code for the TAB (**T**ransfer **A** accumulator to accumulator **B**) instruction?
16-14. What does the TBA instruction do?
16-15. What does the mnemonic CLRA stand for?
16-16. Which instruction would you use to copy the contents of accumulator B into a memory location?
16-17. Write a program which will
a. Place the number 89_{16} in accumulator B.
b. Copy the contents of accumulator B to accumulator A.
c. Stop.

8080/8085/Z80 Family

16-18. What is the op code for the NOP instruction?
16-19. What is the op code for the HALT instruction?
16-20. What is the op code for the Mov A,D *[LD A,D]* instruction?
16-21. What does the MOV B,C *[LD B,C]* instruction do?
16-22. What does the mnemonic MVI A,dd *[LD A,dd]* stand for?
16-23. Which instruction would you use to store the contents of the accumulator in a memory location?
16-24. Write a program which will
a. Place the number 78_{16} immediately into the accumulator.
b. Copy the contents of the accumulator into register C.
c. Stop.

8086/8088 Family

16-25. What is the DOS utility which we are using in this text to do assembly, disassembly, running, and debugging of 8086/8088 assembly-language programs?
16-26. What three areas can serve as a source for the 8088 **MOV**e instruction?

16-27. What are the two areas which can serve as destinations for the 8088 MOV instruction?

16-28. Which area *cannot* be both a source and a destination at the same time?

16-29. What is the source of a MOV AL,DL instruction?

16-30. What is the destination of a MOV AL,76 instruction?

16-31. Does the instruction MOV B,[4456] move the number 4456 or the contents of memory location 4456_{16} to register B?

16-32. What does the DEBUG command "r" stand for and what does it do?

16-33. What does the DEBUG command "a" stand for and what does it do?

16-34. What does the DEBUG command "u" stand for and what does it do?

16-35. What does the DEBUG trace command do?

16-36. What is the DEBUG quit command?

16-37. Using DEBUG, write an 8086/8088 assembly program which will

a. Place the number 89_{16} into the register BL.

b. Copy the contents of BL into CL.

(*Note:* Use DEBUG's trace command to execute the program to see if it works.)