# Module 3

## Arithmetic and Logical Instructions

FEU ALABANG FEU DILIMAN FEU TECH

*Technology Driven by Innovation*

Subtopic 1

## Arithmetic and Logical Instructions

## Objectives

- Apply the different arithmetic instructions.
- Understand the assembly logical instructions.

## The INC Instruction

**The INC instruction is used for incrementing an operand by one. It works on a single operand that can be either in a register or in memory.**

**The INC instruction has the following syntax:**

```
INC destination
```

**The operand destination could be an 8-bit, 16-bit or 32-bit operand.**
**Example**

```
INC EBX ; Increments 32-bit register
INC DL ; Increments 8-bit register
INC [count] ; Increments the count variable
```

# The DEC Instruction

**The DEC instruction is used for decrementing an operand by one. It works on a single operand that can be either in a register or in memory.**

**The DEC instruction has the following syntax:**

```
DEC destination
```

**The operand destination could be an 8-bit, 16-bit or 32-bit operand.**

**Example**

```
segment .data
         count dw 0
         value db 15
segment .text
         inc [count]
         dec [value]
         mov ebx,
         count inc
         word [ebx]
         mov esi, value
         dec byte [esi]
```

## The ADD and SUB Instructions

**The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte, word and doubleword size, i.e., for adding or subtracting 8-bit, 16- bit or 32-bit operands, respectively.**

**The ADD and SUB instructions have the following syntax:**

```
ADD/SUB destination, source
```

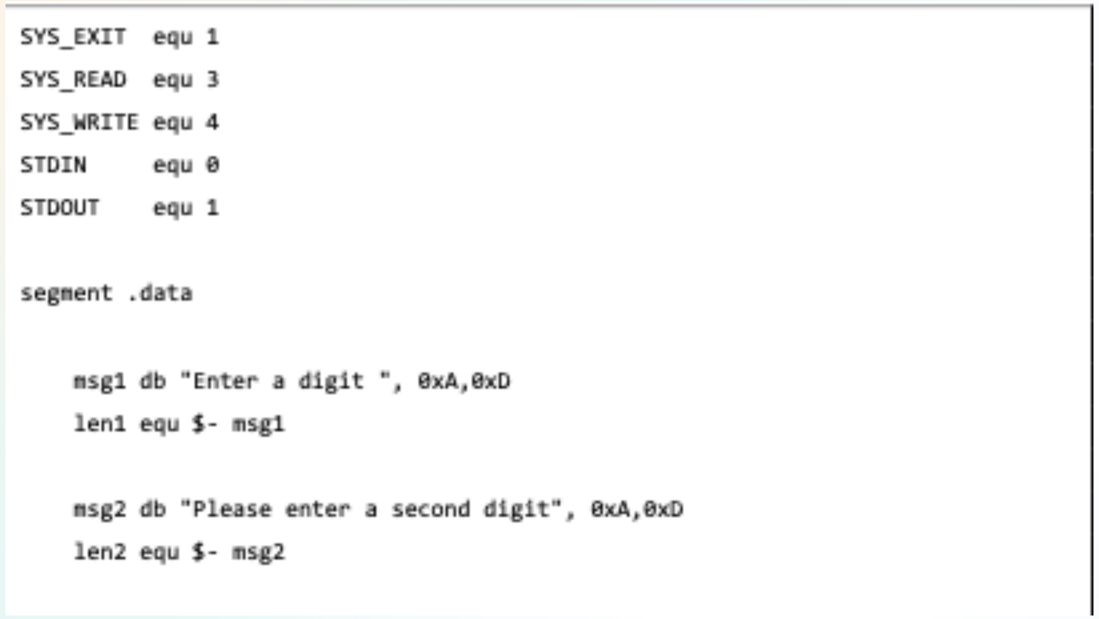
**The ADD/SUB instruction can take place between:**

- **Register to register**
- **Memory to register**
- **Register to memory**
- **Register to constant data**
- **Memory to constant data**

