21

ADDRESSING MODES—II

New Concepts

In this chapter we'll study some of the more complex addressing modes. The different microprocessor families will show more variation at this point than they did in our earlier chapter on basic addressing modes.

The 6502 has more addressing modes than any other 8-bit microprocessor. Some are used quite often, but several are used with only a few instructions. Since the 6502 has no general-purpose registers and only one accumulator, it must use memory very often and is therefore said to have a *memory-intensive* architecture.

The 6800/6808 has a moderate number of different addressing modes, and students learning about it should not have difficulty. The 6800/6808 also lacks general-purpose registers but does have two accumulators. It is also considered to have a memory-intensive architecture.

The 8080/8085 has the fewest number of addressing modes of any of the 8-bit microprocessors. Students will find it easiest to learn in this respect. (The Z80 has more addressing modes, but those beyond the ones the 8080/8085 has will not be studied at this time.) The 8080/8085 has six general-purpose registers in addition to an accumulator and is therefore said to have a *register-intensive* architecture.

The 8086/8088, being a successor to and relative of the 8080/8085, has many general-purpose registers. Because it is a 16-bit microprocessor, it also has many addressing modes.

To summarize, the 6502 has 56 different instructions which use one or more of 13 addressing modes. When you combine the instructions and addressing modes, you produce 152 different op codes.

The 6800/6808 has 107 different instructions which use one or more of seven addressing modes. The 6800/6808 has 197 different op codes.

The 8080/8085 has 246 different instructions which have only one addressing mode each. There are five different addressing modes. This provides a total of 246 different op codes.

The 8086/8088 has 24 addressing modes (they are presented in 11 addressing-mode categories in this text) and approximately 91 different assembly-language instructions. This is just part of the picture, however.

Each 8086/8088 instruction can have many variations, the **MOV**e instruction probably being the best example. MOV is considered one assembly-language instruction; yet the 8086/8088 recognizes 28 different assembly-language forms of the MOV instruction (*move to a register, move immediate, move byte to memory, move word to register*, and so on). Each of the 28 assembly-language forms can have many different machine-level instructions which may be composed of up to 6 bytes (with eight 8-bit registers; the ability to move any one of them to any other produces 10s of different machine-level instructions just for moving 8-bit registers).

To put it simply, there are hundreds of variations of the MOV instruction alone. The possible variations of all 91 different assembly-language instructions number somewhere between 3,000 and 4,000.

How can anyone learn so many combinations? First, if you are using the 8086 or 8088, you will be concentrating on learning about the 91 different assembly-language instructions, not every possible variation. Second, once you learn any one instruction, MOV, for example, most of the variations will seem very natural. It's not like rote memorization.

Which microprocessor is easiest to learn? That's hard to say. They each have strengths and weaknesses. And which feature is a strength and which is a weakness depend on what you as the programmer want to do.

(*Note:* Do not try to memorize all of these addressing modes at this time. Read this chapter and then refer back to it as you need to in the chapters to come.)

(*Additional Note:* Reference will be made in this chapter to concepts and instructions which have not yet been covered. This is necessary to explain the various advanced addressing modes. This method of organizing the text has the great advantage of placing all necessary information regarding addressing modes in two easy-to-locate chapters.)

21-1 ADVANCED ADDRESSING MODES

Some addressing modes which will be described in this chapter use a multistep process to find the address of the data or the next instruction to be executed. There may be one or more intermediate addresses, but the final address at which the data or instruction is to be found will be referred to as the *effective address*.

There are three fundamental advanced addressing modes, although some microprocessors also feature variations of these three.

Relative Addressing

Relative addressing is a mode in which your destination is described relative to where you are now. You aren't directed to an absolute memory location but rather to an address higher or lower than where you are now.

This form of addressing is not used to describe where to find data but rather where the program should find its next instruction. But let's back up just a bit.

In an earlier chapter we described the program counter and its function (the 8086/8088 uses the term *instruction pointer* instead of program counter). It keeps track of the next memory location to be accessed. Normally the locations are taken in order. The microprocessor gets an instruction, goes to the next byte in memory to get the next instruction or data, then to the next, and so forth. Sometimes, however, we need to "jump" or "branch" to a different area in memory to get our next instruction, for example, when we want to repeat a section of the program. (This saves time compared to writing a portion of a program many times if it is to be executed many times.)

Relative addressing involves 2 bytes (on 8-bit microprocessors). The first is the op code for the jump or branch instruction. The second byte tells how far and in what direction the microprocessor should jump. The second byte is a signed binary number—that is, it can be positive or negative. If it's positive, the microprocessor jumps forward in memory (to a higher-numbered address). If it's negative, it jumps backward (to a lower-numbered address). There is a limit, however, to how far you can jump with this form of addressing. On 8-bit microprocessors the range is from -128_{10} to $+127_{10}$ bytes. On 16-bit microprocessors the range is from $-32{,}768_{10}$ to $+32{,}767_{10}$ bytes.

The next task is to determine exactly what point we start counting from. For example, if we tell the microprocessor to jump forward 10 memory locations, where do we start counting from? We must again look at the program counter.

```
0000 01   NOP
0001 01   NOP
0002 20   BRA $02
0003 02
0004 01   NOP
0005 01   NOP
0006 01   NOP
0007 01   NOP
0008 01   NOP
```

Note what is happening here. The **BR**anch **A**lways instruction causes the microprocessor to branch forward 2 places *from the next instruction* in memory! Thus the next instruction *to be executed* is at memory location 0006.

Fig. 21-1 An example of relative branching forward using the 6800/6808.

The program counter always points to the *next* memory location to be accessed. In the case of relative jumps, it points to the next instruction after the jump instruction.

We start counting from the memory location being pointed to by the program counter when the jump instruction is being executed. This memory location is *not* the location of the jump instruction itself, and it is *not* the byte after the jump instruction, but is the *next instruction* in memory, which is usually two memory locations *after* the jump instruction.

Let's look at an example. Refer to Fig. 21-1. The 6800/6808 has an instruction called BRA (**BR**anch **A**lways), which uses relative addressing.

The four-digit numbers in the left column are memory addresses. The two-digit numbers in the next column are op codes. The third column contains the assembly-language mnemonics. Memory location 0002 contains the op code 20, which is the op code for the BRA instruction. The next memory location, 0003, contains the number 02, which is the same 02 referred to in the BRA $02 instruction.

The NOPs are simply dummy instructions placed there, in this example, so that we have something to skip over when the branch is implemented. Again, memory address 0002 contains the op code for BRA, which is 20. Address 0003 contains the number of places we wish to move *relative to where the program counter will be while it's executing this instruction!* Since the program counter is always pointing to the next instruction in memory, it will contain 0004. $0004_{16} + 02_{16} = 0006_{16}$. This is the next instruction to be executed.

Now let's try branching backward. Figure 21-2 shows an example.

At this point a review of 2's-complement negative numbers may be in order. Remember the odometer? Let's look at it again, in decimal first.

```
0000 01   NOP
0001 01   NOP
0002 01   NOP
0003 01   NOP
0004 20   BRA $FC
0005 FC
0006 01   NOP
0007 01   NOP
0008 01   NOP
```

This program branches backward 4 places from address 0006. This is because FC_{16} is the 2's-complement hexadecimal number for -4_{16}. The NOP at memory location 0002 will be the next instruction to be executed.

Fig. 21-2 An example of relative branching backward using the 6800/6808.

Negative 2's-Complement Numbers

Let's say you buy a brand-new car and the odometer reads 00,000. Now suppose your odometer rolls forward if the car drives forward, and rolls backward if the car drives backward. Let's drive backward from 00,000.

00,000
99,999
99,998
99,997
99,996

We could say that driving backward is like creating negative numbers: 99,999 is 1 mile less than 00,000. What's 1 less than 0? Minus one, of course. 99,998 is 2 miles less than 00,000. What's 2 less than 0? Minus two is. Let's look at some odometer readings from driving backward and their negative equivalents, along with some odometer readings from driving forward and their positive equivalents.

00,003	+3
00,002	+2
00,001	+1
00,000	0
99,999	−1
99,998	−2
99,997	−3
99,996	−4

Now let's show the same situation with a 1-byte hexadecimal odometer.

03	+3
02	+2
01	+1
00	0
FF	−1
FE	−2
FD	−3
FC	−4

Now look at Fig. 21-2 again. Do you see where the FC came from? It's −4.

What if you had to have a negative number like -40_{10}? Counting backward in hexadecimal would require too much time. There are several options. First, experiment with your calculator. Most scientific calculators now convert numbers back and forth between decimal, binary, octal, and hexadecimal. Many even do calculations in all number bases. Try entering -4_{10} and converting it to hexadecimal. If the calculator handles negative conversions, you'll get many F's and a C at the end. Simply ignore all the leading F's and use just the last two digits, the final FC.

If your calculator does conversions between decimal and hexadecimal but won't handle negative numbers, you can use another technique. A two-digit hexadecimal number is made up of 8 binary bits, each representing a power of 2. Find 2^8 and then subtract the number you wish to make negative. In the case of −4, for instance, take $2^8_{10} - 4_{10} = 252_{10}$. Now convert 252_{10} to hexadecimal; it should be FC. (To do the same thing with a 16-bit number, use 2^{16} instead of 2^8.)

Or, should no calculator be handy at the time, use the technique described in Chap. 6, that of taking the 2's complement of the number you wish to make negative. In the case of −4 it looks like this:

	0000 0100	+4
	1111 1011	1's complement (invert all bits)
+	1	add 1
	1111 1100	2's complement for −4
	↓ ↓	
	F C	converted to hexadecimal

Indirect Addressing

Indirect addressing is an addressing mode in which the *data* does not appear after the op code (as in immediate addressing), nor does *its memory location* appear after the op code (as in direct addressing), but rather a memory location follows the op code, and in this location is *another* address where the data may be found. It's like finding the address of an address. (*Indirect addressing* is indeed a fitting name.)

There are two basic types of indirect addressing: absolute indirect addressing and register indirect addressing. The 6502 uses absolute indirect addressing. The 8080/8085/Z80 uses register indirect addressing. The 8086/8088 uses register indirect addressing for data and program indirect addressing for jumps (which we'll study later). The 6800/6808 has no indirect addressing (indirect addressing was added to the 6809).

Let's look at an example of this addressing mode and then develop the topic further in the Specific Microprocessor Families section of this chapter. The 6502 has an instruction which looks like this

```
JMP ($aaaa)
```

which means **JuMP** indirect (indicated by the parentheses) to the address indicated by aaaa. If the address were 1000_{16}, it would be written as

```
JMP ($1000)
```

This tells us that at memory location 1000 and 1001 we can find the address the microprocessor should jump to. The address found at these two locations is loaded into the program counter. (It takes two locations because addresses in the 6502 are 16 bits wide but memory locations are only 8 bits wide.)

Indexed Addressing

Indexed addressing involves using a register called an *index register*, with a number called an *offset*, to calculate the address where the data is located. Let's look at an example using the 6800/6808.

One version of the 6800/6808's load accumulator A instruction looks like this

LDAA $ff,X

which means

> **L**oa**D** Accumulator **A** with the value in the memory location found by adding the contents of the X register to the hexadecimal offset ff.

For example, if the X register contains the number 1000_{16} and the instruction is written as

LDAA $22,X

we calculate the address where the data is located in this way

$$X + ff = \text{address}$$
$$1000_{16} + 22_{16} = 1022_{16}$$

The microprocessor then goes to address 1022 and places a copy of its contents in accumulator A.

You might be curious as to why we would want an addressing mode like this. One reason is its usefulness in accessing individual pieces of data in a data table. The index register can be incremented (increased by 1) or decremented (decreased by 1) easily, allowing the programmer to access each item in the table.

The 6502 microprocessor has two index registers, the X register and the Y register, and it has six different types of indexed addressing! The 6800/6808 has only one index register, the X register, with only one type of indexed addressing. The 8080/8085 has no index registers at all (the Z80 has two, X and Y) and has no indexed addressing mode. The 8086/8088 has two index registers, the source index and the destination index, and has several types of indexed addressing.

Specific Microprocessor Families

Go to the section which discusses your particular microprocessor.

21-2 6502 FAMILY

The 6502's numerous addressing modes make it unusual among 8-bit microprocessors. It has 13 different addressing modes. Allow us to offer a few words of encouragement at this time.

First, don't expect everything to make sense in the beginning. It takes time before all these new concepts become clear and you feel comfortable with them. Incidentally, the subject of addressing modes is the *only* difficult aspect of the 6502. In fact, the 6502 has the fewest different *instructions* of any of the 8-bit microprocessors—only 56 (the 6800/6808 has 107; the 8080/8085 has 246).

Relative Addressing

The relative addressing mode occurs in only one category of 6502 instruction, the Conditional Jump (Branch) category. Look at that section of the Expanded Table of 6502 Instructions Listed by Category. No other category uses this type of addressing, and this category uses no other type of addressing.

The subject of branching is coming in a later chapter, but it is necessary to discuss branching instructions for a moment to continue our coverage of the relative addressing mode.

The status register is where the 6502's flags are located. They keep track of certain events. If the result of the last calculation were 0, for instance, the zero flag bit would contain a 1. If we wanted to know whether the last result was a 0, we would check the zero flag. A 1 would mean yes, and a 0 would mean no. If we wanted the program to perform one action if the result of the last operation was a 0, and another if the result of the last operation was not a 0, we would write our program so that it would check the zero flag.

Let's look at the BEQ instruction. The assembler notation looks like this

BEQ $rr

which means

> **B**ranch rr bytes from where the program counter is now and do what it says to do there if the result of the last operation was **EQ**ual to 0.

You'll notice that the Operation column of the instruction table has a shorter version of that description.

Let's look at a program fragment. Refer to Fig. 21-3.

After the BEQ instruction and its operand in locations 0007 and 0008 have been fetched, the program counter will have already incremented to 0009, which is where we start counting for the branch (jump).

```
...
0005 EA    NOP
0006 EA    NOP
0007 F0    BEQ  $03
0008 03
0009 EA    NOP ──┐
000A EA    NOP   │
000B EA    NOP   │
000C EA    NOP ◄─┘
000D EA    NOP
000E EA    NOP
...
```

Memory location 0007 contains F0, the op code for BEQ. The next location, 0008, contains 03_{16}, which is the distance the program is going to jump relative to where the program counter is at the end of this instruction. Remember: this jump occurs only if the last operation set the zero flag (which we are assuming for this example).

Fig. 21-3 6502 example of relative addressing. *Note:* The zero flag is assumed to be set from a previous operation.

Refer back to the New Concepts section of this chapter to see how a backward branch or jump would work and how to use 2's-complement negative numbers.

Indirect Addressing

There is only one 6502 instruction which uses the indirect addressing mode. That instruction is the JMP instruction, which is found in the Unconditional Jump Instructions category in the Expanded Table of 6502 Instructions Listed by Category.

This particular instruction can be used with two different addressing modes. In the absolute addressing mode, the microprocessor simply jumps to the specified address. When written this way

JMP $aaaa

it means

> **JuMP** to address $aaaa_{16}$ and continue program execution from that point.

In the indirect addressing mode, however, it would be written this way

JMP ($aaaa)

and would mean

> **JuMP** to the address which can be found at memory location aaaa and aaaa + 1.

```
0000 6C    JMP ($0004)
0001 04
0002 00
0003 EA    NOP
0004 1F◄────────low byte ──┐
0005 01◄────────high byte ─┘
...
011F next instruction  ◄──
0120
...
```

This is where the effective address is being stored. 011F is placed in the program counter.

This is the location of the next instruction to be executed.

Fig. 21-4 Example of 6502 indirect addressing mode.

This would load the contents of memory location aaaa into the low byte of the program counter (PC_L). The contents of memory location aaaa + 1 would be loaded into the high byte of the program counter (PC_H). (This reverse low-byte/high-byte order is normal for the 6502.)

Let's look at an example. If you refer to Fig. 21-4, you will see that the instruction

JMP ($0004)

does not mean that address 0004 is where the program is supposed to jump to, but rather that location 0004 contains the address it's supposed to jump to.

Indexed Addressing

Indexed addressing is the subject of the remainder of this 6502 section. There are four basic indexed addressing modes, and two more which use a mixture of indexed and indirect addressing.

It should be noted that while the 6502 family has a great number of addressing modes which use the index registers, it is the only family which has index registers which are only 8-bits wide. The 6800/6808, Z80, and 8086/8088 all have 16-bit index registers. Keep this in mind if you use the 6502 in addition to one of the other microprocessors.

Zero Page,X and Absolute,X Addressing

You may remember from the New Concepts section of this chapter that the 6502 has two index registers, X and Y, and six different forms of indexed addressing. Here are the first two of the six forms. The difference between these two forms is the range of addresses possible.

These first two forms, and the next two, are so similar to the description in the New Concepts section that you will probably have little difficulty understanding them. If you don't remember how the indexed form of addressing works, go back and reread the description now.

Look in the Data Transfer Instructions category of the Expanded Table of 6502 Instructions Listed by Category. We will use the LDA instruction to illustrate the *zero page,X* and *absolute,X* addressing modes.

First notice the Assembler Notation column for the zero page,X and absolute,X forms of the LDA instruction. For these two the assembler notation is

LDA \$ff,X ← zero page,X
LDA \$ffff,X ← absolute,X

In both cases the offset (ff or ffff) is a hexadecimal number which is going to be added to the value in the X register. The sum of these two values provides the address of the data which is to be loaded into the accumulator.

For example, if the X register contained the hexadecimal number 10, the instruction

LDA \$034E,X

would add those two values,

$$034E_{16} + 10_{16} = 035E_{16}$$

and place a copy of the contents of memory location $035E_{16}$ in the accumulator.

When zero page,X addressing is used, the offset (the number being added to the X register) is two hex digits wide and the X register is also two hex digits wide. Two hex digits can address memory locations only in page 0 (00_{16} to FF_{16}). When this addressing mode is used, it is assumed that the data is somewhere in page 0. *If the sum of the offset and the X register is greater than* FF_{16} *then the most significant digit is truncated and only the first two digits are used!* For example, if the X register contained FF, the instruction

LDA \$04,X

would add the offset to the X register

$$04_{16} + FF_{16} = 103_{16} \qquad \textit{(The 1 will be dropped.)}$$

so the data will be retrieved from location 03_{16}! Numbers larger than FF_{16} *wrap around* to the beginning of page 0.

When absolute,X addressing is used, the offset is a four-digit hexadecimal number ranging from 0000_{16} to $FFFF_{16}$. This allows the data to be located anywhere in the entire 6502 address range. If the sum of the offset and the X register exceeds $FFFF_{16}$, then the microprocessor again performs a *wraparound* back to 0000_{16}.

Zero Page,Y and Absolute,Y Addressing

Notice in the Data Transfer Instructions section of the Expanded Table of 6502 Instructions Listed by Category that the LDX instruction uses both *absolute,Y* and *zero page,Y* addressing. These work exactly the same as absolute,X and zero page,X, except that they use the Y register instead.

The absolute,X, absolute,Y, and zero page,X addressing modes are used by many 6502 instructions. Zero page,Y addressing is used by only two instructions, however—LDX and STX.

Indirect Indexed Addressing

Indirect indexed addressing, as the name implies, is a mixture of indirect addressing and indexed addressing. Notice that the word "indirect" is first, and the word "indexed" is next. In this form of addressing, the indirect part of the address calculation is accomplished first; then the indexing is taken into consideration.

Refer to this form of the LDA instruction in the Data Transfer Instructions section of the Expanded Table of 6502 Instructions Listed by Category. Remember the word order—*indirect*, then *indexed*; and notice the assembler notation—LDA (\$aa),Y.

To understand the assembler notation for this form of addressing, it helps to remember one of the rules of algebra. In algebra, expressions are read from left to right, and when parentheses are encountered, they are read from the inside to the outside. Let's look at an example.

LDA (\$aa),Y

The \$aa stands for a two-digit hexadecimal address. Because only two digits are allowed, this address must be between 00_{16} and FF_{16}. At this address, and the one following it (aa and aa + 1), is a 16-bit address stored in reverse low-byte/high-byte order. This address is then added to the Y register to produce the actual (effective) address where the operand (data) is stored. Notice that we worked our way from left to right and from the inside toward the outside as we analyzed this instruction.

