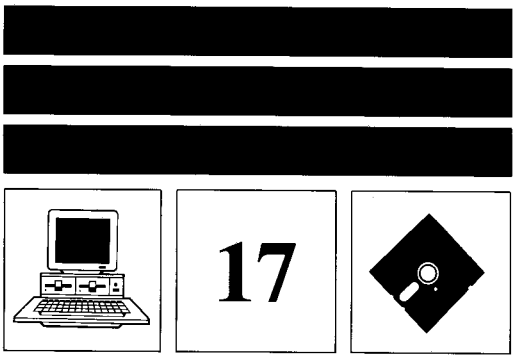


17

ADDRESSING MODES—I

New Concepts

In this chapter we will study the simplest of the different addressing modes. This will provide a foundation for the next couple of chapters. In a later chapter we will look at the more complex addressing modes. First we need to learn what an addressing mode is.

17-1 WHAT IS AN ADDRESSING MODE?

In an earlier chapter we used the system of addressing homes as a way to describe memory addressing. Let's use the same idea to describe addressing modes.

If you are moving and want to describe to the movers how to get to your new home so that they can deliver your belongings, you would give them the name of the state, city, street, and house number.

But what if you were moving to an apartment in Canada? In that case you would give them the name of the country, province, city, street, apartment complex, and apartment number.

Or what if you were moving to a backwoods cabin for a summer in the wilderness? You would give them the name of the state, county, county road, the direction and number of miles to travel on that county road, and finally landmarks to help them find the cabin. (OK, you probably won't have a truck moving all of your belongings to a wilderness cabin, but the analogy worked well up to that point.)

You can see that we need more than one way to "address" or describe a location because not every method works in every circumstance. This is what addressing modes are about.

How you describe a location you want to transfer a number to can depend on several factors. Remember that while the addressing mode which should be used is very apparent in some cases, in other cases choosing the best addressing mode requires skill that must be developed over time.

17-2 THE PAGING CONCEPT

Before we go any further into the subject of addressing modes, we need to look at the concept of paging. *Paging* is the concept of dividing memory into blocks of 256 bytes each. Each block is called a *page*. We have to look at how we count in hexadecimal to see why this number was chosen.

The number 256 was chosen because that is how far you can count using only two hex digits. Actually, FF, which is the highest two-digit hex number, is 255 (decimal), but if you count 00, you have 256 different numbers, or in this case, memory locations.

Counting from 00_{16} to FF_{16} using four hex digits looks like this:

```
0000
0001
0002
:
:
00FE
00FF
```

Notice that the left two digits are always 0. The range of hex numbers from 00 to FF is called *page 00* (sometimes called the *zero page*).

The next number after 00FF is 0100. Let's continue counting from there:

```
0100
0101
0102
:
:
01FE
01FF
```

Notice that the left two digits are 01. This is called *page one*.

The next number in the sequence is 0200. This is the beginning of page two. Page two ends with 02FF, after which comes 0300, the beginning of page three.

This process continues up to $FFFF_{16}$. There are 256 of these pages, with 256 bytes per page.

The addressing modes of the 8085 do not reference page numbers; however, the 6800/6808 and 6502 do have addressing modes that depend upon the concept of paging.

17-3 BASIC ADDRESSING MODES

We are now going to study the four most basic addressing modes. As you read about each mode, first and foremost try to understand the concept. The actual name of the addressing mode may be different for the microprocessor which you are using. After you read about these four modes, go to the section which covers your particular microprocessor for specific details.

Implied Addressing

In *implied addressing*, sometimes called *inherent addressing*, no address is necessary because the location is implied in the instruction itself. It is the simplest of all addressing modes. You used this mode in Chap. 16, which discussed the CPU control instructions. Remember the NOP (no operation) instruction? Do we have to tell it where to do nothing? No. No addressing is necessary.

Another example would be the case of a microprocessor which has only one accumulator and a certain index register. The 6502, for example, has an instruction called

TAX

which means

Transfer **A**ccumulator to **X** register

There is only one accumulator, and the specified register is the X register. The microprocessor knows exactly where the accumulator and the X register are, so we say that the address is *implied* in the instruction itself. The data will be transferred from the accumulator to the X register.

Register (Accumulator) Addressing

Register addressing, sometimes called *accumulator addressing*, involves only internal registers or an accumulator(s) and no external RAM. For example, the 8085 microprocessor has an instruction called

MOV A,B

which means

MOVe data to **A** from **B**

Since the data is being moved from one register to another, no other address information is needed. The names of the registers are enough.

It should be noted that with some microprocessors it is not clear whether this is considered to be a separate addressing mode or a special subtype of the implied addressing mode. See your particular microprocessor section for details.

Immediate Addressing

Immediate addressing is a mode in which the number or data to be operated on or moved is in the memory location *immediately* following the instruction op code. For example, the 6800/6808 microprocessor has an instruction called

LDAA #$dd

which means

> **LoaD** Accumulator **A** with the two hexadecimal ($) digits of data (dd) immediately (#) following this op code

In the computer's memory there will be the hex number 86, which is the op code for the LDAA immediate instruction, followed immediately by the two hex digits which we have called dd (since we don't know what their actual value is right now).

Direct Addressing

Direct addressing uses an op code followed by a 1- or 2-byte memory address where the data which is to be used can be found. The data is outside of the microprocessor itself, in one of the many thousands of memory locations. The Z80 has an instruction called

LD A, (aaaa)

which means

> **LoaD** the **A**ccumulator with the data found at memory location (aaaa)

Here of course the aaaa is four hex digits, which makes a 16-bit address. The microprocessor will go to memory address aaaa and place a copy of the contents of that address in the accumulator.

Keep in mind that some microprocessors do not call this "direct" addressing and that some have more than one form of this addressing mode.

Specific Microprocessor Families

Go to the section which discusses the microprocessor you are using.

17-4 6502 FAMILY

The 6502 uses the implied and immediate modes as described in the New Concepts section of this chapter. The register mode and direct mode are a little different.

Implied Addressing

For an example of implied addressing refer to the Data Transfer Instructions part of the 6502 instruction set in the Expanded Table of 6502 Instructions Listed by Category.

Find the TAX instruction, which is an example of implied addressing as noted in the Address Mode column. Notice that it **T**ransfers the contents of the **A**ccumulator to the **X** register as indicated in the Operation and Boolean/Arithmetic Operation columns. Of course, no other information is needed since both of these locations are inside the microprocessor itself.

Register (Accumulator) Addressing

The 6502 doesn't use register addressing as a dominant addressing mode like the 8080/8085 does. It does use it in four instances, however, and calls it *accumulator* addressing.

For example, the 6502 instruction

ASL

which stands for **A**rithmetic **S**hift **L**eft, shifts every bit in the accumulator to the left one place. The operand is in the accumulator.

There are only four 6502 instructions which use the register or accumulator addressing mode: they are ASL A (**A**rithmetic **S**hift **L**eft **A**ccumulator), LSR A (**L**ogical **S**hift **R**ight **A**ccumulator), ROL A (**RO**tate **L**eft **A**ccumulator), and ROR A (**RO**tate **R**ight **A**ccumulator). All these instructions can be found in the Rotate and Shift Instructions section of the Expanded Table of 6502 Instructions Listed by Category.

Immediate Addressing

Now let's look at an example of immediate addressing. In the Data Transfer section of the 6502 instruction set, notice the first form of the LDA instruction. It uses immediate addressing, which is what the # in the Assembler Notation column stands for. The $ means that the number is hexadecimal (not decimal). The dd stands for two hex digits such as 35 or E2. To load the accumulator with the hex number E2, you would type

LDA #$E2

Direct Addressing

The 6502 has two different types of direct addressing. One is called zero page addressing, and the other absolute addressing.

Zero page addressing is direct addressing in which the target address is in page zero of memory, somewhere in the first 256 bytes of memory, between 0000_{16} and $00FF_{16}$. Since the first two hex digits of any address in zero page are 00s, the 00s can be omitted, making it possible to describe the address with only 1 byte.

Absolute addressing is a form of direct addressing in which the target address can be anywhere from 0000_{16} to $FFFF_{16}$. This requires four hex digits, which is a 2-byte address.

Referring again to the LDA instruction, the third form down is the zero page addressing form of the instruction. Notice that the assembler notation form appears as

LDA $aa

The two lowercase a's indicate a two-digit hex address. The second form of the LDA instruction is the absolute addressing form. The assembler notation in this case appears as

LDA $aaaa

which means that the address consists of four hex digits (2 bytes).

(*Note:* The 6502 microprocessor expresses addresses in reverse low-byte/high-byte order!)

6502 Summary

Some examples are

```
NOP        ← Implied addressing
ASL        ← Register (accumulator) addressing
LDA #$35   ← Immediate addressing
LDA $1E    ← Direct (zero page) addressing
LDA $123D  ← Direct (absolute) addressing
```

17-5 6800/6808 FAMILY

The 6800/6808 uses the implied and immediate modes as described in the New Concepts section. The register and direct modes are a little different.

Implied Addressing

For an example of implied addressing, refer to the Data Transfer Instructions part of the Expanded Table of 6800 Instructions Listed by Category.

Find the TAB instruction, which is an example of implied addressing as noted in the Address Mode column. Notice that it transfers the contents of accumulator A to accumulator B as indicated in the Operation and Boolean/Arithmetic Operation columns. No other information is needed since both of these locations are inside of the microprocessor itself.

Register (Accumulator) Addressing

The 6800/6808 doesn't use register addressing as a dominant addressing mode the way the 8080/8085 does. Technically, it does use it, however, and calls it *accumulator* addressing. Since it is often considered a special form of implied addressing by many who use the 6800/6808, it has not been included in the Address Mode column of the instruction sheets but rather falls under the title of Implied addressing.

Immediate Addressing

Now let's look at an example of immediate addressing. In the Data Transfer section of the 6800/6808 instruction set notice the first form of the LDAA instruction. It uses immediate addressing, which is what the # in the Assembler Notation column stands for. The $ means that the number is hexadecimal (not decimal). The dd stands for two hex digits such as E2. The instruction which would **LoaD** accumulator **A** with the value E2 would appear as

LDAA #$E2

Direct Addressing

The 6800/6808 has two different types of direct addressing. One is called direct addressing, and the other extended addressing.

Direct addressing is a form of direct addressing in which the target address is in page zero of memory—that is, somewhere in the first 256 bytes of memory between 0000_{16} and $00FF_{16}$. Since the first two hex digits of any address in this range are 00, the 00s can be omitted, making it possible to designate the address with only 1 byte.

Extended addressing is a form of direct addressing in which the target address can be anywhere from 0000_{16} to $FFFF_{16}$. This requires four hex digits, which is a 2-byte address.

Referring to the LDAA instruction, notice that the second form down is the direct addressing form of the instruction. The assembler notation appears as

LDAA $aa

which means there are only two hex address digits, indicated by aa (a stands for address). The fourth LDAA form is the extended addressing form of the instruction. The assembler notation in this case appears as

LDAA $aaaa

which means that there are four hex address digits (2 bytes).

6800/6808 Summary

Some examples are

TAB	← Implied addressing
TAB	← Register (accumulator) addressing
LDAA #$35	← Immediate addressing
LDAA $1E	← Direct addressing
LDAA $123D	← Direct (extended) addressing

17-6 8080/8085/Z80 FAMILY

The 8080/8085/Z80 uses the implied, immediate, register, and direct addressing modes as described in the New Concepts section of this chapter.

Note that the Z80 has all of the addressing modes that the 8080/8085 has, plus a number of addressing modes that the 8080/8085 does not have. We do not include these additional modes of the Z80 in either the text or the instruction set tables in this book. Refer to one of the many books available about the Z80 to learn about these other modes.

We need to bring your attention to a sometimes confusing fact about the 8080/8085/Z80 mnemonics. Look at the Data Transfer Instructions section of the Expanded Table of 8080/8085/Z80 (8080 subset) Instructions Listed by Category. Now look at the MOV A,B *[Z80 = LD A,B]* instruction (the second instruction in this section). Notice in the Boolean/Arithmetic Operation column that the data is moving from B toward A. This means that the mnemonic places the destination register before the source register. This is true of the entire 8080/8085/Z80 instruction set. The MOV A,B instruction is moving data *to* A *from* B. (*Note:* The 6502 and 6800/6808 are just the reverse.)

Implied Addressing

An example of implied addressing can be seen in the CPU Control Instructions section of the 8080/8085/Z80 instruction set. The NOP instruction uses implied addressing since no address is necessary. In the Flag Instructions section you can see another example. The STC (**SeT** Carry flag) instruction uses implied addressing. The carry flag is inside the 8080/8085/Z80 microprocessor. Therefore no other address information is needed.

Register (Accumulator) Addressing

This form of addressing is called *register addressing* with the 8080/8085 (in contrast to the term *accumulator addressing* used by the 6502 and 6808). The 8080/8085/Z80 uses this form of addressing very frequently. In fact, if you browse through the Data Transfer Instructions section of the Expanded Table of 8085/8080 and Z80 (8080 Subset) Instructions Listed by Category, you will find that most of these instructions use this form of addressing.

For example, the instruction MOV A,B *[Z80 = LD A,B]* moves or makes a copy of the data in the B register and places it in the A register. (We normally call this the accumulator.) Since external memory is not utilized, and both the source of the data and its destination are inside the microprocessor, this information is sufficient.

Immediate Addressing

The 8080/8085/Z80 microprocessors use the immediate mode as described in the New Concepts section at the beginning of the chapter.

To see an example of this mode, scan through the 8080/8085/Z80 instruction set in the Data Transfer Section until you come to the MVI A,dd *[Z80 = LD A,dd]* instruction (the 64th instruction in that section). You'll notice in the Address Mode column that this is labeled as using the immediate addressing mode. This means that the op code for this instruction (3E) would be followed immediately by the two hex digits we want moved.

If the Hex number C8 was the value we wanted to load into the accumulator, the 8080/8085 assembly-language notation would appear as

MVI A,C8 *[LD A,C8]*

The second instruction, in brackets and in italics, is the Z80 form.

Direct Addressing

The direct addressing mode as implemented in the 8080/8085/Z80 microprocessors works as described in the New Concepts section of this chapter.

The 8080/8085/Z80 has only one form of direct addressing. (The 6502 and the 6800/6808 have two forms of this addressing mode.)

To find an example of this mode, scan through the Data Transfer Instructions section of the 8080/8085/Z80 instruction set until you find the LDA aaaa *[LD A,(aaaa)]* instruction (the 78th instruction in this section). The op code for this instruction is 3A. It uses 3 bytes of memory. The 1st byte will be the op code, 3A. The 2d and 3d bytes will be the address of the memory location where the data can be found.

(*Note:* The 8080/8085/Z80 microprocessors express addresses in reverse low-byte/high-byte order!) If we wanted to load the accumulator from memory location 1234, the 3 bytes of object code would be

3A 34 12

in the op code/high-byte/low-byte sequence.

The assembly-language notation for this instruction would appear as

LDA 1234 *[LD A, (1234)]*

8080/8085/Z80 Summary

Some examples are

NOP	← Implied addressing
MOV A,B *[LD A,B]*	← Register addressing
MVI A,C8 *[LD A,C8]*	← Immediate addressing
LDA 1234 *[LD A, (1234)]*	← Direct addressing

17-7 8086/8088 FAMILY

Most of the 8086/8088 instructions are implemented as described in the New Concepts section of this chapter.

We need to bring to your attention a sometimes confusing fact about 8086/8088 mnemonics. The 8086/8088 mnemonics place the destination register before the source register. This is true of the entire 8086/8088 instruction set. The MOV AL,BL instruction is moving data *to* AL *from* BL. (*Note:* This is similar to the 8080/8085/Z80 microprocessors.)

Implied Addressing

Implied addressing works on the 8086/8088 microprocessors as described in the New Concepts section of this chapter. Two examples are HLT (halt) and NOP (no operation).

Register Addressing

Register addressing also works as described in the New Concepts section of this chapter. Since the 8086/8088 chips have eight 8-bit (or four 16-bit) general-purpose registers in addition to a number of other special-purpose registers, there are *hundreds* of move combinations. Let's look at one of them.

The instruction which moves the contents of the CX register into the BX register looks like this:

MOV BX,CX

Again you should notice that where the data is going to (BX) is written *first*, and where the data is coming from (CX) is written *last*.

Since only registers are involved, all of which are inside the microprocessor, no other information is needed by the microprocessor.

Immediate Addressing

Immediate addressing on the 8086/8088 is as described in the New Concepts section of this chapter. For example, the instruction MOV AL,37 would place the hexadecimal number 37 in the AL register.

Memory Segmentation

Before we can discuss direct addressing, we need to look at a feature of the 8086/8088 microprocessors which does not exist in any of the 8-bit microprocessors used in this book. That feature is memory segmentation.

Earlier in this chapter we discussed the paging concept. Segmentation is an extension of that concept. The 8-bit microprocessors use 16-bit addresses. That gives them a range from 0000_{16} to $FFFF_{16}$. In decimal that is 65,535, which gives us a total of 65,536 different memory locations counting location 0000_{16}. Another way to express this is as 64 kilobytes, or 64K. Notice that the addresses from 0000_{16} to $FFFF_{16}$ use four hex digits. The two right-most digits express which byte is being referred to. The two left-most digits express which page the bytes are in. There are 256 bytes per page and 256 pages from 0000_{16} to $FFFF_{16}$.

The 8086/8088 chips use a larger 20-bit address instead of the 16-bit address used by the 8-bit chips. Twenty bits is five hexadecimal digits. This provides a range from 00000_{16} to $FFFFF_{16}$. In decimal this is 1,048,575, which gives us 1,048,576 memory locations (since we can count 00000_{16}), or 1 megabyte of memory.

A segment is a 64K block of memory; thus there could be as many as 16 nonoverlapping segments in 1M (megabyte) of memory. Unlike a memory page, however, a segment is not bound to a certain location. The only requirement is that a segment must start on a 16-byte memory boundary. Segments can be nonoverlapping, they can partially overlap, or they can be superimposed with one exactly on top of the other. The 8086/8088 has four segment registers and so can manage four different segments at a time.

Direct addressing uses not only the address specified in the instruction but also the address in one of the segment registers. In the case of move instructions, the data segment register is used. The process involves adding the address you have specified to the address in the data segment register after shifting the data segment register to the left one hexadecimal digit. For example, if you said

MOV DL,[0100]

and if the data segment register contained 2000, the address would be calculated in the following manner.

2000	data segment register (shifted left)
+ 0100	address
20100	effective address

Notice that the contents of the data segment register have been shifted to the left one place. (You can think of it as adding a 0 to the right side of the data segment register.) So the MOV DL,[0100] instruction places a copy of the data found at memory location 20100_{16} (*not* location 0100_{16}) in the DL register.

We generally won't be concerned with segment registers in this text since our programs are simple and very small. All the segment registers will be the same, so the offset (the address of the instruction pointer) will be all we must pay attention to.

Direct Addressing

Except for memory segmentation, direct addressing on the 8086/8088 is quite like that used on the 8-bit chips. When we use the term *direct addressing* in reference to the 8086/8088, we are referring to the direct form of addressing used when manipulating data. (See the following topic, Program Direct Addressing, for the other use of direct addressing.)

For example, if the data segment register contains the number 0723, and the instruction

MOV BL,[0100]

is encountered, the contents of memory location 07330_{16} (07230 + 0100 = 07330) would be copied into the BL register.

Program Direct Addressing

Program direct addressing is no different from direct addressing: It is simply direct addressing used for a different purpose.

Program direct addressing is used with JMP and CALL instructions. These instructions direct the "flow" of the program. They are not used to manipulate data. Which instruction or subroutine is to be executed next can be altered with the JMP and CALL instructions.

For example, the instruction

JMP 100

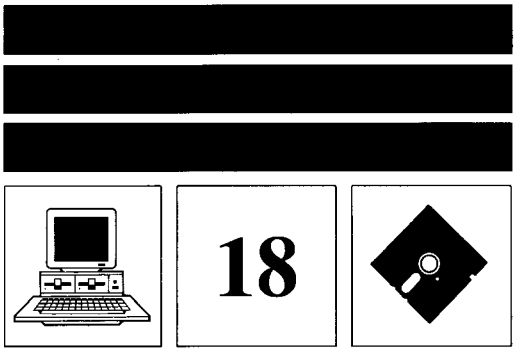
tells the microprocessor to execute the instruction found at location 0100 (hex) in the program segment. This is an example of program direct addressing.

The offset (100 in the above example) is added to the *code segment* register rather than the data segment register. Remember that the contents of the code segment register, like the data segment register, are shifted one hexadecimal place to the left before being added to the offset.

8086/8088 Summary

Some examples are

```
NOP             ← Implied addressing
MOV BX,CX       ← Register addressing
MOV AL,37       ← Immediate addressing
MOV BL,[0100]   ← Direct addressing
JMP 100         ← Direct (program direct) addressing
```



ARITHMETIC AND FLAGS

In this chapter we will study the arithmetic instructions of each of our microprocessor families. We will also look at the closely related topic of flags, at how they react to arithmetic instructions, and at the instructions which control them.

New Concepts

There are several main topics in this chapter. We will learn about (and review) the number systems microprocessors use. We will study addition and subtraction (as well as multiplication and division on the 16-bit 8086/8088). And finally we will study the flags which are affected by these arithmetic operations and how to alter the condition of those flags.

18-1 MICROPROCESSORS AND NUMBERS

We must first look at the kind of numbers a microprocessor performs arithmetic operations on. You have already studied much of this in earlier chapters.

Binary and Hexadecimal Numbers

We introduced binary numbers in Chap. 1. If that was the first time you had ever seen numbers in another base system, the whole subject may have been a bit confusing. It all becomes quite natural, though, with time and experience.

At this point there are a couple of very important skills which you must have. You should be able to look at an 8-bit binary number and know the decimal value of each of the bit's positions. This is illustrated in Fig. 18-1.

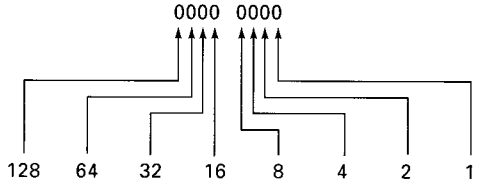


Fig. 18-1 Decimal values of each bit of an 8-bit binary number.

You should also be able to add the decimal values of each binary digit to determine the decimal value of the complete binary number. See Chap. 1 if you have forgotten how to do this.

Another skill which was stressed in Chap. 1 is now necessary if you are to work with microprocessors effectively. This is the ability to recognize any 4-bit binary number, its hexadecimal equivalent, and its decimal equivalent. The table which illustrates this appeared in Chap. 1 as Table 1-4 and is repeated here as Fig. 18-2.

If you are unsure about any of these concepts, review Chap. 1.

Binary-Coded Decimal Numbers

Binary-coded decimal numbers are just that: They are decimal numbers that happen to have each digit represented by its 4-bit binary equivalent. For example

$$0100_2 = 4_{10} \quad \text{and} \quad 0001_2 = 1_{10}$$

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

Fig. 18-2 Hexadecimal-binary-decimal conversion chart.

Therefore the BCD (Binary Coded Decimal) equivalent of the decimal number 41 is

0100 0001

Each nibble (group of 4 bits) stands for one decimal digit. The number as a whole is still a decimal number, however.

ASCII

ASCII code is different from decimal, binary, hexadecimal, and BCD in that it is not a number system but rather a way to represent various symbols with different patterns of 1s and 0s. Each pattern of 1s and 0s stands for a different letter of the alphabet (uppercase or lowercase), digit, punctuation mark, or other useful character.

We use number systems to count and to perform mathematical computations. We don't use ASCII for these purposes. We use ASCII code to represent characters used in normal written communication.

Do not try to memorize the ASCII code. Using charts when needed will suffice. If a large amount of data is necessary, we usually have some device, primarily the standard computer keyboard, to create these ASCII characters. A table (Table 1-6) showing the ASCII code appears in Chap. 1.

Microprocessors and Number Conversions

Microprocessors "think" in binary numbers. They use binary numbers for calculations and logical operations. Since binary numbers can be displayed as hexadecimal numbers with fewer digits, we often display binary numbers as their hexadecimal equivalents when people must enter or interpret those numbers.

The BCD numbers are used in certain situations to aid the people who must read them. For this reason some microprocessors have instructions which can convert answers resulting from binary mathematical operations to binary-coded decimal numbers. We will look at these operations later in this chapter.

Bit Positions

Sometimes students are confused when people talk about a certain "bit." There are two ways to describe a particular bit: by the binary power of 2 reflected in its position and by its location, from right to left. Look at Fig. 18-3.

You will see both methods used in the workplace and in other textbooks, so you should become comfortable with each.

18-2 ARITHMETIC INSTRUCTIONS

We will now review basic binary math and look at typical microprocessor instructions which perform mathematical

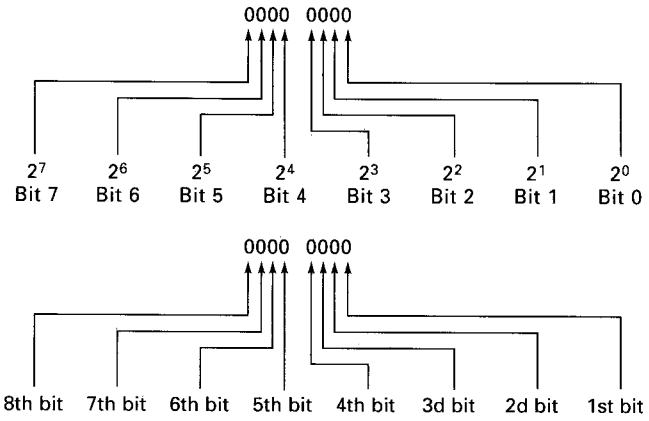


Fig. 18-3 Two methods for describing bit positions.

computations. Remember that we are now discussing techniques and instructions which are common to most microprocessors. We will study instructions specific to each microprocessor family in its appropriate section later in this chapter.

Addition

Each microprocessor family included in this text has at least one addition instruction. Most have more than one.

When adding binary numbers the microprocessor produces two types of information: (1) the sum of the two numbers (answer), (2) and information indicating whether there were carries in certain columns.