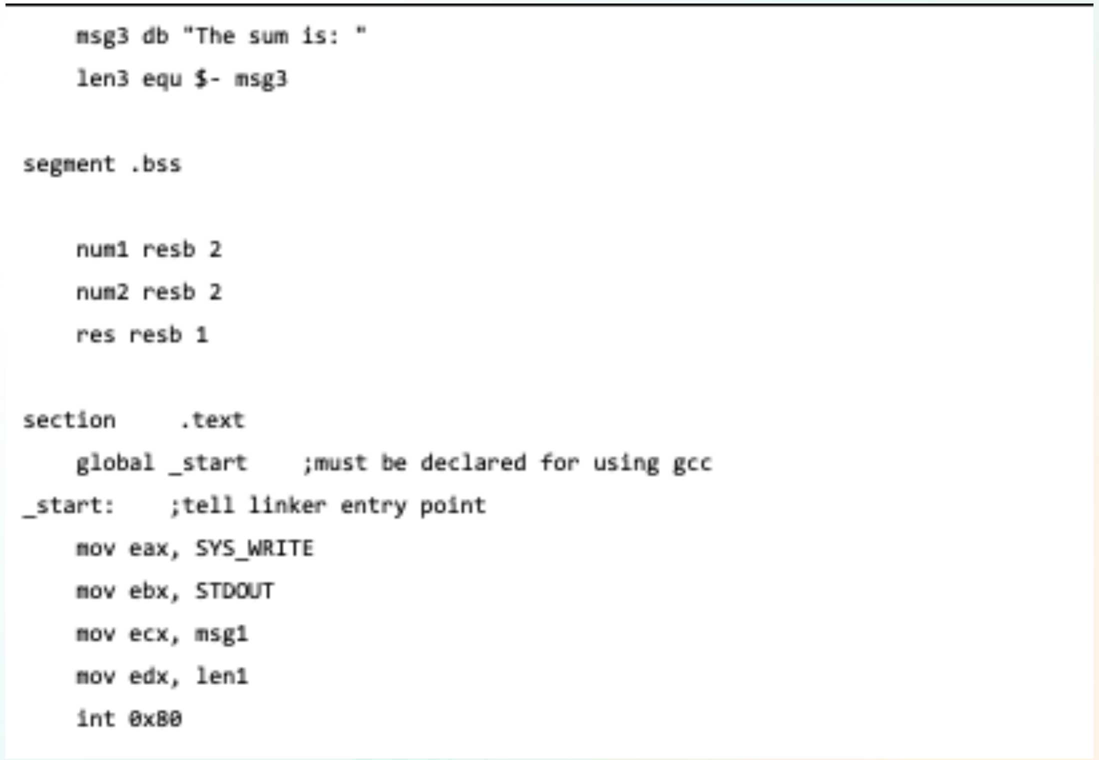
**However, like other instructions, memory-to-memory operations are not possible using ADD/SUB instructions. An ADD or SUB operation sets or clears the overflow and carry flags.**

## The ADD and SUB Instructions

**The following example will ask two digits from the user, store the digits in the EAX and EBX register, respectively, add the values, store the result in a memory location 'res' and finally display the result.**



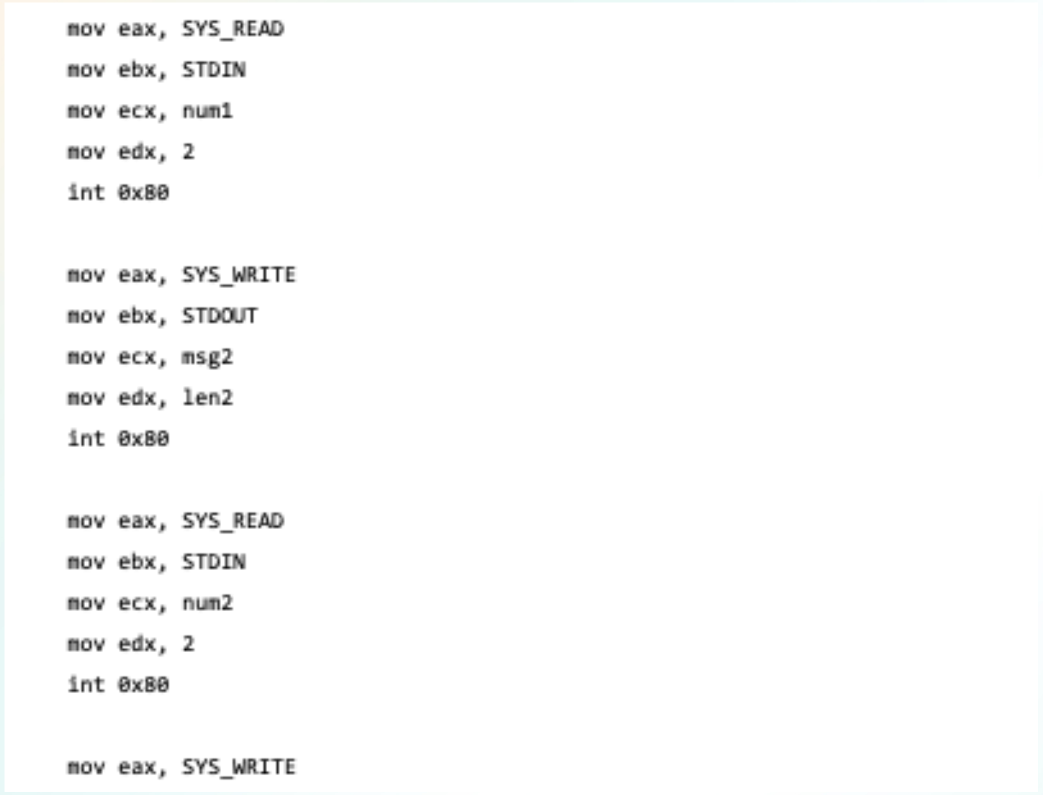
```
SYS_EXIT  equ 1
SYS_READ  equ 3
SYS_WRITE equ 4
STDIN     equ 0
STDOUT    equ 1

segment .data

    msg1 db "Enter a digit ", 0xA,0xD
    len1 equ $- msg1

    msg2 db "Please enter a second digit", 0xA,0xD
    len2 equ $- msg2
```