For example, let's say that

$$\begin{aligned} \text{Y register} &= 10_{16} \\ \text{memory location 2D} &= 00 \\ \text{memory location 2E} &= \text{C0} \end{aligned}$$

If we write the instruction

LDA (\$2D),Y

the microprocessor will look in addresses 2D and 2E and use their contents to form another address, C000. It will then take the number $C000_{16}$ and add it to the Y register:

$$C000_{16} + 10_{16} = C010_{16}$$

$C010_{16}$ is where the data is actually stored.

To summarize,

LDA ($aa),Y

means

> **LoaD** the **A**ccumulator with the contents of an address formed by adding the contents of memory location aa and aa + 1 (low-byte/high-byte order) to the Y register.

Indexed Indirect Addressing

This form of addressing is also a mixture of indexed and indirect addressing, but it is the reverse of the previous indirect indexed addressing.

It will be helpful here, as in the previous explanation, to think of how algebraic expressions are written, from left to right and from the inside to the outside.

We will again use the LDA instruction. Look at the indexed indirect form of this instruction. In the Assembler Notation column it appears as

LDA ($ff,X)

In this form of addressing, the microprocessor takes the two-digit offset (ff_{16}) and then adds it to the value found in the X register. (If the sum of ff and X is greater than FF_{16}, the sum will be truncated so that only the two least significant digits remain.) The address formed by the sum of ff and the X register and the following address contain the effective address stored in reverse low-byte/high-byte order.

Let's try an example. If

$$\text{X register} = 10_{16}$$

and we write the instruction

LDA ($11,X)

then the microprocessor will add 11_{16} to the X register

$$11_{16} + 10_{16} = 21_{16}$$

creating the address 21_{16}. *However, this is not where the operand (data) is stored!* At addresses 21_{16} and 22_{16} the effective address is stored in reverse low-byte/high-byte order. So if

memory location 21 = 00
memory location 22 = C0

then the address $C000_{16}$ is created. *Memory address $C000_{16}$ does contain the operand!*

To summarize,

LDA ($ff,X)

means

> **LoaD** the **A**ccumulator with the contents of the memory location *pointed to* by the contents of memory location ff + X and ff + X + 1.

21-3 6800/6808 FAMILY

The 6800/6808 microprocessor has only two addressing modes which must be covered in this chapter—relative addressing and indexed addressing. (The 6800/6808 has no form of indirect addressing.)

Relative Addressing

The 6800/6808 uses relative addressing with all of its branch instructions. These fall into three instruction categories, Unconditional Jump (Branch) Instructions, Conditional Jump (Branch) Instructions, and Subroutine Instructions. This form of addressing works exactly as described in the New Concepts section of this chapter. (In fact, the 6800/6808 was used as our example in that section.)

Let's go over this mode again by using the program fragment in Fig. 21-5.

Since 02_{16} is a positive number, we branch forward by that many spaces *starting with the memory location which will be pointed to by the program counter after the BRA instruction and its operand have been fetched.*

It is important to remember that the BRA operand is a 2's-complement signed binary number and thus can be either negative or positive within a range from $+127_{10}$ to -128_{10}. A negative number indicates a backward branch, and a positive number indicates a forward branch.

Indexed Addressing

The subject of indexed addressing, as discussed in the New Concepts section, was illustrated by using the 6800/6808. We present that information again here for your convenience.

```
0010 20    BRA $(02)
0011 02
0012 01    NOP
0013 01    NOP
0014 01    NOP
0015 01    NOP
```

Fig. 21-5 An example of relative addressing.

One version of the 6800/6808's load accumulator A instruction looks like this

LDAA $ff,X

which means

> **LoaD** Accumulator **A** with the value in the memory location found by adding the contents of the X register to the hexadecimal offset ff.

For example, if the X register contained the number 1000_{16} and the instruction were written as

LDA $22,X

we would calculate the address where the data was located in this way:

$$X + ff = \text{address}$$
$$1000_{16} + 22_{16} = 1022_{16}$$

We would go to address 1022 and place a copy of its contents in accumulator A.

21-4 8080/8085/Z80 FAMILY

The 8080/8085 microprocessor is easier to learn in some respects than the other 8-bit microprocessors. One reason is that the 8080/8085 has the fewest number of addressing modes. And while the 8080/8085 has the most number of different instructions (246, in contrast to the 6502 with only 56 and the 6800/6808 with 107), each instruction works with only one addressing mode (in contrast to the 6502, which has some instructions which operate in as many as eight different addressing modes).

As we talk about the 8080/8085/Z80 family, you should remember that although the Z80 is treated as a part of the 8080/8085 family in this text, it is a significantly enhanced member of the 8080/8085 family. It has many multibyte instructions and several addressing modes which the 8080/8085 does not have. At this time we will cover only those aspects of the Z80 which it has in common with the 8080/8085.

Register Indirect Addressing

The only advanced addressing mode which the 8080/8085 has is register indirect addressing. Although indirect addressing was covered in the New Concepts section of this chapter, register indirect addressing was not covered since it is a variation of indirect addressing which, among the 8-bit microprocessors, is unique to this family.

Register indirect addressing uses the contents of a 16-bit register pair (most often the HL register pair) as a pointer for the operand.

For example, refer to the Data Transfer Instructions section of the Expanded Table of 8085/8080 and Z80 (8080 Subset) Instructions Listed by Category and look at the MOV A,M [Z80 = LD A,(HL)] instruction. (The MOV A,M instruction is the eighth instruction in this category.) The 8085 form is written

MOV A,M

which means

> **MOV**e to the **A**ccumulator the number found at the **M**emory location pointed to by the HL register pair.

The Z80 form is written

LD A,(HL)

which means

> **LoaD** the **A**ccumulator with the number found at the memory location pointed to (parens) by the **HL** register pair.

which says the same thing the 8085 form did but in different words.

To give an example, if

$$\text{register pair HL} = 1000_{16}$$

and you entered MOV A,M [Z80 LD A,(HL)] into your assembler, the microprocessor would go to memory location 1000_{16} and place a copy of its contents in the accumulator.

There are a few occasions when either the BC or the DE register pair is used instead of the HL pair. You may want to page through the Expanded Table of 8085/8080 and Z80 (8080 Subset) Instructions Listed by Category to see some of the instructions that use this addressing mode.

21-5 8086/8088 FAMILY

Because the 8086/8088 is a 16-bit microprocessor, it uses a greater number of addressing modes than the 8-bit microprocessors, and the modes are more complex. We covered the basic 8086/8088 addressing modes in a previous chapter and will try to give a simple, yet sufficiently complete description of each of the advanced modes at this time.

Register Relative Addressing

Register relative addressing uses two numbers, added together, to determine the address of the source. This form of addressing is especially useful in addressing arrays (tables of data).

Some examples of register relative addressing using the format used by DEBUG (an MS-DOS utility which helps to "debug" programs and includes an assembler and disassembler) are

```
MOV     AL,[BX+0100]
MOV     AX,[DI+0200]
MOV     [SI+0500],CL
MOV     [BP+20],BL
MOV     DI,[BX+0400]
```

Figure 21-6 illustrates how this form of addressing works. The instruction

```
MOV     AL,[BX+0100]
```

is used as an example. Notice first the brackets surrounding the BX+0100. This is required by DEBUG and indicates that the two numbers added together (the value in register BX + 0100_{16}) will point to the location of the data being moved to AL.

We can use the number in the source (0100) to indicate the location of the beginning of the table. The value in the register indicated in the source operand tells us which item in the table is the desired data item.

Notice in Fig. 21-6 that 0100 is the beginning of the table and that 03 (the value in BX) is the data item we need. We need the fourth item in the table starting at address 0100. The contents of memory location 0103 (E3) have been copied to register AL.

It is important to remember that we have added the displacement (0100) to the value in the indicated register (BX) to form an address (0103) *in the current data segment!*

Program Relative Addressing

Program relative addressing is used with JMP and CALL instructions. This mode specifies where the next program instruction is located without using absolute addressing. This allows you to write relocatable assembly-language programs.

Figure 21-7 shows an 8086/8088 instruction which is *not* using program relative addressing. (We'll show you program relative addressing in a moment.) This figure is using direct addressing. We have listed the same line of code three times.

The first line shows the code as it appeared on our computer after being disassembled by DEBUG.

The second line shows DEBUG's disassembly broken into its major components. The *address* is the address of the current memory location. We did not type the address; DEBUG picked that address for us. The *machine code* contains the actual bits which will tell the 8086/8088 what to do. The *assembly language* is what we typed in when using DEBUG.

The third line shows even greater detail. Notice that the code segment the program is to jump to (8888) and location within the segment (0100) are actually contained in the machine code (the bytes are reversed).

Figure 21-8 shows a JMP instruction written using DEBUG which does use program relative addressing, instead of direct addressing as in Fig. 21-7.

Line one shows the information as it appeared on our screen when disassembled by DEBUG.

Line two illustrates the major components of the disassembly. We typed in the assembly language, and DEBUG provided us with the machine code.

The third line shows the components in greater detail. The most interesting fact is that the address we specified as our target address is not the same as the address DEBUG generated. Let's see what DEBUG did.

The JMP op code, EB, is in memory location 0100 as indicated in the "location within segment" portion of the

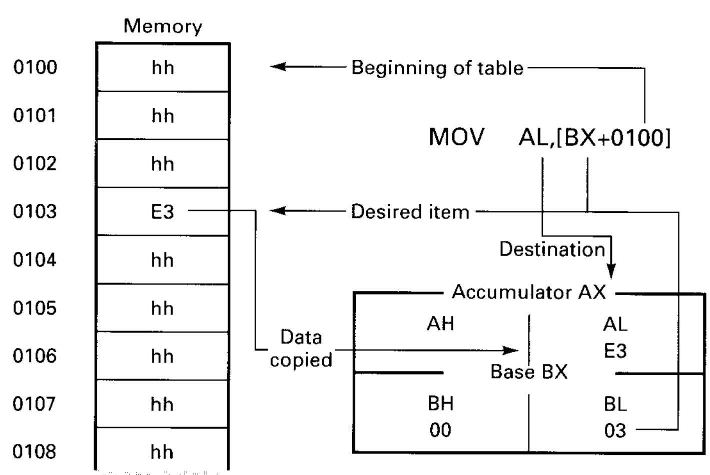

Fig. 21-6 Register relative addressing.

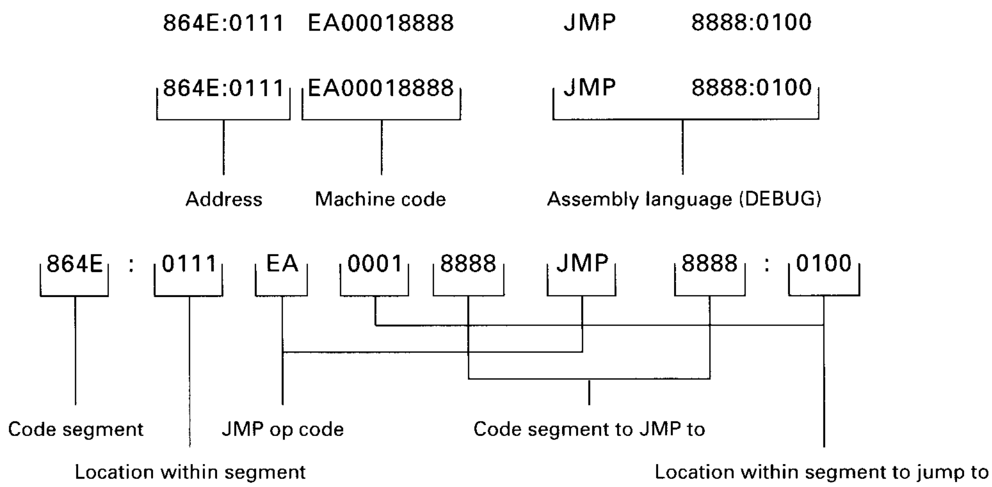

Fig. 21-7 Direct addressing.

line. That means the next *byte*, 0E, is in address 0101. (DEBUG does not show the 0101.) Therefore, the next *instruction* is at memory location 0102.

How far is it from memory location 0102 to our target address of 0110? Remember, these are *hexadecimal* numbers.

$$0110_{16} - 0102_{16} = E_{16}$$

To reach the target address of 0110, the microprocessor will have to jump forward a number of spaces from the point (the instruction) at which the instruction pointer is pointing when this instruction is executed; the number of spaces is E_{16}. The 0E in Fig. 21-8 was calculated by DEBUG as the position of our target *relative* to where the instruction pointer will be when this instruction is being executed.

Relative addressing tells the microprocessor how far to jump forward or backward from the instruction after the JMP instruction. The next instruction is used because the instruction pointer always points to the *next* instruction *to be* executed.

A positive relative address signifies a jump forward; a negative relative address signifies a jump backward.

Register Indirect Addressing

Register indirect addressing uses a register to point to a memory location rather than specifying that location directly. BX, BP, SI, and DI are used as pointers. All of them except BP point to locations in the *data segment*; BP points to a location in the *stack segment*. The registers can point to either the source or the destination operand.

An assembly-language instruction which uses indirect addressing is shown in Fig. 21-9.

The format of the instruction line in bold print in Fig. 21-9 is the format that DEBUG uses. (The code segment on your computer will probably not be the same as the one shown in Fig. 21-9.)

Most of the different components of the instruction line in bold have been identified in the figure. [BX] is labeled as the source. The brackets around BX indicate that the operand is not the contents of BX; rather the operand will be found at the address *pointed to* by BX.

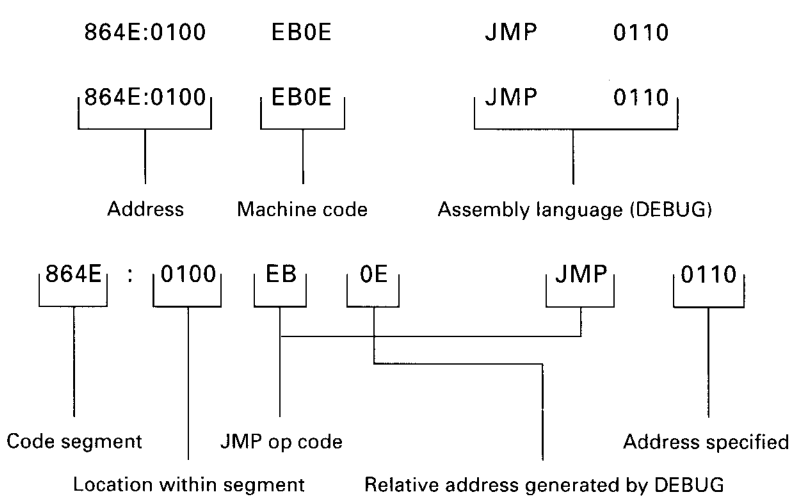

Fig. 21-8 Program relative addressing.

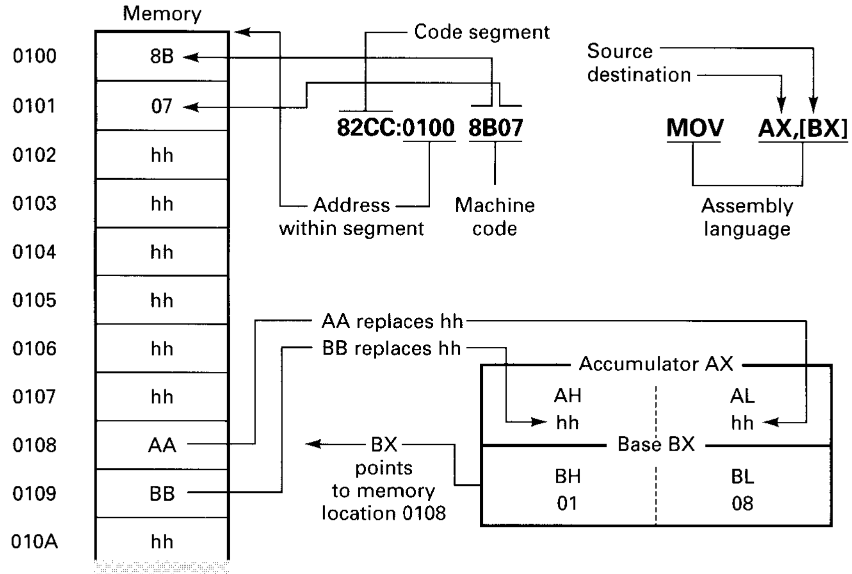

Fig. 21-9 Register indirect addressing.

If you look at the contents of BX, you will see the value 0108. That means that the actual operand is in memory location 0108. In this case we are moving a 16-bit word rather than an 8-bit byte. Since it takes two memory locations to hold a whole word, we will find the operand in locations 0108 and 0109. The 16-bit values in locations 0108 and 0109 are copied into AX, which is the destination.

Figure 21-10 is a screen dump of Fig. 21-9 obtained by using DEBUG.

In the first line

-d 100 10f

tells DEBUG to "dump" the contents of memory locations 0100_{16} through $010F_{16}$ to the screen so that we can see them. The hyphen halfway through the memory dump separates those 16 bytes into two sections to make the display easier to read. We have shown the contents of locations 0100 and 0101 in bold because they are the object code for the

MOV AX,[BX]

instruction. The contents of memory locations 0108 and 0109 are in bold because they are the locations being pointed to by register BX.

The -r tells DEBUG to display its registers. We have shown the contents of registers AX and BX in bold in this illustration because they are the two registers being referred to in this example.

The -t is the DEBUG trace command. This tells DEBUG to execute the next instruction and then stop. The next instruction is

MOV AX,[BX]

Notice the contents of register AX after the trace command. The contents of memory locations 0108 and 0109 have been copied to register AX as was illustrated in Fig. 21-9.

Take some time to compare Figs. 21-9 and 21-10. You may notice that the code segments in the two figures differ. That's because we created the figures on two different days, and the memory arrangement in our computer was not exactly the same both days. This is normal and something you should expect to see as you try these figures and

```
-d 100 10f
9029:0100  8B 07 00 00 00 00 00 00-AA BB 00 00 00 00 00 00   ................
-
-r
AX=0000  BX=0108  CX=0000  DX=0000  SP=FD6E  BP=0000  SI=0000  DI=0000
DS=9029  ES=9029  SS=9029  CS=9029  IP=0100   NV UP EI PL NZ NA PO NC
9029:0100 8B07          MOV     AX,[BX]                          DS:0108=BBAA
-
-t

AX=BBAA  BX=0108  CX=0000  DX=0000  SP=FD6E  BP=0000  SI=0000  DI=0000
DS=9029  ES=9029  SS=9029  CS=9029  IP=0102   NV UP EI PL NZ NA PO NC
9029:0102 0000          ADD     [BX+SI],AL                       DS:0108=AA
```

Fig. 21-10 DEBUG screen dump of Fig. 21-9.

examples on your computer. Everything will be the same except the code segment, and that will almost never match ours.

Again, in the case of register indirect addressing, at least one of the operands is in a memory location pointed to by the value in a register (BX, BP, SI, DI).

Program Indirect Addressing

Program indirect addressing is used by CALL and JMP instructions. It allows the memory location where the program is to fetch its next instruction to be stored in a register, in a memory location pointed to by a register, or in a memory location pointed to by a register with a displacement.

Normally instructions are stored in memory in sequential order, with the microprocessor fetching one after another. When a JMP instruction uses direct addressing, the address the microprocessor is to jump to is placed immediately after the jump instruction itself.

A CALL instruction causes the microprocessor to go to another area of memory where a subroutine is stored, execute the subroutine, and then return to where it left off before it began the subroutine. The CALL, like the JMP instruction, can use direct addressing and place the location of the subroutine immediately after the CALL instruction.

When either the CALL or the JMP instruction uses one of the 16-bit registers (AX, BX, CX, DX, SP, BP, SI, or DI), it means that the destination for the JMP or CALL is located in that register. For example

```
JMP     AX
```

instructs the microprocessor to look in register AX and jump to the location stored in AX. That is, AX "points" to the correct memory location.

When either the JMP or the CALL instruction uses a register placed inside brackets ([BX], [BP], [SI], or [DI]), it means that register contains an address, and that address contains another address, which is the actual destination for the JMP or CALL. For example,

```
JMP     [BX]
```

instructs the microprocessor to look in register BX. Let's say BX = 0200. Next the microprocessor looks at address 0200 and 0201. There it will find another address which is its actual destination.

