Chapter 6

The aim of this chapter is to understand and work with **Boolean operations**, **conditional jumps** and **selection statements.**

Binary processing is one of the basics that should be studied in conditional processing; specifically, we study the AND, OR, XOR, NOT and TEST operations.

AND Instruction:

Performs a bitwise AND operation between each pair of matching bits in two operands and places the result in the destination operand. Note that operands can be 8-, 16-, or 32-bit, and they should have **the same size**.

Syntax:

AND *destination , source*

The following are the permitted operand operations:

AND *reg , reg*
AND *reg , mem*
AND *reg , imm*
AND *mem . reg*
AND *mem , imm*

How does the AND operation works? For each matching bit in each operand, if both bits equal 1, the result bit is 1; otherwise, it is zero.

The ***Overflow*** and the ***Carry*** flags are always cleared by the AND instruction. The ***Sign, Zero,*** and ***Parity*** flags are modified according to the destination operand.

Example:
Covert a lower case character ('a' → 61h) to upper case ('A→ 41h'):

61h: 0 1 **1** 0 0 0 0 1
41h: 0 1 **0** 0 0 0 0 1

Thus, to convert from small letter to capital letter, we have to AND the small letter character with 11011111.

```
.data
array BYTE 50 DUP(?)
.code
    mov ecx LENGTHOF array
    mov esi, OFFSET array
L1:
    And BYTE PTR [esi] , 11011111b
    inc esi
    loop L1
```

Note that we have to use the BYTE PTR otherwise the assembler cannot tell whether esi points to a BYTE, WORD, or a DWORD.

OR Instruction:

Performs a bitwise (Boolean) OR operation between each pair of matching bits in two operands and places the result in the destination operand. Note that operands can be 8-, 16-, or 32-bit, and they should have **the same size**.

Syntax: OR *destination , source*

Similar to AND instruction, the following are the permitted operand operations:

OR *reg , reg*
OR *reg , mem*
OR *reg , imm*
OR *mem . reg*
OR *mem , imm*

How does the OR operation works? For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1.

Example:

OR instruction can be used to convert from (0 – 9) integer into (0-9) ASCII:

0 0 0 0 0 1 0 1 : 5h
0 0 1 1 0 0 0 0 : 30h
0 0 1 1 0 1 0 1 :

OR instruction clears the Carry and Overflow flag. It modifies the Sign, Zero, and Parity flag according to the value in the destination operand.

XOR Instruction:

Performs a bitwise (Boolean) exclusive-OR operation between each pair of matching bits in two operands and places the result in the destination operand. Note that operands can be 8-, 16-, or 32-bit, and they should have **the same size**.

Syntax: XOR *destination , source*

Similar to AND instruction, the following are the permitted operand operations:

XOR *reg , reg*
XOR *reg , mem*
XOR *reg , imm*
XOR *mem . reg*
XOR *mem , imm*

How does the XOR operation works? For each matching bit in the two operands, the output bit is 0 when both bits are different, and 1 if both bits are similar.

XOR instruction clears the Carry and Overflow flag. It modifies the Sign, Zero, and Parity flag according to the value in the destination operand.

NOT Instruction:

The NOT instruction complements all the bits in an operand.

Syntax: NOT operand

Permitted operand types are:

NOT reg
NOT mem

TEST Instruction:

Similar to the AND instruction but it does not modify the destination operand. It is used to check which individual bits are set.

If the result of a TEST operation is all zero, the Zero flag is set; otherwise it is cleared.

CMP Instruction:

It performs subtraction of a source operand from a destination operand. (N.B: operands are not modified.):

Syntax:

CMP destination, source

The result of the CMP instruction affects the FLAGS as follows:

CMP Results	**ZF**	**CF**
destination < source	0	1
destination > source	0	0
destination = source	1	0

CMP Results	**Flags**
destination < source	SF != OF
destination > source	SF = OF
destination = source	ZF = 1

CMP uses the same operand combinations as the AND instruction.

In all cases, we benefit from the result of CMP by using the following conditional jump instructions:

Syntax:

Jcond destination

where destination is a label in the code segment. *Jcond* will move the program execution to the destination if the result of the Jcond is true based in the result from the CMP instruction.

Jcond instructions:

Jc	Jump if carry flag is set.
Jnc	Jump if carry flag is not set (clear).

Jz	Jump if zero flag is set.
Jnz	Jump if zero flag is not set (clear).
Jo	Jump if overflow flag is set.
Jno	Jump if overflow flag is not set (clear).
Js	Jump if signed flag is set.
Jns	Jump if signed flag is not set (clear).

Equality comparisons:

Je	Jump if left operand = right operand.
Jne	Jump if left operand != right operand.
JCXZ	Jump if CX = 0.
JECXZ	Jump if ECX = 0.

Unsigned comparisons:

JA	Jump if above (left operand > right operand).
JNBE	Same as JA.
JAE	Jump if above or equal (left >= right).
JNB	Same as JAE.
JB	Jump if below (left < right).
JNAE	Same as JB.
JBE	Jump if below or equal (left <= right).
JNA	Same as JBE.

Signed comparisons:

JG	Jump if greater (left > right).
JNLE	Same as JG.
JGE	Jump if greater or equal (left >= right).
JNL	Jump if not less than (Same as JGE).
JL	Jump if less (left < right).
JNGE	Same as JL.
JLE	Jump if less than or equal (left <= right).
JNG	Same as JLE.

Example:

Finding an element in an array.
Encrypting a text.

LOOPZ and LOOPE Instructions:

Loope and Loopz (loop if zero): this instruction loops as long as the Zero flag is set and the value of EXC is greater than zero.

LOOPNZ and LOOPNE

Loopnz and Loopne: loops as long as the Zero flag is clear and ECX is greater than zero.