If you don't remember how to add binary numbers, you may want to review Chap. 6 now. There are really only five binary addition combinations to remember:

(1)	(2)	(3)	(4)	(5)
0	0	1	1	1
+0	+1	+0	+1	1
0	1	1	10	+1
				11

The first three combinations produce the same answer as they do in the decimal number base system. Combination #4 is simply saying that 1 + 1 = 2, except that the 2 is binary ($10_2 = 2_{10}$). You should say combination #4 to yourself as, "1 plus 1 equals 0, carry 1." Likewise, the fifth combination is saying that 1 + 1 + 1 = 3, except that the 3 is binary ($11_2 = 3_{10}$). You should express combination #5 as, "1 plus 1 plus 1 equals 1, carry 1." The last two combinations are the only new ones that you should memorize, since they are the only two that are different from our decimal number system.

To continue our review, let's see how to add several columns. It is common (and very practical) to show 8-bit binary numbers in two groups of four (as 2 nibbles). Refer to Fig. 18-4.

As you study Fig. 18-4, you will see that each of the

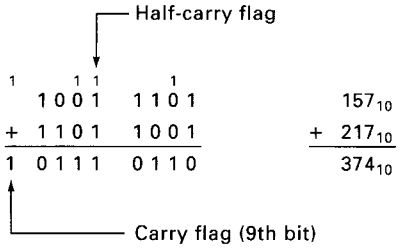


Fig. 18-4 Multi-column addition.

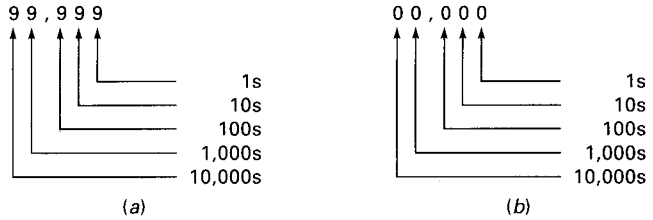


Fig. 18-5 (*a*) Automobile odometer. (*b*) Automobile odometer reset.

individual additions in each column is one of the five combinations we presented a moment ago.

Now let's continue using this example as we talk about two other closely related subtopics.

Carry Flag

The first flag we'll study is the carry flag. The *carry flag*, during addition, lets us know that the 8-bit sum is not the complete answer. If the carry flag is set (has a value of 1), it indicates that a 9th bit was produced.

Let's look again at Fig. 18-4. Notice the sum shown in the decimal version of the example. The decimal answer is 374. Now look at the binary version of the example. If you were to use only the right-most 8 bits (the 8 least significant bits), the sum would appear to be 118_{10} ($0111\ 0110_2 = 118_{10}$), which is not the correct sum. The 9th bit, which appears at the far left (the most significant bit), would not appear in an 8-bit accumulator. The 9th bit would exist in the carry flag (so to speak). The 1 in the carry flag would indicate a carry from column 8 to column 9. Again, we cannot see a 9th bit since the accumulator only holds 8 bits. (If you are using a 16-bit microprocessor, the function of the carry flag is the same as that described above except that it indicates the presence of a 17th bit, which will not fit into a 16-bit accumulator.

Substraction also affects the carry flag. We will discuss that a little later in this chapter.

Half-Carry Flag

Some (but not all) of our microprocessors have a half-carry flag. A *half-carry flag* indicates that a carry has occurred from the 4th-bit column to the 5th-bit column. The half-carry has been marked in Fig. 18-4.

Overflow Flag

The *overflow flag* alerts the programmer to a condition that is similar to, but not the same as, that to which the carry flag alerts the programmer. All our featured microprocessor families have an overflow flag except the 8080/8085. To understand what the overflow flag does, we need to take a closer look at 2's-complement arithmetic and signed binary numbers.

Each of our microprocessor families has one or more accumulators. All are 8-bit accumulators except the 8086/8088, which has a 16-bit accumulator. Let's focus our discussion on the 8-bit microprocessors.

If we do not expect to ever need negative numbers in a particular application, we can let the binary range of 0000 0000 to 1111 1111 represent decimal numbers 0 to 255. These are called *unsigned binary numbers*. However, if we need to represent negative numbers, we must use the 2's-complement form of the numbers we wish to make negative. When we allow both positive and negative numbers, we are using *signed binary numbers*.

We introduced 2's-complement numbers in Chap. 6. The concept was compared to that of the odometer on a car. Remember that the accumulator, like the odometer of a car, can contain only a certain number of digits. Most cars display 5 digits plus 10ths of a mile. If we disregard the 10ths digit, we have just 5 places. Of course, the highest number which can be represented is 99,999 miles. There aren't enough digits to show 100,000 miles. The 1 is lost, and only the 00,000 remains. The odometer has reset. Figure 18-5 illustrates this.

The accumulator of a microprocessor has this same limitation. If you continuously increment an 8-bit accumulator, you will eventually reach a maximum number beyond which the accumulator would have to have another digit. Figure 18-6 illustrates this. The accumulator, like the odometer, will reset to zero if it is incremented one more time.

When working with 2's-complement binary numbers, we assume that the accumulator can also be rolled backward, so to speak, to represent negative numbers. One less than zero is 11111111_2, which would be equal to -1_{10}. One less than that would be 11111110_2, which would be equal to -2_{10}. This process would continue as shown in Fig. 18-7.

As Fig. 18-7 illustrates, -128_{10} is as far as we can go

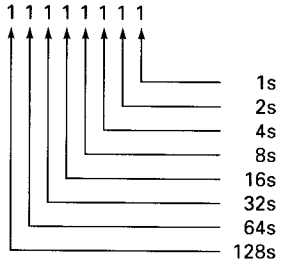


Fig. 18-6 Eight-bit accumulator.

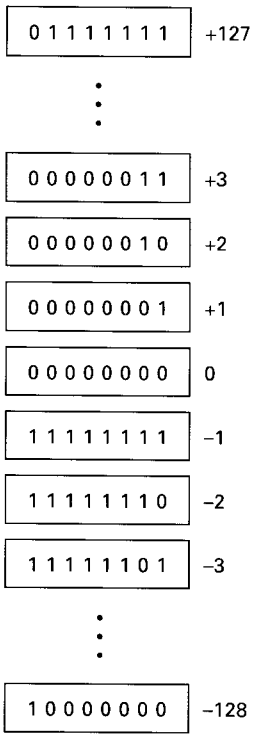


Fig. 18-7 Eight-bit 2's-complement range.

on the negative end. The reason for this is that one less than 10000000_2 is 01111111_2, which, if you look at the top of Fig. 18-7, is already being used as the equivalent of $+127_{10}$. When working with 8-bit 2's-complement numbers, we regard all numbers which have a 1 as the MSB (most significant bit) as negative. Numbers with a 0 in the MSB are positive. This means that the range for 8-bit 2's-complement binary numbers is $+127_{10}$ to -128_{10} inclusive.

Let's review a little. If we are using all 8 bits to represent numbers from 00_{10} to 255_{10}, we refer to these numbers as unsigned binary numbers. If we are using the MSB to signify whether a number is positive or negative, we have a range of -128_{10} to $+127_{10}$. These are called signed binary numbers.

There is a simple procedure by which you can determine how to form a negative binary (or hexadecimal) number. First, write the binary equivalent of the positive form of the number. For example

$$10_{10} = 0000\ 1010_2 = 0A_{16}$$

Now invert each bit of the binary number.

0000 1010 becomes 1111 0101

Then add 1.

$$\begin{array}{r} 1111\ 0101 \\ +\qquad\quad 1 \\ \hline 1111\ 0110 \end{array}$$

Therefore

$$-10_{10} = 1111\ 0110_2 = F6_{16}$$

Notice that the MSB of the binary number is 1, as we said it would be.

To determine what value a negative-signed binary number represents, reverse the above process. If you had the binary number

1111 0110 (the number created a moment ago)

invert each bit

0000 1001

and then add 1.

$$\begin{array}{r} {\scriptstyle 1}\ \ \ \ \ \ \ \ \\ 0000\ 1001 \\ +\qquad\quad 1 \\ \hline 0000\ 1010 \end{array}$$

Notice that we now have the binary number for 10_{10}. (A small 1 indicates a carry.) We have found that the binary number 1111 0110 is the signed binary number for -10_{10}.

The question now is how to interpret certain numbers. For example,

$$\begin{array}{r} 125_{10} \\ +\ 50_{10} \\ \hline 175_{10} \end{array} \qquad \begin{array}{r} 0111\ 1101_2 \\ +0011\ 0010_2 \\ \hline 1010\ 1111_2 \end{array} \qquad \begin{array}{r} 125_{10} \\ +\ 50_{10} \\ \hline -81_{10} \end{array}$$

We know that $125_{10} + 50_{10} = 175_{10}$. As you will notice in this example, however, the binary number for 175_{10} (which is $1010\ 1111_2$) is also the binary number for -81_{10}. So if we didn't know what two numbers this was the sum of, how would we know how to interpret this answer? If we simply found the binary number $1010\ 1111_2$ in a register, how would we know if it was meant to be $+175_{10}$ or -81_{10}? The answer is that we wouldn't. (The number -81_{10} is, of course, the wrong answer. We will deal with that part of the problem in just a bit.) We must know whether we are using unsigned binary numbers or signed binary numbers before we see the answer. It is simply a matter of agreement beforehand.

We have been preparing to explain the purpose of the overflow flag. We are now ready. The previous example, which produced a sum of $+175_{10}$ ($1010\ 1111_2$), would have set the overflow flag in an 8-bit microprocessor. The overflow flag tells the programmer that the last answer produced was outside the range of $+127_{10}$ to -128_{10} ($0111\ 1111_2$ to $1000\ 0000_2$ or $7F_{16}$ to 80_{16}). If the programmer understood this answer to represent an unsigned binary number, he or she would ignore the flag. If, however, this

was intended to be a signed binary number, the programmer would know that this answer, if taken as a signed binary number, is incorrect because it has exceeded the range for 8-bit signed binary numbers.

The range for unsigned 16-bit binary numbers is 0_{10} (0000 0000 0000 0000_2 or 0000_{16}) to $65{,}535_{10}$ (1111 1111 1111 1111_2 or $FFFF_{16}$). The range for signed 16-bit binary numbers is $+\ 32{,}767_{10}$ (0111 1111 1111 1111_2 or $7FFF_{16}$) to $-32{,}768_{10}$ (1000 0000 0000 0000_2 or 8000_{16}).

Addition-with-Carry

The previous section on addition discussed the carry flag. The carry flag signals the programmer that the result of an operation has exceeded 8 bits.

The carry flag has another use, though. The carry from the 8th bit to the 9th bit (which is what the carry represents) can be used during multiple-precision arithmetic. We use multiple-precision arithmetic when the accumulator cannot accept numbers large enough for the desired operation.

Multiple-Precision Binary Numbers

Until now we have assumed that any numbers we want to add would occupy only 1 byte of memory. This is called a *single-precision* number. One-byte unsigned numbers can range from 0 to 255. Two-byte unsigned numbers can range from 0 to 65,535. These are called *double-precision* binary numbers. Three-byte unsigned binary numbers can range from 0 to 16,777,215. These are called *triple-precision* numbers.

When we construct a double-precision number, we use the same techniques to determine its value as when we work with a single-precision number. Recall from earlier chapters that each binary position has a value and that each value is twice as large as the value to its right. If you have a calculator which will calculate powers of a number, it is quite easy to determine the value of a double-precision binary number. Refer to Fig. 18-8.

You see that the least significant bit (LSB) has a value of 2^0. This is equal to the number 1. (If you try this on a scientific calculator, it should give you that answer.) To determine the value of a double-precision number, add the value of each position which has a 1 in it. This will give you its decimal value. For example, to calculate the value of the binary number

0100 0001 0000 0010

you would enter

$$2^{14} + 2^8 + 2^1 = 16{,}642_{10}$$

into your calculator to get the above answer.

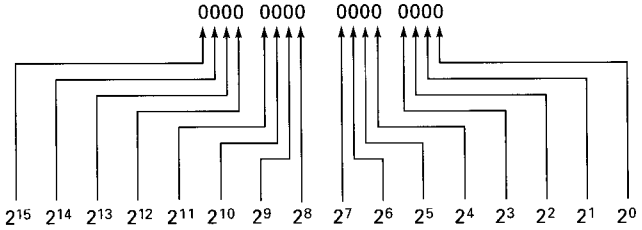


Fig. 18-8 Powers of 2 for a double-precision binary number.

Add-with-Carry

Let's step through a double-precision addition problem. Remember that we will be using the carry flag. Figure 18-9 shows an example.

The least significant bytes (LSBs) are on the right. They occupy the positions which have the least value. The most significant bytes are on the left. They occupy the positions which have the most value.

As you can see, several carries occur in this example. We are interested in the carry from the LSB to the MSB. That carry would actually be held in the carry flag of the microprocessor.

A typical microprocessor program to add these two binary numbers (using English phrases instead of microprocessor instructions) would appear as follows:

CLEAR CARRY FLAG
LOAD ACCUMULATOR WITH LSB OF ADDEND
ADD THE LSB OF THE AUGEND TO THE ACCUMULATOR
STORE THE LSB OF THE SUM IN MEMORY
LOAD THE ACCUMULATOR WITH THE MSB OF THE ADDEND
ADD-WITH-CARRY THE MSB OF THE AUGEND TO THE ACCUMULATOR
STORE THE MSB OF THE SUM IN MEMORY

Notice that we simply add the LSB of each number, but we *add-with-carry* the MSB of each number. When the microprocessor sees the add-with-carry instruction, it actually adds three numbers. It adds the addend (MSB), augend (MSB), and the carry flag. This brings the carry from the LSB into the MSB.

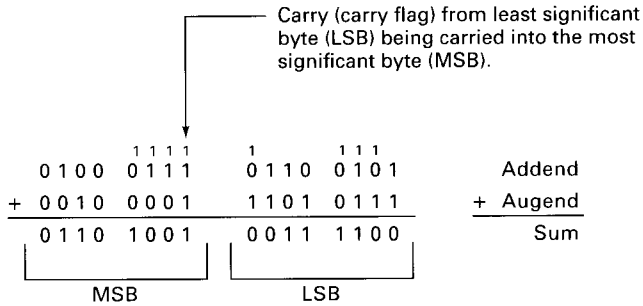


Fig. 18-9 Double-precision addition-with-carry.

Subtraction

Each of the microprocessor families included in this text has at least one subtraction instruction. Most have more than one.

When subtracting binary numbers, the microprocessor produces two types of information: (1) The difference between the two numbers (answer) and (2) whether there were borrows in certain columns.

If you don't remember how to subtract binary numbers, you may want to review Chap. 6 now. There are really only four binary combinations you need to remember:

(1)	(2)	(3)	(4)
0	1	1	$^{1}0$
−0	−0	−1	−1
0	1	0	1

The first three combinations produce the same answer as they do in the decimal-number base system. Combination #4 requires a borrow, which is shown by the small 1 set as a superscript. You cannot have 0 and subtract 1 from it. If you can borrow a 1 from the next-higher column, the subtraction becomes possible. If there is a higher column from which to borrow, this combination is really $2_{10} - 1_{10} = 1_{10}$. That is, 10_2 is created after the borrow occurs, and now the top number is larger than the bottom number. The carry flag is used if there is no higher column from which to borrow. You might say that it now becomes a "borrow" flag.

The last combination is the only new one that you will need to memorize since it is the only one that is different from our decimal number system.

The above discussion appeared in Chap. 6 and has been reviewed here for your convenience.

To continue our review, let's see how to subtract several columns. As in addition, it is common (and very practical) to show 8-bit binary numbers in two groups of four (as two nibbles). Refer to Fig. 18-10.

As you study Fig. 18-10, you will see that each individual subtraction in each column is one of the four combinations we presented a moment ago. When a borrow occurs, we have shown the borrowed 1 as a superscript 1 next to the 0 which needed it. The 1 that was borrowed from is crossed off, and its new value, 0, is shown above it.

Negative (Sign) Flag

The *negative flag*, sometimes called the *sign flag*, tells us whether the number in the accumulator is a positive or negative number. Since the most signficant bit of the accumulator is the sign bit (when using signed binary numbers), the negative flag simply reflects the status of that bit. If the most significant bit is 0, the negative flag is 0, and this is a positive number. If the most significant bit is 1, the negative flag will be 1, and this is a negative number.

While the negative flag always indicates the status of bit 7 of the accumulator, it is up to the programmer to determine whether the number is to be interpreted as a signed or unsigned binary number.

Figure 18-11 illustrates how the negative flag works.

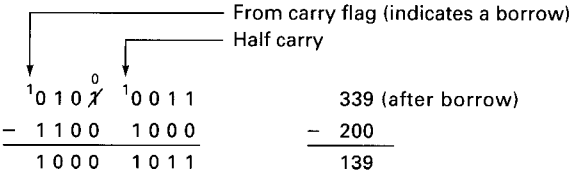


Fig. 18-10 Subtraction of binary numbers.

Zero Flag

The *zero flag* shows that the last operation produced a result of 0. This does not apply just to the accumulator but can apply to other registers as well. This is especially helpful when repeatedly decrementing (reducing by 1) an index register to determine the number of times a loop has executed. Knowing when a register has reached 0 is also useful when branching to other parts of a program and when determining whether or not to activate (call or enter) certain subroutines.

The one unusual feature of the zero flag is that it contains a 1 when the result is 0, and the flag is 0 when the result is anything other than 0. While this may appear confusing at first, it becomes second nature as you gain experience with microprocessors.

The idea here is that a 0 says that something is false or has not occurred. A 0 says, "No, this number was *not* the number zero."

A 1 says that something is true or has occurred. A 1 says, "Yes, this number *is* the number zero."

Subtraction-with-Carry (Borrow)

The same carry flag that informs us that an addition problem produced a sum which carried a 1 into the 9th bit also tells us something about subtraction problems. Now it tells us that to produce the answer (difference) the microprocessor had to borrow a 1 from a 9th bit. This occurs when the top number (minuend) is smaller than the bottom number (subtrahend).

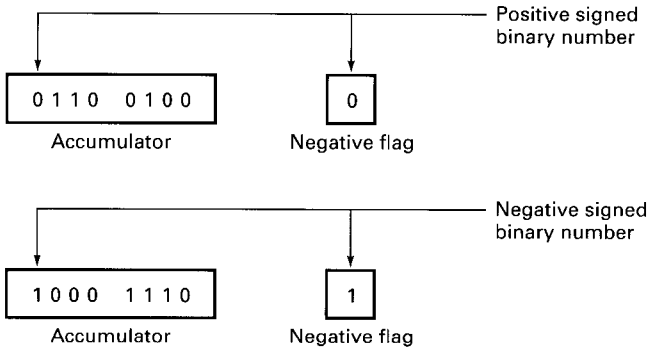


Fig. 18-11 The negative flag.

Refer again to Fig. 18-10. Notice that a borrow was required from a column that doesn't actually exist. There is no 9th column. The carry flag acts as that column. It tells us that a borrow from this "imaginary" column was necessary.

Most microprocessors set the carry flag (make it a 1) when a borrow is necessary (1 = true). The exception to this is the 6502 microprocessor. It clears the carry flag, as though the borrow actually came from the flag itself. In the 6502 you must set the carry flag before you start a subtraction problem so that, if a borrow is necessary, a 1 will be present.

Some microprocessors also monitor the 4th bit during subtraction. This is the half-carry flag which was mentioned earlier in this chapter.

Multiplication and Division

The 8-bit microprocessors featured in this text do not have multiplication or division instructions (the 6809, a relative of the 6800 and 6808, does have a **MUL**tiply instruction). However, the 16-bit 8086/8088 has both multiply and divide instructions, which will be discussed in the 8086/8088 section of this chapter.

There are several software algorithms for both multiplication and division which work well with the 8-bit microprocessors.

18-3 FLAG INSTRUCTIONS

Each of our microprocessors has instructions to alter the state of its flags. Which of their flags and how many of their flags can be directly altered vary.

The 8080/8085 has the fewest instructions for setting and clearing flags. The 6502, 8086/8088, and 6800/6808 all have the ability to set and clear many of their flags directly. The 6800/6808 has an instruction which makes it possible to move the status of all the flags into accumulator A and to copy the contents of accumulator A into the flag register. All our microprocessors except the 6800/6808 have the ability to push all the flags onto the stack and retrieve them from the stack. The 6800/6808 can accomplish the same task by transferring the flags to accumulator A and then to the stack in a two-step process.

We'll discuss the specific uses for each flag instruction in the Specific Microprocessor Families section of this chapter. The uses for flag instructions can be generalized, however. We use the flags primarily during arithmetic operations and for control of loops, branches, and subroutines.

Since we use the flags to give us information about the outcome of arithmetic operations, we often need to set or clear flags before these math operations so that we are certain of their exact condition before the operation begins.

We use flags to determine whether or not certain loops should be repeated, whether branches into other parts of the program should be taken, and whether certain subroutines should be called. Flags are used to make decisions about which microprocessor instructions should be executed next. This is the same as saying that the flags are used by the program to make decisions. For these reasons we may want to set or clear certain flags before or after certain instructions are executed.

Specific Microprocessor Families

Let's study the arithmetic and flag instructions for each of our microprocessor families. We'll be using short routines to study operations for which each microprocessor has specific instructions. We will not develop long routines to facilitate arithmetic operations which are not inherent to each microprocessor family. This will help you to become familiar with your microprocessor's basic arithmetic and flag instructions.

18-4 6502 FAMILY

The 6502 probably has the fewest different arithmetic instructions of any of our microprocessor families. However, by conscientiously setting and clearing the appropriate flags before arithmetic operations, this chip performs math operations adequately.

Arithmetic Instructions

The 6502 does not have normal *add* and *subtract* instructions. It has only *add-with-carry* and *subtract-with-carry*. Both of these instructions use the value in the accumulator as one of their operands with another value which can be an immediate value, or a value in memory, in addition to the value in the carry flag. The value in memory can be addressed any one of seven different ways. Let's see how to use these instructions.

Addition-with-Carry

Let's start with a very simply addition program. Figure 18-12 illustrates this type of program.

Notice first that we have used the CLC (**CL**ear **C**arry) instruction before we even loaded the accumulator with our first operand. This is necessary when using the 6502 microprocessor. If the carry flag is set from a previous operation, the ADC (**AD**d-with-**C**arry) instruction will add

```
    1 1
  0100  1001      49₁₆      73₁₀
+ 0001  1110    + 1E₁₆    + 30₁₀
  0110  0111      67₁₆      103₁₀
```

Addr	Obj	Assembler	Comment
0340	18	CLC	Prepare for addition problem
0341	A9	LDA #$49	Load accumulator with first number (49)
0342	49		
0343	69	ADC #$1E	Add 1E to the number in the accumulator and place the answer in the accumulator
0344	1E		
0345	00	BRK	Stop

Fig. 18-12 Simple 6502 addition problem.

the 1 in the carry flag to the answer and will cause the answer to be incorrect (it will be 1 greater than the correct result).

Pay particular attention to the accumulator and the processor status register. Notice their contents both before and after you run the program. (You may want to write down their values before and after so that you can study their behavior.) You will find that the accumulator will have the number 67_{16} in it (which is the correct answer) and that only the BRK (**BR**ea**K**) flag will be set.

Let's look at the processor status register a little more closely. Refer to Fig. 18-13 now.

Examining the flags from right to left, let's consider each and why it was or was not set during the last problem.

The carry flag would have been set if a carry from the 8th bit to the 9th bit (which doesn't exist, so it goes into the carry flag) had occurred, but none did.

The zero flag would have been set if the answer had been 0, but it wasn't.

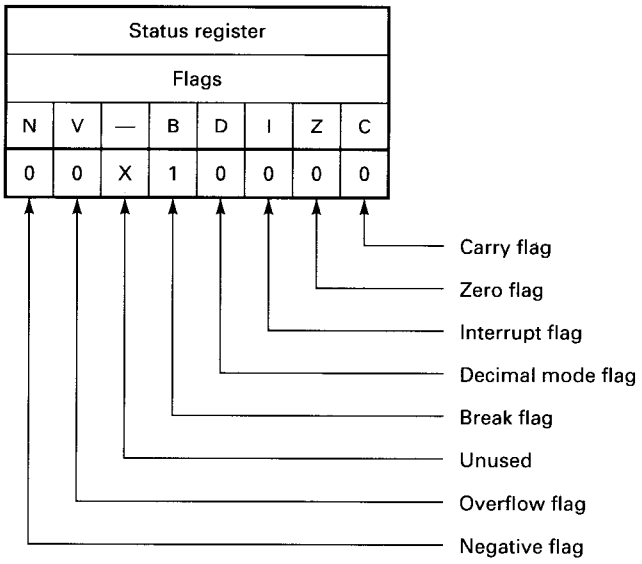


Fig. 18-13 6502 processor status register.

Don't worry about the interrupt flag since we haven't introduced this subject yet.

We dealt with the two operands as though they were hexadecimal numbers so we didn't set the decimal flag.

The break flag was set because we used the break instruction to stop the program.

The status of the unused flag doesn't matter.

We did not exceed the range of decimal +127 to −128 (hexadecimal 7F to 80); therefore the overflow flag was not set.

Finally, we did not have a 1 in the 8th bit of the accumulator so the answer could not have been negative; therefore the negative flag was not set.

The Negative Flag

Let's look at a problem which produces a negative answer. Refer to Fig. 18-14 now.

Notice that this is exactly the same problem that was used in Fig. 18-12 except that we have changed the first operand, which used to be 49_{16} into $C9_{16}$, which is the decimal number -55_{10}, if we consider these numbers to be signed binary numbers. We know that $-55_{10} + 30_{10} = -25_{10}$. Since this is a negative answer, we know that the negative flag should be set after the program is run.

Load the program and run it. Again write down the contents of the accumulator and processor status register before and after running the program so that you can compare them. After the program is run, the accumulator should contain the value $E7_{16}$. The processor status register should contain B0.

Let's examine the status register again. The binary value for $B0_{16}$ is $1011\ 0000_2$. If you put those bits into the appropriate positions in the status register as shown in Fig. 18-15, you will see that 3 bits or flags are set.