When either the JMP or the CALL instruction uses one of the registers with brackets ([BX], [BP], [SI], or [DI]) and a displacement, the microprocessor is instructed to add the displacement to the contents of the register, forming an address, and then to look at that address and get another address, which is the actual destination. For example

```
JMP     [BX+0100]
```

instructs the microprocessor to add 0100_{16} to the value in BX. Let's say that BX contains 0500_{16}.

$$0500_{16} + 0100_{16} = 0600_{16}$$

The microprocessor now looks in addresses 0600 and 0601 and gets another address. This is the destination address where the next instruction is to be fetched or the subroutine begins.

Base plus Index Addressing

Base plus index addressing also uses the concept of calculating the address where data is located rather than using direct addressing, which explicitly states where the data is located.

When *base plus index* addressing is used, the contents of one of the *base registers* (either BX or BP) and the contents of one of the *index registers* (either SI or DI) are added to calculate the address of the operand. For example,

```
MOV     AX,[BX+DI]
```

instructs the microprocessor to add the value in register BX to the value in register DI. This sum is the location of the data which is to be copied into register AX. This is illustrated in Fig. 21-11.

Base plus index addressing is useful for working with tables of data. The base register (BX or BP) can point to the beginning of the data table. The index register (SI or DI) can then point to the specific piece of data within the table. The program can then increment or decrement the index register to point to the next or preceding piece of data in the table.

Base Relative plus Index Addressing

Base relative plus index addressing combines the features of base plus index addressing and register relative addressing. Examples of base relative plus index addressing are

```
MOV     DX,[BX+SI+10]
MOV     [BX+DI+20],AX
```

In the first example, the microprocessor would add the values in registers BX and SI and the number 10_{16}. The sum is the memory location of the data which is to be copied into register DX.

In the second example, the microprocessor would copy the contents of register AX to a memory location whose address would be calculated by finding the sum of 20_{16}, the value in register BX, and the value in register DI.

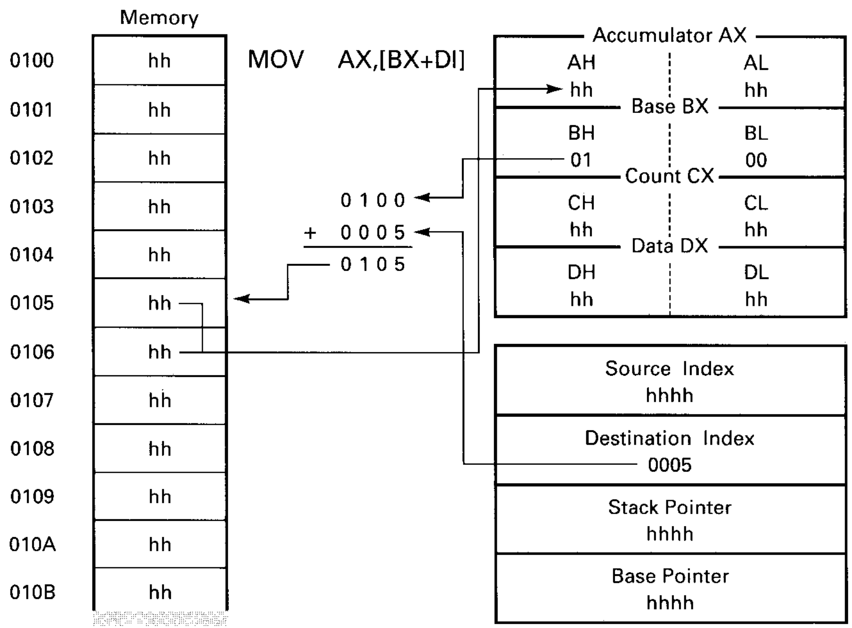

Fig. 21-11 Base plus index addressing.

This addressing mode is useful for working with two-dimensional data tables. The displacement (the number) can point to the beginning of the table, since this is the constant value. The base register (BX) can point to the first of the two dimensions (for example, a record in a file or an area in a data table). The index register (SI or DI) can then point to the specific memory location containing the desired data (for example, a field within a record within a file, or a specific piece of data in a data table). The program can then increment or decrement the base register to point to the next or previous record in the file and increment or decrement the index register to point to the next or previous field in the record.

22

BRANCHING AND LOOPS

In this chapter we'll study branching and loops. A branch instruction (or jump instruction) causes the program to "skip" forward or backward and to execute instructions from this new memory location.

A loop involves executing a series of microprocessor instructions and then branching backward to repeat the same set of instructions. This "loop" is finally broken, or exited from, when some condition is met.

The previous chapter introduced you to the remainder of the addressing modes (the more difficult ones) which had not been covered in the earlier chapter on addressing. From this point on we will use many of the different types of addressing modes available to each microprocessor. You should refer back to either of the chapters on addressing whenever necessary.

New Concepts

We'll study unconditional branching (or jumping) first; then we'll discuss the slightly more difficult subject of conditional branching. Later we'll look at loops and how to control them through the use of conditions and counters.

22-1 UNCONDITIONAL JUMPS

The simplest type of branch or jump is an unconditional one. This means that the program will jump to the indicated memory location every time this part of the program is run. The jump can be forward or backward.

With unconditional jumps, most of the microprocessors featured in this text use some form of direct or indirect addressing to indicate where the next instruction should be fetched from. The exceptions to this are the 6800/6808, which can also use relative addressing, and the 8086/8088, which also uses relative addressing, at least for jumps within a single memory segment.

To jump forward, you simply indicate the address of the next instruction to be executed. We'll look at exactly how the different addressing modes are used in the Specific Microprocessor Families section of this chapter.

22-2 CONDITIONAL BRANCHING

Conditional branching, like unconditional branching, causes program execution to continue with an instruction which is not the next instruction in memory. We either skip forward or backward from where we are now. Whether or not program execution does skip depends on a certain condition.

The microprocessor determines whether a condition is true or not true by the condition of the flags. To be able to predict whether or not a condition will be true when the microprocessor reaches the point at which the conditional branch occurs, one must know how the preceding instructions affect the flags. How each instruction affects each of the flags is shown in several of the instruction-set tables for each microprocessor.

When we branch *forward*, we have the effect of skipping over a certain number of instructions, if certain conditions exist, and not skipping over them if those conditions do not exist. Figure 22-1 shows a generic example of branching forward.

When we branch *backward*, the instructions between where we branched from and where we branched to are executed again. They could in fact be executed many times. This creates a loop which will not be exited from until some condition is met. Figure 22-2 shows a generic example of branching backward.

In Fig. 22-2, we are not branching backward from address 0009 because of the instruction at that memory location. Rather, we are branching backward because of the instruc-

```
0000        INSTRUCTION
0001        DATA
0002        INSTRUCTION
0003        DATA
0004        INSTRUCTION
0005        INSTRUCTION
0006        INSTRUCTION
0007        COND JUMP
0008        00A0
.
.
.
.
.
00A0        INSTRUCTION
00A1        DATA
00A2        INSTRUCTION
00A3        END
```

This area is skipped over if condition exits. If condition doesn't exist, this area is not skipped over.

Fig. 22-1 Example of generic forward conditional jump.

tion at location 0007 and the address at location 0008. The arrow is drawn from location 0009 because that will be the instruction pointed to by the program counter or instruction pointer when the branch occurs. Remember, the instruction pointer or program counter points to the *next* instruction to be executed, not the one currently being executed.

22-3 COMPARE AND TEST INSTRUCTIONS

Many (but not all) microprocessor instructions affect the flags. The flags then tell something about the results of the instruction. There are instructions, however, *compare* and *test* instructions, which actually do nothing except affect flags.

For example, the arithmetic instructions actually accomplish some task, such as adding, subtracting, multiplying, or dividing, and also affect the flags depending on the result of the operation. Compare and test instructions, however, compare a register or memory location to another, to zero, or AND two registers, without producing any result or changing any register or memory location—that is, no answer is produced. The flags, however, respond just as if an answer had been produced. A conditional branch instruction can then check the flags and determine whether a certain condition is true or false and then branch or not branch accordingly.

22-4 INCREMENT AND DECREMENT INSTRUCTIONS

Sometimes you may want to repeat a section of your program a certain number of times. A register or memory location is used to count how many times the section has been repeated. This register or memory location being used as a counter can either count up (increment) to a certain value or count down (decrement) to a certain value. Since it is easy to test for the occurrence of zero (just check the zero flag), counters often start at a certain number and decrement to zero. When the counter reaches zero, we know how many times that section of the program has repeated.

This technique produces a loop and uses conditional branching in a way that is similar to that discussed in the last section, although the intent is a little different. In the last section we were talking about situations when you want to branch if an operation produces a certain result. In this section we are discussing situations when we simply want something to be repeated a certain number of times.

22-5 NESTED LOOPS

It's possible to nest loops one inside the other. Figure 22-3 shows what this looks like.

The operand immediately following the conditional branch instruction may not be the actual address to branch to but rather the value needed by some other form of addressing such as relative addressing.

Remember also that we do not branch from the memory location containing the conditional branch instruction; nor do we branch from the next address which determines where we branch to, but from the instruction after that.

In Fig. 22-3 you can see that an inner loop will be repeated until the conditions necessary for the program to "drop through" the bottom of the loop exist, in which case the program may go back to the beginning of the outer loop, depending again on the conditions which exist.

```
0000        INSTRUCTION
0001        DATA
0002        INSTRUCTION
0003        DATA
0004        INSTRUCTION
0005        INSTRUCTION
0006        INSTRUCTION
0007        COND JUMP
0008        0000
0009        INSTRUCTION
```

This area is repeated if certain condition exists. This area is not repeated if condition does not exist.

Fig. 22-2 Example of generic backward conditional jump.

```
0000   INSTRUCTION
0001   data
0002   INSTRUCTION
0003   data
0004   INSTRUCTION
0005   data
0006   INSTRUCTION
0007   data
0008   INSTRUCTION
0009   data
000A   CONDITIONAL BRANCH BACKWARDS
000B   0006
000C   INSTRUCTION
000D   data
000E   INSTRUCTION
000F   data
0010   CONDITIONAL BRANCH BACKWARDS
0011   0000
0012   INSTRUCTION
```

Fig. 22-3 Generic nested loops.

Specific Microprocessor Families

Let's look at each of our microprocessors' instructions to see how branching and loops are handled.

22-6 6502 FAMILY

The 6502 microprocessor family has a variety of instructions to handle unconditional jumps, conditional branching, comparing, incrementing, and decrementing. We'll look at several tasks and see how the 6502 microprocessor handles them.

You should enter each program into your computer or microprocessor trainer and single-step through it, watching the appropriate registers, memory locations, and flags to understand how each program works.

Unconditional Jumps

The forward unconditional jump using absolute addressing is easiest to understand. An example is shown in Fig. 22-4.

The program begins by loading the accumulator with FF_{16}. In a moment we are going to subtract another number from FF_{16}. First, however, we need to jump to the area of memory where the subtract instruction is. We have placed the subtract instruction several memory locations forward from this point to show, in a very simple manner, how the unconditional jump instruction operates.

The next instruction is our jump instruction. In the source code column of line 0004 the instruction

JMP MINUS

appears, which might be different from what you were expecting.

The instruction is saying to jump to a place called MINUS. To be able to jump to a place with a certain name is not a native ability of the 6502 microprocessor. Our assembler is making this possible. Line 0008 has the label *MINUS* in the label column. This is the place we want to jump to. Notice the address at the MINUS label. The address is 0348. Now look back at line 0004. In the op code column you see 4C, which is the op code for an unconditional jump. Then come the numbers 48 03. If you reverse those two sets of numbers, you have 0348. This is the memory location of the instruction labeled MINUS. If you use an assembler, you can use labels and the assembler will calculate the address for you. If you are hand-assembling these programs, you must enter the address as shown in the op code column, in the reverse low-byte/high-byte order. If you are using an assembler which does not allow labels, you will need to use the format shown in the 6502 tables. Namely,

JMP $0348

```
0001  0340                       .org $0340      ;beginning of code
0002  0340
0003  0340 A9 FF        START:   LDA #$FF        ;minuend
0004  0342 4C 48 03              JMP MINUS       ;forward unconditional jump
0005  0345 EA                    NOP
0006  0346 EA                    NOP             ;misc. instructions
0007  0347 EA                    NOP
0008  0348 38           MINUS:   SEC             ;prepare for subtraction
0009  0349 E9 EE                 SBC #$EE        ;subtrahend
0010  034B 8D 4F 03              STA ANSWER      ;store difference
0011  034E 00                    BRK             ;stop
0012  034F
0013  034F 00           ANSWER   .db     $00     ;memory area for answer
0014  0350                                       ;  (initialized to 00)
0015  0350                       .end
```

Fig. 22-4 Forward unconditional jump with the 6502 microprocessor.

After the jump instruction are several NOPs which could be other instructions or just unused memory in a particular microprocessor system.

Line 0008 is the next instruction to be executed. It sets the carry flag in preparation for the subtraction instruction. In line 0009 we subtract EE_{16} from FF_{16} (in the accumulator). In line 0010 we store the result of our subtraction in a memory location called *ANSWER*. Look at line 0013, labeled ANSWER. In the op code column are the initials .db. They stand for *define byte*. We are telling the assembler to reserve a memory location, namely, a single byte of memory, with the name ANSWER. The assembler is initializing the memory location ANSWER with a value of 0. Our program can then put any other number we wish in that location.

Notice also that the memory location of ANSWER is $034F_{16}$. In the op code column of line 0010 we see 8D 4F 03. 8D is the op code for storing the value of the accumulator in a certain memory location. If you reverse the order of 4F 03, you have 034F, which is the memory location of ANSWER. Again, the assembler made life simpler by figuring out where the next available memory location would be and setting aside that location for the ANSWER.

Finally, in line 0011 the program stops.

You should enter this program and single-step through it, making sure that everything works as described.

Conditional Branches

Now let's see an example of conditional branching. Figure 22-5 shows such an example.

In this program we are going to do several things differently from the way they were done in the last program. First, we are using a conditional jump or branch rather than an unconditional one. Second, we are branching backward rather than forward. Third, we are creating a loop by branching backward and repeating a section of the program. Finally, we are using a register as a counter to control how many times the loop repeats.

In line 0003 we place the number 3_{16} in the X register. This register *controls* how many times we will branch backward. In line 0004 we clear the Y register making it 00_{16} so that it can be used to *count* how many times the loop repeats.

Line 0005 marks the beginning of the loop; we have named that location *REPEAT*. In this line we increment the Y register since we are beginning to pass through the loop, in this case for the first time. The Y register is keeping track of how many times the loop is passed through. Line 0006 represents the fact that there could be many instructions inside the loop which are going to be repeated.

Line 0007 decrements (reduces by 1) the X register. The X register keeps track of how many times through the loop are remaining.

Line 0008 is where we meet our conditional branch instruction. BNE means **B**ranch if **N**ot **E**qual. Your first thought might be, "Not equal to what?" If you check the Expanded Table for the 6502, you'll see it is **B**ranch if the last result is **N**ot **E**qual to 0.

All the conditional branch instructions are influenced by the most recent instruction that affected the flag they check. In this case the zero flag is checked. What was the last instruction which sets or clears the zero flag? The DEX (**DE**crement **X** register) instruction. If the X register were reduced to 0, the zero flag would be set. Has the X register been reduced to 0? On this first pass through the loop, it gets reduced from 3 to 2. No, the X register is not equal to 0.

The branch instruction says, "**B**ranch if the last result is **N**ot **E**qual to 0." Clearly this is true: the last result is not 0, so we branch. Branch to where? We branch to the memory location known as *REPEAT*. Notice that the location called REPEAT, in line 0005, is memory location 0344_{16}. Now look again at line 0008. D0 is the op code for the BNE instruction, and FB is where it is branching to. Is FB the memory location of REPEAT? No. The BNE instruction uses relative addressing. FB_{16} is a negative-signed binary number telling us how many places to move from where we are now. FB_{16} is -5_{10}. We must branch five memory location backward from memory location 0349_{16}.

It will be helpful to enter this program into your computer or microprocessor trainer and single-step through it. We've gone through the loop only once in our discussion here.

```
0001  0340                .ORG $0340
0002  0340
0003  0340 A2 03   START:  LDX #$03       ;initialize X (repeats)
0004  0342 A0 00           LDY #$00       ;initialize Y
0005  0344 C8      REPEAT: INY            ;times loop has repeated
0006  0345 EA              NOP            ;misc instructions
0007  0346 CA              DEX            ;decrement X
0008  0347 D0 FB           BNE REPEAT     ;if X not equal to 0 then
0009  0349                                ;  branch back to start of
0010  0349                                ;  loop
0011  0349 00              BRK            ;stop
0012  034A
0013  034A                 .END
```

Fig. 22-5 A backward conditional jump creating a loop with the 6502 microprocessor.

Pay special attention to the X register, the Y register, and the zero flag.

Compare Instructions

The *compare* instructions allow us to compare the values in two registers and/or memory locations, and to set the flags accordingly, without changing either of the original values. The appropriate branch instruction can then cause program execution to continue at the desired location. The program in Fig. 22-6 will allow you to observe the compare instructions.

The program simply loads the value 05_{16} into the accumulator and compares the numbers 04_{16}, 06_{16}, and 05_{16} to it. If you will refer to the Expanded Table of 6502 Instructions and look in the Operation column, you will see what we mean by "compare."

To "compare" means to subtract the number you are "comparing" from the number being "compared to." For example, line 0004 of the program in Fig. 22-6 sets the flags as though 04_{16} had been subtracted from 05_{16}, without actually changing the value in the accumulator.

Lines 0005 and 0006 likewise subtract 06_{16} and 05_{16}, respectively, from the value in the accumulator without altering the accumulator.

A point needs to be made at this time about the carry flag in the 6502 microprocessor. Most microprocessors set a flag (value of 1) to say, "Yes, this condition exists." For example, setting the zero flag (value of 1) means, "Yes, the last value (or current value) is a zero." When a flag is reset (value of zero) it means "No, this condition does not exist."

The 6502 handles the carry flag in an unusual way. It is inverted. After addition this flag will appear as expected. A 1 means that a carry occurred, and a 0 means that a carry did not occur. After subtraction, however, a 1 means that a borrow did not occur, and a 0 means that a borrow did occur. Be careful to remember this exception when using 6502 compare instructions to prepare for branch instructions.

This program's only purpose is to allow you to see how the flags are affected by each compare instruction. Enter the program and single-step through it. Watch the flags after each instruction and make sure that you understand why they react the way they do.

An Example Program

We'll now look at an example program which uses a compare instruction, increment instructions, and a conditional branch instruction. This program looks at two numbers in memory, determines which is larger, and then places the larger value in a third memory location. It also uses a form of indexed addressing. Refer to Fig. 22-7 at this time.

After entering this program into your computer or trainer, but before running it, you must place values of your choice into the two memory locations indicated in the notes at the beginning of the program.

This program uses the X register to help point to the next memory location to load a number from or store a number in. The first instruction in line 0008 initializes the X register with a value of 00_{16}.

Memory location $03A0_{16}$ is the beginning of a series of memory locations which this program uses. A common way to address successive memory locations is to use some form of indexed addressing. Location $03A0_{16}$ is the beginning of the list, and the X register will point to each successive number in the list. In line 0009 we load the accumulator with the first number from the list. The memory location of this number is formed by adding $03A0_{16}$ to the value of the X register, which is 00_{16} at this moment, to form the address of the first number in the list, in location $03A0_{16}$.

In line 0010 we increment the X register to a value of 01_{16} so that it points to the next number.

In line 0011 we compare the value held in memory location $03A1_{16}$ to the value in the accumulator. If the value in the accumulator is larger, then no borrow will be needed to perform the comparison (which involves subtraction). Therefore the carry flag will be set.

We find in line 0012 that, if the carry flag is set, then we branch forward to line 0014. This will be the case if the value in the accumulator is the larger value. In line 0014 the X register is incremented so that it points to the last memory location. In line 0015 we store the value now in the accumulator in that final memory location.