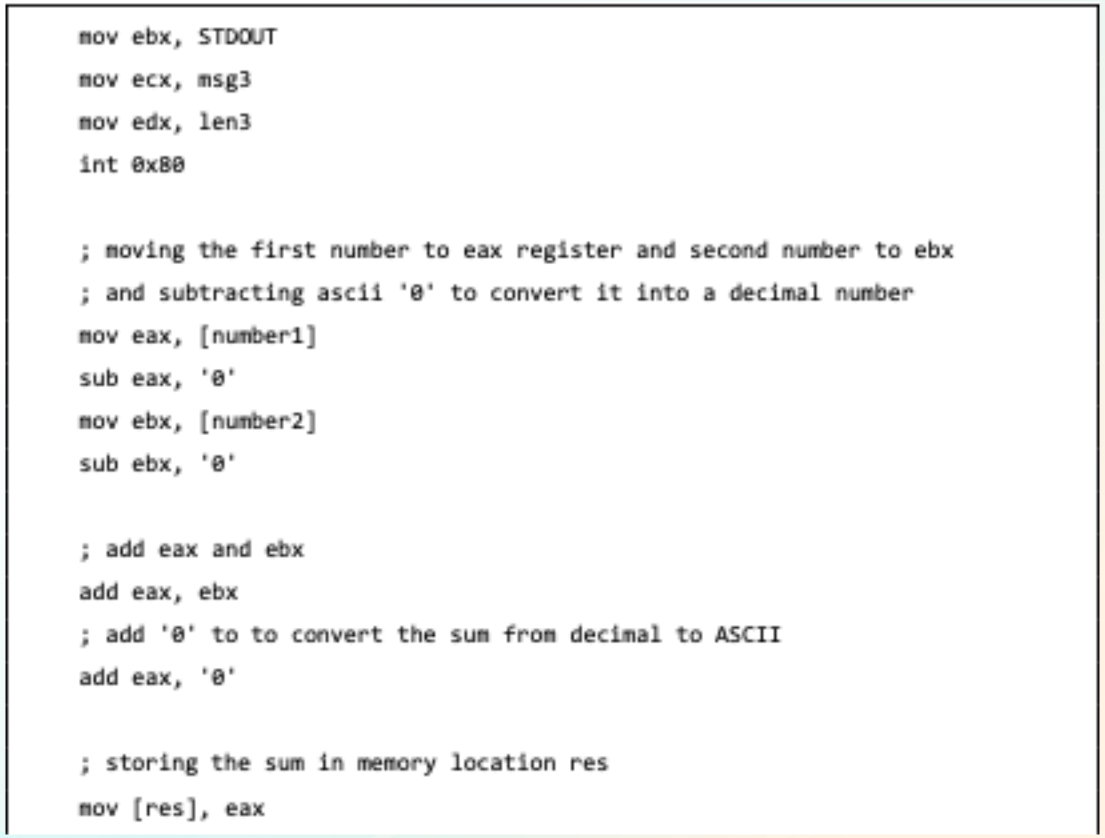
```
    msg3 db "The sum is: "
    len3 equ $- msg3

segment .bss

    num1 resb 2
    num2 resb 2
    res resb 1

section     .text
    global _start    ;must be declared for using gcc
_start:    ;tell linker entry point
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, msg1
    mov edx, len1
    int 0x80
```

# The ADD and SUB Instructions

**The following example will ask two digits from the user, store the digits in the EAX and EBX register, respectively, add the values, store the result in a memory location 'res' and finally display the result.**