The break flag is again set because we used the break instruction to stop the program. We do not care about the status of the unused bit.

```
    1 1
  1 1 0 0   1 0 0 1      C9₁₆      -55₁₀
+ 0 0 0 1   1 1 1 0    + 1E₁₆    + 30₁₀
  1 1 1 0   0 1 1 1      E7₁₆      -25₁₀
```

Addr	Obj	Assembler	Comment
0340	18	CLC	Prepare for addition problem
0341	A9	LDA #$C9	Load accumulator with first number (C9)
0342	C9		
0343	69	ADC #$1E	Add 1E to the number in the accumulator and place the answer in the accumulator
0344	1E		
0345	00	BRK	Stop

Fig. 18-14 Simple 6502 addition problem with negative answer.

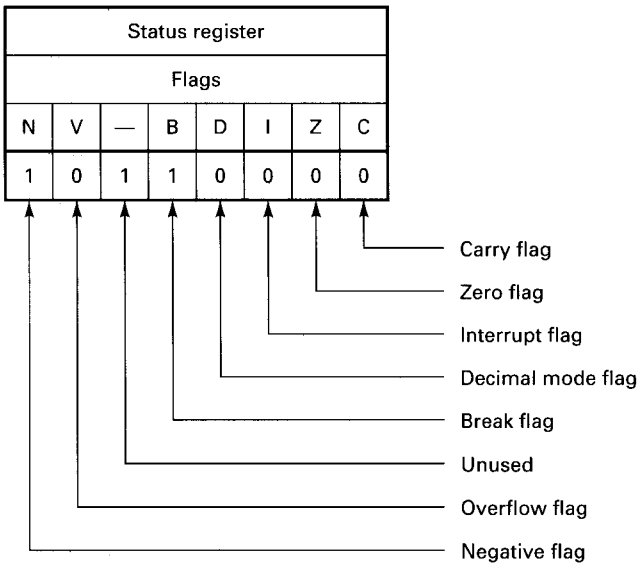


Fig. 18-15 6502 status register after an addition problem which produces a negative answer.

The negative flag is now set, however. This is what we expected to see. The sum of the addition problem was -25_{10} ($E7_{16}$). If we assume that our numbers are signed binary numbers, then any number that has a 1 in the 8th bit is negative. $E7_{16}$ has a 1 in the 8th bit. The negative flag simply reflects the state of the 8th bit.

The Zero Flag

Now let's change the program so that we get a sum of 0. Then we can see how the flags react to this situation.

Figure 18-16 shows the problem and the program to solve the problem.

We are again assuming that our numbers are signed binary numbers. The problem is $C9_{16} + 37_{16} = 00_{16}$, which is $-55_{10} + 55_{10} = 0_{10}$. You should go through the binary addition now before you run the program. Notice both the answer and any carries.

Write down the contents of the accumulator and the processor status register before and after running the program. You will notice that we are using the same program as in the last problem but have again changed one of the operands.

```
1  1 1 1 1   1 1 1 1
     1 1 0 0   1 0 0 1      C9₁₆      -55₁₀
+    0 0 1 1   0 1 1 1    + 37₁₆    + 55₁₀
1    0 0 0 0   0 0 0 0      00₁₆       0₁₀
```

Addr	Obj	Assembler	Comment
0340	18	CLC	Prepare for addition problem
0341	A9	LDA #$C9	Load accumulator with first number (C9)
0342	C9		
0343	69	ADC #$37	Add 37 to the number in the accumulator and place the answer in the accumulator
0344	37		
0345	00	BRK	Stop

Fig. 18-16 Simple 6502 addition problem which produces a sum of 0.

Now enter and run the program. The accumulator should contain 00_{16}, and the status register should contain 33. If you place the bits of the status register in their proper places as shown in Fig. 18-17, you will see how the flags have responded to this problem.

Notice that the break flag and unused flag have again been set as before. The value of the unused flag has no meaning, and the break flag simply shows that we used a break to stop the program.

The zero flag is set, as we supposed it would be. The carry flag is also set. Notice in the binary addition that a carry did indeed occur from the 8th to a nonexistent 9th bit (which the carry flag acts as).

Status register							
Flags							
N	V	—	B	D	I	Z	C
0	0	1	1	0	0	1	1

Fig. 18-17 6502 status register after an addition problem which produces a sum of 0.

The Overflow Flag

When the overflow flag is set, it tells us that if the numbers which were just added or subtracted are signed binary numbers, then the valid range for such numbers has been exceeded and the result is incorrect. The valid range for 8-bit microprocessors, which the 6502 is, is $+127$ to -128. Let's change our problem to create an overflow.

Figure 18-18 shows our problem and program. Notice in this problem that we are assuming that all values are to be interpreted as signed binary values.

The problem shown here is $123_{10} + 111_{10} =$ _____. First go through the binary addition and enter the program. Then write down the values in the accumulator and processor status register, run the program, and finally write down the ending values of the accumulator and status register.

Status register							
Flags							
N	V	—	B	D	I	Z	C
1	1	1	1	0	0	0	0

Fig. 18-19 6502 status register after an addition problem which creates an overflow.

Figure 18-19 shows what the value in the status register should be.

You should have a sum of EA_{16} in the accumulator and $F0_{16}$ in the status register. EA_{16} is the correct sum if you are using *unsigned* binary numbers! If you interpret EA_{16} as a signed binary number, it has a value of -22_{10}. This is *not* the correct answer. We have exceeded our valid range for signed binary numbers.

The status register has a value of F0. This means that in addition to the unused flag and the break flag, both the overflow and the negative flags have been set.

It makes sense for the negative flag to be set because the 8th bit of the accumulator is set. This indicates a negative number if the value is a signed binary number.

The overflow flag is set because we have exceeded our range of $7F_{16}$ (127_{10}) to 80_{16} (-128_{10}), giving an incorrect result.

The Decimal Flag

Because of differences in the way binary and decimal numbers round, and because numeric output to humans is usually decimal, it is sometimes better to actually do arithmetic calculations by using decimal numbers rather than binary numbers. Actually, true decimal numbers are not used. Rather, a mixture of binary and decimal, called binary-coded decimal, is used. (The method used to create BCD numbers is covered in Chap. 1 and they have been discussed subsequently. You should review that section of

```
  1 1 1 1   1 1 1
  0 1 1 1   1 0 1 1
+ 0 1 1 0   1 1 1 1
-----------------
  1 1 1 0   1 0 1 0
```

```
   7B16      12310
+  6F16    + 11110
-------    -------
   EA16      23410
```

Addr	Obj	Assembler	Comment
0340	18	CLC	Prepare for addition problem
0341	A9	LDA #$7B	Load accumulator with first number (7B)
0342	7B		
0343	69	ADC #$6F	Add 6F to the number in the accumulator and place the answer in the accumulator
0344	6F		
0345	00	BRK	Stop

Fig. 18-18 Simple 6502 addition problem which produces an overflow.

Chap. 1 now if you are unsure of what BCD numbers are or how they are formed.)

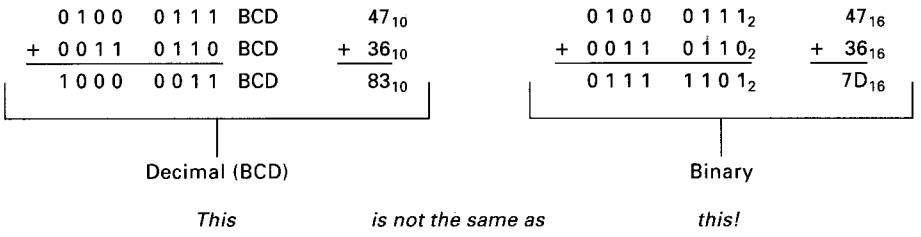
One of the problems encountered when using BCD numbers is that, as the binary nibbles are added, invalid results are sometimes obtained.

Most microprocessors have an instruction called *decimal adjust* (or something similar). This instruction changes the number in the accumulator to what it would be if the last two numbers operated on had been BCD numbers instead of binary numbers. The 6502 handles this a little differently. It requires that you set a flag designed just for this purpose and enter a "decimal mode," so to speak. When the decimal flag is set, all operands are assumed to be packed BCD numbers.

Let's look at an example. In Fig. 18-20 we have compared a decimal addition problem to the binary version of the same problem.

First notice the difference between BCD and binary addition. BCD addition is not the same as binary addition. BCD is *decimal* addition using four binary digits to represent each decimal digit.

The program shown in Fig. 18-20 will help you understand the difference between binary and BCD addition (and subtraction). This program does the addition problem twice,



Addr	Obj	Assembler	Comment
0340	D8	CLD	Prepare to do ***binary*** addition
0341	18	CLC	
0342	A9	LDA #$47	This is being interpreted as a binary number
0343	A7		
0344	69	ADC #$36	This also is being considered a binary number
0345	36		
0346	8D	STA $03A0	We'll store the ***binary*** answer in memory location 03A0
0347	A0		
0348	03		
0349	08	PHP	Put the flags on the stack
034A	68	PLA	Transfer flags from stack to accumulator
034B	8D	STA $03A1	We'll store the status of the flags from the ***binary*** addition in the memory location immediately following the ***binary*** sum, which is location 03A1
034C	A1		
034D	03		
034E	F8	SED	Prepare for ***decimal*** addition
034F	18	CLC	
0350	A9	LDA #$47	This number is being interpreted as a ***decimal*** number
0351	47		
0352	69	ADC #$36	This number likewise is being considered a ***decimal*** number
0353	36		
0354	8D	STA $03A2	We'll store the ***decimal*** answer in memory location 03A2
0355	A2		
0356	03		
0357	08	PHP	Put the flags on the stack
0358	68	PLA	Transfer the flags to the accumulator
0359	8D	STA $03A3	We'll store the status of the flags resulting from this ***decimal*** addition in the memory location immediately following the ***decimal*** sum, that is, location 03A3
035A	A3		
035B	03		
035C	00	BRK	Stop

Fig. 18-20 Binary vs. BCD addition.

once using binary numbers and once using BCD numbers. The result of the binary addition is stored in memory location $03A0_{16}$, and the resulting flags in location $03A1_{16}$. The result of the BCD addition is stored in location $03A2_{16}$, and the resulting flags in location $03A3_{16}$. Enter and run this program to see what results you get. (Don't be concerned about the reference to the stack in the program. We'll study the stack in a later chapter. For now just think of it as a temporary storage area.) When we ran the program we found the following:

location $03A0_{16}$ = binary sum = 7D
location $03A1_{16}$ = binary flags = 30
location $03A2_{16}$ = BCD sum = 83
location $03A3_{16}$ = BCD flags = F8

The status of the binary flags indicates only that the break instruction had been used to stop the program. No other flags were set.

The status of the flags after the BCD addition indicates that the decimal flag was set. (We set this flag to get into the "decimal mode.") The negative flag was set but has no valid meaning. It was simply following the state of the 8th bit of the accumulator. The overflow flag was set, but it also has no valid meaning in BCD arithmetic.

Subtraction-with-Carry

Subtraction-with-carry is the opposite of addition-with-carry. As in addition, there is no simple subtract instruction, only subtract-with-carry.

The 6502 handles borrows differently from the way most other microprocessors do. Most microprocessors set the carry flag if either a carry or a borrow occurs. The 6502 sets the flag if a carry occurs and clears the flag if a borrow occurs. It is important to remember that *the carry flag must be set before a subtraction problem (or the first section of a multiple-precision subtraction problem) so that if a borrow is needed, it can clear the carry flag,* which then indicates that the borrow has occurred. *If the carry flag is not set before starting the subtraction, the answer will be incorrect. (It will be 1 less than the correct result.)*

Figure 18-21 illustrates the correct way to write a program to do single-precision subtraction.

You should assemble and run this program. When we did, we found that the result in the accumulator was FF. We also found that the overflow and negative flags had been set. The negative flag was set because the 8th bit of the answer is a 1, which indicates a negative-signed binary number. The overflow flag was set because $7F_{16} = 127_{10}$, and $80_{16} = -128_{10}$; therefore

$$\begin{array}{r} 127 \\ --128 \\ \hline 255 \end{array}$$

and 255_{10} is outside the valid range for 8-bit signed binary numbers. (The valid range is $+127_{10}$ to -128_{10}.)

18-5 6800/6808 FAMILY

The 6800/6808 has a variety of *add* and *subtract* instructions which can use either of its two accumulators and can address memory locations in several ways. The 6800/6808 can also add and subtract binary-coded decimal (BCD) numbers.

Arithmetic Instructions

The 6800/6808 has *add, subtract, add-with-carry, subtract-with-carry, add accumulator A to accumulator B, subtract accumulator B from accumulator A,* and *decimal adjust accumulator A* instructions. These instructions use the value in one of the accumulators as one of their operands and another value which can be an immediate value or a value in memory. Let's see how to use these instructions.

Addition

Let's start with a very simple addition program. Figure 18-22 illustrates this type of program.

Pay particular attention to the accumulator and the condition code register (status register). Notice their contents both before and after you run the program. (You may want to write down their values before and after so you can study

Addr	Obj	Assembler	Comment
0340	38	SEC	*Remember this step!*
0341	A9	LDA #$7F	
0342	7F		
0343	E9	SBC #$80	
0344	80		
0345	00	BRK	

Fig. 18-21 Subtraction-with-carry.

$$\begin{array}{r} {\scriptstyle 1\ 1} \\ 0\,1\,0\,0 \quad 1\,0\,0\,1 \\ +\ 0\,0\,0\,1 \quad 1\,1\,1\,0 \\ \hline 0\,1\,1\,0 \quad 0\,1\,1\,1 \end{array} \qquad \begin{array}{r} 49_{16} \\ +\ 1E_{16} \\ \hline 67_{16} \end{array} \qquad \begin{array}{r} 73_{10} \\ +\ 30_{10} \\ \hline 103_{10} \end{array}$$

Addr	Obj	Assembler	Comment
0000	86	LDAA #$49	Load accumulator with first number (49)
0001	49		
0002	8B	ADDA #$1E	Add 1E to the number in the accumulator and place the answer in the accumulator
0003	1E		
0004	3E	WAI	Stop

Fig. 18-22 Simple 6800/6808 addition problem.

their behavior.) You will find that the accumulator will have the number 67_{16} in it (which is the correct answer) and that only the half-carry flag will be set.

Let's look at the status register a little more closely. Refer to Fig. 18-23 now.

Examining the flags from right to left, let's consider each and why it was or was not set.

The carry flag would have been set if a carry from the 8th bit to the 9th bit (which doesn't exist, so it goes into the carry flag) had occurred, but none did.

We did not exceed the range of $+127_{10}$ to -128_{10} (hexadecimal 7F to 80); therefore the overflow flag was not set.

The zero flag would have been set if the answer had been zero, but it wasn't.

We did not have a 1 in the 8th bit of the accumulator so the answer could not have been negative; therefore the negative flag was not set.

Don't worry about the interrupt flag since we haven't introduced this subject yet.

The half-carry flag was set because we had a carry from the 4th bit to the 5th bit. (Information about the half-carry is useful when dealing with BCD numbers.)

The status of the unused flags doesn't matter.

The Negative Flag

Now let's look at a problem that produces a negative answer. See Fig. 18-24.

Notice that this is exactly the same problem as the last one except that we have changed the first operand, which was 49_{16}, into $C9_{16}$, which is the number -55_{10} if we consider these numbers to be signed binary numbers. We know that $-55_{10} + 30_{10} = -25_{10}$. Since this is a negative answer, we know that the negative flag should be set after the program is run.

Write down the contents of the accumulator and processor status register before running the program so that you know what the initial conditions are. Now load the program and run it. After you run the program, the accumulator should

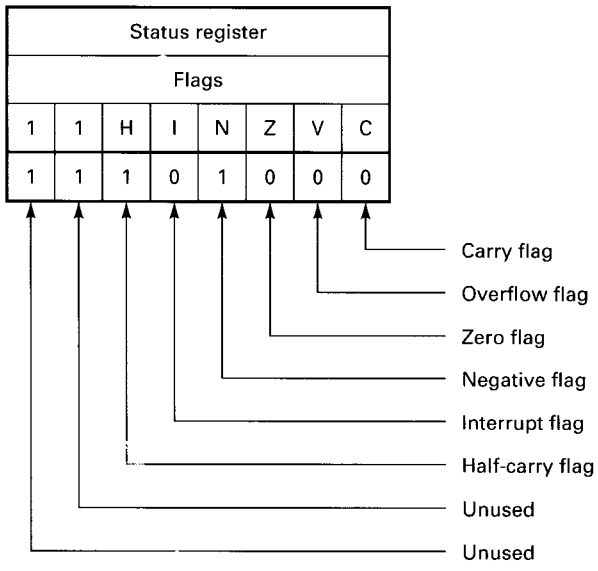


Fig. 18-23 6800/6808 status register.

```
    1 1
  1 1 0 0   1 0 0 1        C9₁₆      -55₁₀
+ 0 0 0 1   1 1 1 0      + 1E₁₆    + 30₁₀
  1 1 1 0   0 1 1 1        E7₁₆      -25₁₀
```

Addr	Obj	Assembler	Comment
0000	86	LDAA #$C9	Load accumulator with first number (C9)
0001	C9		
0002	8B	ADDA #$1E	Add 1E to the number in the accumulator and place the answer in the accumulator
0003	1E		
0004	3E	WAI	Stop

Fig. 18-24 Simple 6800/6808 addition problem with negative answer.

contain the value $E7_{16}$. The status register should contain XX101000.

Let's examine the status register again. If you put the bits into their appropriate positions in the status register as shown in Fig. 18-25, you will see that 2 bits or flags are set.

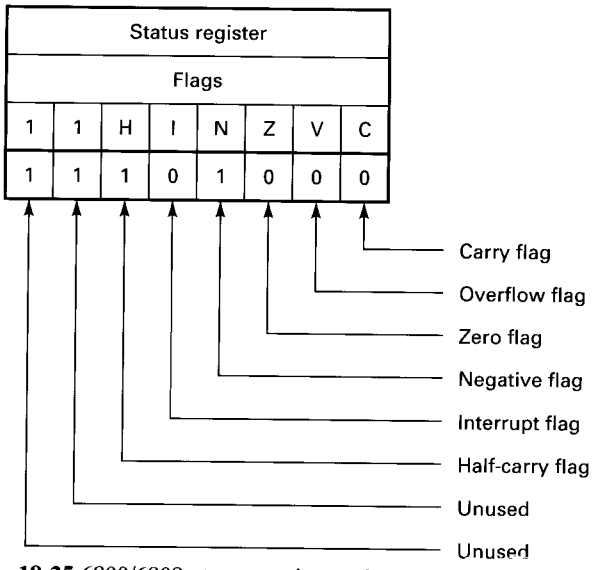


Fig. 18-25 6800/6808 status register after an addition problem which produces a negative answer.

The half-carry flag is again set because we had a carry from the 4th to the 5th bit of the result. The difference is that the negative flag is now set. This is what we expected to see. The sum of the addition problem was -25_{10} ($E7_{16}$). If we assume that our numbers are signed binary numbers, then any number that has a 1 in the 8th bit is negative. $E7_{16}$ has a 1 in the 8th bit.

The Zero Flag

Now let's change the program slightly so that we get a sum of 0. Then we can see how the flags react to this situation.

Figure 18-26 shows the problem and the program to solve the problem.

We are again assuming that our numbers are signed binary numbers. The problem is $C9_{16} + 37_{16} = 00_{16}$ ($-55_{10} + 55_{10} = 0_{10}$). You should go through the binary addition of these two numbers now before you run the program. Notice both the answer and the carries.

Again write down the contents of the accumulator and the status register before and after running the program. You will notice that we are using the same program as the ast example but have changed one of the operands.

Now enter and run the program. The accumulator should contain 00_{16}, and the status register should contain $XX100101_2$. If you place the bits of the status register value in their

```
  1 1 1 1 1   1 1 1
    1 1 0 0   1 0 0 1        C9₁₆      -55₁₀
+   0 0 1 1   0 1 1 1      + 37₁₆    + 55₁₀
  1 0 0 0 0   0 0 0 0        00₁₆        0₁₀
```

Addr	Obj	Assembler	Comment
0000	86	LDAA #$C9	Load accumulator with first number (C9)
0001	C9		
0002	8B	ADDA #$37	Add 37 to the number in the accumulator and place the answer in the accumulator
0003	37		
0004	3E	WAI	Stop

Fig. 18-26 Simple 6800/6808 addition problem which produces a sum of 0.

Status register							
Flags							
1	1	H	I	N	Z	V	C
1	1	1	0	0	1	0	1

Fig. 18-27 6800/6808 status register after an addition problem which produces a sum of 0.

proper places as shown in Fig. 18-27, you will see how the flags have responded to this problem.

Notice that the half-carry flag has again been set. The zero flag is set, as we supposed it would be. The carry flag is also set. Notice in the binary addition that a carry did indeed occur from the 8th to a nonexistent 9th bit (which the carry flag acts as).

The Overflow Flag

When the overflow flag is set, it tells us that if the numbers which the microprocessor just added or subtracted are signed binary numbers, the valid range for such numbers has been exceeded and the result is incorrect. The valid range for 8-bit microprocessors is $+127$ to -128. Let's change our problem to create an overflow.

Figure 18-28 shows our problem and program. Note that in this problem we are assuming that all values are to be interpreted as *signed binary* values.

This problem is $123_{10} + 111_{10} =$ _____. First go through the binary addition and enter the program. Then write down the values of the accumulator and status register, run the program, and finally write down the final values of the accumulator and status register.

Figure 18-29 shows what the value in the status register should be.

You should have a sum of EA_{16} in the accumulator and $XX101010_2$ in the status register. EA_{16} is the correct sum if you are using *unsigned* binary numbers! If you interpret EA_{16} as a signed binary number, it has a value of -22_{10}.

Status register							
Flags							
1	1	H	I	N	Z	V	C
1	1	1	0	1	0	1	0

Fig. 18-29 6800/6808 status register after an addition problem which creates an overflow.

This is *not* the correct answer. We have exceeded our valid range for signed binary numbers.

The status register has a value of XX101010. This means that in addition to the half-carry flag, both the overflow and the negative flags have been set.

It makes sense for the negative flag to be set because the 8th bit of the accumulator is set. This indicates a negative number if the value is a signed binary number.

The overflow flag is set because we have exceeded our range of $7F_{16}$ (127_{10}) to 80_{16} (-128_{10}), and the result is incorrect.

Decimal Addition

Because of differences in the way binary and decimal numbers round, and because numeric output to humans is usually decimal, it is sometimes helpful to actually do arithmetic calculations by using decimal numbers rather than binary numbers. Actually, true decimal numbers are not used. Rather a mixture of binary and decimal, called binary-coded decimal, is used. (The method used to create BCD numbers is covered in Chap. 1, and they have been discussed subsequently. You should review that section of Chap. 1 now if you are at all unsure of what BCD numbers are or how they are formed.)

One of the problems encountered in using BCD numbers is that, as the binary nibbles are added, invalid results are sometimes obtained.

Most microprocessors have an instruction called *decimal adjust* (or something similar). This instruction changes the

```
  1 1 1 1   1 1 1
  0 1 1 1   1 0 1 1        7B16      12310
+ 0 1 1 0   1 1 1 1      + 6F16    + 11110
-----------------------   ------    -------
  1 1 1 0   1 0 1 0        EA16      23410
```

Addr	Obj	Assembler	Comment
0000	86	LDAA #$7B	Load accumulator with first number (7B)
0001	7B		
0002	8B	ADDA #$6F	Add 6F to the number in the accumulator and place the answer in the accumulator
0003	6F		
0004	3E	WAI	Stop

Fig. 18-28 Simple 6800/6808 addition problem which produces an overflow.

number in the accumulator to what it would be if the last two numbers operated on were packed BCD (binary-coded decimal) numbers instead of binary numbers.

Let's look at an example. Figure 18-30 compares a decimal addition problem to the binary version of the same problem.

Notice first the difference between BCD and binary addition. BCD addition is not at all the same as binary addition. BCD is *decimal* addition using four binary digits to represent each decimal digit.

The program shown in Fig. 18-30 will help you understand the difference between binary and BCD addition (and subtraction). This program does the addition problem twice, once using binary numbers and once using BCD numbers. The result of the binary addition is stored in memory location $A0_{16}$, and the resulting flags in location $A1_{16}$. The result of the BCD addition is stored in location $A2_{16}$, and the resulting flags in location $A3_{16}$. Enter and run this program to see what results you obtain. When we ran the program, we found the following:

location $A0_{16}$ = binary sum = 7D
location $A1_{16}$ = binary flags = 000000

location $A2_{16}$ = BCD sum = 83
location $A3_{16}$ = BCD flags = 001000

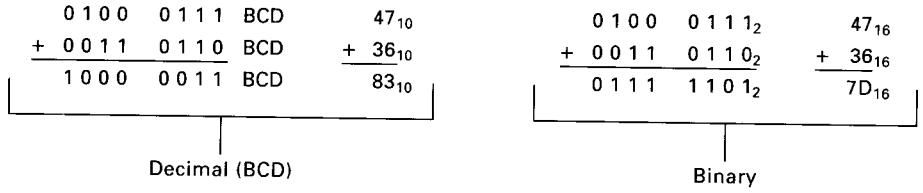
The status of the binary flags indicates that no flags were set as a result of the binary addition. After the BCD addition, the negative flag was set but has no valid meaning. It is simply following the state of the 8th bit of the accumulator.

Subtraction

Subtraction is the opposite of addition. All the flags operate the same except the carry flag. After subtraction, the carry flag indicates whether or not a borrow has occurred. You can think of it as a "borrow" flag. A 1 in the carry flag position indicates that a borrow from the nonexistent 9th bit was required to do the subtraction. A 0 indicates that no borrow from the 9th bit was required.

Figure 18-31 illustrates how to write a program to do single-precision subtraction.

You should assemble and run this program. When we did, we found that the result in the accumulator was FF.



Addr	Obj	Assembler	Comment
0000	86	LDAA #$47	This is being interpreted as a binary number
0001	47		
0002	8B	ADDA #$36	This also is being considered a binary number
0003	36		
0004	97	STAA #A0	We'll store the ***binary*** answer in memory location 03A0
0005	A0		
0006	07	TPA	Transfer flags to accumulator
0007	97	STAA #A1	We'll store the status of the flags from the ***binary*** addition in memory location A1
0008	A1		
0009	86	LDAA #$47	This number is being interpreted as a ***decimal*** number
000A	47		
000B	8B	ADDA #$36	This number likewise is being considered a ***decimal*** number
000C	36		
000D	19	DAA	Make the answer decimal
000E	97	STAA $A2	We'll store the ***decimal*** answer in memory location 03A2
000F	A2		
0010	07	TPA	Transfer the flags to the accumulator
0011	97	STAA $A3	We'll store the status of the flags resulting from this ***decimal*** addition in memory location A3
0012	A3		
0013	3E	WAI	Stop

Fig. 18-30 Binary vs. BCD addition.