If during the comparison in line 0011 the value in the accumulator is smaller, a borrow is required to perform the comparison (involving subtraction) and the carry flag is cleared. In line 0012 the carry flag is not set and the branch does not occur. Therefore, the next instruction in line 0013 is executed. This instruction loads the second number into

```
0001  0340                       .org $0340
0002  0340
0003  0340 A9 05       START:    LDA #$05       ;initial value
0004  0342 C9 04                 CMP #$04       ;compare each of these numbers
0005  0344 C9 06                 CMP #$06       ;  to A and set flags as though
0006  0346 C9 05                 CMP #$05       ;  each had been subtracted from A
0007  0348 00                    BRK
0008  0349
0009  0349                       .end
```

Fig. 22-6 Using the compare instruction.

```
0001  0000                   ;place a number in memory location $0340 and another in $03A1,
0002  0000                   ;  this program will determine which is larger and place
0003  0000                   ;  the larger in location $03A2  (Note: Do not use two
0004  0000                   ;  numbers which are equal.)
0005  0000
0006  0340                           .org $0340
0007  0340
0008  0340 A2 00             START:  LDX #$00        ;initialize X register
0009  0342 BD A0 03                  LDA $03A0,X     ;load A from mem 03A0 + 00 = 03A0
0010  0345 E8                        INX             ;point to next mem loc
0011  0346 DD A0 03                  CMP $03A0,X     ;compare data in mem 03A0 +
                                                       01 = 03A1 to A
0012  0349 B0 03                     BCS FOUND       ;if A is larger jump forward to Found;
0013  034B BD A0 03                  LDA $03A0,X     ;  otherwise load A from mem 03A0 +
                                                         01 = 03A1
0014  034E E8                FOUND:  INX             ;point to next mem loc
0015  034F 9D A0 03                  STA $03A0,X     ;store A in mem 03A0 + 02 = 03A2
0016  0352 00                        BRK             ;stop
0017  0353
0018  0353                           .end
```

Fig. 22-7 An example 6502 program.

the accumulator. Obviously, if the first number is not the larger, the second one must be. After loading the accumulator with the second number in line 0013, we continue in lines 0014 and 0015 to store that value in the third memory location.

This program will give you an idea how to use some of the new instructions in this chapter and how to use indexed addressing.

22-7 6800/6808 FAMILY

The 6800/6808 microprocessor family has a variety of instructions to handle unconditional jumps and branches, conditional branching, comparing, incrementing, and decrementing. We'll look at several tasks and see how the 6800/6808 microprocessor handles them.

You should enter each program into your computer or microprocessor trainer and single-step through it, watching the appropriate registers, memory locations, and flags to understand how each program works.

Unconditional Jumps

The forward unconditional jump using extended addressing is probably easiest to understand. An example is shown in Fig. 22-8.

(*Technical Note:* We have started this program at address 0100_{16} rather than our usual 0000_{16}. Addresses from 0000_{16} to $00FF_{16}$ form page 0 of memory. Some instructions can use direct addressing, if the desired location is on page 0, or extended addressing, if the desired location is on a memory page other than page 0. Our particular assembler had trouble handling forward references on page 0. Switching to a page other than page 0 provided a simple solution to this problem.)

The program begins by loading accumulator A with FF_{16}. In a moment we are gong to subtract another number from this one. First we need to jump to the area of memory where the subtract instruction is. We have placed the subtract instruction several memory locations forward from this point to show, in a very simple manner, how the unconditional jump instruction operates.

```
0100                      .org $0100      ;beginning of code
0100
0100 86 FF       START:   LDAA #$FF       ;minuend
0102 7E 01 08             JMP MINUS       ;forward unconditional jump
0105 01                   NOP
0106 01                   NOP             ;misc. instructions
0107 01                   NOP
0108 80 EE       MINUS:   SUBA #$EE       ;subtrahend
010A B7 01 0E             STAA ANSWER     ;store difference
010D 3E                   WAI             ;stop
010E
010E 00          ANSWER   .db     $00     ;memory area for answer
010F                                      ;  (initialized to 00)
010F                      .end
```

Fig. 22-8 Forward unconditional jump with the 6800/6808 microprocessor. (*Note* that address is $0100 rather than $0000. This prevents an assembler error caused by a forward reference to a label on zero page.)

The next instruction is our jump instruction. In the source code column of line 0004 the instruction

JMP MINUS

appears, which might be different than what you were expecting.

The instruction is saying to jump to a place called *MINUS*. To be able to jump to a place with a certain name is not a native ability of the 6800/6808 microprocessor. Our assembler is making this possible. Line 0008 has the label MINUS in the label column. This is the place we want to jump to. Notice the address at the MINUS label. The address is 0108. Now look back at line 0004. In the op code column you see 7E, which is the op code for an unconditional jump. Then come the numbers 01 08. This is the memory location of the instruction labeled MINUS. If you use an assembler, you can use labels and the assembler will calculate the address for you. If you are hand-assembling these programs, you must enter the address as shown in the op code column. If you are using an assembler which does not allow labels, you will need to use the format shown in the 6800/6808 instruction-set tables. Namely

JMP $0108

After the jump instruction are several NOPs which could be other instructions or just unused memory in a particular microprocessor system.

In line 0008 we subtract EE_{16} from FF_{16} (in accumulator A). In line 0009 we store the result of our subtraction in a memory location called *ANSWER*. Look at line 0012, labeled ANSWER. In the op code column are the initials .db. They stand for *define byte*. We are telling the assembler to reserve a memory location, namely, a single byte of memory, with the name ANSWER. The assembler is initializing the memory location ANSWER with a value of 0. Our program can then put any other number we wish in that location.

Notice also that the memory location of ANSWER is $010E_{16}$. In the op code column of line 0009 we see B7 01 0E. The op code for storing the value of the accumulator in a certain memory location is B7. 010E is the memory location of ANSWER. Again, the assembler made life simpler by figuring out where the next available memory location would be and setting aside that location for the ANSWER.

Finally, in line 0010 the program stops.

You should enter this program and single-step through it, making sure everything works as described.

Conditional Branches

Now let's see an example of conditional branching. Figure 22-9 shows such an example.

In this program we are going to do several things differently from the way they were done in the last program. First, we are using a conditional jump or branch rather than an unconditional one. Second, we are branching backward rather than forward. Third, we are creating a loop by branching backward and repeating a section of the program. Finally, we are using a register as a counter to control how many times the loop repeats.

In line 0003 we place the number 3_{16} in the X register. This register *controls* how many times we will branch backward. In line 0004 we clear accumulator B, making it 00_{16} so that it can be used to *count* how many times the loop repeats.

Line 0005 marks the beginning of the loop, and we have named that location *REPEAT*. In this line we increment accumulator B since we are beginning to pass through the loop, in this case for the first time. Accumulator B is keeping track of how many times the loop is passed through. Line 0006 represents the fact that there could be many instructions inside this loop which are going to be repeated.

Line 0007 decrements (reduces by 1) the X register. The X register keeps track of how many times to go through the loop remain.

Line 0008 is where we meet our conditional branch instruction. BNE means **B**ranch if **N**ot **E**qual. Your first thought might be, "Not equal to what?" If you check the Expanded Table for the 6800/6808, you'll see that it is **B**ranch if **N**ot **E**qual to 0.

```
0001  0000                       .ORG $0000
0002  0000
0003  0000 CE 00 03    START:    LDX #$0003        ;initialize X (repeats)
0004  0003 C6 00                 LDAB #$00         ;initialize B
0005  0005 5C          REPEAT:   INCB              ;times loop has repeated
0006  0006 01                    NOP               ;misc. instructions
0007  0007 09                    DEX               ;decrement X
0008  0008 26 FB                 BNE REPEAT        ;if X not equal to 0 then
0009  000A                                         ;  branch back to start of
0010  000A                                         ;  loop
0011  000A 3E                    WAI               ;stop
0012  000B
0013  000B                       .END
```

Fig. 22-9 A backward conditional jump creating a loop with the 6502 microprocessor.

All the conditional branch instructions are influenced by the most recent instruction that affected the flag they check. In this case the zero flag is checked. What was the last instruction which sets or clears the zero flag? The DEX (**DE**crement **X** register) instruction. If the X register was reduced to 0, the zero flag would be set. Has the X register been reduced to 0? On this first pass through the loop, it gets reduced from 3 to 2. No, the X register is not equal to 0.

The branch instruction says, "**B**ranch if **N**ot **E**qual to 0." Clearly this is true: the last result is not 0, so we branch. Branch to where? We branch to the memory location known as REPEAT. Notice that the location called REPEAT, in line 0005, is memory location 0005_{16}. Now look again at line 0008. The op code for the BNE instruction is 26, and FB is where it's branching to. Is FB the memory location of REPEAT? No. The BNE instruction uses relative addressing. FB_{16} is a negative-signed binary number telling us how many places to move from where we are now. FB_{16} is -5_{10}. We must branch five memory locations backward from memory location $000A_{16}$.

It will be helpful to enter this program into your computer or microprocessor trainer and single-step through it. Pay special attention to the X register, accumulator B, and the zero flag.

Compare Instructions

The *compare* instructions allow us to compare the values in two registers and/or memory locations and to set the flags accordingly without changing either of the original values. The appropriate branch instruction can then cause program execution to continue at the desired location. The program in Fig. 22-10 allows you to observe how the compare instructions work.

The program simply loads the value 05_{16} into accumulator A and compares the numbers 04_{16}, 06_{16}, and 05_{16} to it. If you refer to the Expanded Table of 6800/6808 Instructions and look in the Operation column, you will see what we mean by "compare."

To "compare" means to subtract the number you are "comparing" from the number being "compared to." For example, line 0004 of the program in Fig. 22-10 sets the flags as though 04_{16} had been subtracted from 05_{16}, in accumulator A, without actually changing the value in the accumulator.

```
0001  0000                      .org $0000
0002  0000
0003  0000 86 05      START:    LDAA #$05     ;initial value
0004  0002 81 04                CMPA #$04     ;compare each of these numbers
0005  0004 81 06                CMPA #$06     ;  to A and set flags as though
0006  0006 81 05                CMPA #$05     ;  each had been subtracted from A
0007  0008 3E                   WAI
0008  0009
0009  0009                      .end
```

Fig. 22-10 Using the compare instruction.

Line 0005 and 0006 likewise subtract 06_{16} and 05_{16}, respectively, from the value in accumulator A without altering the accumulator.

This program's only purpose is to allow you to see how the flags are affected by each compare instruction. Enter the program and single-step through it. Watch the flags after each step and make sure that you understand why they react the way they do.

An Example Program

We'll now look at an example program which uses a compare instruction, increment instructions, and a conditional branch instruction. This program looks at two numbers in memory, determines which is larger, and then places the larger value in a third memory location. It also uses a form of indexed addressing. Refer to Fig. 22-11 at this time.

After entering this program into your computer or trainer but before running it, you must place values of your choice into the two memory locations indicated in the notes at the beginning of the program.

This program uses the X register to help point to the next memory location to load a number from or store a number in. The first instruction in line 0008 initializes the X register with a value of $01A0_{16}$.

Memory location $01A0_{16}$ is the beginning of a series of memory locations which this program uses. A common way to address successive memory locations is to use some form of indexed addressing. Location $01A0_{16}$ is the beginning of the list, and the X register will point to each successive number in the list. In line 0009 we load the accumulator with the first number from the list. The memory location of this number is formed by adding 00_{16} to the value in the X register, which is $01A0_{16}$, to form the address of the first number in the list, at location $01A0_{16}$.

In line 0010 we increment the X register to a value of $01A1_{16}$ so that it points to the next number.

In line 0011 we compare the value held in memory location $01A1_{16}$ to the value in accumulator A. If the value in the accumulator is larger, then no borrow will be needed to perform the comparison (which involves subtraction). Therefore the carry flag will be clear.

We find in line 0012 that, if the carry flag is clear, then we branch forward to line 0014. This will be the case if the value in the accumulator is the larger value. In line 0014 the X register is incremented, so it points to the last

```
0001  0000                  ;place a number in memory location $01A0 and another in $01A1;
0002  0000                  ;  this program will determine which is larger and place
0003  0000                  ;  the larger in location $01A2  (Note: Do not use two
0004  0000                  ;  numbers which are equal.)
0005  0000
0006  0100                          .org $0100
0007  0100
0008  0100 CE 01 A0         START:  LDX #$01A0       ;initialize X register
0009  0103 A6 00                    LDAA $00,X       ;load A from mem 01A0 + 00 = 01A0
0010  0105 08                       INX              ;point to next mem loc
0011  0106 A1 00                    CMPA $00,X       ;compare data in mem 01A0 + 00 =
                                                        01A1 to A
0012  0108 24 02                    BCC FOUND        ;if A is larger jump forward to Found;
0013  010A A6 00                    LDAA $00,X       ;  otherwise load A from mem 01A1 +
                                                        00 = 01A1
0014  010C 08               FOUND:  INX              ;point to next mem loc
0015  010D A7 00                    STAA $00,X       ;store A in mem 01A2 + 00 = 01A2
0016  010F 3E                       WAI              ;stop
0017  0110
0018  011-                          .end
```

Fig. 22-11 An example 6800/6808 program.

memory location. In line 0015 we store the value now in accumulator A in that final memory location.

If, during the comparison in line 0011 the value in the accumulator is smaller, a borrow is required to perform the comparison (involving subtraction) and the carry flag is set. In line 0012 the carry flag is not clear and the branch does not occur. Therefore, the next instruction in line 0013 is executed. This instruction loads the second number into the accumulator. Obviously, if the first number is not the larger, the second one must be. After loading accumulator A with the second number in line 0013, we continue in lines 0014 and 0015 to store that value in the third memory location.

This program will give you an idea how to use some of the new instructions in this chapter and how to use indexed addressing.

22-8 8080/8085/Z80 FAMILY

The 8080/8085/Z80 microprocessor family has a variety of instructions to handle unconditional jumps, conditional branching, comparing, incrementing, and decrementing. We'll look at several tasks and see how the 8080/8085/Z80 microprocessor handles them.

You should enter each program into your computer or microprocessor trainer and single-step through it, watching the appropriate registers, memory locations, and flags to understand how each program works.

Remember that we will show both 8080/8085 and Z80 programs in the figures and that in the text we will show 8080/8085 mnemonics first with Z80 mnemonics in brackets.

Unconditional Jumps

The forward unconditional jump using direct addressing is probably easiest to understand. An example is shown in Fig. 22-12.

The program begins by loading the accumulator with FF_{16}. In a moment we are going to subtract another number from this one. First we need to jump to the area of memory where the subtract instruction is. We have placed the subtract instruction several memory locations forward from this point to show, in a very simple manner, how the unconditional jump instruction operates.

The next instruction is our jump instruction. In the source code column of line 0004 the instruction

JMP MINUS [JP MINUS]

appears, which might be different than what you were expecting.

The instruction is saying to jump to a place called *MINUS*. To be able to jump to a place with a certain name is not a native ability of the 8080/8085/Z80 microprocessor. Our assembler is making this possible. Line 0008 has the label MINUS in the label column. This is the place we want to jump to. Notice the address at the MINUS label. The address is 1808. Now look back at line 0004. In the op code column you see C3, which is the op code for an unconditional jump. Then come the numbers 08 18. If you reverse these two sets of numbers, you have 1808. This is the memory location of the instruction labeled MINUS. If you use an assembler, you can use labels and the assembler will calculate the address for you. If you are hand-assembling these programs, you must enter the address as shown in the op code column, in the reverse low-byte/high-byte order. If you are using an assembler which does not allow labels, you will need to use the format shown in the 8080/8085/Z80 instruction-set tables. Namely

JMP aaaa [JP aaaa]

After the jump instruction are several NOPs which could be other instructions or just unused memory in a particular microprocessor system.

8080/8085 program

```
0001  1800                .org 1800h      ;beginning of code
0002  1800
0003  1800 3E FF   START:  MVI A,0FFh      ;minuend
0004  1802 C3 08 18        JMP MINUS       ;forward unconditional jump
0005  1805 00              NOP
0006  1806 00              NOP             ;misc. instructions
0007  1807 00              NOP
0008  1808 D6 EE   MINUS:  SUI 0EEh        ;subtrahend
0009  180A 32 0E 18        STA ANSWER      ;store difference
0010  180D 76              HLT             ;stop
0011  180E
0012  180E 00      ANSWER  .db     00h     ;memory area for answer
0013  180F                                 ;  (initialized to 00)
0014  180F                 .end
```

Z80 program

```
0001  1800                .org 1800h      ;beginning of code
0002  1800
0003  1800 3E FF   START:  LD A,0FFh       ;minuend
0004  1802 C3 08 18        JP MINUS        ;forward unconditional jump
0005  1805 00              NOP
0006  1806 00              NOP             ;misc. instructions
0007  1807 00              NOP
0008  1808 D6 EE   MINUS:  SUB 0EEh        ;subtrahend
0009  180A 32 0E 18        LD (ANSWER),A   ;store difference
0010  180D 76              HALT            ;stop
0011  180E
0012  180E 00      ANSWER  .db     00h     ;memory area for answer
0013  180F                                 ;  (initialized to 00)
0014  180F                 .end
```

Fig. 22-12 Forward unconditional jump with the 8080/8085/Z80 microprocessor.

In line 0008 we subtract EE_{16} from FF_{16} (in the accumulator). In line 0009 we store the result of our subtraction in a memory location called *ANSWER*. Look at line 0012, labeled ANSWER. In the op code column are the initials .db. They stand for *define byte*. We are telling the assembler to reserve a memory location, namely, a single byte of memory, with the name ANSWER. The assembler is initializing the memory location ANSWER with a value of 0. Our program can then put any other number we wish in that location.

Notice also that the memory location of ANSWER is $180E_{16}$. In the op code column of line 0009 we see 32 0E 18. The op code for storing the value of the accumulator in a certain memory location is 32. If you reverse 0E 18, you have 180E, which is the memory location of ANSWER. Again the assembler made life simpler by figuring out where the next available memory location would be and setting aside that location for the ANSWER.

Finally, in line 0010, the program stops.

You should enter this program and single-step through it, making sure that everything works as described.

Conditional Branches

Now let's see an example of conditional branching. Figure 22-13 shows such an example.

In this program we are going to do several things differently from the way they were done in the last program. First, we are using a conditional jump or branch rather than an unconditional one. Second, we are branching backward rather than forward. Third, we are creating a loop by branching backward and repeating a section of the program. Finally, we are using a register as a counter to control how many times the loop repeats.

In line 0003 we place the number 3_{16} in register B. This register *controls* how many times we will branch backward. In line 0004 we clear register C making it 00_{16} so that it can be used to *count* how many times the loop repeats.

Line 0005 marks the beginning of the loop, and we have named that location *REPEAT*. In this line we increment register C since we are beginning to pass through the loop, in this case for the first time. Register C is keeping track of how many times the loop is passed through. Line 0006 represents the fact that there could be many instructions inside this loop which are going to be repeated.

Line 0007 decrements (reduces by one) register B. Register B keeps track of how many times we have left to go through the loop.

Line 0008 is where we meet our conditional branch instruction. JNZ means **J**ump if **N**ot **Z**ero. [JP NZ means **JumP** if **N**ot **Z**ero.] Your first thought might be, "If what isn't zero?"

```
8080/8085 program

0001   1800                    .ORG 1800h
0002   1800
0003   1800 06 03     START:   MVI B,03h       ;initialize B (repeats)
0004   1802 0E 00              MVI C,00h       ;initialize C
0005   1804 0C        REPEAT:  INR C           ;times loop has repeated
0006   1805 00                 NOP             ;misc instructions
0007   1806 05                 DCR B           ;decrement B
0008   1807 C2 04 18           JNZ REPEAT      ;if B not equal to 0 then
0009   180A                                    ;  branch back to start of
0010   180A                                    ;  loop
0011   180A 76                 HLT             ;stop
0012   180B
0013   180B                    .END
```

```
Z80 program

0001   1800                    .ORG 1800h
0002   1800
0003   1800 06 03     START:   LD B,03h        ;initialize B (repeats)
0004   1802 0E 00              LD C,00h        ;initialize C
0005   1804 0C        REPEAT:  INC C           ;times loop has repeated
0006   1805 00                 NOP             ;misc instructions
0007   1806 05                 DEC B           ;decrement B
0008   1807 C2 04 18           JP NZ,REPEAT    ;if B not equal to 0 then
0009   180A                                    ;  branch back to start of
0010   180A                                    ;  loop
0011   180A 76                 HALT            ;stop
0012   180B
0013   180B                    .END
```

Fig. 22-13 A backward conditional jump creating a loop with the 8080/8085/Z80 microprocessor.

All the conditional branch instructions are influenced by the most recent instruction that affected the flag they check. In this case the zero flag is checked. What was the last instruction which sets or clears the zero flag? The DCR B (**D**e**CR**ement **B**) [DEC B (**DEC**rement **B**)] instruction. If register B were reduced to 0, the zero flag would be set. Has register B been reduced to zero? On this first pass through the loop, it gets reduced from 3 to 2. No, register B is not equal to 0.