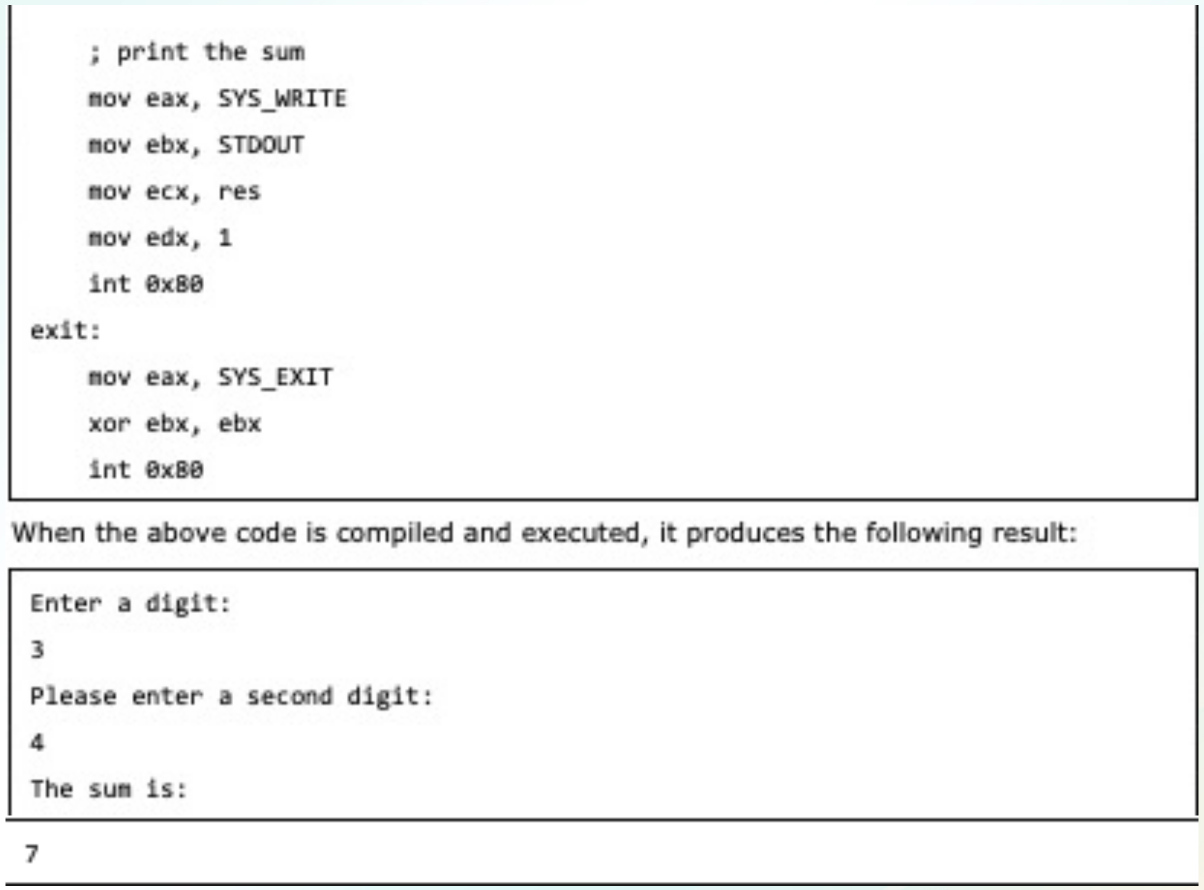
```
    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, num1
    mov edx, 2
    int 0x80

    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, msg2
    mov edx, len2
    int 0x80

    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, num2
    mov edx, 2
    int 0x80

    mov eax, SYS_WRITE
```



```
    mov ebx, STDOUT
    mov ecx, msg3
    mov edx, len3
    int 0x80

    ; moving the first number to eax register and second number to ebx
    ; and subtracting ascii '0' to convert it into a decimal number
    mov eax, [number1]
    sub eax, '0'
    mov ebx, [number2]
    sub ebx, '0'

    ; add eax and ebx
    add eax, ebx
    ; add '0' to to convert the sum from decimal to ASCII
    add eax, '0'

    ; storing the sum in memory location res
    mov [res], eax
```

## The ADD and SUB Instructions

**The following example will ask two digits from the user, store the digits in the EAX and EBX register, respectively, add the values, store the result in a memory location 'res' and finally display the result.**