Addr	Obj	Assembler	Comment
0000	86	LDAA #$7F	
0001	7F		
0002	80	SUBA #$80	
0003	80		
0004	3E	WAI	

Fig. 18-31 Subtraction.

We also found that the overflow, negative, and carry flags had been set. The negative flag was set because the 8th bit of the answer is a 1, which indicates a negative-signed binary number. The overflow flag was set because $7F_{16} = 127_{10}$, and $80_{16} = -128_{10}$; therefore

$$\begin{array}{r} 127 \\ -(-128) \\ \hline 255 \end{array}$$

and 255_{10} is outside the valid range for 8-bit signed binary numbers (the valid range is $+127_{10}$ to -128_{10}). The carry flag was set because a borrow from a 9th bit was needed to complete the subtraction.

18-6 8080/8085/Z80 FAMILY

The 8080/8085/Z80 family has a variety of *add* and *subtract* instructions. The 8080/8085/Z80 can also work with binary-coded decimal (BCD) numbers.

Arithmetic Instructions

The 8080/8085/Z80 family has *add, subtract, add-with-carry, subtract-with-borrow, immediate mode* and *decimal adjust accumulator A* instructions. These instructions use the value in the accumulator as one of their operands and another value in one of the other registers as the other operand. Let's see how to use these instructions.

Addition

Let's start with a very simple addition program. Figure 18-32 illustrates this type of program.

Pay particular attention to the accumulator and the status register. Notice their contents both before and after you run the program. (You may want to write down their values before and after so that you can study their behavior.) You will find that the accumulator will have the number 67_{16} in it (which is the correct answer) and that only the half-carry flag will be set.

Let's look at the status register a little more closely. Refer to Fig. 18-33.

$$\begin{array}{r} {\scriptstyle 1\ 1} \\ 0100\ 1001 \\ +\ 0001\ 1110 \\ \hline 0110\ 0111 \end{array} \qquad \begin{array}{r} 49_{16} \\ +\ 1E_{16} \\ \hline 67_{16} \end{array} \qquad \begin{array}{r} 73_{10} \\ +\ 30_{10} \\ \hline 103_{10} \end{array}$$

Addr	Obj	Assembler	Comment
1800	3E	MVI A,49	Load accumulator with first number (49)
1801	49		
1802	C6	ADI 1E	Add 1E to the number in the accumulator and place the answer in the accumulator
1803	1E		
1804	76	HALT	Stop

(8080/8085 mnemonics)

Addr	Obj	Assembler	Comment
1800	3E	LD A,49	Load accumulator with first number (49)
1801	49		
1802	C6	ADD A,1E	Add 1E to the number in the accumulator and place the answer in the accumulator
1803	1E		
1804	76	HALT	Stop

(Z80 mnemonics)

Fig. 18-32 Simple 8080/8085/Z80 addition problem.

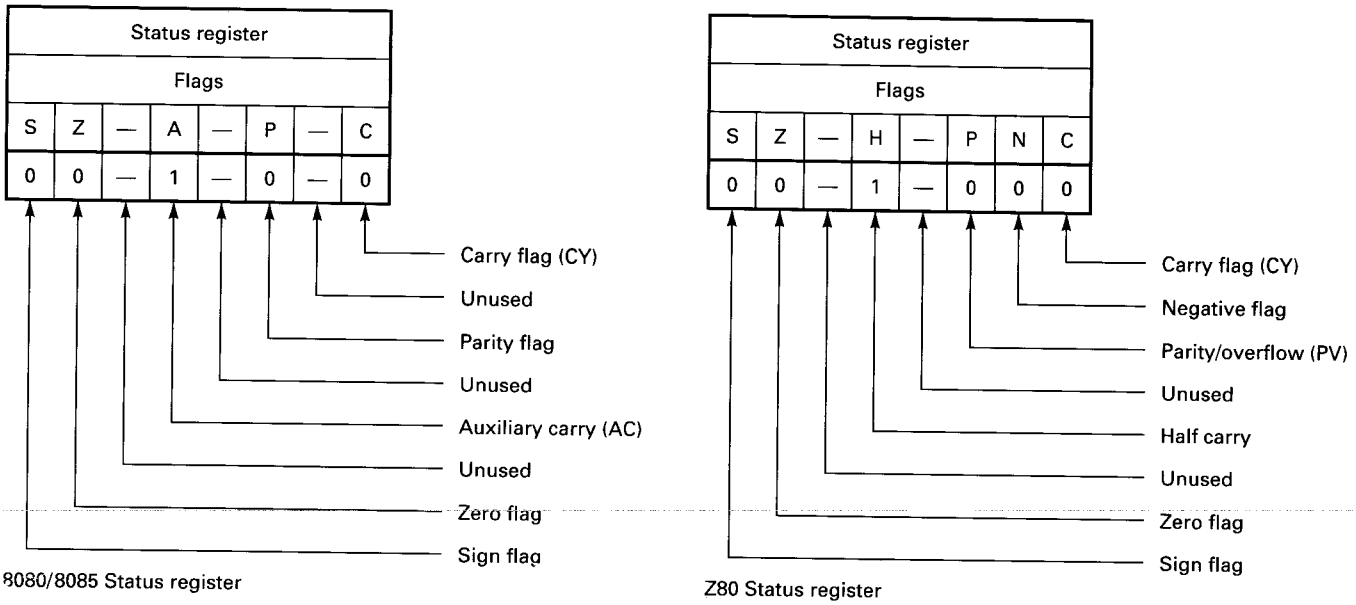


Fig. 18-33 8080/8085/Z80 status registers after addition problem.

Examining the flags from right to left, let's consider each and why it was or was not set.

The carry flag would have been set if a carry from the 8th bit to the 9th bit (which doesn't exist, so it goes into the carry flag) had occurred, but none did.

(Note to Z80 users: Ignore the negative flag.)

The parity flag was not set because the answer 0110 0111_2 has an odd number of 1s. That is to say it has odd parity, which is indicated by a 0. *(Note to Z80 users: We did not exceed the range of decimal +127 to −128—hexadecimal 7F to 80; therefore the parity/overflow flag was not set.)*

The microprocessor set the auxiliary carry (half-carry) flag because we had a carry from the 4th bit to the 5th bit. (Information about the half-carry is useful when dealing with BCD numbers.)

The zero flag would have been set if the answer had been zero, but it wasn't.

We did not have a 1 in the 8th bit of the accumulator so the answer could not have been negative; therefore the sign flag was not set.

The status of the unused flags doesn't matter.

The Sign Flag

Let's look at a problem that produces a negative answer. See Fig. 18-34.

Notice that this is the same problem as the last one except that we have changed the first operand. It used to be 49_{16}, but it is now $C9_{16}$, which is the decimal number -55_{10} if we consider these numbers to be signed binary numbers. We know that $-55_{10} + 30_{10} = -25_{10}$. Since this is a negative answer, we know that the sign flag should be set after the program is run.

Write down the contents of the accumulator and status register before running the program so that you know the initial conditions. Load the program and run it. After the program is run, the accumulator should contain the value $E7_{16}$. The status register should contain 10-1-1-0 *[Z80 = 10-1-000]*.

Let's examine the status register. If you put the status register bits into the appropriate positions in the status register as shown in Fig. 18-35, you will see what the bits indicate.

The auxiliary-carry *[half-carry]* is set again because we had a carry from the 4th to the 5th bit of the result.

The difference this time is that the sign flag is now set. This is what we expected to see. The sum of the addition problem was -25_{10} ($E7_{16}$). If we assume our numbers are signed binary numbers, then any number that has a 1 in the 8th bit is negative. $E7_{16}$ has a 1 in the 8th bit. The sign flag simply reflects the state of the 8th bit.

[*Note to 8085 users:* Your parity flag is 1 this time because the answer ($E7_{16}$) has an even number of 1s in it and even parity is indicated by a 1 in the parity flag. *Note to Z80 users: Your parity/overflow flag is 0 just like last time because the answer did not exceed the range from $+127_{10}$ to -128_{10}.*]

The Zero Flag

Now let's change the program slightly so that we get a sum of 0. That way we can see how the flags react to this situation.

Figure 18-36 shows the problem and the program to solve the problem.

We are again assuming that our numbers are signed binary numbers. The problem is $C9_{16} + 37_{16} = 00_{16}$

$$\begin{array}{rrrr} & {\scriptstyle 1\ 1} & & \\ 1\,1\,0\,0 & 1\,0\,0\,1 & C9_{16} & -55_{10} \\ +\ 0\,0\,0\,1 & 1\,1\,1\,0 & +\ 1E_{16} & +\ 30_{10} \\ \hline 1\,1\,1\,0 & 0\,1\,1\,1 & E7_{16} & -25_{10} \end{array}$$

Addr	Obj	Assembler	Comment
1800	3E	MVI A,C9	Load accumulator with first number (C9)
1801	C9		
1802	C6	ADI 1E	Add 1E to the number in the accumulator and place the answer in the accumulator
1803	1E		
1804	76	HALT	Stop

(8080/8085 mnemonics)

Addr	Obj	Assembler	Comment
1800	3E	LD A,C9	Load accumulator with first number (C9)
1801	C9		
1802	C6	ADD A,1E	Add 1E to the number in the accumulator and place the answer in the accumulator
1803	1E		
1804	76	HALT	Stop

(Z80 mnemonics)

Fig. 18-34 Simple 8080/8085/Z80 addition problem with negative answer.

($-55_{10} + 55_{10} = 0_{10}$). You should go through the binary addition of these two numbers now before you run the program. Notice both the answer and the carries.

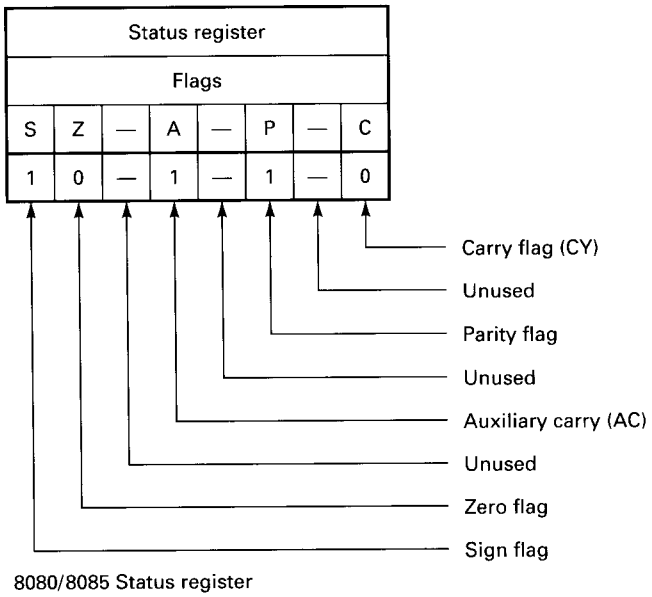
Again write down the contents of the accumulator and the status register before and after running the program. You will notice that we are using the same program but have changed one of the operands.

Now enter and run the program. The accumulator should contain 00_{16} and the status register should contain 01-1-1-1 *[Z80 = 01-1-001]*. If you place the bits of the status register value in their proper places as shown in Fig. 18-37, you will see how the flags have responded to this problem.

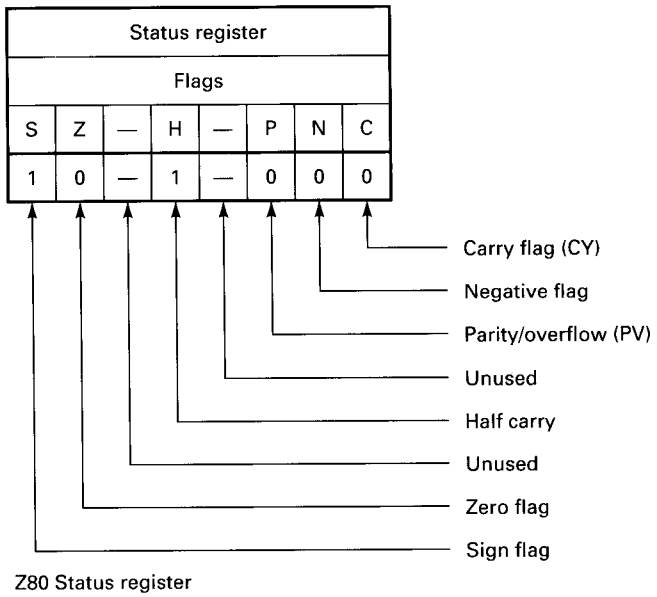
Notice that the half-carry flag has again been set.

The zero flag is set, as we supposed it would be.

The carry flag is also set. Notice in the binary addition



8080/8085 Status register



Z80 Status register

Fig. 18-35 8080/8085/Z80 status registers after an addition problem which produces a negative answer.

```
  1 1111  111
   1100  1001        C9₁₆      −55₁₀
+  0011  0111      + 37₁₆    + 55₁₀
-------------     --------  --------
 1 0000  0000        00₁₆       0₁₀
```

Addr	Obj	Assembler	Comment
1800	3E	MVI A,C9	Load accumulator with first number (C9)
1801	C9		
1802	C6	ADI 37	Add 37 to the number in the accumulator and place the answer in the accumulator
1803	37		
1804	76	HALT	Stop

(8080/8085 mnemonics)

Addr	Obj	Assembler	Comment
1800	3E	LD A,C9	Load accumulator with first number (C9)
1801	C9		
1802	C6	ADD A,37	Add 37 to the number in the accumulator and place the answer in the accumulator
1803	37		
1804	76	HALT	Stop

(Z80 mnemonics)

Fig. 18-36 Simple 8080/8085/Z80 addition problem which produces a sum of 0.

that a carry did indeed occur from the 8th bit to a nonexistent 9th bit (which the carry flag acts as).

The *8085 parity flag* is set indicating an even number of 1s. (*Note to Z80 users: Your parity/overflow flag is cleared indicating you have not exceeded the range for 8-bit signed binary numbers, from* $+127_{10}$ *to* -128_{10}.)

The Parity Flag *[Z80: Parity/Overflow Flag]*

The 8080/8085 and Z80 microprocessors differ slightly in the function of this flag. Let's look at the 8080/8085 first.

Status register							
Flags							
S	Z	—	A	—	P	—	C
0	1	—	1	—	1	—	1

8080/8085 Status register

Status register							
Flags							
S	Z	—	H	—	P	N	C
0	1	—	1	—	0	0	1

Z80 Status register

Fig. 18-37 8080/8085/Z80 status registers after an addition problem which produces a sum of 0.

The 8080/8085 microprocessors have a parity flag which simply tells us how many 1s are in the accumulator after an arithmetic or a logic operation. Even parity exists when an even number of 1s are in the accumulator. Odd parity exists when an odd number of 1s exist in the accumulator. Even parity is shown by a 1 in the parity flag, and odd parity by a zero in the parity flag.

The Z80 has a combination parity/overflow flag. During logic operations it indicates parity as just described for the 8080/8085. During arithmetic operations, however, it acts as an overflow flag.

When an overflow flag is set, it tells us that if the numbers which were just added or subtracted are signed binary numbers, then the valid range for such numbers has been exceeded and the result is incorrect. The valid range for 8-bit microprocessors is $+127$ to -128. Let's change our problem to create an overflow.

Figure 18-38 shows our problem and program. In this problem it is important to note that we are assuming that all values are to be interpreted as *signed binary* values.

This problem is $+123_{10} + 111_{10} =$ _____. First go through the binary addition and enter the program. Then write down the values in the accumulator and status register, run the program, and finally write down the values of the accumulator and status register after the program has run.

Figure 18-39 shows what the value in the status register should be.

You should have a sum of EA_{16} in the accumulator and 10-1-0-0 *[Z80: 10-1-100]* in the status register. EA_{16} is the correct sum if you are using *unsigned* binary numbers! If

```
  1 1 1 1   1 1 1
  0 1 1 1   1 0 1 1        7B16        12310
+ 0 1 1 0   1 1 1 1      + 6F16      + 11110
  1 1 1 0   1 0 1 0        EA16        23410
```

Addr	Obj	Assembler	Comment
1800	3E	MVI A,7B	Load accumulator with first number (7B)
1801	7B		
1802	C6	ADI 6F	Add 6F to the number in the accumulator and place the answer in the accumulator
1803	6F		
1804	76	HALT	Stop

(8080/8085 mnemonics)

Addr	Obj	Assembler	Comment
1800	3E	LD A,7B	Load accumulator with first number (7B)
1801	7B		
1802	C6	ADD A,6F	Add 6F to the number in the accumulator and place the answer in the accumulator
1803	6F		
1804	76	HALT	Stop

(Z80 mnemonics)

Fig. 18-38 Simple 8080/8085/Z80 addition problem which produces an overflow.

you interpret EA_{16} as a signed binary number, it has a value of -22_{10}. This is *not* the correct answer. We have exceeded our valid range for signed binary numbers.

The status registers of both the 8080/8085 and the Z80 microprocessors have a 1 in the half-carry flag as before. Now however, both also have a sign flag that is set. It makes sense for the sign flag to be set because the 8th bit of the accumulator is set. This indicates a negative number if the value is a signed binary number.

Status register							
Flags							
S	Z	—	A	—	P	—	C
1	0	—	1	—	0	—	0

8080/8085 Status register

Status register							
Flags							
S	Z	—	H	—	P	N	C
1	0	—	1	—	1	0	0

Z80 Status register

Fig. 18-39 8080/8085/Z80 status registers after an addition problem which creates an overflow.

The parity flag of the 8080/8085 is 0 because the answer (EA_{16}) contains five 1s and 5 is an odd number. However, the parity/overflow flag of the Z80 acts as an overflow flag during an arithmetic instruction and is 1 because we have exceeded our range of $7F_{16}$ (127_{10}) to 80_{16} (-128_{10}) for 8-bit signed binary numbers, and the result is therefore incorrect.

Decimal Addition

Because of differences in the way binary and decimal numbers round, and because numeric output to humans is usually decimal, it is sometimes useful to do arithmetic calculations by using decimal numbers rather than binary numbers. Actually, true decimal numbers are not used. Rather a mixture of binary and decimal, called binary-coded decimal is used. (The method used to create BCD numbers is covered in Chap. 1, and they have been discussed subsequently. You should review that section of Chap. 1 now if you are unsure of what BCD numbers are or how they are formed.)

One of the problems involved in using BCD numbers is that as the binary nibbles are added, invalid results are sometimes obtained.

Most microprocessors have an instruction called *decimal adjust* (or something similar). This instruction changes the number in the accumulator to what it would be if the last two numbers operated on had been packed BCD numbers instead of binary numbers.

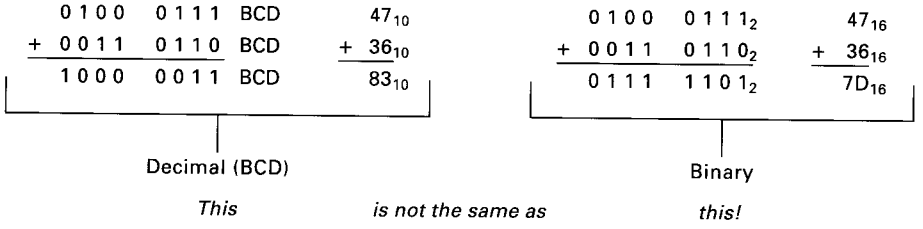
Let's look at an example. Figure 18-40 compares a decimal addition problem to the binary version of the same problem.

Notice first the difference between BCD and binary addition. BCD addition is not at all the same as binary addition. BCD is *decimal* addition using 4 bits to represent each decimal digit.

The program shown in Fig. 18-40 will help you understand the difference between binary and BCD addition (and subtraction). This program does the addition problem twice, once using binary numbers and once using BCD numbers. The result of the binary addition is stored in memory location $18A0_{16}$, and the resulting flags in location $18A1_{16}$. The result of the BCD addition is stored in location $18A2_{16}$, and the resulting flags in location $18A3_{16}$. Enter and run this program to see what results you get. When we ran the program, we found the following:

location $18A0_{16}$ = binary sum = 7D
location $18A1_{16}$ = binary flags = 00-0-1-0
[Z80:00-0-000]
location $18A2_{16}$ = BCD sum = 83
location $18A3_{16}$ = BCD flags = 10-1-0-0
[Z80: 10-1-000]

The status of the flags after the binary addition indicates that no flags were set (except the 8080/8085 parity flag indicating even parity).



Addr	Obj	Assembler	Comment
1800	3E	MVI A,47	This is being interpreted as a binary number
1801	47		
1802	C6	ADI 36	This also is being considered a binary number
1803	36		
1804	32	STA 18A0	We'll store the ***binary*** answer in memory location 18A0
1805	A0		
1806	18		
1807	F5	PUSH PSW	Put the flags and accumulator in stack
1808	C1	POP B	Retrieve flags and accumulator into register B and C
1809	79	MOV A,C	Move the flags from register C to the accumulator
180A	32	STA 18A1	We'll store the status of the flags from the ***binary*** addition in memory location 18A1
180B	A1		
180C	18		
180D	3E	MVI A,47	This is being interpreted as a ***decimal*** number
180E	47		
180F	C6	ADI 36	This also is being considered a ***decimal*** number
1810	36		
1811	27	DAA	Convert the answer to ***decimal***
1812	32	STA 18A2	We'll store the ***decimal*** answer in memory location 18A2
1813	A2		
1814	18		
1815	F5	PUSH PSW	Put the flags and accumulator in stack
1816	C1	POP B	Retrieve flags and accumulator into registers B and C
1817	79	MOV A,C	Move the flags from register C to the accumulator
1818	32	STA 18A3	We'll store the status of the flags from the ***binary*** addition in memory location 18A3
1819	A3		
181A	18		
181B	76	HALT	Stop

(8080/8085 mnemonics)

Fig. 18-40 Binary vs. BCD addition. (Continued on next page.)

Addr	Obj	Assembler	Comment
1800	3E	LD A,47	This is being interpreted as a binary number
1801	47		
1802	C6	ADD A,36	This also is being considered a binary number
1803	36		
1804	32	LD (18A0),A	We'll store the ***binary*** answer in memory location 18A0
1805	A0		
1806	18		
1807	F5	PUSH AF	Put the flags and the accumulator in stack
1808	C1	POP BC	Retrieve flags and accumulator into registers B and C
1809	79	LD A,C	Move the flags from register C to the accumulator
180A	32	LD (18A1),A	We'll store the status of the flags from the ***binary*** addition in memory location 18A1
180B	A1		
180C	18		
180D	3E	LD A,47	This is being interpreted as a ***decimal*** number
180E	47		
180F	C6	ADD A,36	This also is being considered a ***decimal*** number
1810	36		
1811	27	DAA	Convert the answer to ***decimal***
1812	32	LD (18A2),A	We'll store the ***decimal*** answer in memory location 18A2
1813	A2		
1814	18		
1815	F5	PUSH AF	Put the flags and accumulator in stack
1816	C1	POP BC	Retrieve flags and accumulator into registers B and C
1817	79	LD A,C	Move the flags from register C to the accumulator
1818	32	LD (18A3),A	We'll store the status of the flags from the ***binary*** addition in memory location 18A3
1819	A3		
181A	18		
181B	76	HALT	Stop

(Z80 mnemonics)

Fig. 18-40 (Continued)

After the BCD addition, the sign flag was set, but it has no valid meaning. It simply follows the state of the 8th bit of the accumulator.

Several points must be kept in mind when doing decimal addition and subtraction on the 8080/8085 and Z80 microprocessors.

With the 8080/8085 microprocessors, the DAA instruction only works after addition. Also, the DAA instruction works only with the accumulator.

With the Z80 microprocessor, the DAA instruction can be used after either addition or subtraction. This is made possible by the addition of the negative flag which the 8080/8085 does not have. This flag simply keeps track of whether an addition or subtraction was just performed. This flag is used in combination with the half-carry flag to correct the BCD answers.

Subtraction

Subtraction is the opposite of addition. All the flags operate the same except the carry flag. After subtraction, the carry flag indicates whether a borrow has occurred. You can think of it as a "borrow" flag. A 1 in the carry flag position indicates that a borrow from the nonexistent 9th bit was required to do the subtraction. A 0 indicates that no borrow from the 9th bit was required.

Figure 18-41 illustrates how to write a program to do single-precision subtraction.

You should assemble and run this program. When we did, we found that the result in the accumulator was FF. We also found that the overflow (*Z80*), sign, and carry flags had been set. The sign flag was set because the 8th bit of the answer is a 1 which indicates a negative-signed binary number. The overflow flag was set because $7F_{16} = 127_{10}$ and $80_{16} = -128_{10}$; therefore

$$\begin{array}{r} 127 \\ -(-128) \\ \hline 255 \end{array}$$

and 255_{10} is outside the valid range for 8-bit signed binary numbers. (The valid range is $+127_{10}$ to -128_{10}.) The

Addr	Obj	Assembler	Comment
1800	3E	MVI A,7F	
1801	7F		
1802	D6	SUI 80	
1803	80		
1804	76	HALT	

(8080/8085 mnemonics)

Addr	Obj	Assembler	Comment
1800	3E	LD A,7F	
1801	7F		
1802	D6	SUB A,80	
1803	80		
1804	76	HALT	

(Z80 mnemonics)

Fig. 18-41 Subtraction.

carry flag was set because a borrow from a 9th bit was needed to complete the subtraction.

18-7 8086/8088 FAMILY

The 8086/8088 has a variety of arithmetic instructions and various support instructions. The 8086/8088 can also work with ASCII and binary-coded decimal (BCD) numbers.

Arithmetic Instructions

The 8086/8088 has *add, subtract, add-with-carry, subtract-with-borrow, ASCII adjust, multiply, divide, integer multiply, integer divide,* and *conversion* instructions. These instructions use a value in one of the registers, memory, or an immediate number as their operands. Let's see how to use these instructions.