The jump instruction says, "Jump if not zero." Clearly this is true: the last result is not 0, so we do jump. Jump to where? We jump to the memory location known as *REPEAT*. Notice that the location called REPEAT, in line 0005, is memory location 1804_{16}. Now look again at line 0008. C2 is the op code for the JNZ [JP NZ] instruction. If you reverse the two sets of numbers 04 18, you form 1804, which is the memory location of the REPEAT label.

It will be helpful to enter this program into your computer or microprocessor trainer and single-step through it. Pay special attention to register B, register C, and the zero flag.

Compare Instructions

The *compare* instructions allow us to compare the values in two registers and/or memory locations and to set the flags accordingly without changing either of the original values. The appropriate jump instruction can then cause program execution to continue at the desired location. The program in Fig. 22-14 allows you to experiment with the compare instructions.

This program loads the value 05_{16} into the accumulator and compares the numbers 04_{16}, 06_{16}, and 05_{16} to it. If you will refer to the Expanded Table of 8080/8085/Z80 Instructions and look in the Operation column, you will see what we mean by "compare."

To "compare" means to subtract the number you are comparing from the number being "compared to." For example, line 0004 of the program in Fig. 22-14 sets the flags as though 04_{16} had been subtracted from 05_{16}, without actually changing the value in the accumulator.

Lines 0005 and 0006 likewise subtract 06_{16} and 05_{16}, respectively, from the value in the accumulator without altering the accumulator.

This program's only purpose is to allow you to see how the flags are affected by each compare instruction. Enter the program and single-step through it. Watch the flags after each step and make sure you understand why they react the way they do.

An Example Program

We'll now look at an example program which uses a compare instruction, increment instructions, and a condi-

```
8080/8085 program

0001   1800                          .org 1800h
0002   1800
0003   1800 3E 05        START:   MVI A,05h        ;initial value
0004   1802 FE 04                 CPI 04h          ;compare each of these numbers
0005   1804 FE 06                 CPI 06h          ;  to A and set flags as though
0006   1806 FE 05                 CPI 05h          ;  each had been subtracted from A
0007   1808 76                    HLT
0008   1809
0009   1809                       .end
```

```
Z80 program

0001   1800                          .org 1800h
0002   1800
0003   1800 3E 05        START:   LD A,05h         ;initial value
0004   1802 FE 04                 CP 04h           ;compare each of these numbers
0005   1804 FE 06                 CP 06h           ;  to A and set flags as though
0006   1806 FE 05                 CP 05h           ;  each had been subtracted from A
0007   1808 76                    HALT
0008   1809
0009   1809                       .end
```

Fig. 22-14 Using the compare instruction.

tional branch instruction. This program looks at two numbers in memory, determines which is larger, and then places the larger value in a third memory location. It also uses register indirect addressing. Refer to Fig. 22-15 at this time.

After entering this program into your computer or trainer, but before running it, you must place values of your choice into the two memory locations indicated in the notes at the beginning of the program.

This program uses the HL register pair to help point to the next memory location to load a number from or store a number in. The first instruction in line 0008 initializes the HL register pair with a value of $18A0_{16}$.

Memory location $18A0_{16}$ is the beginning of a series of memory locations which this program uses. A common way to address successive memory locations is to use some form of indexed addressing. The 8080/8085 does not actually have an index register; however, the HL register pair can be used with register indirect addressing to accomplish much the same thing. Location $18A0_{16}$ is the beginning of the list, and the HL register pair will point to each successive number in the list. In line 0009 we load the accumulator with the first number from the list. The memory location of this number is pointed to by the value in the HL register pair.

In line 0010 we increment the HL register pair to a value of $18A1_{16}$ so that it points to the next number.

In line 0011 we compare the value held in memory location $18A1_{16}$ to the value in the accumulator. If the value in the accumulator is larger, then no borrow will be needed to perform the comparison (which involves subtraction). Therefore the carry flag will be clear.

We find in line 0012 that, if the carry flag is clear, then we branch forward to line 0014. This will be the case if the value in the accumulator is the larger value. In line 0014 the HL register pair is incremented so that it points to the last memory location. In line 0015 we store the value now in the accumulator in that final memory location.

If, during the comparison in line 0011 the value in the accumulator is smaller, a borrow is required to perform the comparison (involving subtraction) and the carry flag is set. In line 0012 the carry flag is not clear and the branch does not occur. Therefore the next instruction in line 0013 is executed. This instruction loads the second number into the accumulator. Obviously, if the first number is not the larger, the second one must be. After loading the accumulator with the second number in line 0013, we continue in lines 0014 and 0015 to store that value in the third memory location.

This program will give you an idea how to use some of the new instructions in this chapter and how to use register indirect addressing.

22-9 8086/8088 FAMILY

The 8086/8088 microprocessor family has a variety of instructions to handle unconditional jumps, conditional branching, comparing, incrementing, and decrementing. We'll look at several typical tasks and see how the 8086/8088 microprocessor handles them.

You should enter each program into your computer or microprocessor trainer and single-step through it, watching the appropriate registers, memory locations, and flags to understand how each program works.

```
8080/8085 program

0000              ;place a number in memory location 18A0h and another in 18A1h;
0000              ;  this program will determine which is larger and place
0000              ;  the larger in location 18A2h  (Note: Do not use two
0000              ;  numbers which are equal.)
0000
1800                        .org 1800h
1800
1800 21 A0 18     START:    LXI H,18A0h      ;initialize HL register
1803 7E                     MOV A,M          ;load A from mem 18A0
1804 23                     INX H            ;point to next mem loc
1805 BE                     CMP M            ;compare data in mem 18A1 to A
1806 D2 0A 18               JNC FOUND        ;if A is larger jump forward to Found;
1809 7E                     MOV A,M          ;  otherwise load A from mem 18A1
180A 23           FOUND:    INX H            ;point to next mem loc
180B 77                     MOV M,A          ;store A in mem 18A2
180C 76                     HLT              ;stop
180D
180D                        .end
```

```
Z80 program

0000              ;place a number in memory location 18A0h and another in 18A1h;
0000              ;  this program will determine which is larger and place
0000              ;  the larger in location  18A2h  (Note: Do not use two
0000              ;  numbers which are equal.)
0000
1800                        .org 1800h
1800
1800 21 A0 18     START:    LD HL,18A0h      ;initialize HL register
1803 7E                     LD A,(HL)        ;load A from mem 18A0
1804 23                     INC HL           ;point to next mem loc
1805 BE                     CP (HL)          ;compare data in mem 18A1 to A
1806 D2 0A 18               JP NC,FOUND      ;if A is larger jump forward to Found;
1809 7E                     LD A,(HL)        ;  otherwise load A from mem 18A1
180A 23           FOUND:    INC HL           ;point to next mem loc
180B 77                     LD (HL),A        ;store A in mem 18A2
180C 76                     HALT             ;stop
180D
180D                        .end
```

Fig. 22-15 An example 8080/8085/Z80 program.

Using An Assembler

We need to explain a few things about using an assembler with the 8086/8088 microprocessor. Look at Fig. 22-16 for a moment. The

```
page ,132
```

command tells the assembler to create a list file (Fig. 22-16 is a list file) that is up to 132 columns wide. This gives us more room for the comments at the ends of the lines.

The top portion above the program, which reads

```
CODE    SEGMENT
        ASSUME CS:CODE, DS:CODE, SS:CODE
        ORG 100h
```

and the bottom portion, which reads

```
CODE    ENDS
        END     START
```

are required by the assembler. This information has to do with where in memory we want the program to be and how we want to handle memory segmentation. This model allows the program to be assembled and linked to form an .EXE file which can then be converted to a .COM file with the EXE2BIN DOS utility. A complete discussion of these concepts is beyond the scope of this text. If you will use this model, however, you will be able to use DEBUG to examine the file and use the trace command to single-step through it.

After you assemble and link the file, use the EXE2BIN

```
                              page ,132

0000                          CODE    SEGMENT
                                      ASSUME CS:CODE, DS:CODE, SS:CODE
0100                                  ORG 100h

0100  B0 FF                   START:  MOV AL,0FFh       ;minuend
0102  EB 03                           JMP SHORT MINUS ;forward unconditional jump
0104  90                              NOP
0105  90                              NOP               ;misc. instructions
0106  90                              NOP
0107  2C EE                   MINUS:  SUB AL,0EEh       ;subtrahend
0109  A2 010E R                       MOV ANSWER,AL     ;store difference
010C  CD 20                           INT 20h           ;stop

010E  00                      ANSWER  DB     00h        ;memory area for answer
                                                        ;  (initialized to 0)

010F                          CODE    ENDS

                                      END    START
```

Fig. 22-16 Forward unconditional jump with the 8086/8088 microprocessor (using an assembler).

utility to change it to a .COM file. Then load the file (filename.ext) by typing

debug filename.ext

at the DOS prompt.

Unconditional Jumps

The forward unconditional jump using direct addressing is probably the easiest to understand. Look again at Fig. 22-16. The same program entered with DEBUG is shown in Fig. 22-17.

The program begins by loading AL with FF_{16}. In a moment we are going to subtract another number from this one. First we need to jump to the area of memory where the subtract instruction is. We have placed the subtract instruction several memory locations forward from this point to show, in a very simple manner, how the unconditional jump instruction operates.

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0100   NV UP EI PL NZ NA PO NC
3F3D:0100 B0FF          MOV     AL,FF
-
-a
3F3D:0100 MOV AL,FF          ;minuend
3F3D:0102 JMP 0107           ;forward unconditional jump
3F3D:0104 NOP
3F3D:0105 NOP                ;misc. instructions
3F3D:0106 NOP
3F3D:0107 SUB AL,EE          ;subtrahend
3F3D:0109 MOV [010E],AL      ;store difference
3F3D:010C INT 20             ;stop
3F3D:010E
-
-
-u 100 10d
3F3D:0100 B0FF          MOV     AL,FF
3F3D:0102 EB03          JMP     0107
3F3D:0104 90            NOP
3F3D:0105 90            NOP
3F3D:0106 90            NOP
3F3D:0107 2CEE          SUB     AL,EE
3F3D:0109 A20E01        MOV     [010E],AL
3F3D:010C CD20          INT     20
-
```

Fig. 22-17 Forward unconditional jump with the 8086/8088 microprocessor (using DEBUG).

The next instruction is our jump instruction. In the source-code column of line 8 in Fig. 22-16 the instruction

JMP SHORT MINUS

appears, which might be different from what you were expecting.

The instruction is saying to jump to a placed called *MINUS*. To be able to jump to a place with a certain name is not a native ability of the 8086/8088 microprocessor. Our assembler is making this possible. Line 12 has the label MINUS in the label column. This is the place we want to jump to. Notice the address at the MINUS label. The address is 0107. Now look back at line 8. In the op code column you see EB, which is the op code for an unconditional jump. Then comes the number 03. This is the number of memory locations by which we must move forward from the instruction after the JMP instruction. Moving forward 03 places takes us to memory location 0107. This is the memory location of the instruction labeled MINUS. If you use an assembler, you can use labels and the assembler will calculate the relative address for you. The term *SHORT* tells the assembler that this place called MINUS is within 127 bytes of our current location.

If you are using DEBUG to assemble these programs, you must enter the program as shown in Fig. 22-17. Toward the top of Fig. 22-17 we simply say

JMP 0107

Notice further down in Fig. 22-17 where we disassembled the program that JMP 0107 disassembles to EB03. Our assembler and DEBUG produced the same code.

After the jump instruction are several NOPs which could be other instructions or just unused memory in a particular microprocessor system.

In line 12 of Fig. 22-16 we subtract EE_{16} from FF_{16} (in AL). In line 13 we store the result of our subtraction in a memory location called *ANSWER*. Look at line 16, labeled ANSWER. In the op code column are the initials DB. This stands for *define byte*. We are telling the assembler to reserve a memory location, namely, a single byte of memory, with the name ANSWER. The assembler is initializing the memory location ANSWER with a value of 0. Our program can then put any other number we wish in that location.

Notice also that the memory location of ANSWER is $010E_{16}$. In the op code column of line 13 we see A2 010E. A2 is the op code for storing the value of AL in a certain memory location. Again the assembler made life simpler by figuring out where the next available memory location would be and setting aside that location for the ANSWER.

If you used DEBUG as shown in Fig. 22-17, then you had to specify memory location 010E as shown.

Finally, in line 14 of Fig. 22-16, the program stops.

You should enter this program and single-step through it, making sure that everything works as described. This is shown in Fig. 22-18.

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0100   NV UP EI PL NZ NA PO NC
3F3D:0100 B0FF          MOV     AL,FF
-
-t

AX=00FF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0102   NV UP EI PL NZ NA PO NC
3F3D:0102 EB03          JMP     0107
-
AX=00FF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0107   NV UP EI PL NZ NA PO NC
3F3D:0107 2CEE          SUB     AL,EE
-t

AX=0011  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0109   NV UP EI PL NZ NA PE NC
3F3D:0109 A20E01        MOV     [010E],AL                         DS:010E=83
-t

AX=0011  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=010C   NV UP EI PL NZ NA PE NC
3F3D:010C CD20          INT     20
-
-d 0100 010F
3F3D:0100  B0 FF EB 03 90 90 90 2C-EE A2 0E 01 CD 20 11 3F   ......,.......?
-
-
```

Fig. 22-18 Forward unconditional jump with the 8086/8088 microprocessor (single-stepping with the Trace command).

```
 1                          page ,132
 2
 3 0000                     CODE    SEGMENT
 4                                  ASSUME CS:CODE, DS:CODE, SS:CODE
 5 0100                             ORG 100h
 6
 7 0100   B1 03             START:  MOV CL,03h     ;initialize CL (repeats)
 8 0102   B5 00                     MOV CH,00h     ;initialize CH
 9 0104   FE C5             REPEAT: INC CH         ;times loop has repeated
10 0106   90                        NOP            ;misc. instructions
11 0107   FE C9                     DEC CL         ;decrement CL
12 0109   75 F9                     JNZ REPEAT     ;if CL not equal to 0 then
13                                                 ;  branch back to start of
14                                                 ;  loop
15 010B   CD 20                     INT 20h        ;stop
16
17 010D                     CODE    ENDS
18
19                                  END     START
```

Fig. 22-19 A backward conditional jump creating a loop with the 8086/8088 microprocessor (using an assembler).

Conditional Branches

Now let's see an example of conditional branching. Figure 22-19 shows such an example using an assembler.

Figure 22-20 shows the same program using DEBUG.

In this program we are going to do several things differently from the way they were done in the last program. First, we are using a conditional jump or branch rather than an unconditional one. Second, we are branching backward rather than forward. Third, we are creating a loop by branching backward and repeating a section of the program. Finally, we are using a register as a counter to control how many times the loop repeats.

In line 7 of Fig. 22-19 we place the number 3_{16} in CL. This register *controls* how many times we will branch backward. In line 8 we clear CH, making it 00_{16} so that it can be used to *count* how many times the loop repeats.

Line 9 marks the beginning of the loop, and we have named that location *REPEAT*. In this line we increment CH since we are beginning to pass through the loop, in this case for the first time. Register CH is keeping track of how many times the loop is passed through. Line 10 represents the fact that there could be many instructions inside this loop which are going to be repeated.

Line 11 decrements (reduced by 1) register CL. Register CL keeps track of how many times we have left to go through the loop.

Line 12 is where we meet our conditional branch instruction. JNZ means **J**ump if **N**ot **Z**ero. Your first thought might be, "If what isn't zero?"

All the conditional branch instructions are influenced by the most recent instruction that affected the flag they check. In this case the zero flag is checked. What was the last instruction which sets or clears the zero flag? The DEC CL

```
-a 100
77B3:0100 MOV CL,03         ;initialize CL (repeats)
77B3:0102 MOV CH,00         ;initialize CH
77B3:0104 INC CH            ;times loop has repeated
77B3:0106 NOP               ;misc instructions
77B3:0107 DEC CL            ;decrement CL
77B3:0109 JNZ 0104          ;if CL not equal to 0 then
77B3:010B                   ;  branch back to start of
77B3:010B                   ;  loop
77B3:010B INT 20            ;stop
77B3:010D
-
-u 100 10c
77B3:0100 B103          MOV     CL,03
77B3:0102 B500          MOV     CH,00
77B3:0104 FEC5          INC     CH
77B3:0106 90            NOP
77B3:0107 FEC9          DEC     CL
77B3:0109 75F9          JNZ     0104
77B3:010B CD20          INT     20
-
```

Fig. 22-20 A backward conditional jump creating a loop with the 8086/8088 microprocessor (using DEBUG).

(**DEC**rement **CL**) instruction. If register CL were reduced to 0, the zero flag would be set. Has CL been reduced to 0? On this first pass through the loop it gets reduced from 3 to 2. No, CL is not equal to 0.

The jump instruction says, "Jump if not zero." Clearly this is true: the last result is not 0, so we do jump. Jump to where? We jump to the memory location known as *REPEAT*. Notice that the location called REPEAT, in line 9, is memory location 0104_{16}. Now look again at line 12. The op code for the JNZ instruction is 75. F9 is a negative-signed binary number telling us how many places to move backward through memory to reach the place labeled REPEAT.

If you are using DEBUG to enter this program a shown in Fig. 22-20, you will actually enter address 0104. DEBUG then calculates the relative address (F9) for you as shown in the disassembled area at the bottom of Fig. 22-20.

It will be helpful to enter this program into your computer and single-step through it. Pay special attention to register CL, register CH, and the zero flag.

Compare Instructions

The *compare* instructions allow us to compare the values in two registers and/or memory locations and to set the flags accordingly without changing either of the original values. The appropriate jump instruction can then cause program execution to continue at the desired location. The program in Figs. 22-21 and 22-22 allows you to observe how the compare instructions work.

The program simply loads the value 05_{16} into AL and compares the numbers 04_{16}, 06_{16}, and 05_{16} to it. If you will refer to the Expanded Table of 8086/8088 Instructions and read the description, you will see what we mean by "compare."

To "compare" means to subtract the number you are "comparing" from the number being "compared to." For example, line 8 of the program in Fig. 22-21 sets the flags as though 04_{16} had been subtracted from 05_{16}, without actually changing the value in AL. Lines 9 and 10 likewise subtract 06_{16} and 05_{16}, respectively, from the value in AL without altering AL.

This program's only purpose is to allow you to see how the flags are affected by each compare instruction. Enter the program and single-step through it. Watch the flags after each step and make sure that you understand why they react the way they do. This has been done in Fig. 22-22.

An Example Program

We'll now look at an example program which uses a compare instruction, increment instructions, and a conditional branch instruction. This program looks at two numbers in memory, determines which is larger, and then places the larger value in a third memory location. It also uses register indirect addressing. Refer to Figs. 22-23 and 22-24 at this time.

After entering this program into your computer or trainer but before running it, you must place values of your choice into the two memory locations indicated in the note at the top of Fig. 22-23.

This program uses BX to help point to the next memory location to load a number from or store a number in. The first instruction in line 12 of Fig. 22-23 initializes BX with a value of 00_{16}.

Memory location 0119_{16} (referred to as DATA, line 22) is the beginning of a series of memory locations which this program uses. A common way to address successive memory locations is to use register relative addressing. Location 0119_{16} is the beginning of the list, and the BX register will point to each successive number in the list. In line 13 we load the accumulator with the first number from the list. The memory location of this number is pointed to by adding 0119_{16} (DATA) to the value in BX.

In line 14 we increment the BX register to a value of 01_{16} so that we can point to the next number.

In line 15 we compare the value held in memory location [DATA + BX] to the value in the accumulator. If the value in the accumulator is larger, then no borrow will be needed to perform the comparison (which involves sub-

```
 1                       page ,132
 2
 3 0000                          CODE    SEGMENT
 4                                       ASSUME CS:CODE, DS:CODE, SS:CODE
 5 0100                                  ORG 100h
 6
 7 0100   B0 05                  START:  MOV AL,05h        ;initial value
 8 0102   3C 04                          CMP AL,04h        ;compare each of these numbers
 9 0104   3C 06                          CMP AL,06h        ;  to AL and set flags as though
10 0106   3C 05                          CMP AL,05h        ;  each had been subtracted from AL
11 0108   CD 20                          INT 20h
12
13 010A                          CODE    ENDS
14
15                                       END     START
```

Fig. 22-21 Using the compare instruction (8086/8088 using an assembler).