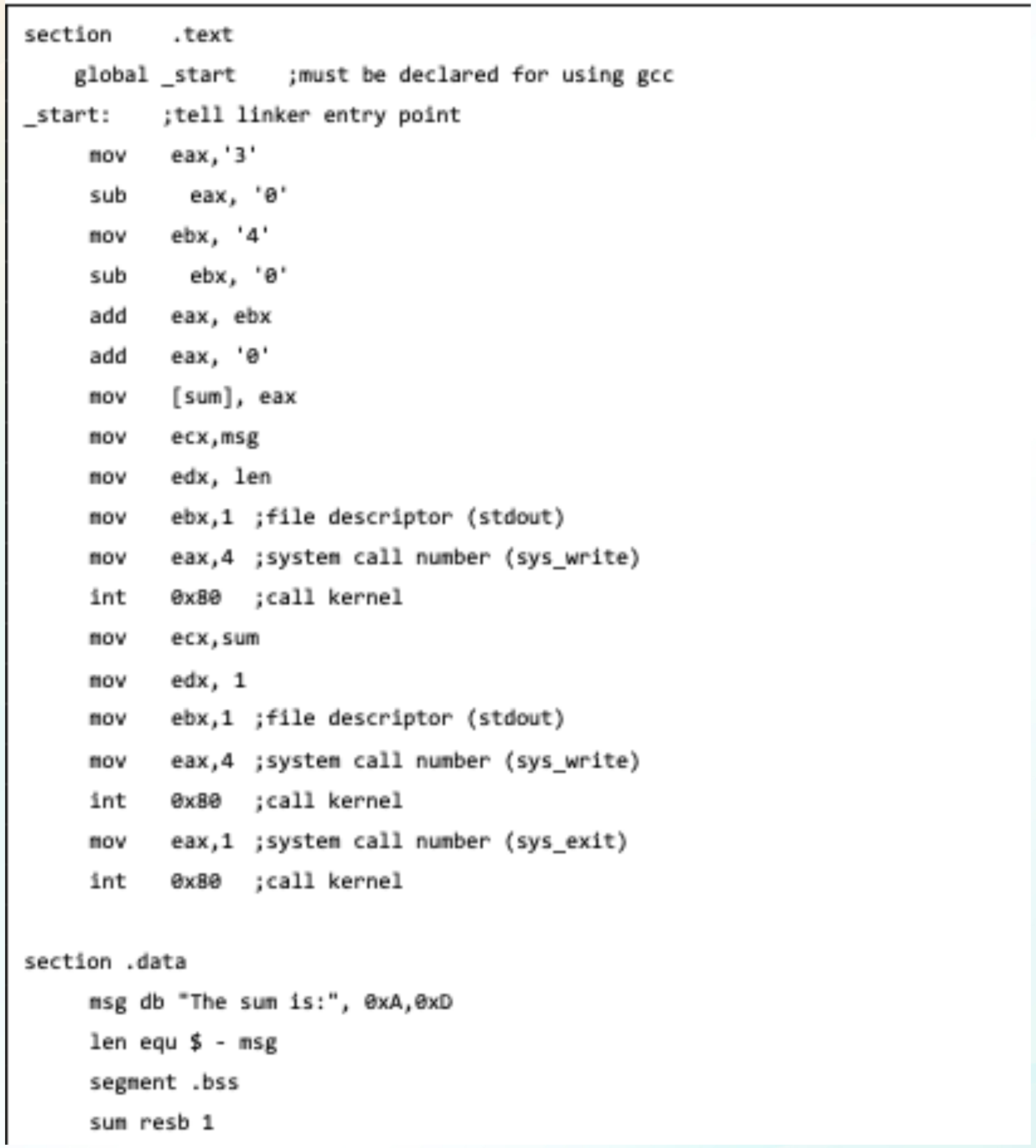


```
    ; print the sum
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, res
    mov edx, 1
    int 0x80
exit:
    mov eax, SYS_EXIT
    xor ebx, ebx
    int 0x80
```

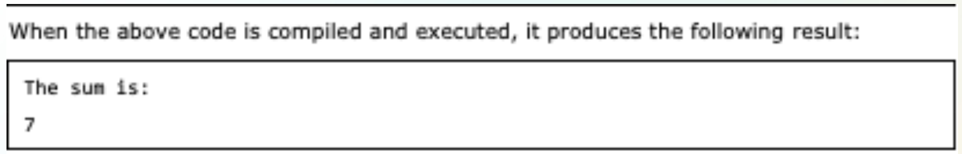
When the above code is compiled and executed, it produces the following result:

```
Enter a digit:
3
Please enter a second digit:
4
The sum is:
7
```

## The program with hardcoded variables:



```
section     .text
    global _start    ;must be declared for using gcc
_start:    ;tell linker entry point
     mov    eax,'3'
     sub     eax, '0'
     mov    ebx, '4'
     sub     ebx, '0'
     add    eax, ebx
     add    eax, '0'
     mov    [sum], eax
     mov    ecx,msg
     mov    edx, len
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    ecx,sum
     mov    edx, 1
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    eax,1  ;system call number (sys_exit)
     int    0x80   ;call kernel

section .data
     msg db "The sum is:", 0xA,0xD
     len equ $ - msg
     segment .bss
     sum resb 1
```