DEBUG Revisited

In just a moment we are going to begin studying some sample arithmetic programs for the 8086/8088 microprocessor. However, we must first learn more about the DEBUG utility.

Until now, we have assembled each program with DEBUG and then executed the program by using the *trace* command. Trace executes one instruction, displays the contents of the registers and flags, and then stops. This works well when the program is only a few lines long or when you must carefully observe the effect each instruction has on the registers. It is very slow, however.

DEBUG has another command which executes an entire program without stopping until the end. This is the g (**go**) command. Of course, the computer has to know where to start. If you just use the g command the computer assumes that it should start program execution at the memory location indicated by the instruction pointer (IP). If that is not where you want to start, such as when you want to execute a program for the second time, you have two ways to specify where to start. One way is to change the instruction pointer with the r (**register**) command. This is accomplished as follows:

```
-rip
IP 0100
:0100
-
```

You should start your assembly-language programs at or after address 0100H. The other way is to specify a starting point as part of the g (**go**) command. To start at memory location 0100H, for example, you would type

```
g=0100
```

Execution would start at address 0100 even though the instruction pointer might not contain that address.

When you use the g (**go**) command, the computer also has to know where to stop. You might think that the HLT (**HaLT**) instruction would work just fine. When you are using DEBUG, however, a different instruction is needed to stop the program. You are using DEBUG to control the computer. When your assembly-language routine is finished, control of the computer must be returned to DEBUG. DOS (the computer's disk operating system) has a routine which will do this. This routine is accessed by executing the

INT 20 instruction. For example, the arithmetic program we're going to study shortly looks like this:

```
MOV AL,49
ADD AL,1E
INT 20
```

Notice the use of INT 20 to stop program execution. There are a number of these DOS functions which handle the computer's housekeeping chores.

The g command is faster than individual t (trace) commands, and we can tell the computer where to start and stop, but it has one major disadvantage. When you use the r (register) command to view the registers after the program has run, they will have the same values they had in them before the program was run. This doesn't give you a chance to study the registers and flags to learn about how the program works.

The solution to this problem is breakpoints. A *breakpoint* is an address where you want program execution to stop. A breakpoint is specified as part of the g command. The difference between using a breakpoint to stop the program and INT 20 is that, when the breakpoint is reached, all the registers will be automatically displayed and their contents will not have been returned to their previous values. This allows you to see what all the registers and flags look like at that exact point in the program. For example:

```
g 0104
```

tells DEBUG to start program execution at the address indicated by the instruction pointer and to stop at address 0104. Notice that the instruction at address 0104 will not be executed. Instructions or data at address 0103 will be the last that the program will use. After the program stops at address 0104, the contents of the registers and flags will be automatically displayed.

The starting and stopping points for program execution can be combined into one command. For example:

```
g=0100 0104
```

will cause program execution to start at address 0100 and to stop at address 0104. The contents of the registers and flags will be automatically displayed.

When you run programs using a breakpoint, you need to remember that the instruction pointer will not be reset. Therefore, you'll have to change it back to the program's starting point if you wish to run the program more than once or specify the starting point in the g command as just shown.

If you wish, the t (trace) command can still be used to execute instructions one at a time.

Now let's try running a short program which will show you how to use these DEBUG commands and will allow you to learn about the 8086/8088 ADD instruction.

Addition

Let's start with a very simple addition program. Figure 18-42 illustrates this type of program.

We have shown the program twice: the first time using the g command without a breakpoint, showing that the registers will in fact be the same as before the program was run, and the second time using the g command with a breakpoint, showing that the contents of the registers will reflect how the program alters them. From this point on in this text we will use a breakpoint to stop program execution. Being able to see how a program affects the registers is important since our primary purpose is to explain what the program has accomplished and how it functions by studying the registers and flags after it has run.

Pay particular attention to AL and the flags. Notice their contents both before and after the program is run. You will find that the accumulator will have the number 67_{16} in it (which is the correct answer) and that only the auxiliary flag will be set.

Let's look at the flags a little more closely. Examining the flags from right to left, let's consider each and why it was or was not set. Refer to the bottom portion of Fig. 18-42.

There was no carry (NC) because no carry from the 8th bit to the 9th bit (which doesn't exist so it goes into the carry flag) occurred.

The parity was odd (PO) because the answer, $0110\,0111_2$, has an odd number of 1s.

There was an auxiliary carry (AC) because we had a carry from the 4th bit to the 5th bit. (Information about the half-carry is useful when dealing with BCD numbers.)

There was no zero (NZ) because the answer wasn't 0.

The answer was positive or "plus" (PL) because we did not have a 1 in the 8th bit of the accumulator so the answer could not have been negative.

Don't worry about the enable interrupt (EI) or auto-increment (UP) flags for the moment.

There was no overflow (NV) because we did not exceed our range for valid 8-bit signed binary numbers $+127$ to -128.

In fact, if you compare the state of the flags before the program was run with the state of the flags after it was run, only one of them changed. That was the auxiliary carry (AC) flag.

The Sign Flag

Let's look at a problem that produces a negative answer. See Fig. 18-43 at this time.

Notice that this is exactly the same problem as the last one except that we have changed the first operand, which used to be 49_{16} into $C9_{16}$ (-55_{10} if we consider these numbers to be signed binary numbers). We know that $-55_{10} + 30_{10} = -25_{10}$. Since this is a negative answer, we know that the sign flag should be set after the program is run.

```
                 1 1
   0100  1001      49₁₆      73₁₀
 + 0001  1110    + 1E₁₆    + 30₁₀
 ------------    --------  --------
   0110  0111      67₁₆     103₁₀
```

```
B>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 7420          JZ      0122
-
-a
8FFD:0100 mov al,49
8FFD:0102 add al,1e
8FFD:0104 int 20
8FFD:0106
-
-u 0100 0105
8FFD:0100 B049          MOV     AL,49
8FFD:0102 041E          ADD     AL,1E
8FFD:0104 CD20          INT     20
-
-g

Program terminated normally
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 B049          MOV     AL,49
-
```

Program assembled, unassembled, and executed without using a breakpoint to stop program (notice that registers have returned to their previous state)

```
B>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 7420          JZ      0122
-
-a
8FFD:0100 mov al,49
8FFD:0102 add al,1e
8FFD:0104 int 20
8FFD:0106
-
-u 0100 0105
8FFD:0100 B049          MOV     AL,49
8FFD:0102 041E          ADD     AL,1E
8FFD:0104 CD20          INT     20
-
-g 0104

AX=0067  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0104   NV UP EI PL NZ AC PO NC
8FFD:0104 CD20          INT     20
```

Program assembled, unassembled, and executed using a breakpoint to stop and display contents of registers and flags (notice that registers have *not* returned to their previous state)

Fig. 18-42 Simple 8086/8088 addition problem.

```
                    1 1
                  1100  1001        C9₁₆      -55₁₀
                + 0001  1110      + 1E₁₆    + 30₁₀
                  1110  0111        E7₁₆      -25₁₀

-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 B0C9          MOV     AL,C9
-a
8FFD:0100 mov al,c9
8FFD:0102 add al,1e
8FFD:0104 int 20
8FFD:0106
-
-u 0100 0104
8FFD:0100 B0C9          MOV     AL,C9
8FFD:0102 041E          ADD     AL,1E
8FFD:0104 CD20          INT     20
-
-g 104

AX=00E7  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0104   NV UP EI NG NZ AC PE NC
8FFD:0104 CD20          INT     20
```

Fig. 18-43 Simple 8086/8088 addition problem with a negative answer.

Assemble the program and run it. Observe the contents of AL and the status register before and after running the program so that you can compare them. After the program is run, AL should contain the value $E7_{16}$. The status register shows that only two flags have changed in exactly the same way as in the last example. The parity flag indicates even parity, and we have an auxiliary carry, just like the last example.

The difference this time is that we have a negative (NG) answer. This is what we expected to see. The sum of the addition problem was -25_{10} ($E7_{16}$). If we assume that our numbers are 8-bit signed binary numbers, then any number that has a 1 in the 8th bit is negative. $E7_{16}$ has a 1 in the 8th bit. The sign flag simply reflects the state of the most significant bit (8th or 16th depending on whether we are using 8-bit or 16-bit numbers).

The Zero Flag

Now let's change the program slightly so that we obtain a sum of 0. Then we can see how the flags react to this situation.

Figure 18-44 shows the problem and the program to solve the problem.

We are again assuming that our numbers are signed binary numbers. The problem is $C9_{16} + 37_{16} = 00_{16}$, which is $-55_{10} + 55_{10} = 0_{10}$. You should go through the binary addition of these two numbers now before you run the program. Notice both the answer and the carries. Notice also that we are using the same program as in the last example but have changed one of the operands.

We have a carry out of the most-significant bit (CY), we have a half-carry (AC), we have even parity (PE), and of course the zero flag indicates that our answer was in fact 0 (ZR). You should be able to look at the problem itself and at the flags before and after the program was run and be able to see why the flags have responded the way they have.

The Parity Flag

The 8086/8088 microprocessor has a parity flag which simply tells us how many 1s are in the accumulator after an arithmetic or logic operation. Even parity exists when an even number of 1s are in the accumulator. Odd parity exists when an odd number of 1s exist in the accumulator. Even parity (PE) and odd parity (PO) are indicated in the flags section of the DEBUG display.

Overflow Flag

When the overflow flag is set, it tells us that if the numbers which were just added or subtracted are signed binary numbers, then the valid range for such numbers has been exceeded and the result is incorrect. The valid range for 8-bit calculations is $+127$ to -128. The valid range for 16-bit calculations is $+32{,}767$ to $-32{,}768$. Let's modify our problem to create an overflow.

Figure 18-45 shows our problem and program. Remember that we are assuming that all values are to be interpreted as 8-bit *signed binary* values.

This problem is $123_{10} + 111_{10} =$ _____. First go through the binary addition and enter the program. Then write down the values you think will be found in AL and

```
                    1 1111  1111
                      1100  1001        C9₁₆      -55₁₀
                    + 0011  0111      + 37₁₆    + 55₁₀
                    ---------------   --------  --------
                    1 0000  0000        00₁₆       0₁₀

-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 7420          JZ      0122
-
-a
8FFD:0100 mov al,c9
8FFD:0102 add al,37
8FFD:0104 int 20
8FFD:0106
-
-u 0100 0104
8FFD:0100 B0C9          MOV     AL,C9
8FFD:0102 0437          ADD     AL,37
8FFD:0104 CD20          INT     20
-
-g 104

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0104   NV UP EI PL ZR AC PE CY
8FFD:0104 CD20          INT     20
-
```

Fig. 18-44 Simple 8086/8088 addition problem which produces a sum of 0.

the status register, run the program, and finally note the final values of AL and the status register.

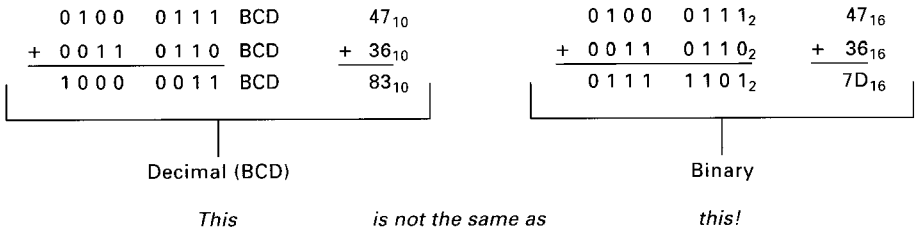
You should have a sum of EA_{16} in the accumulator and should find that there has been an auxiliary carry (AC), that the sign bit indicates that this is a negative number (NG), and that there has been an overflow (OV). EA_{16} is the correct sum *if you are using unsigned binary numbers!* If you interpret EA_{16} as a signed binary number, it has a value of -22_{10}. This is *not* the correct answer. We have exceeded our valid range for 8-bit signed binary numbers.

```
                      1111  1111
                      0111  1011        7B₁₆      123₁₀
                    + 0110  1111      + 6F₁₆    +111₁₀
                    ---------------   --------  --------
                      1110  1010        EA₁₆      234₁₀

-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 7420          JZ      0122
-
-a
8FFD:0100 mov al,7b
8FFD:0102 add al,6f
8FFD:0104 int 20
8FFD:0106
-
-u 0100 0104
8FFD:0100 B07B          MOV     AL,7B
8FFD:0102 046F          ADD     AL,6F
8FFD:0104 CD20          INT     20
-
-g 104

AX=00EA  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0104   OV UP EI NG NZ AC PO NC
8FFD:0104 CD20          INT     20
-
```

Fig. 18-45 Simple 8086/8088 addition problem which produces an overflow.




```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=0100   NV UP EI PL NZ NA PO NC
8FFD:0100 7420          JZ      0122
-
-a
8FFD:0100 mov al,47       ;1st operand (binary)
8FFD:0102 add al,36       ;add 2d, put sum in al (binary)
8FFD:0104 mov [01A0],al   ;store sum
8FFD:0107 pushf           ;copy flags
8FFD:0108 pop bx          ;retrieve flags
8FFD:0109 mov [01A1],bx   ;store flags
8FFD:010D mov al,47       ;1st operand
8FFD:010F add al,36       ;add 2d, put sum in al (binary)
8FFD:0111 daa             ;convert sum to BCD
8FFD:0112 mov [01A3],al   ;store BCD sum
8FFD:0115 pushf           ;copy flags
8FFD:0116 pop bx          ;retrieve flags
8FFD:0117 mov [01A4],bx   ;store flags
8FFD:011B int 20          ;return to DEBUG
8FFD:011D
-
-g 011b

AX=0083  BX=FA92  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=8FFD  ES=8FFD  SS=8FFD  CS=8FFD  IP=011B   OV UP EI NG NZ AC PO NC
8FFD:011B CD20          INT     20
-
-d 01A0 01AF
8FFD:01A0  7D 06 F2 83 92 FA 79 6E-74 61 78 20 65 72 6F  }. . . . . yntax erro
```

Fig. 18-46 Binary vs. BCD addition.

Decimal Addition

Because of differences in the way binary and decimal numbers round, and because numeric output to humans is usually decimal, it is sometimes helpful to do arithmetic calculations by using decimal numbers rather than binary numbers. Actually, true decimal numbers are not used. Rather a mixture of binary and decimal, called binary-coded decimal, is used. (The method used to create BCD numbers is covered in Chap. 1, and they have been discussed subsequently. You should review that section of Chap. 1 now if you are unsure of what BCD numbers are or how they are formed.)

One of the problems involved in using BCD numbers is that as the binary nibbles are added invalid results are sometimes obtained.

Most microprocessors have an instruction called *decimal adjust* (or something similar). This instruction changes the number in the accumulator to what it would be if the last two numbers operated on had been packed BCD numbers instead of binary numbers.

Let's look at an example. Figure 18-46 is a decimal addition problem which is compared to the binary version of the same problem.

Notice first the difference between BCD and binary addition. BCD addition is not at all the same as binary addition. BCD is *decimal* addition using 4 bits to represent each decimal digit.

The program shown in Fig. 18-46 will help you understand the difference between binary and BCD addition (and subtraction). This program does the addition problem twice, once using binary numbers and once using BCD numbers. The result of the binary addition is stored in memory location $01A0_{16}$, and the resulting flags in locations $01A1_{16}$ and $01A2_{16}$. The result of the BCD addition is stored in location $01A3_{16}$, and the resulting flags in locations $01A4_{16}$ and $01A5_{16}$. Assemble and run this program to see whether you obtain the same results. When we ran the program we found the following:

location $01A0_{16}$ = binary sum = 7D
location $01A1_{16}$ = binary flags (low byte) = 06

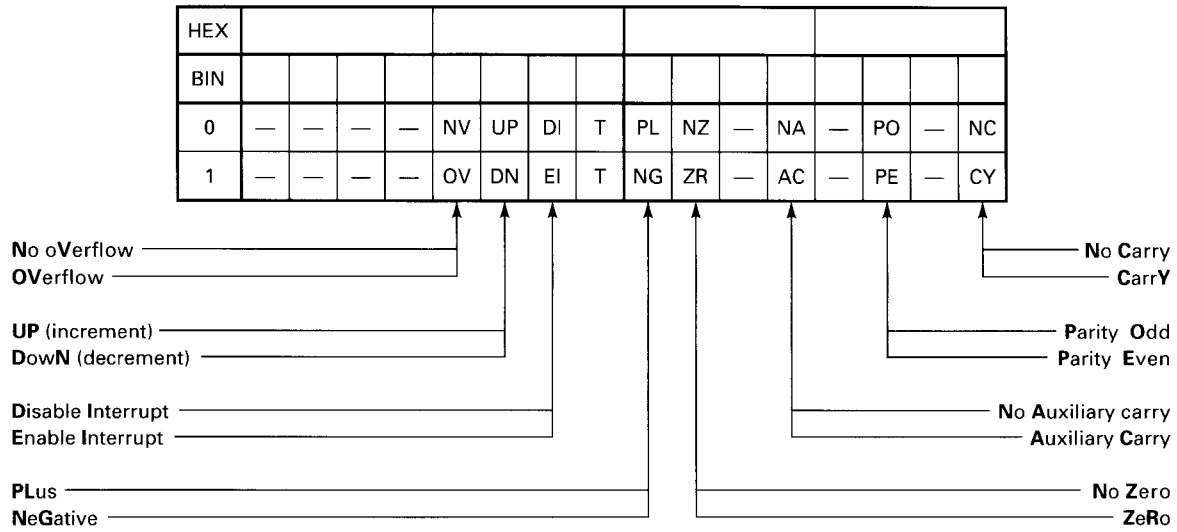


Fig. 18-47 8086/8088 flag chart.

location $01A2_{16}$ = binary flags (high byte) = F2
location $01A3_{16}$ = BCD sum = 83
location $01A4_{16}$ = BCD flags (low byte) = 92
location $01A5_{16}$ = BCD flags (high byte) = FA

Figure 18-47 will help you understand what the stored flag values mean.

When you store the value of the flag register (status register), you can place the hexadecimal values on the chart in Fig. 18-47. You can then convert the hexadecimal values to binary values and look in the 0 row or 1 row to see what conditions the flags indicate existed at a certain point in the program.

In this example we have placed the values of the flags after the binary addition in Fig. 18-48.

First notice that we have reversed the order of the hexadecimal values for the flags. The PUSHF instruction pushes the current value of the flags onto the stack. The POP BX then retrieves that value into the BX register. At this point the values are still in their correct order. In fact, if you will refer to Fig. 18-46, those areas have been printed in bold to illustrate this fact. Notice that BX contains $FA92_{16}$. Look at memory locations $01A4_{16}$ and $01A5_{16}$. Notice that those 2 bytes have been reversed. PUSH and POP instructions do not reverse the bytes. However, MOV instructions do. The MOV instruction places the data into memory in a low-byte/high-byte order, which has the effect of reversing the bytes when the memory locations are examined.

The binary addition problem produced an answer of $7D_{16}$, as we expected. There were no carries or overflows, and we have even parity. These conditions are shown in bold type in Fig. 18-48.

The BCD addition produced a sum of 83_{BCD}, as we thought it would. Don't be concerned about the flags at this point. Simply notice that they are different. They reflect

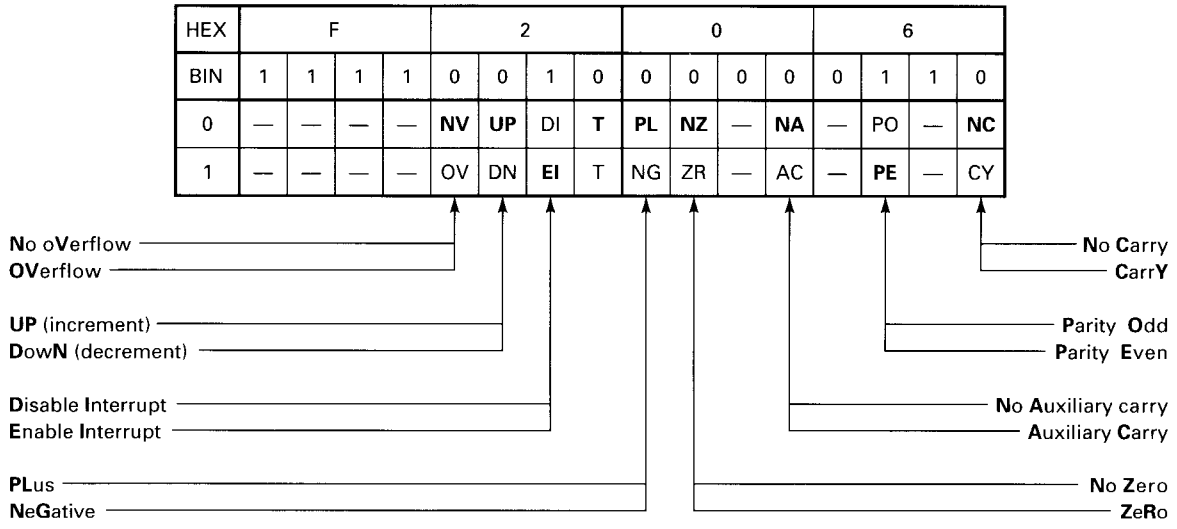


Fig. 18-48 Conditions after the binary addition.

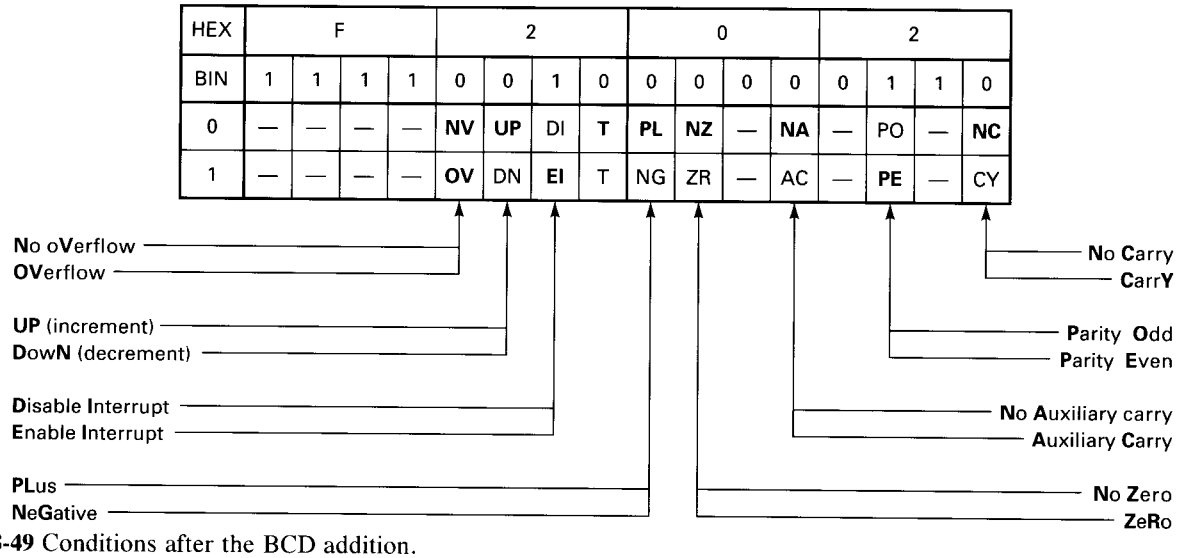


Fig. 18-49 Conditions after the BCD addition.

conditions that result from the conversion from binary to BCD (see Fig. 18-49.)

Subtraction

Subtraction is the opposite of addition. All the flags operate the same except the carry flag. After subtraction, the carry flag indicates whether a borrow has occurred or not. You can think of it as a "borrow" flag. A 1 in the carry flag position indicates that a borrow from a nonexistent bit was required to do the subtraction. A 0 indicates that no borrow was required.

Figure 18-50 illustrates how to write a program to do single-precision subtraction.

You should assemble and run this program. When we did, we found that the result in AL was FF. We also found that there was an overflow, the answer was negative, and there was a carry. The answer was negative because the 8th bit of the answer is a 1, which indicates an 8-bit negative signed binary number. There was an overflow because $7F_{16} = 127_{10}$ and $80_{16} = -128_{10}$; therefore

$$\begin{array}{r} 127 \\ -(-128) \\ \hline 255 \end{array}$$

and 255_{10} is outside the valid range for 8-bit signed binary numbers. (The valid range is $+127_{10}$ to -128_{10}.) There was a carry because a borrow from a 9th bit was needed to complete the subtraction.

Multiplication

The 8-bit microprocessors featured in this text do not have a multiply instruction. To multiply, the programmer must use many instructions to accomplish what the 8086/8088 does with just one instruction.

There are several ways the 8086/8088 can multiply. It can multiply signed binary numbers by using the **I**nteger **MUL**tiply (IMUL) instruction. It can also multiply unsigned binary numbers by using the **MUL**tiply (MUL) instruction.

Whether signed or unsigned, the 8086/8088 can multiply two 8-bit binary numbers to produce a 16-bit answer or two 16-bit binary numbers to produce a 32-bit answer.

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=379E  BP=0000  SI=0000  DI=0000
DS=9C86  ES=9C86  SS=9C86  CS=9C86  IP=0100   NV UP EI PL NZ NA PO NC
9C86:0100 8BFE          MOV     DI,SI
-
-a
9C86:0100 mov al,7f       ;load first operand
9C86:0102 sub al,80       ;subtract second operand
9C86:0104 int 20          ;return control to DEBUG
9C86:0106
-
-g 0104

AX=00FF  BX=0000  CX=0000  DX=0000  SP=379E  BP=0000  SI=0000  DI=0000
DS=9C86  ES=9C86  SS=9C86  CS=9C86  IP=0104   OV UP EI NG NZ NA PE CY
9C86:0104 CD20          INT     20
-
```

Fig. 18-50 Subtraction.