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0100   NV UP EI PL NZ NA PO NC
3F3D:0100 B005          MOV     AL,05
-
-a
3F3D:0100 MOV AL,05       ;initial value
3F3D:0102 CMP AL,04       ;compare each of these numbers
3F3D:0104 CMP AL,06       :  to AL and set flags as though
3F3D:0106 CMP AL,05       ;  each had been subtracted from AL
3F3D:0108 INT 20
3F3D:010A
-
-u 100 108
3F3D:0100 B005          MOV     AL,05
3F3D:0102 3C04          CMP     AL,04
3F3D:0104 3C06          CMP     AL,06
3F3D:0106 3C05          CMP     AL,05
3F3D:0108 CD20          INT     20
-
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0100   NV UP EI PL NZ NA PO NC
3F3D:0100 B005          MOV     AL,05
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0102   NV UP EI PL NZ NA PO NC
3F3D:0102 3C04          CMP     AL,04
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0104   NV UP EI PL NZ NA PO NC
3F3D:0104 3C06          CMP     AL,06
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0106   NV UP EI NG NZ AC PE CY
3F3D:0106 3C05          CMP     AL,05
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0108   NV UP EI PL ZR NA PE NC
3F3D:0108 CD20          INT     20
-
```

Fig. 22-22 Using the compare instruction (8086/8088 using DEBUG).

traction), nor will the result of the comparison be 0; therefore both the carry flag and the zero flag will be clear.

We find in line 16 that, if both the carry flag and the zero flag are clear, then we branch forward to line 18. This will be the case if the value in AL is the larger value. In line 18 the BX register is incremented so that it points to the last memory location. In line 19 we store the value now in AL in that final memory location.

If during the comparison in line 15 the value in the accumulator is smaller, a borrow is required to perform the comparison (involving subtraction) and the carry flag is set. In line 16 the carry flag is not clear and the jump does not occur. Therefore the next instruction in line 17 is executed. This instruction loads the second number into AL. Obviously, if the first number is not the larger, the second one must be. After loading the accumulator with the second number in line 17, we continue in lines 18 and 19 to store that value in the third memory location.

This program will give you an idea how to use some of the new instructions in this chapter and how to use register relative addressing.

Compare the program as shown in Figs. 22-23 and 22-24. In Fig. 22-24 the program is entered by using DEBUG and then single-stepping through (using trace). As in all programs shown in this text, you'll learn the most if you enter the program yourself and experiment with it.

```
                       page ,132

                       ;place a number in memory location DATA and another in DATA+1;
                       ;  this program will determine which is larger and place
                       ;  the larger in location DATA+2  (Note: Do not use two
                       ;  numbers which are equal.)

0000                   CODE    SEGMENT
                               ASSUME CS:CODE, DS:CODE, SS:CODE
0100                           ORG 0100h

0100  BB 0000          START:  MOV BX,00h           ;initialize BX register
0103  8A 87 0119 R             MOV AL,[DATA + BX]   ;move byte to AL from mem loc DATA
0107  43                       INC BX               ;point to next mem loc (DATA + 1)
0108  3A 87 0119 R             CMP AL,[DATA + BX]   ;compare byte in mem DATA + 1 to AL
010C  77 04                    JA FOUND             ;if AL is larger jump forward to Found;
010E  8A 87 0119 R             MOV AL,[DATA + BX]   ;  otherwise move byte to AL from mem
                                                       DATA + 1
0112  43               FOUND:  INC BX               ;point to next mem loc (DATA + 2)
0113  88 87 0119 R             MOV [DATA + BX],AL   ;move byte in AL to mem DATA + 2
0117  CD 20                    INT 20h              ;stop

0119  05 04 00         DATA    DB      05h,04h,00h  ;you can use different values for the
                                                    ;  first two numbers

011C                   CODE    ENDS

                               END     START
```

Fig. 22-23 An example 8086/8088 program (using an assembler).

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0100   NV UP EI PL NZ NA PO NC
3F3D:0100 BB0000        MOV     BX,0000
-
-a
3F3D:0100 MOV     BX,0000              ;initialize BX register
3F3D:0103 MOV     AL,[BX+0119]         ;move byte to AL from mem loc 0119 + 0
3F3D:0107 INC     BX                   ;point to next mem loc 0119 + 1
3F3D:0108 CMP     AL,[BX+0119]         ;compare byte in mem 0119 + 1 to AL
3F3D:010C JA      0112                 ;if AL is larger jump forward to 0112,
3F3D:010E MOV     AL,[BX+0119]         ;  otherwise move byte to AL from 0119 + 1
3F3D:0112 INC     BX                   ;point to next mem loc 0119 + 2
3F3D:0113 MOV     [BX+0119],AL         ;move byte in AL to mem 0119 + 2
3F3D:0117 INT     20                   ;stop
3F3D:0119
-
-
-u 0100 0118
3F3D:0100 BB0000        MOV     BX,0000
3F3D:0103 8A871901      MOV     AL,[BX+0119]
3F3D:0107 43            INC     BX
3F3D:0108 3A871901      CMP     AL,[BX+0119]
3F3D:010C 7704          JA      0112
3F3D:010E 8A871901      MOV     AL,[BX+0119]
3F3D:0112 43            INC     BX
3F3D:0113 88871901      MOV     [BX+0119],AL
3F3D:0117 CD20          INT     20
```

Fig. 22-24 An example 8086/8088 program (using DEBUG).

```
-
-e 0119
3F3D:0119  5E.05   F6.04   8B.00   07.00
-
-d 0110 011f
3F3D:0110  19 01 43 88 87 19 01 CD-20 05 04 00 00 89 46 EE   ..C..... .....F.
-


-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0100   NV UP EI PL NZ NA PO NC
3F3D:0100 BB0000        MOV     BX,0000
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0103   NV UP EI PL NZ NA PO NC
3F3D:0103 8A871901      MOV     AL,[BX+0119]                       DS:0119=05
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0107   NV UP EI PL NZ NA PO NC
3F3D:0107 43            INC     BX
-t

AX=0005  BX=0001  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0108   NV UP EI PL NZ NA PO NC
3F3D:0108 3A871901      CMP     AL,[BX+0119]                       DS:011A=04
-t

AX=0005  BX=0001  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=010C   NV UP EI PL NZ NA PO NC
3F3D:010C 7704          JA      0112
-t

AX=0005  BX=0001  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0112   NV UP EI PL NZ NA PO NC
3F3D:0112 43            INC     BX
-t

AX=0005  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0113   NV UP EI PL NZ NA PO NC
3F3D:0113 88871901      MOV     [BX+0119],AL                       DS:011B=00
-t

AX=0005  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3F3D  ES=3F3D  SS=3F3D  CS=3F3D  IP=0117   NV UP EI PL NZ NA PO NC
3F3D:0117 CD20          INT     20
-
-
-d 0110 011f
3F3D:0110  19 01 43 88 87 19 01 CD-20 05 04 05 00 89 46 EE   ..C..... .....F.
-
-
```

Fig. 22-24 (cont.)

GLOSSARY

decrement To decrease. Most microprocessors decrement registers or memory locations by 1.

increment To increase. Most microprocessors increment registers or memory locations by 1.

loop A group of instructions which can be executed more than once. The program "falls through" the loop when some condition exists or when the loop has been executed a predetermined number of times.

nest To fit one inside another. Loops can be nested by having one small loop executing within a larger loop.

SELF-TESTING REVIEW

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

1. Branches or jumps can be made to execute all the time or only when certain conditions exist. That is, branches and loops can be ________ or ________.
2. *(conditional, unconditional)* When a program branches backward and repeats a group of instructions, it is called a ________.
3. *(loop)* Compare instructions generally (though not always) set and clear the microprocessor's flags as though ________ had occurred.
(subtraction)

PROBLEMS

Solve the following problems by using the microprocessor of your choice.

You may have some difficulty with the following two problems; therefore only two are given. As you begin each problem, do not immediately think of which microprocessor instructions to use. Instead, think about the problem itself and visualize what the memory locations will contain. Think of how to move the data between registers and memory locations to solve the problem, and *then* think about what instructions can be used to accomplish the moves.

22-1. Write a program which will use the first number in a list of unsigned binary numbers as a reference, will compare that number to each of the following numbers in the list, and will then stop when it finds the first number in the list which is smaller than or equal to the reference number. Finally, the program should store that first number which was smaller or equal to the reference number in a memory location called ANSWER.

(*Important:* The numbers in the list must be considered unsigned binary numbers. At least *one number* in the list *must* be smaller than or equal to the reference number. All the numbers *may* be smaller or equal to the reference. The program will be most interesting if more than one, but not all, the numbers are smaller than or equal to the reference.)

(*Note:* You will need to enter the list of numbers before running the program. *The list must have a minimum of two numbers* and can have as many additional numbers as you wish. We have started the list of numbers at memory location \$03A0 for the 6502, \$01A0 for the 6800/6808, and at 18A0h for the 8080/8085/Z80, and at a location labeled LIST for the 8086/8088.)

22-2. Write a program which will look at a list of numbers which you will store in memory. The end of this list will be indicated by the number 00. The number 00 cannot be used anywhere in the list except to mark its end. Write the program so that it will add each pair of consecutive numbers. That is, if the list contained the numbers 06_{16}, $2E_{16}$, 36_{16}, 42_{16}, and 00_{16}, it would perform the following additions:

$$06_{16} + 2E_{16} = 34_{16}$$
$$2E_{16} + 36_{16} = 64_{16}$$
$$36_{16} + 42_{16} = 78_{16}$$

The program should not add the 00_{16} to the preceding number since 00_{16} is not one of the numbers in the list but indicates the end of the list.

When the program adds the first two numbers, it should place their sum in a memory location called *LRGST* (largest). As it adds each of the following pairs, it should compare their sum with the number in LRGST. If the new sum is larger than the number in LRGST, then the new largest number should be placed in LRGST. Thus, after the program has added all the pairs together, LRGST will contain the largest sum that was created. All numbers should be considered unsigned binary numbers.

(*Note:* The list must contain at least one number, with the number 00 following it to indicate the end of the list. In this case no sum should appear in LRGST because there can be no sum with a list of only one number. The list can contain any number of numbers beyond one.)

(*Note:* We have used the numbers $2E_{16}$, $3C_{16}$, $1B_{16}$, 46_{16}, and 00_{16} to end the list, in that order, in the answer key. You should try altering your list to make sure it works under various circumstances.)

23

SUBROUTINE AND STACK INSTRUCTIONS

At this point we have covered most of the instruction set of each of the microprocessors featured in this text. Two final topics, however, the stack and subroutines, may be the most important ones. Without subroutines, programs written for these microprocessors would be unmanageable. Subroutines are used when there are tasks which must be executed or used many times. The subroutine provides a way to write a program segment which can handle a specific task and be reused.

The stack is important because it supports subroutines by storing information the microprocessor needs when it tries to return from a subroutine.

New Concepts

This chapter deals with subroutines and with the stack, especially as the stack relates to subroutines. The use of the stack in passing parameters between subroutines or in mixed-language programs is beyond the scope of this text and is not discussed.

We discussed the stack in Chap. 15. We'll review a portion of that chapter here.

23-1 STACK AND STACK POINTER

The stack, in the case of the microprocessors used in this text, is located in RAM. Refer to Fig. 23-1.

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, and 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. 23-1 has come to be. To do that, refer to Fig. 23-2. Data item #1 is the first item we wish to place on the stack.

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. 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. 23-3.

We have pushed data item #1 onto the stack, and the stack pointer has been decremented or decreased by 1,

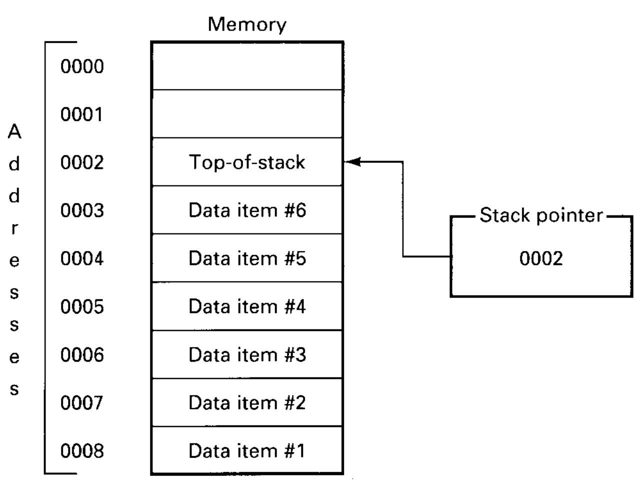

Fig. 23-1 Typical stack and stack pointer.

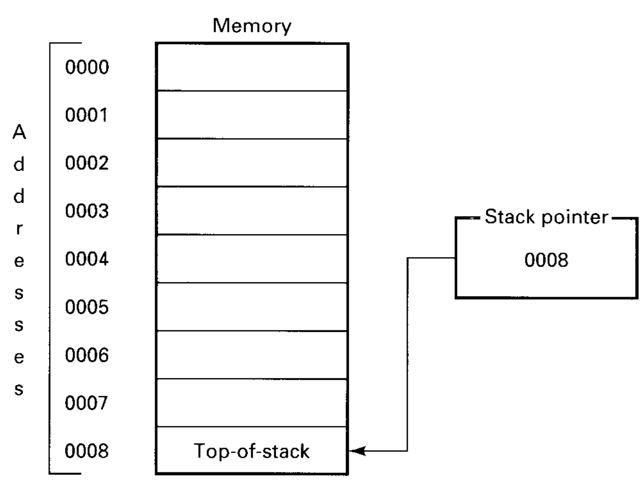

Fig. 23-2 Typical stack and stack pointer.

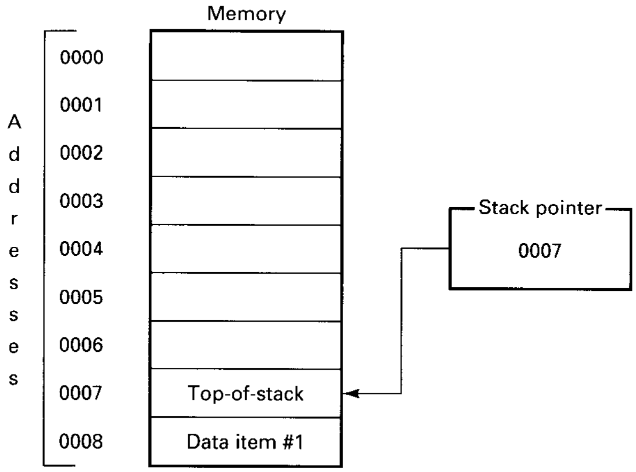

Fig. 23-3 Typical stack and stack pointer.

which means that it is now pointing to memory location 0007. Now 0007 is the top-of-the-stack. Now let's push data item #2 onto the stack. The stack will appear as it does in Fig. 23-4.

When data item #2 was *pushed* onto the stack, it went into the location which was being pointed to by the stack pointer, which was 0007. The stack pointer was then decremented to 0006. This process will be repeated until the stack appears as it did in Fig. 23-1.

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 increment or point to the next-greater memory address.

23-2 BRANCHING VERSUS SUBROUTINES

In Chap. 22, where branching was discussed, we saw that branching causes program execution to *jump* or *branch* to another section of the program. This may be an unconditional jump or a conditional jump. In either case the instructions immediately following the jump instruction may not be executed. If we branch to another section of the program, it is because we don't want to execute the instructions immediately following the branch instructions.

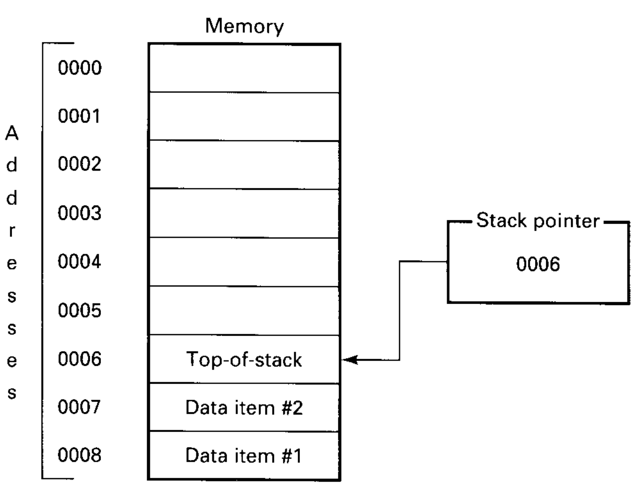

Fig. 23-4 Typical stack and stack pointer.

Subroutines also allow us to jump to another section of the program to execute instructions there. Subroutines differ from jumps or branches, however, in that the instructions which immediately follow the subroutine instruction are executed later. (The act of starting to execute a subroutine is referred to as *jumping to a subroutine* if you are using a 6502 or 6800/6808 microprocessor. It is referred to as *calling a subroutine* if you are using an 8080/8085/Z80 or 8086/8088 microprocessor.)

After the microprocessor jumps to a subroutine or calls a subroutine, the instructions in the subroutine begin to execute. At the end of the subroutine is an instruction called the *return* instruction. The return instruction is usually the last instruction in the subroutine; it tells the microprocessor to go back to the place in the program where it was when the subroutine was called and to pick up where it left off. This is shown in Fig. 23-5.

It is also possible for a subroutine to call another subroutine. These *nested* subroutines then sort of "unwind" and return in the reverse order relative to that in which they were called. This is illustrated in Fig. 23-6.

23-3 HOW DO SUBROUTINES RETURN?

The ability of a subroutine to return to the exact location it came from, especially when nested several layers deep, raises the question of how it knows where to return to.

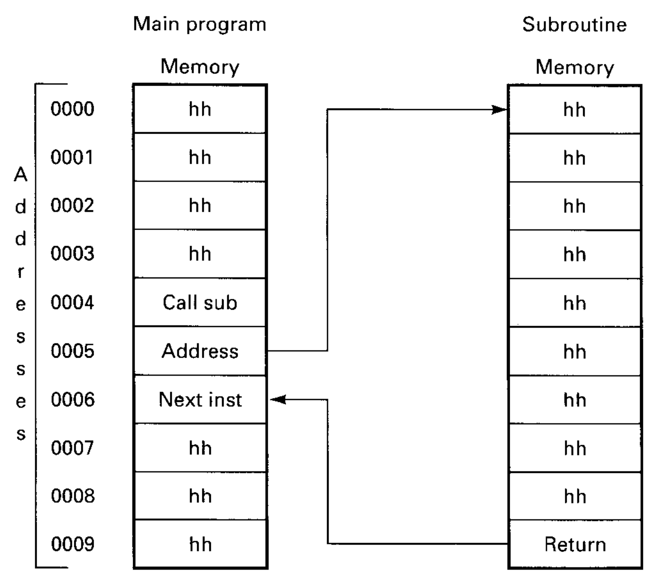

Fig. 23-5 "Calling" or "jumping" to a subroutine.

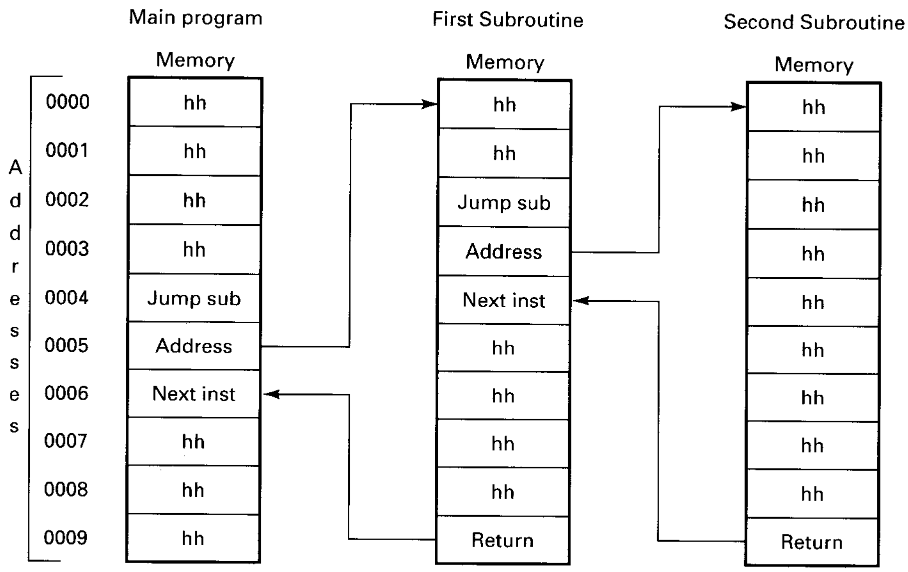

Fig. 23-6 Nested subroutines.