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

```
The sum is:
7
```

## The MUL/IMUL Instruction

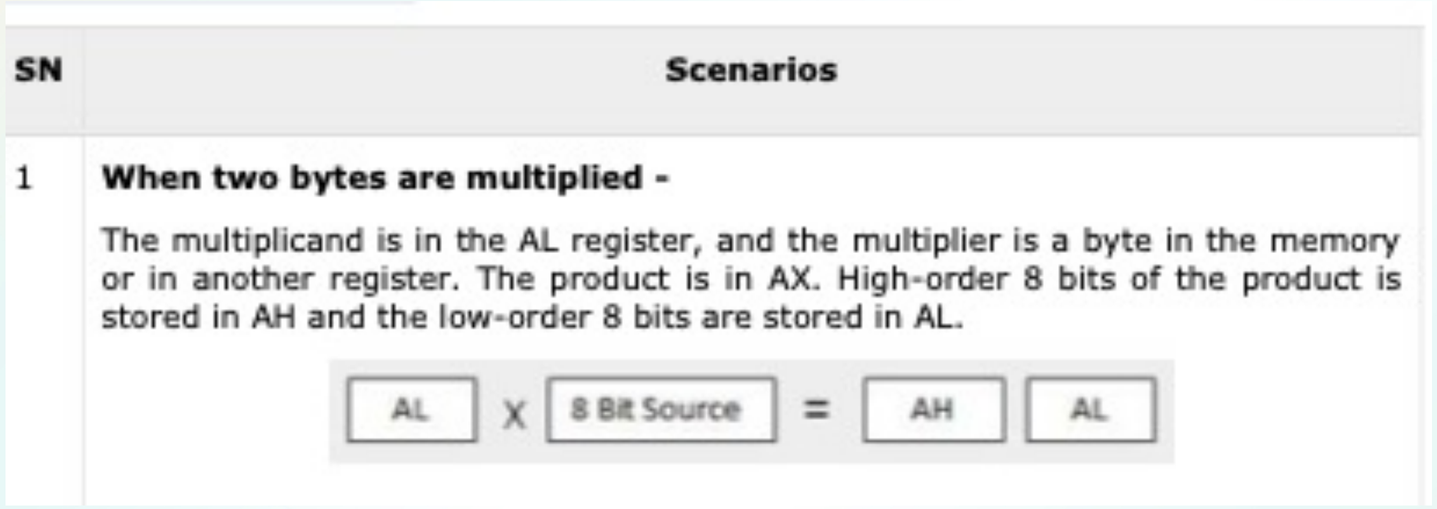
**There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.**

**The syntax for the MUL/IMUL instructions is as follows:**

```
MUL/IMUL multiplier
```

## The MUL/IMUL Instruction

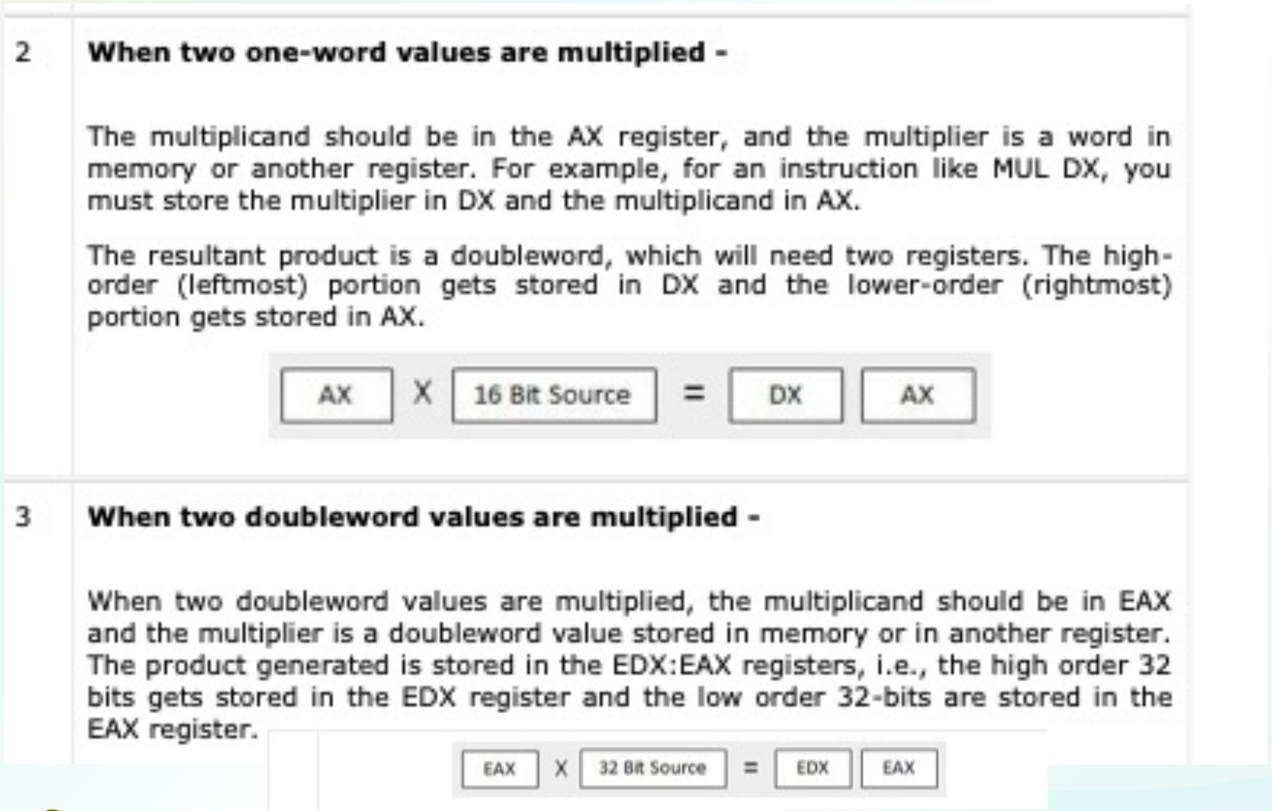
**Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands. Following section explains MUL instructions with three different cases:**



| SN | Scenarios |
|---|---|
| 1 | **When two bytes are multiplied -** <br> The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High-order 8 bits of the product is stored in AH and the low-order 8 bits are stored in AL. <br> AL X 8 Bit Source = AH AL |