```
  1E
× FC
1DBB

B>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=379E  BP=0000  SI=0000  DI=0000
DS=9C86  ES=9C86  SS=9C86  CS=9C86  IP=0100   NV UP EI PL NZ NA PO NC
9C86:0100 8BFE          MOV     DI,SI
-
-a
9C86:0100 mov al,1E       ;first operand
9C86:0102 mov bl,FC       ;second operand (no immediate mode allowed)
9C86:0104 mul bl          ;mul automatically uses value value in al or ax
9C86:0106 int 20          ;return control to DEBUG
9C86:0108
-
-g 106

AX=1D88  BX=00FC  CX=0000  DX=0000  SP=379E  BP=0000  SI=0000  DI=0000
DS=9C86  ES=9C86  SS=9C86  CS=9C86  IP=0106   OV UP EI PL NZ NA PO CY
9C86:0106 CD20          INT     20
-
```

Fig. 18-51 Eight-bit multiplication on the 8086/8088.

If two 8-bit numbers are to be multiplied, one of them must be placed in AL. The other can be in a register or memory location. *Immediate mode multiplication is not allowed.* That is, you cannot do this:

```
mov al,1E
mul al,FC
int 20
```

You cannot specify a number to be multiplied by the number in AL in the instruction itself. You must move it to a register or memory location.

The problem 1E × FC and the program to solve it are shown in Fig. 18-51.

Notice that we moved the first number into AL and then the second into BL. We then only needed to say

```
mul bl
```

because the microprocessor assumes that the first number is in AL. The answer is placed in AX.

The only two flags that have any meaning after a MUL or IMUL instruction are the overflow and carry flags. If the upper byte of the answer (AH) is 00, then both of these flags will be cleared. Any other result in AH causes both of these flags to be set. Since the value in AH in our example is not 0, both the overflow and carry flags are set after the program is run.

Figure 18-52 illustrates a 16-bit multiplication problem.

```
     FFE2
×    12D3
 12DOCB46

B>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=440E  BP=0000  SI=0000  DI=0000
DS=9BBF  ES=9BBF  SS=9BBF  CS=9BBF  IP=0100   NV UP EI PL NZ NA PO NC
9BBF:0100 C3            RET
-
-a
9BBF:0100 mov ax,FFE2     ;first 16-bit operand
9BBF:0103 mov bx,12D3     ;second 16-bit operand
9BBF:0106 mul bx          ;multiply ax by bx
9BBF:0108 int 20          ;return control to DEBUG
9BBF:010A
-
-g 108

AX=CB46  BX=12D3  CX=0000  DX=12D0  SP=440E  BP=0000  SI=0000  DI=0000
DS=9BBF  ES=9BBF  SS=9BBF  CS=9BBF  IP=0108   OV UP EI PL NZ NA PO CY
9BBF:0108 CD20          INT     20
-
-
```

Fig. 18-52 Sixteen-bit multiplication on the 8086/8088.

```
     58 remainder 2
FB ) 564A


B>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=45DE  BP=0000  SI=0000  DI=0000
DS=9BA2  ES=9BA2  SS=9BA2  CS=9BA2  IP=0100   NV UP EI PL NZ NA PO NC
9BA2:0100 2126A105      AND      [05A1],SP                        DS:05A1=E903
-
-a
9BA2:0100 mov ax,564A     ;dividend (16-bits)
9BA2:0103 mov bl,FB       ;divisor (8-bits)
9BA2:0105 div bl          ;divide ax by bl
9BA2:0107 int 20          ;return control to DEBUG
9BA2:0109
-
-g 107

AX=0258  BX=00FB  CX=0000  DX=0000  SP=45DE  BP=0000  SI=0000  DI=0000
DS=9BA2  ES=9BA2  SS=9BA2  CS=9BA2  IP=0107   NV UP EI PL NZ AC PO CY
9BA2:0107 CD20          INT     20
-
```

Fig. 18-53 A 16-bit number divided by an 8-bit number using the 8086/8088 DIV instruction.

The process is similar to that used in 8-bit multiplication. You use 16-bit registers instead of 8-bit, and the answer is 32-bits wide! The upper 2 bytes (16 bits) are found in DX, and the lower 2 bytes are found in AX.

The flags respond as they do for 8-bit multiplication.

Division

We handle division in a way which is similar to, yet the opposite of, the way multiplication is handled.

When division is done, the dividend (number to be divided) must be twice as wide (16 or 32 bits) as the divisor (8 or 16 bits). Figure 18-53 illustrates how a 16-bit dividend is divided by an 8-bit divisor.

Notice how we again moved the operands into a register to prepare for the actual division. Our 16-bit dividend ($564A_{16}$) was placed in AX and the 8-bit divisor (FB_{16}) was placed in BL. Notice that we simply say

```
div bl
```

and the microprocessor assumes we are dividing BL into AX.

Now notice how the answer is displayed. The answer is 58_{16}, with a remainder of 2_{16}. The quotient appears in the lower half of AX (AL), and the remainder is in the upper half of AX (AH). This is where the answer to a problem which divides a 16-bit number by an 8-bit number is found.

Figure 18-54 illustrates how to divide a 32-bit binary number by a 16-bit binary number.

To perform this type of problem, you must place the most significant 16 bits of the dividend in register DX. Place the least significant 16 bits of the dividend in register AX. Then place the 16-bit divisor in BX or CX. After the division the answer (quotient) will be found in register AX, with the remainder in register DX.

GLOSSARY

ASCII American Standard Code for Information Interchange. A binary code in which letters of the alphabet, numbers, punctuation, and certain control characters are represented.

BCD (binary-coded decimal) Decimal numbers which replace each decimal digit with its 4-bit binary equivalent.

multiple-precision number A number which is composed of more than one binary word.

single-precision number A number which is composed of one binary word. In an 8-bit microprocessor this is an 8-bit number, and in a 16-bit microprocessor this is a 16-bit number.

```
          789A remainder 8
45CE ) 20E28DF4


B>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=4ECE  BP=0000  SI=0000  DI=0000
DS=9B13  ES=9B13  SS=9B13  CS=9B13  IP=0100   NV UP EI PL NZ NA PO NC
9B13:0100 7420          JZ      0122
-
-
-a
9B13:0100 mov dx,20E2     ;most significant word of dividend
9B13:0103 mov ax,8DF4     ;least significant word of dividend
9B13:0106 mov bx,45CE     ;divisor
9B13:0109 div bx          ;divide DXAX register pair by BX
9B13:010B int 20          ;return control to DEBUG
9B13:010D
-
-g 10b

AX=789A  BX=45CE  CX=0000  DX=0008  SP=4ECE  BP=0000  SI=0000  DI=0000
DS=9B13  ES=9B13  SS=9B13  CS=9B13  IP=010B   NV UP EI NG NZ AC PE CY
9B13:010B CD20          INT     20
-
```

Fig. 18-54 A 32-bit number divided by a 16-bit number using the 8086/8088 microprocessor.

SELF-TESTING REVIEW

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

1. Binary-coded decimal numbers are decimal numbers in which each digit is represented by its _____-_____ _____ equivalent.
2. (*4-bit binary*) What is the binary value for 10_{10}?
3. (1010_2) When 8-bit binary numbers are added, the carry flag indicates when a carry from the _____ bit to the _____ bit has occurred.
4. (*8th, 9th*) When 8-bit binary numbers are added, the half-carry flag indicates when a carry from the _____ bit to the _____ bit has occurred.
5. (*4th, 5th*) A number which can be represented by 1 byte is called a _____-precision number.
6. (*single*) After subtraction, the carry flag indicates whether or not a _____ has occurred.
7. (*borrow*) Do the 8-bit microprocessors featured in this text have multiply or divide instructions? *(No)*

PROBLEMS

General

18-1. What two types of information are generated by a microprocessor during addition?

18-2. What does $1_2 + 1_2 + 1_2 = ?$

18-3. What is

$$\begin{array}{r} 1010\ 1110_2 \\ +\ 0011\ 0111_2 \\ \hline \end{array}$$

18-4. What is

$$\begin{array}{r} 0111\ 1111\ 0110\ 1101 \\ +\ 0001\ 1000\ 1111\ 0110 \\ \hline \end{array}$$

18-5. When we are using all 8 bits to represent the numbers 1_{10} to 255_{10}, we refer to these as _____ binary numbers.

18-6. When we use 8-bit binary numbers to represent values from -128_{10} to $+127_{10}$, we refer to these as _____binary numbers.

18-7. Find the 8-bit signed binary value for -100_{10}.

18-8. What flag warns the programmer that the last answer produced exceeds the valid range for signed binary numbers?

18-9. What flag tells the programmer whether the number in the accumulator is positive or negative?

Specific Microprocessor Families

Solve the following problems using the microprocessor of your choice.

18-10. Write a program which will add the unsigned binary numbers 67_{16} and 23_{16}. Determine which flags are altered by the program and why.

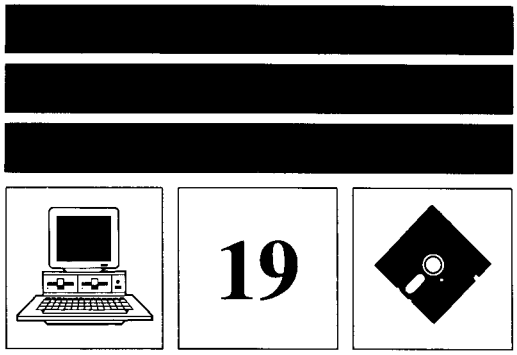
18-11. Write a program which will subtract the signed binary number $4D_{16}$ from $7F_{16}$. Determine which flags are altered by the program and why.

18-12. Write a program which will add the *decimal* numbers 40_{10} and 52_{10}.

18-13. With your computer or microprocessor trainer store the unsigned binary numbers 67_{16} and 23_{16} in two consecutive memory locations. Now write a program which will find the sum of these two numbers and store the sum in a free memory location. (8086/8088 users: To store 67_{16} and 23_{16} in memory locations use the DEBUG e (enter) command. For example, typing

-e 0180 67 23 00 00

will enter 67, 23, 00, and 00 into memory locations 0180, 0181, 0182, and 0183, respectively.)



19

LOGICAL INSTRUCTIONS

This chapter discusses the logical instructions of our featured microprocessors. These instructions, along with the arithmetic and shift and rotate instructions, give us the ability to alter bits and bytes (data) in a predictable fashion.

You may wish to review logic gates before beginning this chapter. Microprocessors use logical instructions the way digital circuits use logic gates.

New Concepts

There are really only four basic logical functions: AND, OR, EXCLUSIVE-OR, and NOT. The NAND, NOR, EXCLUSIVE-NOR, and NEGate functions are simply extensions of the four basic functions.

We will look at each of the basic four plus a couple of other special instructions some of the microprocessors have. We will also discuss masking, a primary use of the logical instructions.

19-1 THE AND INSTRUCTION

When we AND 2 bits or conditions, we are saying that the output bit, or condition, is true only if both the input bits, or conditions, are true. For example, there will be a voltage at the output of a circuit only if there is voltage at both of its inputs. Or, a bit in memory will be 1 only if 2 other input bits are also 1. Or, a drill will begin to lower only if the workpiece has been secured and the worker's hands are away from the bit.

Input		Output
B	A	Y
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 19-1 AND truth table.

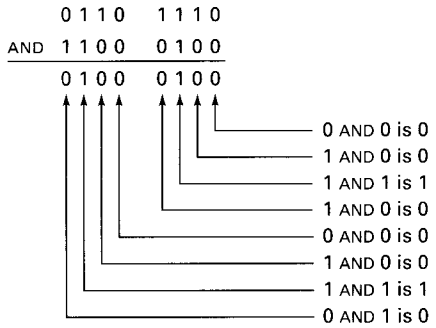


Fig. 19-2 ANDing 2 bytes together.

ANDing Bits

The truth table to AND 2 bits, or conditions, is shown in Fig. 19-1. Notice that the only way to get a 1 out is to put two 1s in.

ANDing Bytes

We can AND entire bytes, or words also. We simply apply the logic shown in the table to each bit. It's almost like turning a truth table on its side. For example, a problem in which we must AND 2 bytes is shown in Fig. 19-2.

Notice that we have applied the logic from the AND truth table to each bit. The only 1s in the answer are in columns where both the inputs are also 1.

EXAMPLE 19-1

Solve the following logical problem.

1011 1110 AND 0111 0001 is ???? ????

SOLUTION

```
  1011 1110
AND 0111 0001
  0011 0000
```

Masking

A common use of the AND instruction is to AND bits or bytes with a mask. A *mask* allows us to change some bits in a certain way while allowing others to pass through unchanged. Look at the example shown in Fig. 19-3.

Notice that the upper nibble of the data byte passed through the 1s of the mask unchanged. However, every bit of the lower nibble passing through the 0s was cleared.

ANDing a mask to data can be viewed in either of two ways. You can say that selected data bits pass through unchanged while all others are cleared. Or, you can say that selected data bits are cleared while others pass through unaltered.

EXAMPLE 19-2

Devise a mask which when ANDed to an 8-bit data byte will clear all bits except the first 2 (2 least significant bits).

SOLUTION

0000 0011

For example:

```
  1111 1111   ← data
AND 0000 0011 ← mask
  0000 0011
```

19-2 THE OR INSTRUCTION

When we OR 2 bits or conditions, we are saying that the output will be true (or 1) if either of the input bits or conditions is true (1) or if both of the input bits or conditions are true.

ORing Bits

The truth table to OR 2 bits or conditions is shown in Fig. 19-4.

Notice that you get a 1 out if any input is a 1. Or, to look at it another way, the only way to get a 0 out is to have 0s at both inputs.

Input		Output
B	A	Y
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 19-4 OR truth table.

ORing Bytes

We can OR entire bytes, or words also. We simply apply the logic shown in the table to each bit. For example, the same problem used in the previous section, but now ORing the 2 bytes together, is shown in Fig. 19-5.

Notice that we have used the logic from the OR truth table and applied it to each bit. The only 0s in the answer are in columns where both the inputs are also 0.

EXAMPLE 19-3

Solve the following logical problem.

1011 1110 OR 0111 0001 is ???? ????

SOLUTION

```
  1011 1110
OR 0111 0001
  1111 1111
```

Masking

A common use of the OR instruction is to OR bits or bytes with a mask. A *mask* allows some bits to pass through unchanged while others are changed in a certain way. Look at the example shown in Fig. 19-6.

```
  1001  1001  ← data
AND 1111  0000  ← mask
  1001  0000
```

Fig. 19-3 Using the AND instruction to mask bits.

```
  0110  1110
OR 1100  0100
  1110  1110
```

0 AND 0 is 0
1 AND 0 is 1
1 AND 1 is 1
1 AND 0 is 1
0 AND 0 is 0
1 AND 0 is 1
1 AND 1 is 1
0 AND 1 is 1

Fig. 19-5 ORing two bytes together.

```
  1001  1001  <-- data
OR 1111  0000  <-- mask
   1111  1001
```

Fig. 19-6 Using the OR instruction to mask bits.

Notice that the lower nibble of the data byte passing through the 0s of the mask was unchanged while every bit of the upper nibble passing through the 1s was set.

ORing a mask to data can be viewed in either of two ways. You can allow selected data bits to pass through unchanged while all others are set. Or, you can allow selected data bits to be set while all others pass through unaltered.

EXAMPLE 19-4

Devise a mask which when ORed to an 8-bit data byte will set all bits except the first 2 (2 least significant bits).

SOLUTION

1111 1100
For example:

```
  0000 0000   <- data
OR 1111 1100  <- mask
   1111 1100
```

19-3 THE EXCLUSIVE-OR (EOR, XOR) INSTRUCTION

When we EXCLUSIVELY OR (EOR, XOR) 2 bits or conditions, we are saying that the output bit or condition is true only if one or the other of the input bits or conditions is true, but not both. For example, there will be a voltage at the output of a circuit only if there is voltage at one or the other, but not both, of its inputs.

XORing Bits

The truth table to XOR 2 bits or conditions is shown in Fig. 19-7.

Input		Output
B	A	Y
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 19-7 XOR truth table.

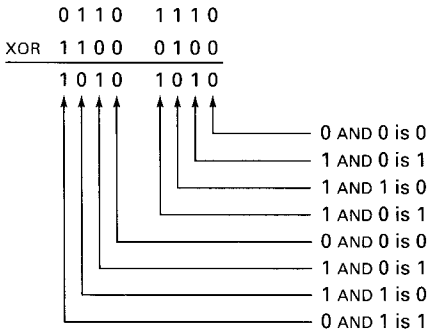


Fig. 19-8 XORing two bytes together.

Notice that the only way to get a 1 out is to have one, but not both, of the inputs be a 1.

XORing Bytes

We can XOR entire bytes, or words also. We simply apply the logic shown in the table to each bit. For example, the same problem shown in the previous two sections, but this time XORing the 2 bytes, is shown in Fig. 19-8.

Notice that we have used the logic from the XOR truth table and applied it to each bit. The only 1s in the answer are in columns where one but not both the inputs are 1.

EXAMPLE 19-5

Solve the following logical problem.

1011 1110 XOR 0111 0001 is ???? ????

SOLUTION

```
  1011 1110
XOR 0111 0001
    1100 1111
```

Masking

A common use of the XOR instruction is to XOR bits or bytes with a mask. A *mask* allows some bits to pass through unchanged while others are changed in a certain way. Look at the example shown in Fig. 19-9.

Notice that the lower nibble of the data byte passed through the 0s of the mask unchanged while every bit of the upper nibble passing through the 1s was inverted.

XORing a mask to data can be viewed in either of two ways. You can allow selected data bits to pass through

```
  1001  1001  <-- data
XOR 1111  0000  <-- mask
    0110  1001
```

Fig. 19-9 Using the XOR instruction to mask bits.

unchanged while all others are inverted. Or, you can allow selected data bits to be inverted while all others pass through unaltered.

EXAMPLE 19-6

Devise a mask which when XORed to an 8-bit data byte will invert all bits except the first 2 (2 least significant bits).

SOLUTION

1111 1100

For example:

```
    1111 1111    ← data
XOR 1111 1100    ← mask
    0000 0011
```

19-4 THE NOT INSTRUCTION

When we NOT or invert bits or conditions, we are saying that the output bit or condition is the opposite of the input bit or condition. For example, if there is a voltage at the input, there will not be one at the output; or if there is *no* voltage at the input, there *will* be a voltage at the output.

NOT-ing (Inverting) Bits

The truth table for the NOT function is shown in Fig. 19-10.

Input	Output
A	Y
0	1
1	0

Fig. 19-10 NOT truth table.

NOT-ing (Inverting) Bytes

We can NOT or invert entire bytes, or words also. We simply apply the logic shown in the table to each bit. An example of inverting or complementing a number is shown in Fig. 19-11.

NOT 1111 0000 is 0000 1111

Fig. 19-11 "NOT-ing" or inverting a binary number.

Notice that we have changed every 0 to a 1 and every 1 to a 0—that is, we have inverted every bit of the byte. This is the 1's complement of the number.

EXAMPLE 19-7

Solve the following logical problem.

NOT 1011 1110 is ???? ????

SOLUTION

0100 0001

19-5 THE NEG (NEGATE) INSTRUCTION

The NEGate instruction finds the 2's complement of a number. To find the 2's complement, we first find the 1's complement and then add 1. An example is shown in Fig. 19-12.

Specific Microprocessor Families

Let's see how these instructions work in the different microprocessor families.

19-6 6502 FAMILY

The 6502 has three of the instructions discussed in the New Concepts section of this chapter plus one instruction not discussed there. These are the AND, OR, EOR, and BIT instructions. Let's look at each.

The AND Instruction

The 6502 AND instruction works exactly as described in the New Concepts section. If we use the same example we used in Fig. 19-3 in the New Concepts section, we will find that the 6502 does in fact AND bytes as discussed.

Figure 19-13 shows our original problem and solution plus a 6502 program which solves the problem. After running this program, you will find that the accumulator contains 90_{16}. This is exactly what we expected after 99_{16} was masked with FO_{16}.

```
  1111  0000  ←—— number
  0000  1111  ←—— 1's complement
+          1  ←—— add 1
  0001  0000  ←—— 2's complement (original number NEGated)
```

Fig. 19-12 NEGating a number (2's complement).

```
      1001  1001  <---- data
AND   1111  0000  <---- mask
      1001  0000
```

```
0340  A9 99     LDA #$99        ;load A with 1001 1001
0342  29 F0     AND #$F0        ;AND mask
0344  00        BRK             ;stop
```

Fig. 19-13 Using the 6502 AND instruction to mask bits.

```
      1001  1001  <---- data
OR    1111  0000  <---- mask
      1111  1001
```

```
0340  A9 99     LDA #$99        ;load A with 1001 1001
0342  09 F0     ORA #$F0        ;OR mask
0344  00        BRK             ;stop
```

Fig. 19-14 Using the 6502 OR instruction to mask bits.

If you check the 6502 instruction set, you will find that the AND instruction affects the negative and zero flags. In this case the negative flag is set because the 8th bit of the accumulator is 1, indicating a 2's-complement negative number.

The OR Instruction

The 6502 OR instruction also works exactly as described in the New Concepts section. If we use the example from Fig. 19-6 in the New Concepts section, we find that the 6502 does OR bytes as discussed there.

Figure 19-14 shows our original problem and solution plus a 6502 program which solves the problem. After entering and running the program, you will find that the accumulator contains $F9_{16}$. This is the value we expected the accumulator to have.

The OR instruction also affects the negative and zero flags. You will find that the negative flag is again set because the 8th bit is 1, indicating a 2's-complement negative number.

The EOR Instruction

The 6502 EOR instruction also works as described in the New Concepts section. If we use the example from Fig. 19-9 in the New Concepts section, we'll find that the 6502 does EOR bytes as discussed there.

Figure 19-15 shows our original problem and solution plus a 6502 program which solves the problem. Enter and run the program. We expected the accumulator to have 69_{16} after EORing.

The EOR instruction also affects the negative and zero flags. This time neither is set; the result is neither negative nor zero.

The BIT Instruction

The BIT instruction was not described in the New Concepts section and is somewhat unusual. Refer to the BIT instruction in the Expanded Table of 6502 Instructions Listed by Category.

The BIT instruction ANDs a memory location with the accumulator. However, the result is not stored anywhere. Neither the accumulator nor the memory location is changed.

If the result of the AND is zero, the zero flag is set. If the result is not zero, the zero flag is not set.

The negative and overflow flags are affected in an unusual way. The status of the negative and overflow flags is not determined by the result of the AND process but rather is copied from bits 6 and 7 (7th and 8th bits) of the memory location.

A program which illustrates the operation of the BIT instruction is shown in Fig. 19-16.

The BIT instruction is useful when using the flags to control branching. You can alter the flags with a logical condition without actually changing the accumulator or memory location.

```
      1001  1001  <---- data
EOR   1111  0000  <---- mask
      0110  1001
```

```
0340  A9 99     LDA #$99        ;load A with 1001 1001
0342  49 F0     EOR #$F0        ;EOR mask
0344  00        BRK             ;stop
```

Fig. 19-15 Using the 6502 EOR instruction to mask bits.

```
0340  A9 C0     LDA #$C0        ;load A with 1100 0000
0342  8D A0 03  STA #03A0       ;store 1100 0000 in location 03A0
0345  A9 00     LDA #$00        ;load A with 0000 0000
0347  2C A0 03  BIT $03A0       ;AND A (0000 0000) with 0340 (1100 0000)
034A  00        BRK             ;stop

After running the program:

negative flag = 1, overflow flag = 1, break flag = 1, zero flag = 1, accumulator = 00
```

Fig. 19-16 Using the 6502 BIT instruction.

19-7 6800/6808 FAMILY

The 6800/6808 has all the instructions discussed in the New Concepts Section plus one instruction not discussed there. These are the ANDA/ANDB, ORAA/ORAB, EORA/EORB, BITA/BITB, COM/COMA/COMB, and NEG/NEGA/NEGB instructions. Notice that each instruction has a mnemonic for each accumulator and that some (COM and NEG) have one for memory locations also. Let's look at each.

Clearing the Flags

The 6800/6808 examples which follow cover both the result of the logical operation and the condition of the flags. It is helpful to be able to clear the flags before the examples are run so that the previous condition of the flags is not confused with the effect the example had on the flags.

Place the following program in an area of memory you do not plan to use for the examples. Then run this program to clear both accumulators and all flags before running each example program.

```
xxxx 4F   CLRA   ;clear A
xxxx 5F   CLRB   ;clear B
xxxx 06   TAP    ;clear flags
xxxx 3E   WAI    ;stop
```

The ANDA/ANDB Instruction

The 6800/6808 AND instruction works exactly as described in the New Concepts section. If we use the same example we discussed in the New Concepts section (Fig. 19-3), we will find that the 6800/6808 does in fact AND bytes as discussed.

Figure 19-17 shows our original problem and solution plus a 6800/6808 program which solves the problem. If you will notice the condition of the accumulator and flags after running this program, you will find that the accumulator contains 90_{16} as we expected.

If you check the 6800/6808 instruction set, you will find that the AND instruction affects the negative and zero flags. (The overflow flag is always cleared.) In this case the negative flag is set because the 8th bit of the accumulator is 1, indicating a 2's-complement negative number. (It is assumed that the flags just discussed were cleared before the program was started.)

The ORAA/ORAB Instruction

The 6800/6808 ORAA/ORAB instruction also works exactly as described in the New Concepts section. We'll use the example found in Fig. 19-6 in the New Concepts section.

Figure 19-18 shows our original problem and solution plus a 6800/6808 program which solves the problem. After entering and running the program, you will find that the accumulator contains $F9_{16}$. This is what we expected.

The OR instruction also affects the negative and zero flags. (The overflow flag is always cleared.) The negative flag is again set because the 8th bit of A is 1, indicating a 2's-complement negative number.

The EORA/EORB Instruction

Let's look at the 6800/6808 EORA/EORB instruction. If we use the example from Fig. 19-9 in the New Concepts section, we will find that the 6800/6808 does EOR bytes as discussed.

Figure 19-19 shows our original problem and solution from Fig. 19-9 plus a 6800/6808 program which solves the problem. Enter and run the program. You will find that the accumulator contains 69_{16}.

The EOR instruction also affects the negative and zero flags. (The overflow flag is always cleared.) In this case neither was set; the result is neither negative nor zero.

```
    1001  1001  <---- data
AND 1111  0000  <---- mask
    1001  0000
```

```
0000  86 99     LDAA #$99       ;load A with 1001 1001
0002  84 F0     ANDA #$F0       ;AND mask
0004  3E        WAI             ;stop
```

Fig. 19-17 Using the 6800/6808 AND instruction to mask bits.