That is, how does it know where it came from? The answer lies in what happens just before the microprocessor leaves the main program, or current subroutine, to go to the subroutine being called.

The microprocessor must know two things before a subroutine can be called or jumped to. First, it must know where it's going, and second, it must know how to get back.

The instruction *jump to subroutine* or *call subroutine* contains the address of the desired subroutine. This may be in the form of an absolute address or an offset of some sort. This is the destination.

The program counter (8086/8088 instruction pointer) contains the address of the next instruction to be executed. This is the point to which the microprocessor needs to return. Refer to Fig. 23-7.

When the subroutine is called, the contents of the program counter are pushed onto the stack. This requires more than one push, since in the case of the 8-bit microprocessors the stack is only 8 bits wide but the program counter is 16 bits wide. (The 8088 stores not only the instruction pointer but may also store the code segment, depending on the type of call—*near* or *far*.)

After the program counter (instruction pointer) is pushed onto the stack, the address of the subroutine which is being called or jumped to is placed in the program counter (instruction pointer), and program execution begins at this new address.

Execution now continues in the subroutine until a return instruction is encountered. Refer to Fig. 23-8.

At this point, the address of the next instruction which was to be executed after the subroutine jump or call, which

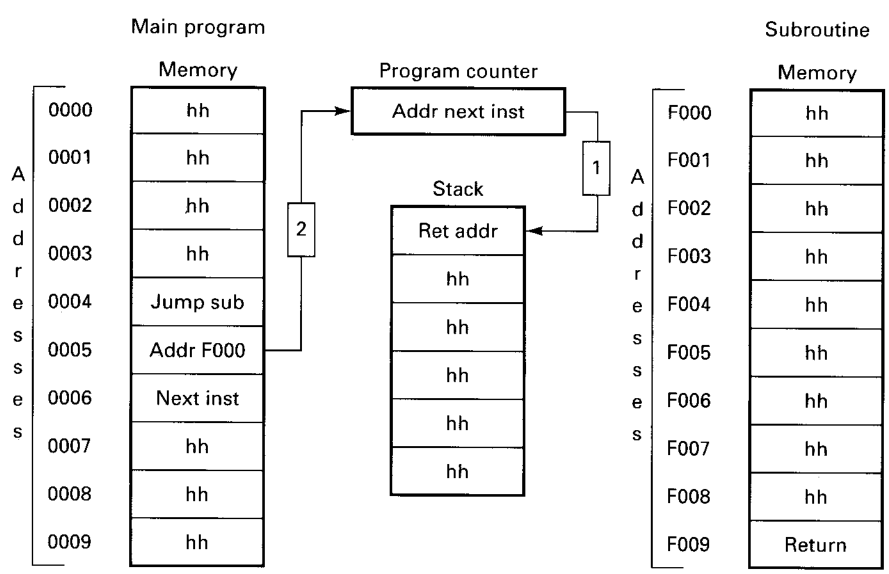

Fig. 23-7 Calling a subroutine.

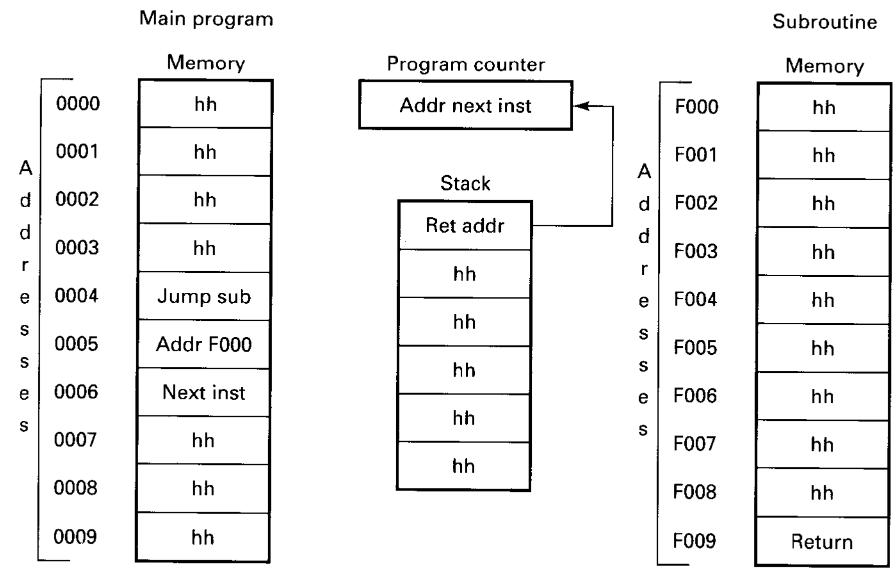

Fig. 23-8 Returning from a subroutine.

has been stored on the stack, is pulled or popped from the stack and placed in the program counter (instruction pointer). Execution then proceeds from that point forward in the main program.

To summarize:

1. The call or jump to subroutine instruction is encountered.
2. The program counter (instruction pointer) is already pointing to the next instruction to be executed (in this section of the program code).
3. The contents of the program counter (instruction pointer) are pushed onto the stack.
4. The address of the subroutine is placed in the program counter (instruction pointer).
5. Program execution now begins in the subroutine.
6. When a return instruction is encountered, the return address, which has been previously stored in the stack, is pulled from the stack and placed in the program counter (instruction pointer).
7. Program execution continues from where it left off before the subroutine was called or jumped to.

23-4 PUSHING AND POPPING REGISTERS

When a subroutine is called or jumped to, the use and operation of the stack are automatic. You don't have to tell the microprocessor to store the return address on the stack. It is done automatically.

In addition to the automatic use of the stack in subroutine calls, the stack can be used directly by the programmer for other purposes. Although each microprocessor is different, in general, you can push onto the stack, and pull from the stack, the contents of some or most of the microprocessor's registers. This is often used to pass values from the main program to subroutines and back, or from subroutine to subroutine. These values are sometimes referred to as *parameters*. The use of the stack in parameter passing, however, is beyond the scope of this text.

Specific Microprocessor Families

Let's look at each of our featured microprocessors. We will not go into great detail about what each microprocessor does automatically before and after a subroutine is called. Rather, we will give examples which show how to call a subroutine and how to nest subroutines.

23-5 6502 FAMILY

The 6502 microprocessor works as described in the New Concepts section of this chapter. There is one point worth noting, however.

The stack pointer of the 6502 is a little different from that of the other microprocessors featured in this text. The changeable portion of the stack pointer is only 8 bits wide (all the others are 16 bits wide) and a 9th bit is always set to 1. This means that the location of the stack must lie in the range from address 0100 to 01FF. This is shown in Fig. 23-9.

Setting the Stack Pointer

Our first example program illustrates how to set the stack pointer to a desired address and then call a subroutine. It

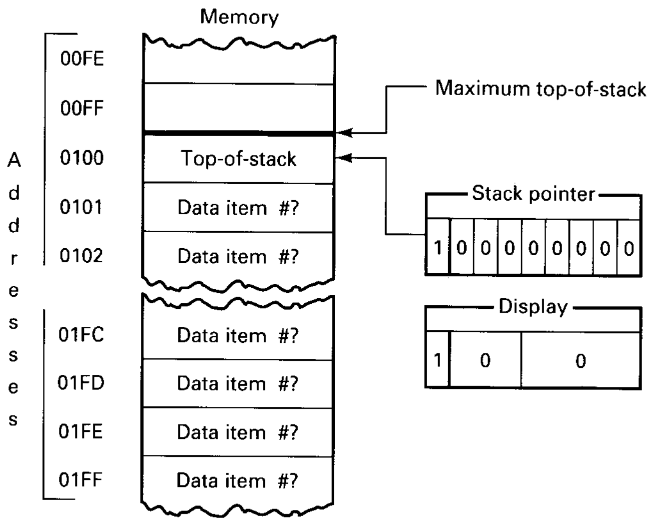

Fig. 23-9 6502 family stack and stack pointer.

is important to note that, with the simple programs we have used throughout this text, setting the stack pointer is normally not required. The microprocessor trainer or computer you are working with will have an operating system that will set the stack pointer to a logical address based on available memory.

Figure 23-10 contains our example program. It sets the stack pointer to a desired address and then calls a subroutine. The subroutine does not actually do anything. It gives you a chance to single-step through a program and watch the stack pointer and program counter.

```
0340                      .ORG $0340
0340                      ;
0340 A2 F9      START:    LDX #$F9     ;load number for stack pointer
0342 9A                   TXS          ;load stack pointer
0343 EA                   NOP          ;misc instructions
0344 20 48 03             JSR SUBRTN   ;jump to subroutine (watch stack pointer)
0347 00                   BRK          ;stop
0348 EA         SUBRTN:   NOP          ;misc instructions
0349 60                   RTS          ;return from subroutine
034A                      ;
034A                      .END
```

Fig. 23-10 6502 program loading stack pointer and calling a subroutine.

```
0340                      .ORG $0340
0340
0340 EA         START:    NOP
0341 20 49 03             JSR RTNE_1   ;
0344 EA                   NOP          ;
0345 20 4B 03             JSR RTNE_2   ;
0348 00                   BRK          ;
0349 EA         RTNE_1:   NOP          ;
034A 60                   RTS          ;
034B EA         RTNE_2:   NOP          ;
034C 60                   RTS          ;
034D
034D                      .END
```

Watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each back to the main program. These subroutines are *not* nested.

Fig. 23-11 6502 program with two subroutines *not* nested.

Calling More than One Subroutine (Not Nested)

Our next example program is shown in Fig. 23-11.

The two subroutines shown here occur one after the other. They are *not* nested. You should single-step through this program and watch the stack pointer and program counter. This is important because the next program will also contain two subroutines, but they *will* be nested. We want you to see the difference between the two.

Again, these first programs do not do anything. Just observe the behavior of the program counter and the stack pointer.

Nesting Subroutines

The program shown in Fig. 23-12 also has two subroutines. They *are* nested, however.

Single-step through this program and watch the stack pointer and the program counter carefully. Notice how they act differently from the way they did in the last program. When you are inside the second subroutine, the stack is holding the return addresses for both subroutines. That's why it decrements further.

Pushing Registers

The example program shown in Fig. 23-13 shows how to use the stack to move information from one register to another.

The program pushes the flags onto the stack and then pulls them from off the stack into the accumulator. The

```
0340                        .ORG $0340
0340
0340 EA          START:     NOP
0341 20 45 03               JSR RTNE_1   ;
0344 00                     BRK          ;
0345 EA          RTNE_1:    NOP          ;
0346 20 4A 03               JSR RTNE_2   ;
0349 60                     RTS          ;
034A EA          RTNE_2:    NOP          ;
034B 60                     RTS          ;
034C
034C                        .END
```

Again, watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each subroutine. These subroutines *are* nested.

Fig. 23-12 6502 program with two *nested* subroutines.

```
0340                        .ORG $0340
0340
0340 08          START:     PHP        :push flags then decrement stack pointer
0341 68                     PLA        ;pull then increment stack pointer
0342 00                     BRK        ;stop
0343
0343                        .END
```

Fig. 23-13 6502 program which pushes a register.

bits of the accumulator, which represent the status of the flags, can now be examined by the program or stored in memory.

A Useful Program Containing a Subroutine

Let's take a look at the program shown in Fig. 23-14.
This program's purpose is as follows:

> This program will read a list of five signed binary numbers. As it reads each number, it will determine whether that number is positive or 0 or negative. If it is positive or 0, it will do nothing with the number. If the number is negative, a subroutine will be entered. This subroutine will find the absolute value of the number (that is, it will make the negative number positive). It will then write this positive number into memory in place of the original negative number. (We used the decimal numbers 3, −4, −2, 0, and 5.) (*Note:* If the microprocessor being used here has a negate instruction, that instruction will not be used.)

Enter this program into your microprocessor trainer or computer and single-step through it. Study the program and make sure that you understand its operation.

```
0340                         .org $0340
0340
0340 A2 00       START:      LDX #$00          ;address of beginning of list
0342 A0 06                   LDY #$06          ;counter
0344 88          GETNUM:     DEY               ;decrement counter
0345 F0 19                   BEQ DONE          ;if no items left end program
0347 BD 61 03                LDA $LIST,X       ;load number from list
034A C9 00                   CMP #$00          ;is it positive/zero or negative?
034C 10 05                   BPL NEXT          ;if positive get next number now
034E F0 03                   BEQ NEXT          ;if zero get next number now
0350 20 57 03                JSR NEGNUM        ;if negative call subroutine
0353 E8          NEXT:       INX               ;point to next number in list
0354 4C 44 03                JMP GETNUM        ;branch back to beginning
0357 49 FF       NEGNUM:     EOR #$FF          ;invert all bits of negative number
0359 18                      CLC               ;prepare for addition
035A 69 01                   ADC #$01          ;add 1 to inverted bits
035C 9D 61 03                STA $LIST,X       ;write absolute value over
                                                  old negative value
035F 60                      RTS               ;return
0360 00          DONE:       BRK               ;stop
0361
0361 03FCFE0005  LIST:       .db    3, -4, -2, 0, 5     ;list of 5 numbers
0366
0366                         .end
```

Fig. 23-14 A useful 6502 program which contains a subroutine.

```
0100                    .ORG $0100
0100                    ;
0100 8E 01 FF  START:   LDS #$01FF   ;load stack pointer
0103 01                 NOP          ;misc instructions
0104 BD 01 08           JSR SUBRTN   ;jump to subroutine (watch stack pointer)
0107 3E                 WAI          ;stop
0108 01        SUBRTN:  NOP          ;misc instructions
0109 39                 RTS          ;return from subroutine
010A                    ;
010A                    .END
```

Fig. 23-15 6800/6808 program loading stack pointer and calling a subroutine.

23-6 6800/6808 FAMILY

The 6800/6808 microprocessor works as described in the New Concepts section of this chapter. We'll look at several sample programs which you can enter into your microprocessor trainer or computer and examine.

Setting the Stack Pointer

Our first example program illustrates how to set the stack pointer to a desired address and then call a subroutine. It is important to note that, with the simple programs we have used throughout this text, setting the stack pointer is normally not required. The microprocessor trainer or computer you are working with will have an operating system that will set the stack pointer to a logical address based on available memory.

Figure 23-15 contains our example program. It sets the stack pointer to a desired address and then calls a subroutine. The subroutine does not actually do anything. It gives you a chance to single-step through a program and watch the stack pointer and program counter.

Calling More than One Subroutine (Not Nested)

Our next example program is shown in Fig. 23-16.

The two subroutines shown here occur one after the other. They are *not* nested. You should single-step through this program and watch the stack pointer and program counter. This is important because the next program will also contain two subroutines, but they *will* be nested. We want you to see the difference between the two.

Again, these first programs do not do anything. Just observe the behavior of the program counter and the stack pointer.

Nesting Subroutines

The program shown in Fig. 23-17 also has two subroutines. They *are* nested, however.

```
0100                    .ORG $0100
0100
0100 01        START:   NOP
0101 BD 01 09           JSR RTNE_1   ;
0104 01                 NOP          ;
0105 BD 01 0B           JSR RTNE_2   ;
0108 3E                 WAI          ;
0109 01        RTNE_1:  NOP          ;
010A 39                 RTS          ;
010B 01        RTNE_2:  NOP          ;
010C 39                 RTS          ;
010D
010D                    .END
```

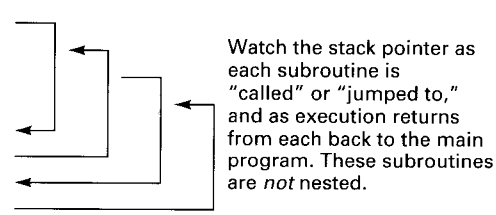

Fig. 23-16 6800/6808 program with two subroutines *not* nested.

```
0100                    .ORG $0100
0100
0100 01        START:   NOP
0101 BD 01 05           JSR RTNE_1   ;
0104 3E                 WAI          ;
0105 01        RTNE_1:  NOP          ;
0106 BD 01 0A           JSR RTNE_2   ;
0109 39                 RTS          ;
010A 01        RTNE_2:  NOP          ;
010B 39                 RTS          ;
010C
010C                    .END
```

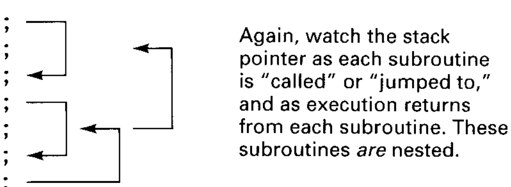

Fig. 23-17 6800/6808 program with two *nested* subroutines.

```
0100                    .ORG $0100
0100
0100 86 12     START:   LDAA #$12    ;load values into
0102 C6 34              LDAB #$34    ;  registers
0104 36                 PSHA         ;push then decrement stack pointer
0105 37                 PSHB         ;push then decrement stack pointer again
0106 32                 PULA         ;pull then increment stack pointer
0107 33                 PULB         ;pull then increment stack pointer again
0108 3E                 WAI          ;stop
0109
0109                    .END
```

Fig. 23-18 6800/6808 program which pushes a register.

Single-step through this program and watch the stack pointer and the program counter carefully. Notice how they act differently from the way they did in the last program. When you are inside the second subroutine, the stack is holding the return addresses for both subroutines. That's why it decrements further.

Pushing Registers

The example program shown in Fig. 23-18 shows how to use the stack to move information from one register to another.

The program loads accumulators A and B with a value, pushes A and B onto the stack, and then pulls them from the stack in reverse order. This places the data that was in A in B and the data that was in B in A.

A Useful Program Containing a Subroutine

Let's take a look at the program shown in Fig. 23-19. This program's purpose is as follows:

> This program will read a list of five signed binary numbers. As it reads each number, it will determine whether that number is positive or 0 or negative. If it is positive or 0, it will do nothing with the number. If the number is negative, a subroutine will be entered. This subroutine will find the absolute value of the number (that is, it will make the negative number positive). It will then write this positive number into memory in place of the original negative number. (We used the decimal numbers 3, -4, -2, 0, and 5.) (*Note:* If the microprocessor being used here has a negate instruction, that instruction will not be used.)

Enter this program into your microprocessor trainer or computer and single-step through it. Study the program and make sure that you understand its operation.

23-7 8080/8085/Z80 FAMILY

The 8080/8085/Z80 microprocessor works as described in the New Concepts section of this chapter. We'll look at several sample programs which you can enter into your microprocessor trainer or computer and examine.

The 8080/8085/Z80 microprocessors do have two features

```
0100                          .org $0100
0100
0100 CE 01 1B       START:    LDX #$LIST       ;address of beginning of list
0103 C6 06                    LDAB #$06        ;counter
0105 5A             GETNUM:   DECB             ;decrement counter
0106 27 12                    BEQ DONE         ;if no items left end program
0108 A6 00                    LDAA $00,X       ;load number from list
010A 81 00                    CMPA #$00        ;is it positive/zero or negative?
010C 2C 03                    BGE NEXT         ;if positive get next number now
010E BD 01 14                 JSR NEGNUM       ;if negative call subroutine
0111 08             NEXT:     INX              ;point to next number in list
0112 20 F1                    BRA GETNUM       ;branch back to beginning
0114 43             NEGNUM:   COMA             ;invert all bits of negative number
0115 8B 01                    ADDA #$01        ;add 1 to inverted bits
0117 A7 00                    STAA $00,X       ;write absolute value over
                                               ;  old negative value
0119 39                       RTS              ;return
011A 3E             DONE:     WAI              ;stop
011B
011B 03FCFE0005     LIST:     .db    3, -4, -2, 0, 5     ;list of 5 numbers
0120
0120                          .end
```

Fig. 23-19 A useful 6800/6808 program which contains a subroutine.

that the other microprocessors featured in this text don't have: They have the ability to perform conditional subroutine calls and to perform conditional returns from subroutines. All the other microprocessors featured in this text have only unconditional calls and unconditional returns.

Setting the Stack Pointer

Our first example program illustrates how to set the stack pointer to a desired address and then call a subroutine. It is important to note that, with the simple programs we have used throughout this text, setting the stack pointer is normally not required. The microprocessor trainer or computer you are working with will have an operating system that will set the stack pointer to a logical address based on available memory.

