Dr. Apurva Muley (Guest Lecturer)

School of Studies in Physics, Vikram University, Ujjain

Lecture for M. Sc. Physics IV Semester students

Paper-IV: Microprocessor

Unit-2 Logical Instructions

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.

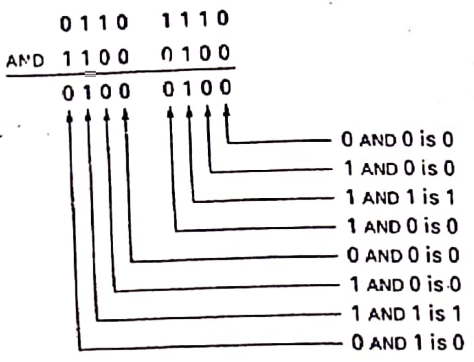

Fig. 19-2 ANDing 2 bytes together.

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

```
     1 0 0 1   1 0 0 1  ←——— data
AND  1 1 1 1   0 0 0 0  ←——— mask
     1 0 0 1   0 0 0 0
```

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

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

Fig. 19-4 OR truth table.

look at it another way, the only way to get a 0 out is to have 0s at both inputs.

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.

```
     0 1 1 0   1 1 1 0
OR   1 1 0 0   0 1 0 0
     1 1 1 0   1 1 1 0
     ↑ ↑ ↑ ↑   ↑ ↑ ↑ ↑
     │ │ │ │   │ │ │ └──── 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.

```
        0110  1110
    XOR 1100  0100
        ----------
        1010  1010
        ││││  ││││
        ││││  │││└──── 0 AND 0 is 0
        ││││  ││└───── 1 AND 0 is 1
        ││││  │└────── 1 AND 1 is 0
        ││││  └─────── 1 AND 0 is 1
        │││└────────── 0 AND 0 is 0
        ││└─────────── 1 AND 0 is 1
        │└──────────── 1 AND 1 is 0
        └───────────── 0 AND 1 is 1
```

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).

```
     0101  1111  ←— original number (95₁₀)
     1010  0000  ←— 1's complement
+             1  ←— plus 1
────────────────
     1010  0001  ←— 2's complement (−95₁₀)

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 FO      MVI B,FO        ;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 FO      LD B,FO         ;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 FO      MVI B,FO        ;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 FO      LD B,FO         ;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.