```
          1001  1001  <---- data
OR        1111  0000  <---- mask
          1111  1001
```

```
0000  86 99     LDAA #$99        ;load A with 1001 1001
0002  8A F0     ORAA #$F0        ;OR mask
0004  3E        WAI              ;stop
```

Fig. 19-18 Using the 6800/6808 ORAA/ORAB instruction to mask bits.

```
          1001  1001  <---- data
XOR       1111  0000  <---- mask
          0110  1001
```

```
0000  86 99     LDAA #$99        ;load A with 1001 1001
0002  88 F0     EORA #$F0        ;EOR mask
0004  3E        WAI              ;stop
```

Fig. 19-19 Using the 6800/6808 EORA/EORB instruction to mask bits.

The BITA/BITB Instruction

The BIT instruction was not described in the New Concepts section. Refer to the BIT instruction in the Expanded Table of 6800/6808 Instructions Listed by Category.

The BIT instruction ANDs a memory location with one of the accumulators. However, the result is not stored anywhere. Neither the accumulator nor the memory location is changed.

If the result of the AND is zero, the zero flag is set. If the result is not zero, the zero flag is not set. If the result of the AND is a negative 2's-complement number, the negative flag is set. Regardless of the result, the overflow flag is cleared.

A program which illustrates the operation of the BIT instruction is shown in Fig. 19-20.

The BIT instruction is useful when the flags are used to control branching. You can alter the flags with a logical condition without actually changing the accumulator or memory location.

The COM/COMA/COMB Instruction

The complement instruction (COM/COMA/COMB) finds the 1's complement of each bit in the byte that's being complemented. That is, it inverts every bit in the byte. An example problem and a 6800/6808 program to solve the problem are shown in Fig. 19-21.

After running this program, you should find 55_{16} in A and the carry flag set.

Referring to the 6800/6808 instruction set, you will find that the COM instructions affect the negative and zero flags. In addition, they always clear the overflow flag and set the carry flag. In this example the negative flag is clear because the result (55_{16}) is not a negative number. Nor is it zero; therefore the zero flag is not set. The overflow flag is automatically cleared, and the carry flag automatically set.

The NEG/NEGA/NEGB Instruction

The NEG/NEGA/NEGB (negate) instructions are very similar to the COM/COMA/COMB instructions. The NEG instructions,

```
0000  86 FF     LDAA #$FF        ;load A with 1111 1111
0002  85 C0     BITA #$C0        ;AND A with 1100 0000
0004  3E        WAI              ;stop
```

```
After running the program:

A = FF          flags = 001000
```

Fig. 19-20 Using the 6800/6808 BITA instruction.

```
          1010  1010  <---- original number (AA16)
          0101  0101  <---- 1's complement of original number (5516)
```

```
0000  86 AA     LDAA #$AA        ;load A with 1010 1010
0002  43        COMA             ;invert all bits (0101 0101) (55h)
0003  3E        WAI              ;stop
```

Fig. 19-21 Using the 6800/6808 COM/COMA/COMB instructions.

```
  0101  1111  <-- original number (95 base 10)
  1010  0000  <-- 1's complement
+          1  <-- plus 1
  1010  0001  <-- 2's complement (-95 base 10)
```

```
0000  86 5F    LDAA#$5F        ;load A with 0101 1111
0002  40       NEGA            ;2's complement of A
0003  3E       WAI             ;stop
```

Fig. 19-22 Using the 6800/6808 NEG/NEGA/NEGB instructions.

however, find the 2's complement of a number instead of the 1's complement. Recall that the 2's complement is found by first finding the 1's complement and then adding 1.

Figure 19-22 shows an example problem and program using the negate instruction.

After running the program you will have $A1_{16}$ in the accumulator and the negative and carry flags set.

The negative flag is set because the 8th bit of A is set indicating a 2's-complement negative number.

Why the carry flag is set requires a little explanation. One way to look at a 2's-complement number is to view it as a 1's-complement number with 1 added to it. There is another point of view, however.

Remember how we described the creation of negative numbers as being like rotating an odometer backward? The original number used in this example is 0101 1111_2, which is 95_{10}. If we rotate our odometer backward from 00 by 95 places, we will arrive at the binary number 1010 0001. Rotating the odometer backward from 00 is the same as subtracting from 00.

Now think about subtracting a number from 00. Would a borrow from the carry bit be required? Yes, because any number is larger than 0 and a borrow would be required to subtract it from 00. To subtract 95 from 00 requires a borrow, which is why the carry flag is set.

If you think about it, the carry flag would have been set regardless of what number we would have used. When you use the NEG instruction, the only time the carry flag won't be set is if you negate the number 00, because subtracting 00 from 00 does not require a borrow.

19-8 8080/8085/Z80 FAMILY

The 8080/8085/Z80 has four of the instructions discussed in the New Concepts section, although one has a different name. These are the AND (ANA [AND]), OR (ORA [OR]), XOR (XRA [XOR]), and NOT (CMA [CPL]) instructions. (Z80 mnemonics are shown in brackets.) Let's look at each.

The ANA [AND] Instruction

The 8080/8085/Z80 ANA [AND] instruction works as described in the New Concepts section. If we use the example from Fig. 19-3 in the New Concepts section, we will find that the 8080/8085/Z80 does in fact AND bytes as discussed.

Figure 19-23 shows our original problem and solution plus an 8080/8085/Z80 program which solves the problem. If you will notice the condition of the accumulator and flags after running this program, you will find that the accumulator has a 90_{16} in it as we expected. The sign, auxiliary carry, and parity flags will be set.

If you check the 8085/Z80 instruction set, you will find that the AND instruction affects the sign, zero, and parity

```
    1001  1001  <-- data
AND 1111  0000  <-- mask
    1001  0000
```

```
8085 program

1800  3E 99     MVI A,99        ;load A with 1001 1001
1802  06 F0     MVI B,F0        ;load B with mask (1111 0000)
1804  A0        ANA B           ;AND A with mask
1805  76        HLT             ;stop

Z80 program

1800  3E 99     LD A,99         ;load A with 1001 1001
1802  06 F0     LD B,F0         ;load B with mask (1111 0000)
1804  A0        AND B           ;AND A with mask
1805  76        HALT            ;stop
```

Fig. 19-23 Using the 8080/8085/Z80 ANA [AND] instruction to mask bits.

```
      1001  1001  <------ data
OR    1111  0000  <------ mask
      1111  1001
```

```
8085 program

1800  3E 99     MVI A,99        ;load A with number (1001 1001)
1802  06 F0     MVI B,F0        ;load B with mask (1111 0000)
1804  B0        ORA B           ;OR number and mask
1805  76        HLT             ;stop

Z80 program

1800  3E 99     LD A,99         ;load A with number (1001 1001)
1802  06 F0     LD B,F0         ;load B with mask (1111 0000)
1804  B0        OR B            ;OR number and mask
1805  76        HALT            ;stop
```

Fig. 19-24 Using the 8085/Z80 OR instruction to mask bits.

flags. The AND instruction *always* sets the auxiliary carry [half-carry] flag and always clears the carry flag. (*Note:* If you are using an 8080 microprocessor, the auxiliary flag works a little differently than it does in the 8085 and Z80. Check the Expanded Table.)

The sign flag is set because this is a negative number. The zero flag is clear because the result was not zero. The auxiliary flag is set because it is always set by this instruction. The parity flag is set because there are an even number of 1s. And the carry flag is clear because that flag is always cleared by the AND instruction.

The ORA [OR] Instruction

The 8085/Z80 OR instruction also works as described in the New Concepts section. We'll use the example from Fig. 19-6 in the New Concepts section.

Figure 19-24 shows our original problem and solution plus an 8085/Z80 program which solves the problem. After entering and running the program, you will find that the accumulator has a value of $F9_{16}$ and that the sign and parity flags have been set.

We expected the accumulator to have $F9_{16}$ after ORing. The OR instruction set the sign flag because $F9_{16}$ is a 2's-complement negative number. The parity flag is set because there are an even number of 1s in $F9_{16}$ ($1111\ 1001_2$). The zero flag is clear because the result ($F9_{16}$) is not zero. All other flags are automatically cleared by the OR instruction.

The XRA [XOR] Instruction

Let's look at the 8085/Z80 XOR instruction. If we use the example from Fig. 19-9 in the New Concepts section, we'll find that the 8085/Z80 does XOR bytes as discussed.

Figure 19-25 shows our original problem and solution plus an 8085/Z80 program which solves the problem. After entering and running the program, you will find that the accumulator contains 69_{16} and that only the parity flag is set. Examine the figure and the Expanded Table to find why this is so.

```
      1001  1001  <------ data
XOR   1111  0000  <------ mask
      0110  1001
```

```
8085 program

1800  3E 99     MVI A,99        ;load A with number (1001 1001)
1802  06 F0     MVI B,F0        ;load B with mask (1111 0000)
1804  A8        XRA B           ;XOR number with mask
1805  76        HLT             ;stop

Z80 program

1800  3E 99     LD A,99         ;load A with number (1001 1001)
1802  06 F0     LD B,F0         ;load B with mask (1111 0000)
1804  A8        XOR B           ;XOR number with mask
1805  76        HALT            ;stop
```

Fig. 19-25 Using the 8085/Z80 XOR instruction to mask bits.

```
                    NOT  1010 1010  is  0101 0101

8085 program

1800  3E AA     MVI A,AA        ;load A with 1010 1010
1802  2F        CMA             ;invert all bits (0101 0101) (55h)
1803  76        HLT             ;stop

Z80 program

1800  3E AA     LD A,AA         ;load A with 1010 1010
1802  2F        CPL             ;invert all bits (0101 0101) (55h)
1803  76        HALT            ;stop
```

Fig. 19-26 Using the 8085/Z80 complement instruction.

The CMA [CPL] Instruction

The complement instruction (CMA [CPL]) finds the 1's complement of each bit in the byte that's being complemented. That is, it inverts every bit in the byte. An example problem and an 8085/Z80 program to solve the problem are shown in Fig. 19-26.

After running this program, you should find the value 55_{16} in A. If you are using an 8085, you will find that none of the flags has been affected or changed by the CMA instruction. If you are using a Z80, you will find that the half-carry and parity flags have been set. The Z80 always sets these two flags after the CPL instruction.

19-9 8086/8088 FAMILY

The 8086/8088 has all the instructions discussed in the New Concepts section. These include the AND, OR, XOR, NOT, and NEG instructions. Let's look at each.

The AND Instruction

The 8086/8088 AND instruction works as described in the New Concepts section. If we use the example from Fig. 19-3 in the New Concepts section, we find that the 8086/8088 does in fact AND bytes as discussed.

Figure 19-27 shows our original problem and solution plus an 8086/8088 program which solves the problem. Notice the condition of the accumulator and flags before and after running this program.

After masking 99_{16} with FO_{16}, 90_{16} is exactly what we expected. If you check the 8086/8088 instruction set, you will find that the AND instruction affects the sign, zero, and parity flags. The overflow and carry flags are always cleared (NV, NC), and the auxiliary flag is undefined. In this case the sign flag is set (NG) because the 8th bit of the accumulator is 1, indicating a 2's-complement negative number.

The OR Instruction

The 8086/8088 OR instruction also works as described earlier in the New Concepts section. We'll use the example from Fig. 19-6 in the New Concepts section.

Figure 19-28 shows our original problem and solution plus an 8086/8088 program which solves the problem. After entering and running the program, you will find that AL has a value of $F9_{16}$ as we expected.

```
          1001  1001  <------ data
AND       1111  0000  <------ mask
          1001  0000

AX=0000  BX=0000  CX=0000  DX=0000  SP=F75E  BP=0000  SI=0000  DI=0000
DS=908A  ES=908A  SS=908A  CS=908A  IP=0100   NV UP EI PL NZ NA PO NC
908A:0100 B099          MOV     AL,99
-
-a 100
908A:0100 MOV AL,99       ;load A with 1001 1001
908A:0102 AND AL,F0       ;AND A with mask
908A:0104 INT 20          ;return control to DEBUG
908A:0106
-
-g 0104

AX=0090  BX=0000  CX=0000  DX=0000  SP=F75E  BP=0000  SI=0000  DI=0000
DS=908A  ES=908A  SS=908A  CS=908A  IP=0104   NV UP EI NG NZ NA PE NC
908A:0104 CD20          INT     20
-
```

Fig. 19-27 Using the 8086/8088 AND instruction to mask bits.

```
          1001  1001  <-- data
OR        1111  0000  <-- mask
          1111  1001
```

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=907C  ES=907C  SS=907C  CS=907C  IP=0100   NV UP EI PL NZ NA PO NC
907C:0100 B099          MOV     AL,99
-
-a
907C:0100 MOV AL,99         ;load A with number (1001 1001)
907C:0102 OR  AL,F0         ;OR number and mask
907C:0104 INT 20            ;stop
907C:0106
-
-g 104

AX=00F9  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=907C  ES=907C  SS=907C  CS=907C  IP=0104   NV UP EI NG NZ NA PE NC
907C:0104 CD20          INT     20
-
```

Fig. 19-28 Using the 8086/8088 OR instruction to mask bits.

The OR instruction also affects certain flags. The sign flag is set (NG) because this is a 2's-complement negative number. The overflow flag is cleared (NV) because the OR instruction always clears it. The carry flag is also cleared for the same reason (NC). We have even parity (PE), and the result is not zero (NZ).

The XOR Instruction

Let's look at the 8086/8088 XOR instruction using the example from Fig. 19-9 in the New Concepts section.

Figure 19-29 shows our original problem and solution plus an 8086/8088 program which solves the problem. After entering and running the program, you will find that AL has a value of 69_{16}. This is what we expected.

The XOR instruction affects the flags in the same way as the OR and AND instructions. Examine the flags that are affected by this instruction to see whether they responded as you expected.

The NOT Instruction

The invert instruction (NOT) finds the 1's complement of each bit in the byte that's being complemented. That is, it inverts every bit in the byte. An example problem and an 8086/8088 program to solve the problem are shown in Fig. 19-30.

After running this program, you should find the value 55_{16} in AL. And since this instruction does not affect *any*

```
          1001  1001  <-- data
XOR       1111  0000  <-- mask
          0110  1001
```

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=909F  ES=909F  SS=909F  CS=909F  IP=0100   NV UP EI PL NZ NA PO NC
909F:0100 7420          JZ      0122
-
-a
909F:0100 MOV AL,99         ;load A with number (1001 1001)
909F:0102 XOR AL,F0         ;XOR number with mask (1111 0000)
909F:0104 INT 20            ;return control to DEBUG
909F:0106
-
-g 104

AX=0069  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=909F  ES=909F  SS=909F  CS=909F  IP=0104   NV UP EI PL NZ NA PE NC
909F:0104 CD20          INT     20
-
```

Fig. 19-29 Using the 8086/8088 XOR instruction to mask bits.

```
                    NOT  1010 1010  is  0101 0101

-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=90AE  ES=90AE  SS=90AE  CS=90AE  IP=0100   NV UP EI PL NZ NA PO NC
90AE:0100 7420          JZ      0122
-
-a
90AE:0100 MOV AL,AA         ;load A with number (1010 1010)
90AE:0102 NOT AL            ;invert all bits of number (0101 0101) (55h)
90AE:0104 INT 20            ;return control to DEBUG
90AE:0106
-
-g 104

AX=0055  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=90AE  ES=90AE  SS=90AE  CS=90AE  IP=0104   NV UP EI PL NZ NA PO NC
90AE:0104 CD20          INT     20
-
```

Fig. 19-30 Using the 8086/8088 NOT instruction.

flags, you should find that every flag is exactly as it was before the instruction was executed.

The NEG Instruction

The NEG (negate) instruction is very similar to the NOT instruction. The NEG instruction, however, finds the 2's complement instead of the 1's complement. Recall that the 2's complement is found by first finding the 1's complement and then adding 1.

Figure 19-31 shows an example problem and program using the negate instruction. After running the program, you will have $A1_{16}$ in the accumulator.

Notice also that the sign flag is set (NG) as well as the carry flag (CY).

The negative flag is set because the 8th bit of AL is set indicating a 2's-complement negative number.

Why the carry flag is set requires a little explanation. One way to look at a 2's-complement number is to view it as a 1's-complement number with 1 added to it. There is another point of view, however.

Remember how we described the creation of negative numbers as being like rotating an odometer backward? The original number we used in this example is $0101\ 1111_2$, which is 95_{10}. If we rotate our odometer backward from 00 by 95 places, we will arrive at the binary number 1010 0001. Rotating the odometer backward from 00 is the same as subtracting from 00.

Now think about subtracting a number from 00. Would a borrow from the carry bit be required? Yes, because any

```
   0101  1111  <-- original number (95 base 10)
   1010  0000  <-- 1's complement
+                1  <-- plus 1
-------------------
   1010  0001  <-- 2's complement (-95 base 10)
```

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=90EA  ES=90EA  SS=90EA  CS=90EA  IP=0100   NV UP EI PL NZ NA PO NC
90EA:0100 7420          JZ      0122
-
-a
90EA:0100 MOV AL,5F         ;load A with number (0101 1111)
90EA:0102 NEG AL            ;find 2's complement of number in AL
90EA:0104 INT 20            ;return control to DEBUG
90EA:0106
-
-g 104

AX=00A1  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=90EA  ES=90EA  SS=90EA  CS=90EA  IP=0104   NV UP EI NG NZ NA PO CY
90EA:0104 CD20          INT     20
-
```

Fig. 19-31 Using the 8086/8088 NEG instruction.

number is larger than 00 and a borrow would be required to subtract it from 00. To subtract 95 from 00 required a borrow, which is why the carry flag was set.

If you think about it, the carry flag would have been set regardless of what number we used. When you use the NEG instruction, the only time the carry flag won't be set is when you negate the number 00 itself, because subtracting 00 from 00 does not require a borrow.

SELF-TESTING REVIEW

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

1. Name the four basic logical instructions.
   _____ _____
   _____ _____
2. *(AND, OR, XOR, and NOT)* When we AND 2 bits, we are saying that the output bit will be 1 only if both inputs bits are _____.
3. *(1)* A mask allows us to change some bits in a byte while allowing others to pass through _____.
4. *(unchanged)* When we _____ two bits together, we are saying that the output bit will be a 1 if either or both of the input bits are 1.
5. *(OR)* When ORing bits, the only way to get a _____ out is to have both inputs be _____.
6. *(0, 0)* When using the XOR instruction, if both input bits are the same, the output bit will be a ___ (0, 1).
7. *(0)* When XORing bits, the only way to get a 1 out is for (both, either) _____ of the input bits to be a 1.
8. *(either)* When we NOT or invert bits, we are saying that the output bit is the _____ (same as, opposite of) the input bit.
9. *(opposite of)* To NEGate a number is to find the 2's complement of the number. This involves finding the _____ _____ and then adding _____. *(1's complement, 1)*

PROBLEMS

General

19-1.
```
  1011 1100
AND 0110 1010
```

19-2. Devise a mask which, used with the AND instruction, would allow all bits to pass through unaltered except the most significant. The most significant should be cleared.

19-3.
```
  0110 1110
OR  0011 0101
```

19-4. Devise a mask which, used with the OR instruction, would allow all bits to pass through unaltered except the most significant. The most significant should be set.

19-5.
```
  0101 0101
XOR 0011 1111
```

19-6. Devise a mask which, used with the XOR instruction, would invert all bits except the 2 most significant. The 2 most significant should pass through unaltered.

19-7. Invert the binary number 0111 1011.

19-8. Negate the number 0110 1110 (8-bit answer).

Specific Microprocessor Families

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

19-9. Write and run a program which will place the binary number 1100 1001 in the accumulator and then AND it with the binary number 1011 1101.

19-10. Write and run a program which will place the number CC_{16} in the accumulator and then use the OR instruction to set every bit in the lower nibble of the accumulator while allowing every bit in the upper nibble to remain unchanged.

Advanced Problems

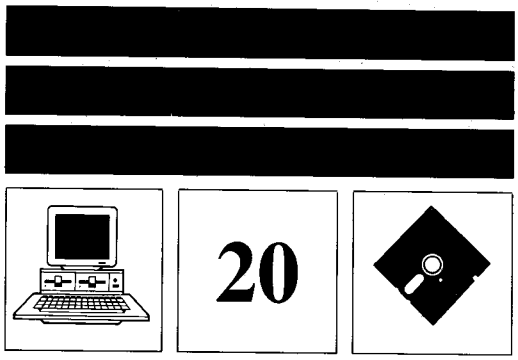
Solve the following problems using the microprocessor of your choice.

19-11. Write and run a program which will:
   a. place 45_{16} in the accumulator.
   b. add $2F_{16}$ to the number in the accumulator.
   c. use a mask to invert every bit in the lower nibble of the sum yet not alter the upper nibble.
   d. subtract $0001\ 1100_2$ from the last result.
   e. create another mask (using the AND instruction) which will allow all bits of the last result to remain unchanged except the least significant 3 bits which should be cleared.

19-12. ASCII values for the digits 0 through 9 are shown below.

0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

It may sometimes be desirable to change an ASCII number into its binary equivalent. For this problem, write and run a program which uses a mask to change the ASCII value for 5 into its binary equivalent.



SHIFT AND ROTATE INSTRUCTIONS

In this chapter we'll study two relatively straightforward concepts—*shifting* and *rotating*. Shifts and rotates can be used for parallel-to-serial data conversion, serial-to-parallel data conversion, multiplication, division, and other tasks.

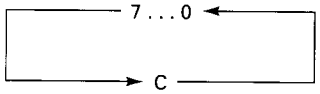


Fig. 20-1 Typical rotate left instruction.

New Concepts

The concepts of rotating and shifting are quite simple. Let's look at each in its "generic" form; then, as usual, we'll study each microprocessor family. The microprocessors' instructions which perform each of these functions differ only slightly.

20-1 ROTATING

Rotating bits is exactly what it sounds like—moving bits in a circle. Let's look at a typical rotate instruction to start our discussion. Figure 20-1 shows a typical *rotate left* instruction.

Figure 20-2 illustrates each step involved when a bit is rotated eight times. Figure 20-2 first shows an 8-bit accumulator and carry flag. The accumulator is loaded with the value 01_{16}, and the carry flag is cleared. Next, a sequence of eight rotate lefts are performed. Notice that the 1 just keeps moving 1 bit position each time.

Microprocessors can rotate toward the right or left. Some also have other forms of rotation in which the carry flag is involved in a slightly different way. We'll look at those in the Specific Microprocessor Families section.

20-2 SHIFTING

Shifting, like rotating, is exactly what it sounds like. And, like rotating, shifting can be toward the left or right. The 8080/8085 is the only microprocessor family being studied in this text which does not have shift instructions. The 8080/8085 has only rotate instructions.

Let's look first at the concept of shifting toward the left. Figure 20-3 illustrates what is known as a *logical shift left* or *arithmetic shift left*. Bits are shifted one at a time toward the left, with the bit in the 8th position (bit 7) being shifted into the carry flag.

Two things should be noticed which make this instruction different from the rotate instruction. First, the contents of the carry flag do not "wrap around" to bit 0; its contents are simply lost. Second, 0s are automatically shifted into bit 0 (least significant bit).

Look at Fig. 20-4 for an example of this type of shifting. We have loaded the value 99_{16} into the accumulator and have cleared the carry flag. Next we execute eight consecutive shifts. Notice that

1. 0s keep coming in from the left.
2. The bits in the accumulator keep shifting 1 bit to the left.
3. The bits shift from the most significant bit of the accumulator into the carry flag.
4. Bits shifting out of the carry flag are lost.

Shifts to the right are possible also. Figure 20-5 shows a typical *logical* shift to the right. This is basically the opposite of the shift left.

Figure 20-6 shows a typical *arithmetic* shift to the right. The *arithmetic shift right* instruction duplicates whatever was in the most significant bit and moves copies of it to the right with each shift.

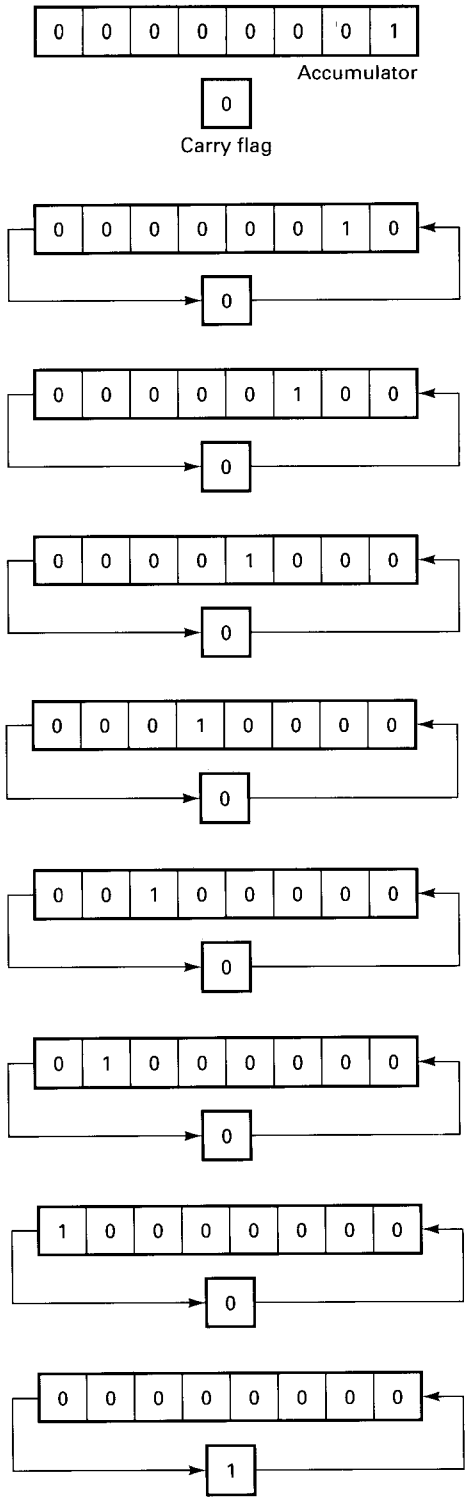


Fig. 20-2 Rotating left eight times.

20-3 AN EXAMPLE

Let's look at an example which uses the rotate instruction. It is often useful to be able to move a nibble of data from one part of a register to the other.

For example, let's say we wanted to clear every bit in the upper nibble of the accumulator and then move every bit of the lower nibble into the upper nibble. There are no instructions for moving a nibble from one place to another. The rotate instruction can help accomplish this, though. Figure 20-7 shows our problem.