## The MUL/IMUL Instruction

**Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands. Following section explains MUL instructions with three different cases:**



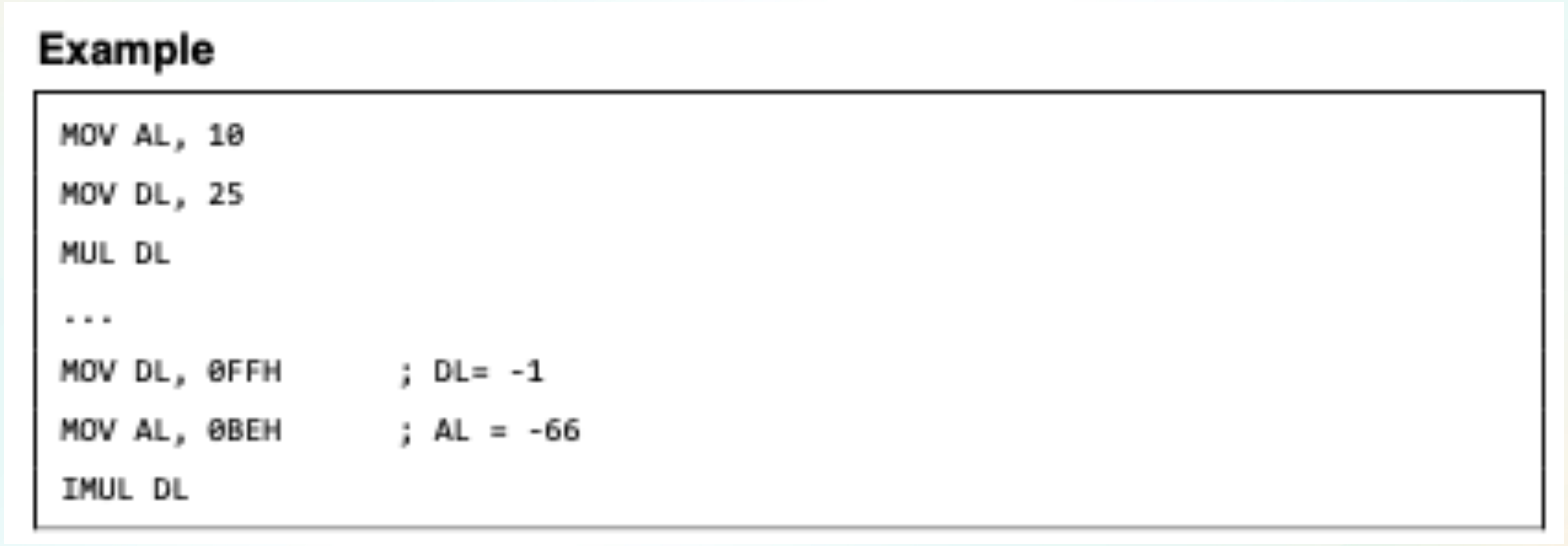
| | |
|---|---|
| 2 | **When two one-word values are multiplied -** <br><br> The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX. <br><br> The resultant product is a doubleword, which will need two registers. The high-order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX. <br><br> AX X 16 Bit Source = DX AX |
| 3 | **When two doubleword values are multiplied -** <br><br> When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register. <br><br> EAX X 32 Bit Source = EDX EAX |

FEU ALABANG FEU DILIMAN FEU TECH

*Technology Driven by Innovation*

## The MUL/IMUL Instruction

**Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands. Following section explains MUL instructions with three different cases:**


