ASSEMBLY - LOGICAL INSTRUCTIONS

The processor instruction set provides the instructions AND, OR, XOR, TEST, and NOT Boolean logic, which tests, sets, and clears the bits according to the need of the program.

The format for these instructions −

SN	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

The first operand in all the cases could be either in register or in memory. The second operand could be either in register/memory or an immediate *constant* value. However, memory-to-memory operations are not possible. These instructions compare or match bits of the operands and set the CF, OF, PF, SF and ZF flags.

The AND Instruction

The AND instruction is used for supporting logical expressions by performing bitwise AND operation. The bitwise AND operation returns 1, if the matching bits from both the operands are 1, otherwise it returns 0. For example −

```
             Operand1:  0101
             Operand2:  0011
----------------------------
After AND -> Operand1: 0001
```

The AND operation can be used for clearing one or more bits. For example, say the BL register contains 0011 1010. If you need to clear the high-order bits to zero, you AND it with 0FH.

```
AND BL,   0FH   ; This sets BL to 0000 1010
```

Let's take up another example. If you want to check whether a given number is odd or even, a simple test would be to check the least significant bit of the number. If this is 1, the number is odd, else the number is even.

Assuming the number is in AL register, we can write −

```
AND AL, 01H     ; ANDing with 0000 0001
JZ    EVEN_NUMBER
```

The following program illustrates this −

Example

```
section .text
   global _start            ;must be declared for using gcc

_start:                     ;tell linker entry point
   mov   ax,   8h            ;getting 8 in the ax
```

```
    and    ax, 1                ;and ax with 1
    jz     evnn
    mov    eax, 4               ;system call number (sys_write)
    mov    ebx, 1               ;file descriptor (stdout)
    mov    ecx, odd_msg         ;message to write
    mov    edx, len2            ;length of message
    int    0x80                 ;call kernel
    jmp    outprog

evnn:

    mov    ah,  09h
    mov    eax, 4               ;system call number (sys_write)
    mov    ebx, 1               ;file descriptor (stdout)
    mov    ecx, even_msg        ;message to write
    mov    edx, len1            ;length of message
    int    0x80                 ;call kernel

outprog:

    mov    eax,1                ;system call number (sys_exit)
    int    0x80                 ;call kernel

section    .data
even_msg  db  'Even Number!' ;message showing even number
len1  equ  $ - even_msg

odd_msg db  'Odd Number!'    ;message showing odd number
len2  equ  $ - odd_msg
```

When the above code is compiled and executed, it produces the following result −

```
Even Number!
```

Change the value in the ax register with an odd digit, like −

```
mov  ax, 9h                  ; getting 9 in the ax
```

The program would display:

```
Odd Number!
```

Similarly to clear the entire register you can AND it with 00H.

The OR Instruction

The OR instruction is used for supporting logical expression by performing bitwise OR operation. The bitwise OR operator returns 1, if the matching bits from either or both operands are one. It returns 0, if both the bits are zero.

For example,

```
             Operand1:     0101
             Operand2:     0011
----------------------------
After OR -> Operand1:    0111
```

The OR operation can be used for setting one or more bits. For example, let us assume the AL register contains 0011 1010, you need to set the four low-order bits, you can OR it with a value 0000 1111, i.e., FH.

```
OR BL, 0FH                   ; This sets BL to  0011 1111
```

Example

The following example demonstrates the OR instruction. Let us store the value 5 and 3 in the AL and the BL registers, respectively, then the instruction,

```
OR AL, BL
```

should store 7 in the AL register −

```
section .text
   global _start            ;must be declared for using gcc

_start:                     ;tell linker entry point
   mov    al, 5              ;getting 5 in the al
   mov    bl, 3              ;getting 3 in the bl
   or     al, bl             ;or al and bl registers, result should be 7
   add    al, byte '0'      ;converting decimal to ascii

   mov    [result],  al
   mov    eax, 4
   mov    ebx, 1
   mov    ecx, result
   mov    edx, 1
   int    0x80

outprog:
   mov    eax,1              ;system call number (sys_exit)
   int    0x80               ;call kernel

section    .bss
result resb 1
```

When the above code is compiled and executed, it produces the following result −

```
7
```

The XOR Instruction

The XOR instruction implements the bitwise XOR operation. The XOR operation sets the resultant bit to 1, if and only if the bits from the operands are different. If the bits from the operands are same $both 0 or both 1$, the resultant bit is cleared to 0.

For example,

```
             Operand1:     0101
             Operand2:     0011
----------------------------
After XOR -> Operand1:    0110
```

> *XORing an operand with itself changes the operand to 0. This is used to clear a register.*

```
XOR     EAX, EAX
```

The TEST Instruction

The TEST instruction works same as the AND operation, but unlike AND instruction, it does not change the first operand. So, if we need to check whether a number in a register is even or odd, we can also do this using the TEST instruction without changing the original number.

```
TEST    AL, 01H
JZ      EVEN_NUMBER
```

The NOT Instruction

The NOT instruction implements the bitwise NOT operation. NOT operation reverses the bits in an operand. The operand could be either in a register or in the memory.

For example,

```
             Operand1:    0101 0011
After NOT -> Operand1:    1010 1100
```