Figure 23-20 contains our example program. It sets the stack pointer to a desired address and then calls a subroutine. The subroutine does not actually do anything. It gives you a chance to single-step through a program and watch the stack pointer and program counter.

Calling More than One Subroutine (Not Nested)

Our next example program is shown in Fig. 23-21.

The two subroutines shown here occur one after the other. They are *not* nested. You should single-step through this program and watch the stack pointer and program counter. This is important because the next program will also contain two subroutines, but they *will* be nested. We want you to see the difference between the two.

Again, these first programs do not do anything. Just observe the behavior of the program counter and the stack pointer.

Nesting Subroutines

The program shown in Fig. 23-22 also has two subroutines. They *are* nested, however.

Single-step through this program and watch the stack pointer and the program counter carefully. Notice how they act differently from the way they did in the last program. When you are inside the second subroutine, the stack is holding the return addresses for both subroutines. That's why it decrements further.

Pushing Registers

The example program shown in Fig. 23-23 shows how to use the stack to move information from one register to another.

The program loads register pairs BC and DE with a value, pushes BC and DE onto the stack, and then pulls them from the stack in reverse order. This places the data that was in BC in DE, and the data that was in DE in BC.

A Useful Program Containing a Subroutine

Let's take a look at the program shown in Fig. 23-24.

This program's purpose is as follows:

> This program will read a list of five signed binary numbers. As it reads each number, it will determine

```
8080/8085 program

1800                          .ORG 1800h
1800                          ;
1800 31 9E 1F   START:        LXI SP, 1F9Eh    ;load stack pointer
1803 00                       NOP              ;misc instructions
1804 CD 08 18                 CALL SUBRTN      ;call subroutine (watch stack pointer)
1807 76                       HLT              ;stop
1808 00         SUBRTN:       NOP              ;misc instructions
1809 C9                       RET              ;return from subroutine
180A                          ;
180A                          .END


Z80 program

1800                          .ORG 1800h
1800                          ;
1800 31 9E 1F   START:        LD SP,1F9Eh      ;load stack pointer
1803 00                       NOP              ;misc instructions
1804 CD 08 18                 CALL SUBRTN      ;call subroutine (watch stack pointer)
1807 76                       HALT             ;stop
1808 00         SUBRTN:       NOP              ;misc instructions
1809 C9                       RET              ;return from subroutine
180A                          ;
180A                          .END
```

Fig. 23-20 8080/8085/Z80 program loading stack pointer and calling a subroutine.

8080/8085 program

```
1800                     .ORG 1800h
1800
1800 00         START:   NOP
1801 CD 09 18            CALL RTNE_1    ;
1804 00                  NOP            ;
1805 CD 0B 18            CALL RTNE_2    ;
1808 76                  HLT            ;
1809 00         RTNE_1:  NOP            ;
180A C9                  RET            ;
180B 00         RTNE_2:  NOP            ;
180C C9                  RET            ;
180D
180D                     .END
```

Watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each back to the main program. These subroutines are *not* nested.

Z80 program

```
1800                     .ORG 1800h
1800
1800 00         START:   NOP
1801 CD 09 18            CALL RTNE_1    ;
1804 00                  NOP            ;
1805 CD 0B 18            CALL RTNE_2    ;
1808 76                  HLT            ;
1809 00         RTNE_1:  NOP            ;
180A C9                  RET            ;
180B 00         RTNE_2:  NOP            ;
180C C9                  RET            ;
180D
180D                     .END
```

Watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each back to the main program. These subroutines are *not* nested.

Fig. 23-21 8080/8085/Z80 program with two subroutines *not* nested.

8080/8085 program

```
1800                     .ORG 1800h
1800
1800 00         START:   NOP
1801 CD 05 18            CALL RTNE_1    ;
1804 76                  HLT            ;
1805 00         RTNE_1:  NOP            ;
1806 CD 0A 18            CALL RTNE_2    ;
1809 C9                  RET            ;
180A 00         RTNE_2:  NOP            ;
180B C9                  RET            ;
180C
180C                     .END
```

Again, watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each subroutine. These subroutines *are* nested.

Z80 program

```
1800                     .ORG 1800h
1800
1800 00         START:   NOP
1801 CD 05 18            CALL RTNE_1    ;
1804 76                  HALT           ;
1805 00         RTNE_1:  NOP            ;
1806 CD 0A 18            CALL RTNE_2    ;
1809 C9                  RET            ;
180A 00         RTNE_2:  NOP            ;
180B C9                  RET            ;
180C
180C                     .END
```

Again, watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each subroutine. These subroutines *are* nested.

Fig. 23-22 8080/8085/Z80 program with two *nested* subroutines.

```
8080/8085 program

1800                         .ORG 1800h
1800
1800 01 34 12   START:       LXI B,1234h   ;load values into
1803 11 78 56                LXI D,5678h   ;  registers
1806 C5                      PUSH B        ;push then decrement stack pointer
1807 D5                      PUSH D        ;push then decrement stack pointer again
1808 C1                      POP B         ;pull then increment stack pointer
1809 D1                      POP D         ;pull then increment stack pointer again
180A 76                      HLT           ;stop
180B
180B                         .END


Z80 program

1800                         .ORG 1800h
1800
1800 01 34 12   START:       LD BC,1234h   ;load values into
1803 11 78 56                LD DE,5678h   ;  registers
1806 C5                      PUSH BC       ;push then decrement stack pointer
1807 D5                      PUSH DE       ;push then decrement stack pointer again
1808 C1                      POP BC        ;pull then increment stack pointer
1809 D1                      POP DE        ;pull then increment stack pointer again
180A 76                      HALT          ;stop
180B
180B                         .END
```

Fig. 23-23 8080/8085/Z80 program which pushes a register.

whether that number is positive or 0 or negative. If it is positive or 0, it will do nothing with the number. If the number is negative, a subroutine will be entered. This subroutine will find the absolute value of the number (that is, it will make the negative number positive). It will then write this positive number into memory in place of the original negative number. (We used the decimal numbers 3, -4, -2, 0, and 5.) (*Note:* If the microprocessor being used here has a negate instruction, that instruction will not be used.)

Enter this program into your microprocessor trainer or computer and single-step through it. Study the program and make sure that you understand its operation.

23-8 8086/8088 FAMILY

The 8086/8088 microprocessor works as described in the New Concepts section of this chapter. The 8086/8088 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.

We'll look at several sample programs which you can enter into your microprocessor trainer or computer and examine.

Setting the Stack Pointer

Our first example program illustrates how to set the stack pointer to a desired address and then call a subroutine. It is important to note that, with the simple programs we have used throughout this text, setting the stack pointer is normally not required. The microprocessor trainer or computer you are working with will have an operating system that will set the stack pointer to a logical address based on available memory.

Figure 23-25 contains our example program. It sets the stack pointer to a desired address and then calls a subroutine. The subroutine does not actually do anything. It gives you a chance to single-step through a program and watch the stack pointer and program counter.

Calling More than One Subroutine (Not Nested)

Our next example program is shown in Fig. 23-26.

The two subroutines shown here occur one after the other. They are *not* nested. You should single-step through this program and watch the stack pointer and program counter. This is important because the next program will also contain two subroutines, but they *will* be nested. We want you to see the difference between the two.

Again, these first programs do not do anything. Just observe the behavior of the program counter and the stack pointer.

Nesting Subroutines

The program shown in Fig. 23-27 also has two subroutines. They *are* nested, however.

8080/8085 program

```
1800                        .org 1800h
1800
1800 21 1C 18    START:   LXI H,LIST      ;address of beginning of list
1803 06 06                MVI B,06h       ;counter
1805 05          GETNUM:  DCR B           ;decrement counter
1806 CA 1B 18             JZ DONE         ;if no items left end program
1809 7E                   MOV A,M         ;load number from list
180A FE 00                CPI 00h         ;is it positive/zero or negative?
180C F2 12 18             JP NEXT         ;if positive get next number now
180F CD 16 18             CALL NEGNUM     ;if negative call subroutine
1812 23          NEXT:    INX H           ;point to next number in list
1813 C3 05 18             JMP GETNUM      ;branch back to beginning
1816 2F          NEGNUM:  CMA             ;invert all bits of negative number
1817 C6 01                ADI 01h         ;add 1 to inverted bits
1819 77                   MOV M,A         ;write absolute value over old negative value
181A C9                   RET             ;return
181B 76          DONE:    HLT             ;stop
181C
181C 03FCFE0005  LIST:    .db     3, -4, -2, 0, 5     ;list of 5 numbers
1821
1821                      .end
```

Z80 program

```
1800                        .org 1800h
1800
1800 21 1C 18    START:   LD HL,LIST      ;address of beginning of list
1803 06 06                LD B,06h        ;counter
1805 05          GETNUM:  DEC B           ;decrement counter
1806 CA 1B 18             JP Z,DONE       ;if no items left end program
1809 7E                   LD A,(HL)       ;load number from list
180A FE 00                CP 00h          ;is it positive/zero or negative?
180C F2 12 18             JP P,NEXT       ;if positive get next number now
180F CD 16 18             CALL NEGNUM     ;if negative call subroutine
1812 23          NEXT:    INC HL          ;point to next number in list
1813 C3 05 18             JP GETNUM       ;branch back to beginning
1816 2F          NEGNUM:  CPL             ;invert all bits of negative number
1817 C6 01                ADD A,01h       ;add 1 to inverted bits
1819 77                   LD (HL),A       ;write absolute value over old negative value
181A C9                   RET             ;return
181B 76          DONE:    HALT            ;stop
181C
181C 03FCFE0005  LIST:    .db     3, -4, -2, 0, 5     ;list of 5 numbers
1821
1821                      .end
```

Fig. 23-24 A useful 8080/8085/Z80 program which contains a subroutine.

Single-step through this program and watch the stack pointer and the program counter carefully. Notice how they act differently from the way they did in the last program. When you are inside the second subroutine, the stack is holding the return addresses for both subroutines. That's why it decrements further.

Pushing Registers

The example program shown in Fig. 23-28 shows how to use the stack to move information from one register to another.

The program loads registers AX and BX with a value, pushes AX and BX onto the stack, then pulls them from the stack in reverse order. This places the data that was in AX in BX, and the data that was in BX in AX.

A Useful Program Containing a Subroutine

Let's take a look at the program shown in Fig. 23-29.

This program's purpose is as follows:

> This program will read a list of five signed binary numbers. As it reads each number, it will determine

8086/8088 program (with assembler)

```
                        page ,132

0000                    CODE    SEGMENT
                                ASSUME CS:CODE, DS:CODE, SS:CODE
0100                            ORG 0100h

0100   BC FFF9          START:  MOV SP,0FFF9h          ;load stack pointer
0103   90                       NOP                    ;misc instructions
0104   E8 0109 R                CALL SHORT SUBRTN      ;call subroutine (watch stack pointer)
0107   CD 20                    INT 20h                ;stop
0109   90               SUBRTN: NOP                    ;misc instructions
010A   C3                       RET                    ;return from subroutine

010B                    CODE    ENDS

                                END     START
```

8086/8088 program (with DEBUG)

```
MOV     SP,FFF9     ;load stack pointer
NOP                 ;misc instructions
CALL    0109        ;call subroutine (watch stack pointer)
INT     20          ;stop
NOP                 ;misc instructions
RET                 ;return from subroutine
```

Fig. 23-25 8086/8088 program loading stack pointer and calling a subroutine.

8086/8088 program (with assembler)

```
                        page ,132

0000                    CODE    SEGMENT
                                ASSUME CS:CODE, DS:CODE, SS:CODE
0100                            ORG 0100h

0100   90               START:  NOP
0101   E8 010A R                CALL SHORT RTNE_1    ;
0104   90                       NOP                  ;
0105   E8 010C R                CALL SHORT RTNE_2    ;
0108   CD 20                    INT 20h              ;
010A   90               RTNE_1: NOP                  ;
010B   C3                       RET                  ;
010C   90               RTNE_2: NOP                  ;
010D   C3                       RET                  ;

010E                    CODE    ENDS

                                END     START
```

Watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each back to the main program. These subroutines are *not* nested.

8086/8088 program (with DEBUG)

```
NOP
CALL    010A    ;
NOP             ;
CALL    010C    ;
INT     20      ;
NOP             ;
RET             ;
NOP             ;
RET             ;
```

Watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each back to the main program. These subroutines are *not* nested.

Fig. 23-26 8086/8088 program with two subroutines *not* nested.

8086/8088 program (with assembler)

```
                     page ,132

  0000               CODE     SEGMENT
                              ASSUME CS:CODE, DS:CODE, SS:CODE
  0100                        ORG 0100h

  0100  90           START:   NOP
  0101  E8 0106 R             CALL SHORT RTNE_1   ;
  0104  CD 20                 INT 20h             ;
  0106  90           RTNE_1:  NOP                 ;
  0107  E8 010B R             CALL SHORT RTNE_2   ;
  010A  C3                    RET                 ;
  010B  90           RTNE_2:  NOP                 ;
  010C  C3                    RET                 ;

  010D               CODE     ENDS

                              END      START
```

Again, watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each subroutine. These subroutines *are* nested.

8086/8088 program (with DEBUG)

```
NOP
CALL    0106    ;
INT     20      ;
NOP             ;
CALL    010B    ;
RET             ;
NOP             ;
RET             ;
```

Again, watch the stack pointer as each subroutine is "called" or "jumped to," and as execution returns from each subroutine. These subroutines *are* nested.

Fig. 23-27 8086/8088 program with two *nested* subroutines.

8086/8088 program (with assembler)

```
                     page ,132

  0000               CODE     SEGMENT
                              ASSUME CS:CODE, DS:CODE, SS:CODE
  0100                        ORG 0100h

  0100  B8 1234      START:   MOV AX,1234h      ;load values into
  0103  BB 5678               MOV BX,5678h      ;  registers
  0106  50                    PUSH AX           ;push then decrement stack pointer
  0107  53                    PUSH BX           ;push then decrement stack pointer again
  0108  58                    POP AX            ;pop then increment stack pointer
  0109  5B                    POP BX            ;pop then increment stack pointer again
  010A  CD 20                 INT 20h           ;stop

  010C               CODE     ENDS

                              END    START
```

8086/8088 program (with DEBUG)

```
MOV     AX,1234    ;load values into
MOV     BX,5678    ;  registers
PUSH    AX         ;push then decrement stack pointer
PUSH    BX         ;push then decrement stack pointer again
POP     AX         ;pop then increment stack pointer
POP     BX         ;pop then increment stack pointer again
INT     20         ;stop
```

Fig. 23-28 8086/8088 program which pushes a register.

8086/8088 program (with assembler)

```
                              page ,132

0000                          CODE    SEGMENT
                                      ASSUME CS:CODE, DS:CODE, SS:CODE
0100                                  ORG 0100h

0100  BB 0000                 START:  MOV BX,00h                  ;address of beginning of list
0103  B1 06                           MOV CL,06h                  ;counter
0105  FE C9                   GETNUM: DEC CL                      ;decrement counter
0107  74 17                           JZ DONE                     ;if no items left end program
0109  8A 87 0122 R                    MOV AL,[BYTE PTR LIST + BX] ;load number from list
010D  3C 00                           CMP AL,00h                  ;is it positive/zero or negative?
010F  7D 03                           JGE NEXT                    ;if positive get next number now
0111  E8 0117 R                       CALL NEGNUM                 ;if negative call subroutine
0114  43                      NEXT:   INC BX                      ;point to next number in list
0115  EB EE                           JMP GETNUM                  ;branch back to beginning
0117  F6 D0                   NEGNUM: NOT AL                      ;invert all bits of negative number
0119  04 01                           ADD AL,01h                  ;add 1 to inverted bits
011B  88 87 0122 R                    MOV [BYTE PTR LIST + BX],AL ;write absolute value over old
                                                                    negative value
011F  C3                              RET                         ;return
0120  CD 20                   DONE:   INT 20h                     ;stop

0122  03 FC FE 00 05          LIST:   db      3, -4, -2, 0, 5      ;list of 5 numbers


0127                          CODE    ENDS

                                      END     START
```

8086/8088 program (with DEBUG)

```
a 0100
MOV     BX,0000          ;address of beginning of list
MOV     CL,06            ;counter
DEC     CL               ;decrement counter
JZ      0120             ;if no items left end program
MOV     AL,[BX+0122]     ;load number from list
CMP     AL,00            ;is it positive/zero or negative?
JGE     0114             ;if positive get next number now
CALL    0117             ;if negative call subroutine
INC     BX               ;point to next number in list
JMP     0105             ;branch back to beginning
NOT     AL               ;invert all bits of negative number
ADD     AL,01            ;add 1 to inverted bits
MOV     [BX+0122],AL     ;write absolute value over old negative value
RET                      ;return
INT     20               ;stop

e 0122  03 FC FE 00 05
```

Fig. 23-29 A useful 8086/8088 program which contains a subroutine.

whether that number is positive or 0 or negative. If it is positive or 0, it will do nothing with the number. If the number is negative, a subroutine will be entered. This subroutine will find the absolute value of the number (that is, it will make the negative number positive). It will then write this positive number into memory in place of the original negative number. (We used the decimal numbers 3, -4, -2, 0, and 5.) (*Note:* If the microprocessor being used here has a negate instruction, that instruction will not be used.)

Enter this program into your microprocessor trainer or computer and single-step through it. Study the program and make sure that you understand its operation.

SELF-TESTING REVIEW

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

1. ________ are used when there are common tasks which must be executed or used many times.
2. *(Subroutines)* The structure of the stack is a ________ type of structure.
3. *(FILO)* The act of putting a piece of data on the top of the stack is called ________ the data onto the stack.
4. *(pushing)* The act of removing a piece of data from the top of the stack is called ________ or ________ the data from the stack.
5. *(pulling, popping)* The instruction that is usually the last instruction in a subroutine, and that tells the microprocessor to go back to the place where it was before the subroutine was called, is the ________ instruction.
6. *(return)* In general, the programmer can push onto and pull from the stack one or more of the microprocessor's ________.

(registers)

PROBLEMS

Solve the following problem using the microprocessor of your choice. This will be the longest program you have written thus far. Therefore, this chapter has only this one program for you to write. The program can be considered correct only if it causes the correct values to be placed in the counter variables *and* alters the original list correctly.

23-1. A 1-byte unsigned number can range from 00 to FF. Each number in this range has a corresponding ASCII value. The primary categories within the ASCII table are shown below. (The characters from 80–FF are not actually official ASCII characters but are used to form the extended IBM character set.)

00 – 1F	various control characters
20 – 2F	punctuation marks
30 – 39	numbers
3A – 40	punctuation marks
41 – 5A	uppercase letters
5B – 60	punctuation marks
61 – 7A	lowercase letters
7B – 7F	punctuation marks
80 – FF	foreign letters, boxes, math symbols, miscellaneous

Write a program in which the main part of the program examines consecutive bytes from a list which ends with the number FF. This main program section then determines which category each value in the list is from. Different subroutines will then be called, depending on which category a value belongs to.

If the value represents a lowercase letter, a subroutine called LOWER will increment a memory location called NUM_LW, which indicates the number of lowercase letters found.

If the value represents an uppercase letter, a subroutine called UPPER will increment a memory location called NUM_UP, which indicates the number of uppercase letters found.

If the value represents a number, a subroutine called NUM will change the number to its corresponding binary value. (The ASCII value for a number and the binary value for that number are not the same.) The subroutine will then store the binary value in the list in place of the original ASCII value and then increment a memory location called NUM_N, which indicates the number of numbers found.

If the number represents a control character, the program will do nothing.

If the value represents a punctuation mark, a subroutine called PUNCT will increment a memory location called NUM_P, which indicates the number of punctuation marks found.

If the value represents one of the special characters in the range from 80 to FF, a subroutine called SPECL will change the uppermost bit of the number from a 1 to a 0. This change will cause the value to fit into one of the previously mentioned categories. The subroutine SPECL will then return to the main program, which is to be arranged in such a way that this converted value will be evaluated a second time to determine its new category and have the appropriate subroutine called.

Place the following hexadecimal values in the list: 00, 1F, 20, 2F, 30, 39, 3A, 40, 41, 5A, 5B, 60, 61, 7A, 7B, 7F, 80, and FF. (FF is not actually a value to be evaluated but marks the end of the list.)