First, we'll use a mask to clear out the upper bit. This is shown in Fig. 20-8.

Next we'll clear the carry bit (since this bit will be rotated into the least significant bit of the lower nibble). Then we'll rotate toward the left four times. This is shown in Fig. 20-9.

If you compare the final value in Fig. 20-9 with our initial value in Fig. 20-9, you'll see that we have moved the lower nibble into the upper nibble, which is what we wanted to do.

Fig. 20-3 Typical arithmetic shift left or logical shift left.

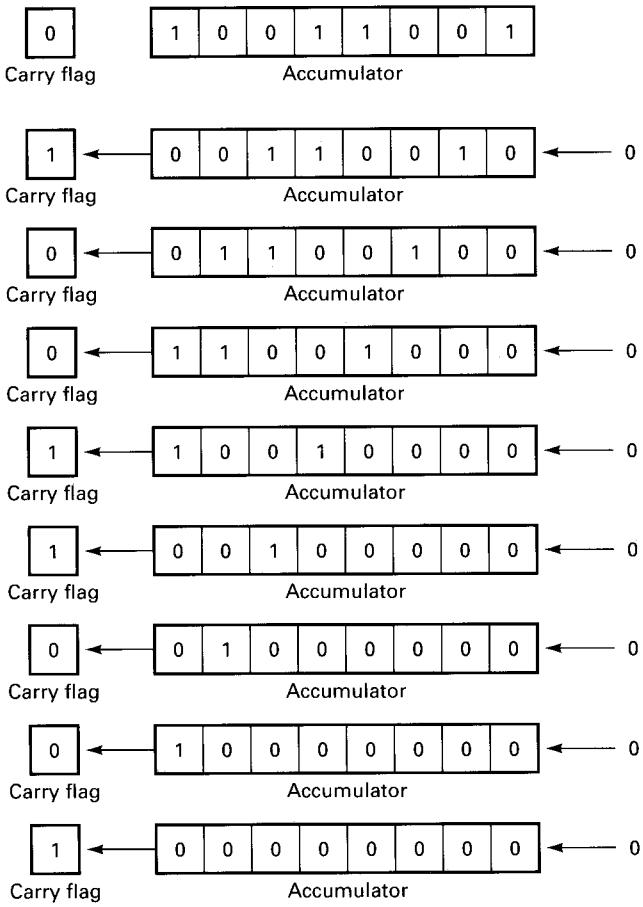


Fig. 20-4 Eight typical arithmetic shift left or logical shift left instructions.

Fig. 20-5 Typical logical shift right.

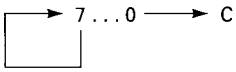


Fig. 20-6 Typical arithmetic shift right.

Upper nibble	Lower nibble
1 1 0 0	1 1 0 1

Fig. 20-7 Situation in which we want to clear the upper nibble and then move every bit of the lower nibble into the upper nibble.

```
    1100  1101
AND 0000  1111
    ----------
    0000  1101
```

Fig. 20-8 Using the AND instruction to mask off the upper nibble.

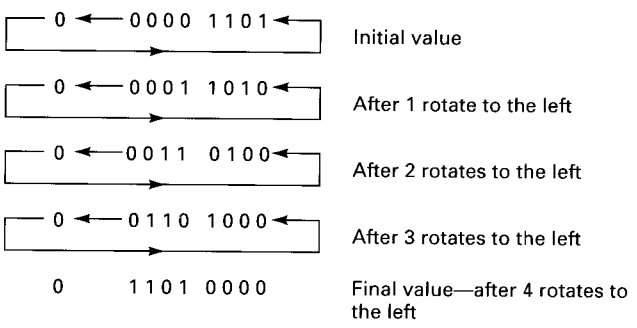


Fig. 20-9 Using the rotate through carry instruction to move the lower nibble into the upper nibble.

Specific Microprocessor Families

Let's study the shift and rotate instructions for each of our microprocessor families.

20-4 6502 FAMILY

The 6502 has two rotate instructions and two shift instructions. Let's look at them.

The ROL and ROR Instructions

The 6502 ROL (**RO**tate **L**eft) instruction works as described in the New Concepts section of this chapter and as shown in Fig. 20-10. Figure 20-10 is taken from the Rotate and Shift Instructions section of the Expanded Table of 6502 Instructions Listed by Category.

In Fig. 20-10 the "7 . . . 0" represents bits 0 through 7 of a byte. Here the "byte" is the value in the accumulator. The "C" represents the carry bit of the status register.

The ROL instruction causes each bit to move to the left one place. Bit 7 moves into the carry bit (flag), and the carry bit moves into bit 0.

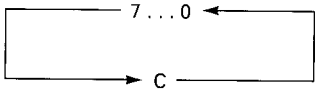


Fig. 20-10 6502 **RO**tate **L**eft instruction.

```
0001  0340          .org $0340
0002  0340          ;
0003  0340 18       CLC
0004  0341 A9 01    LDA #$01
0005  0343 2A       ROL A
0006  0344 2A       ROL A
0007  0345 2A       ROL A
0008  0346 2A       ROL A
0009  0347 2A       ROL A
0010  0348 2A       ROL A
0011  0349 2A       ROL A
0012  034A 2A       ROL A
0013  034B 00       BRK
0014  034C          ;
0015  034C          .end
```

Fig. 20-11 6502 program which rotates left eight times.

The ROR (**RO**tate **R**ight) instruction uses the same concept as the ROL instruction and affects flags in the same way. It simply rotates the bits in the opposite direction.

Figure 20-11 shows a program which clears the carry flag and rotates the accumulator toward the left eight times. If you have a monitor which can single-step ("walk") through the program, cause it to do so, and check the accumulator and carry flag after each step.

If you cannot single-step, then use a break (BRK) instruction after each ROL instruction so that you can observe the movement of the bits in the accumulator. After each BRK you will have to make your trainer or computer begin program execution again at the next ROL instruction to see the shifting action continue.

The ASL and LSR Instructions

The 6502 shift instructions also work as described in the New Concepts section of this chapter. The **A**rithmetic **S**hift **L**eft instruction is shown in Fig. 20-12.

Fig. 20-12 6502 arithmetic shift left instruction.

The **L**ogical **S**hift **R**ight instruction is shown in Fig. 20-13. Both are quite simple.

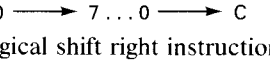


Fig. 20-13 6502 logical shift right instruction.

An Example

Let's look at the same example which was used in the New Concepts section. Remember, our objective was to clear the upper nibble and then to move the lower nibble of the accumulator into the upper nibble of the accumulator. Figure 20-14 shows our original problem.

Upper nibble	Lower nibble
1 1 0 0	1 1 0 1

Fig. 20-14 Situation in which we want to clear the upper nibble and then move every bit of the lower nibble into the upper nibble.

```
0001  0340          .org $0340
0002  0340          ;
0003  0340 29 0F    AND #$0F        ;mask off upper nibble
0004  0342 18       CLC             ;clear the carry flag
0005  0343 2A       ROL A           ;rotate left four times
0006  0344 2A       ROL A
0007  0345 2A       ROL A
0008  0346 2A       ROL A
0009  0347 00       BRK
0010  0348          ;
0011  0348          .end
```

Fig. 20-15 6502 program which clears the upper nibble of the accumulator and then moves the lower nibble into the upper nibble.

A 6502 program which can solve this problem is shown in Fig. 20-15. Manually place the initial value of CD_{16} in the accumulator before running the program. After the program is run, you should find the value $D0_{16}$ in the accumulator.

20-5 6800/6808 FAMILY

The 6800/6808 has two rotate instructions and three shift instructions.

The ROL/ROLA/ROLB and ROR/RORA/RORB Instructions

The 6800/6808 ROL/ROLA/ROLB instructions work as described in the New Concepts section and as shown in Fig. 20-16. Figure 20-16 is taken from the Rotate and Shift Instructions section of the Expanded Table of 6800/6808 Instructions Listed by Category.

In Fig. 20-16 the "7 . . . 0" represents bits 0 through 7 of a byte. In this case the "byte" is the value in a memory location, accumulator A, or accumulator B. The "C" represents the carry bit of the status register.

The ROL/ROLA/ROLB instructions cause each bit to move to the left one place. Bit 7 moves into the carry bit (flag), and the carry bit moves into bit 0.

The ROR/RORA/RORB (**RO**tate **R**ight) instructions use the same concept as the ROL/ROLA/ROLB instructions and affect flags in the same way. They simply rotate the bits in the opposite direction.

Figure 20-17 shows a program which clears the carry flag and rotates accumulator A toward the left eight times.

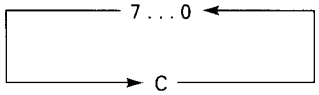


Fig. 20-16 6800/6808 ROL/ROLA/ROLB instructions.

```
0001  0000          .org $0000
0002  0000          ;
0003  0000 0C       CLC
0004  0001 86 01    LDAA #$01
0005  0003 49       ROLA
0006  0004 49       ROLA
0007  0005 49       ROLA
0008  0006 49       ROLA
0009  0007 49       ROLA
0010  0008 49       ROLA
0011  0009 49       ROLA
0012  000A 49       ROLA
0013  000B 3E       WAI
0014  000C          ;
0015  000C          .end
```

Fig. 20-17 6800/6808 program which rotates left eight times.

If you have a monitor which can single-step ("walk") through the program, cause it to do so and check the accumulator and carry flag after each step.

The ASL/ASLA/ASLB, ASR/ASRA/ASRB, and LSR/LSRA/LSRB Instructions

The 6800/6808 shift instructions also work as described in the New Concepts section of this chapter. The **A**rithmetic **S**hift **L**eft instruction is shown in Fig. 20-18.

The **A**rithmetric **S**hift **R**ight instruction is shown in Fig. 20-19. The **L**ogical **S**hift **R**ight instruction is shown in Fig. 20-20. All are quite simple.

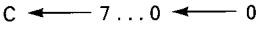


Fig. 20-18 6800/6808 arithmetic shift left instruction.

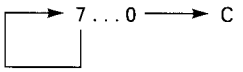


Fig. 20-19 6800/6808 arithmetic shift right instruction.

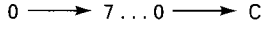


Fig. 20-20 6800/6808 logical shift right instruction.

An Example

Let's look at the same example which was used in the New Concepts section. Remember, our objective was to clear the upper nibble and then to move the lower nibble of the accumulator into the upper nibble of the accumulator. Figure 20-21 shows our original problem.

Upper nibble	Lower nibble
1 1 0 0	1 1 0 1

Fig. 20-21 Situation in which we want to clear the upper nibble and then move every bit of the lower nibble into the upper nibble.

A 6800/6808 program which can solve this problem is shown in Fig. 20-22. Manually place the initial value of CD_{16} in the accumulator before running the program. After the program is run, you should find the value $D0_{16}$ in the accumulator.

20-6 8080/8085/Z80 FAMILY

The 8080 and 8085 have four rotate instructions and no shift instructions. We will place the Z80 form of the instructions in square brackets. (The Z80 does have several multibyte shift instructions which we will not study at this time because the 8080 and 8085 do not share these instructions.)

The RAL [RLA] and RAR [RRA] Instructions

The 8080/8085/Z80 RAL [RLA] (**R**otate **A** **L**eft [**R**otate **L**eft **A**] instructions work as described in the New Concepts section and as shown in Fig. 20-23. Figure 20-23 is taken from the Rotate and Shift Instructions section of the Expanded Table of 8080/8085/Z80 Instructions Listed by Category.

In Fig. 20-23 the "7 . . . 0" represents bits 0 through 7 of a byte. In this case the "byte" is the value in the accumulator. The "C" represents the carry bit of the status register.

```
0001  0000          .org $0000
0002  0000          ;
0003  0000 84 0F    ANDA #$0F       ;mask off upper nibble
0004  0002 0C       CLC             ;clear the carry flag
0005  0003 49       ROLA            ;rotate left four times
0006  0004 49       ROLA
0007  0005 49       ROLA
0008  0006 49       ROLA
0009  0007 3E       WAI
0010  0008          ;
0011  0008          .end
```

Fig. 20-22 6800/6808 program which moves the lower nibble of the accumulator into the upper nibble.

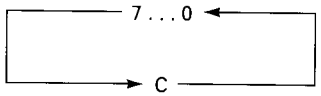


Fig. 20-23 The 8080/8085/Z80 RAL [RLA] instruction.

The RAL [RLA] instruction causes each bit to move to the left one place. Bit 7 moves into the carry bit (flag), and the carry bit moves into bit 0.

The RAR [RRA] instruction uses the same concept as the RAL [RLA] instruction and affects flags in the same way. It simply rotates the bits in the opposite direction.

Figure 20-24 shows a program which clears the carry flag and rotates the accumulator toward the left eight times. If you have a monitor which can single-step ("walk") through the program, cause it to do so and check the accumulator and carry flag after each step.

The RLC [RLCA] and RRC [RRCA] Instructions

The RLC [RLCA] (**R**otate **L**eft with **C**arry [**R**otate **L**eft with **C**arry **A**]) and RRC [RRCA] (**R**otate **R**ight with **C**arry [**R**otate **R**ight with **C**arry **A**]) instructions work just a little differently from the other rotate instructions we have discussed. The RLC [RLCA] instruction is shown in Fig. 20-25.

The RRC [RRCA] instruction is shown in Fig. 20-26.

In the case of the RLC [RLCA] instruction, all bits in the accumulator move toward the left. The bit rotating out of bit 7 goes into the carry flag *and* around into bit 0 of the accumulator.

In the case of the RRC [RRCA] instruction, all bits in the accumulator move toward the right. The bit rotating out of bit 0 goes into the carry flag *and* around into bit 7 of the accumulator.

An Example

Let's look at the same example which was used in the New Concepts section. Remember, our objective was to clear the upper nibble and then to move the lower nibble of the accumulator into the upper nibble of the accumulator. Figure 20-27 shows our original problem.

8080/8085 program

```
0001  1800          .org 1800h
0002  1800          ;
0003  1800 3E 01    MVI A,01H
0004  1802 17       RAL
0005  1803 17       RAL
0006  1804 17       RAL
0007  1805 17       RAL
0008  1806 17       RAL
0009  1807 17       RAL
0010  1808 17       RAL
0011  1809 17       RAL
0012  180A 76       HLT
0013  180B          ;
0014  180B          .end
```

Z80 program

```
0001  1800          .org 1800h
0002  1800          ;
0003  1800 3E 01    LD A,01H
0004  1802 17       RLA
0005  1803 17       RLA
0006  1804 17       RLA
0007  1805 17       RLA
0008  1806 17       RLA
0009  1807 17       RLA
0010  1808 17       RLA
0011  1809 17       RLA
0012  180A 76       HALT
0013  180B          ;
0014  180B          .end
```

Fig. 20-24 8080/8085 and Z80 programs which rotate left eight times.

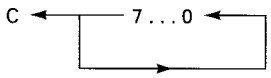


Fig. 20-25 8080/8085/Z80 RLC [RLCA] instruction.

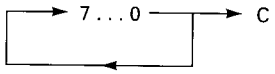


Fig. 20-26 8080/8085/Z80 RRC [RRCA] instruction.

Upper nibble	Lower nibble
1 1 0 0	1 1 0 1

Fig. 20-27 Situation in which we want to clear the upper nibble and then move every bit of the lower nibble into the upper nibble.

An 8080/8085/Z80 program which can solve this problem is shown in Fig. 20-28. Manually place the initial value of CD_{16} in the accumulator before running the program. After the program is run, you should find the value $D0_{16}$ in the accumulator.

20-7 8086/8088 FAMILY

The 8086/8088 has four rotate instructions and three shift instructions. They are discussed starting on the next page.

8080/8085 program

```
0001  1800          .org 1800h
0002  1800          ;
0003  1800 E6 0F    ANI 0FH          ;mask off upper nibble
0004  1802 37       STC              ;set the carry flag then
0005  1803 3F       CMC              ;  complement it
0006  1804 17       RAL              ;rotate left four times
0007  1805 17       RAL
0008  1806 17       RAL
0009  1807 17       RAL
0010  1808 76       HLT
0011  1809          ;
0012  1809          .end
```

Z80 program

```
0001  1800          .org 1800h
0002  1800          ;
0003  1800 E6 0F    AND 0FH          ;mask off upper nibble
0004  1802 37       SCF              ;set the carry flag then
0005  1803 3F       CCF              ;  complement it
0006  1804 17       RLA              ;rotate left four times
0007  1805 17       RLA
0008  1806 17       RLA
0009  1807 17       RLA
0010  1808 76       HALT
0011  1809          ;
0012  1809          .end
```

Fig. 20-28 8080/8085 and Z80 programs which clear the upper nibble of the accumulator and then move the lower nibble into the upper nibble.

The RCL and RCR Instructions

The RCL and RCR instructions work as described in the New Concepts section of this chapter and as shown in Fig. 20-29. Figure 20-29 is taken from the Rotate and Shift Instructions section of the Expanded Table of 8086/8088 Instructions Listed by Category.

In Fig. 20-29 the "MSB . . . LSB" represents bits 0 through 7 of a byte or bits 0 through 15 of a word. The "C" represents the carry bit of the status register.

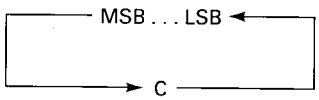


Fig. 20-29 The 8086/8088 RCL and RCR instructions.

The RCL instruction causes each bit to move to the left one place. The MSB moves into the carry bit (flag), and the carry bit moves into the LSB.

The RCR instruction uses the same concept as the RCL instruction and affects flags in the same way. It simply rotates the bits in the opposite direction.

Figure 20-30 shows a program which clears the carry flag and rotates AL toward the left eight times. We then single-step through the program. Follow each step and pay particular attention to AL and the carry flag.

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  BS=775B  IP=0100   NV UP EI PL NZ NA PO NC
775B:0100 7420          JZ      0122
-
-a
775B:0100 CLC
775B:0101 MOV AL,01
775B:0103 RCL AL,1
775B:0105 RCL AL,1
775B:0107 RCL AL,1
775B:0109 RCL AL,1
775B:010B RCL AL,1
775B:010D RCL AL,1
775B:010F RCL AL,1
775B:0111 RCL AL,1
775B:0113 INT 20
775B:0115
-
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0100   NV UP EI PL NZ NA PO NC
775B:0100 F8            CLC
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0101   NV UP EI PL NZ NA PO NC
775B:0101 B001          MOV     AL,01
-t

AX=0001  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0103   NV UP EI PL NZ NA PO NC
775B:0103 D0D0          RCL     AL,1
-t

AX=0002  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0105   NV UP EI PL NZ NA PO NC
775B:0105 D0D0          RCL     AL,1
-t

AX=0004  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0107   NV UP EI PL NZ NA PO NC
775B:0107 D0D0          RCL     AL,1
-t

AX=0008  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0109   NV UP EI PL NZ NA PO NC
775B:0109 D0D0          RCL     AL,1
-t
```

Fig. 20-30 8086/8088 RCL instruction.

```
AX=0010  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=010B   NV UP EI PL NZ NA PO NC
775B:010B D0D0          RCL     AL,1
-t

AX=0020  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=010D   NV UP EI PL NZ NA PO NC
775B:010D D0D0          RCL     AL,1
-t

AX=0040  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=010F   NV UP EI PL NZ NA PO NC
775B:010F D0D0          RCL     AL,1
-t

AX=0080  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0111   OV UP EI PL NZ NA PO NC
775B:0111 D0D0          RCL     AL,1
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775B  ES=775B  SS=775B  CS=775B  IP=0113   OV UP EI PL NZ NA PO CY
775B:0113 CD20          INT     20
-t
```

Fig. 20-30 (cont.)

The ROL and ROR Instructions

The ROL (**RO**tate **L**eft) and ROR (**RO**tate **R**ight) instructions work just a little differently from the other rotate instructions we have discussed. The ROL instruction is shown in Fig. 20-31.

The ROR instruction is shown in Fig. 20-32.

The drawings shown here are slightly different from those shown in the instruction-set description, but if you'll look closely, you'll see that they are really the same.

In the case of the ROL instruction, all bits move toward the left. The bit rotating out of the MSB goes into the carry flag *and* around into the LSB.

In the case of the ROR instruction, all bits move toward the right. The bit rotating out of the LSB goes into the carry flag *and* around into the MSB.

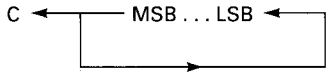


Fig. 20-31 8086/8088 ROL instruction.

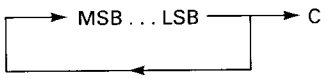


Fig. 20-32 8086/8088 ROR instruction.

The SAL/SHL, SAR, and SHR Instructions

The 8086/8088 shift instructions work as described in the New Concepts Section of this chapter. The **S**hift **A**rithmetic **L**eft/**SH**ift logical **L**eft instruction is shown in Fig. 20-33.

The **S**hift **A**rithmetic **R**ight instruction is shown in Fig. 20-34. The **SH**ift logical **R**ight instruction is shown in Fig. 20-35. All are quite simple.

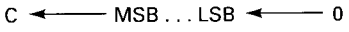


Fig. 20-33 8086/8088 SAL/SHL instruction.

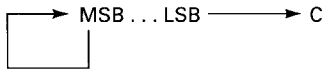


Fig. 20-34 8086/8088 shift arithmetic right instruction.

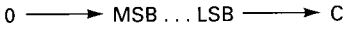


Fig. 20-35 8086/8088 shift logical right instruction.

An Example

Let's look at the same example which was used in the New Concepts section. Remember, our objective was to clear the upper nibble and then to move the lower nibble of AL into the upper nibble of AL. Figure 20-36 shows our original problem.

An 8086/8088 program which can solve this problem is shown in Fig. 20-37. Manually place the initial value of CD_{16} in AL before running the program. After the program is run, you should find the value $D0_{16}$ in AL.

Upper nibble	Lower nibble
1 1 0 0	1 1 0 1

Fig. 20-36 Situation in which we want to clear the upper nibble and then move every bit of the lower nibble into the upper nibble.

```
C>DEBUG
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=77B0  ES=77B0  SS=77B0  BS=77B0  IP=0100   NV UP EI PL NZ NA PO NC
77B0:0100 240F          AND     AL,0F
-
-rax
AX 0000
:00cd
-
-a
77B0:0100 AND AL,0F         ;mask off upper nibble
77B0:0102 CLC               ;clear the carry flag
77B0:0103 RCL AL,1          :rotate left four times
77B0:0105 RCL AL,1
77B0:0107 RCL AL,1
77B0:0109 RCL AL,1
77B0:010B INT 20
77B0:010D
-
-r
AX=00CD  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=77B0  ES=77B0  SS=77B0  CS=77B0  IP=0100   NV UP EI PL NZ NA PO NC
77B0:0100 240F          AND     AL,0F
-
-g 010b

AX=00D0  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=77B0  ES=77B0  SS=77B0  CS=77B0  IP=010B   OV UP EI PL NZ NA PO NC
77B0:010B CD20          INT     20
-
```

Fig. 20-37 8086/8088 program which clears the upper nibble of AL and then moves the lower nibble into the upper nibble.

SELF-TESTING REVIEW

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

1. Does rotating or shifting move the carry bit into one of the ends of the affected register? _____
2. *(Rotating)* Does rotating or shifting move 0s into one of the ends of the affected register? _____
3. *(Shifting)* Which of the following instructions duplicates the current value of the most significant bit and makes it the new value of the most significant bit? Rotate right, rotate left, logical shift right, logical shift left, arithmetic shift right, or arithmetic shift left?
   (Arithmetic shift right)

PROBLEMS

Specific Microprocessor Families

Solve the following problems using the microprocessor of your choice.

20-1. Write a program which will place the number 34_{16} in the accumulator, clear the lower nibble (F), and then move the upper nibble (C) into the lower nibble by using a rotate instruction. (Write the program so that if the carry flag happens to be set (1) prior to running the program, it will *not* rotate the 1 from the carry flag into the upper nibble of the accumulator.)

20-2. The ASCII value for numbers is the same as the hex value for numbers except that the ASCII value has a 3 as a prefix. For example, the ASCII value for 0 is 30, the ASCII value for 1 is 31, the ASCII value for 2 is 32, the ASCII value for 3 is 33, and so on.

Write a program that will place the hex value 23 in the accumulator and will then take the upper nibble (2h), change it to its ASCII value (32h), and store it in a memory location. The program should then take the lower nibble (3h), change it to its ASCII value (33h), and store it in another memory location.

Restrictions: (1) You cannot use shift instructions (but you may use rotate instructions). (2) You must make the program so that it will work for any original value, not just 23h. (That value was picked randomly.)

Hints: (1) You should store the original value (23) in a memory location so that you can use it more than once. (2) You will need to use rotate instructions, masks, and arithmetic instructions. (3) You need to set aside three memory locations: one for the original value (23h), one for theASCII value for 2 (32h), and one for the ASCII value for 3 (33h).

20-3. Place the ASCII value for 8 (38h) in one memory location and the ASCII value for 9 (39h) in another location. Then write a program which will take these two ASCII values, convert them to their hex equivalents (8h and 9h), and combine them into a 1-byte, 2-digit, hex number (89h).

20-4. Since the value of a binary digit doubles in value each time it is moved to the left by one place, and becomes one-half of its previous value each time it is moved to the right one place, it is possible to multiply and divide by shifting/rotating.

Write a program which will load the value $1C_{16}$ into the accumulator and multiply it by 8 by shifting it.

6502, 6800/6808, and 8086/8088 users: You should use the arithmetic shift left type of instruction because it automatically shifts 0s into the least significant bit.

8086/8088 users: You have an actual multiply instruction but shouldn't use it for this program, since this chapter is intended to help you write programs using shift and rotate instructions.

Z80 users: You have an arithmetic shift left type of instruction, but you cannot use it here because it is not part of the 8080/8085 instruction subset. Use the following procedure for the 8080/8085.

8080/8085 users: You do not have any shift instructions; therefore, you should alternately clear the carry flag and rotate to achieve an effect similar to that of the arithmetic shift left instruction.

All users: There are other ways to multiply. This simply illustrates one way, and not necessarily the best or easiest for your particular microprocessor.