**Example**

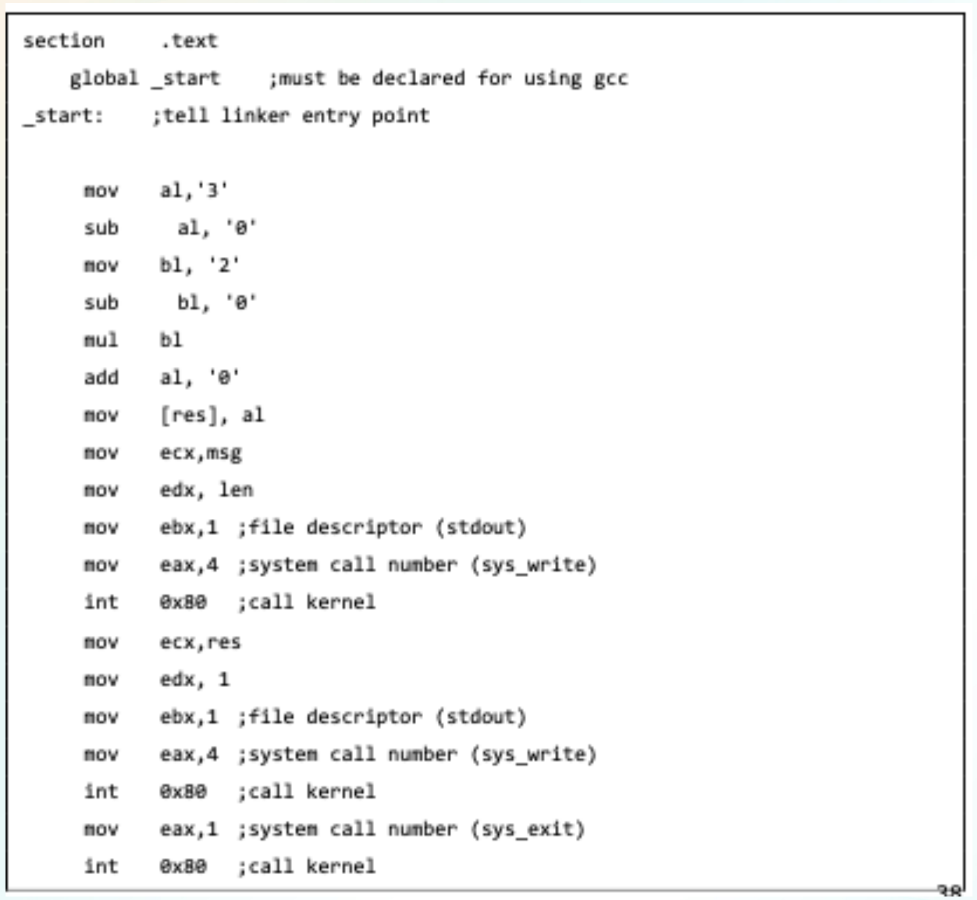
```
MOV AL, 10
MOV DL, 25
MUL DL
...
MOV DL, 0FFH       ; DL= -1
MOV AL, 0BEH       ; AL = -66
IMUL DL
```

FEU ALABANG FEU DILIMAN FEU TECH

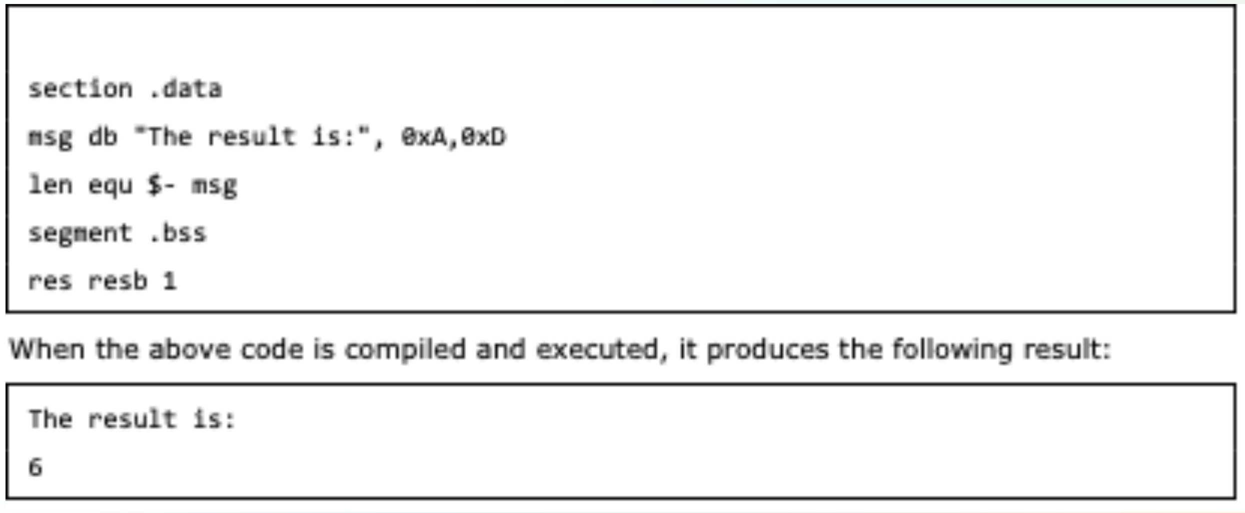
*Technology Driven by Innovation*

# The MUL/IMUL Instruction

The following example multiplies 3 with 2, and displays the result:



```
section     .text
    global _start    ;must be declared for using gcc
_start:     ;tell linker entry point

     mov    al,'3'
     sub      al, '0'
     mov    bl, '2'
     sub      bl, '0'
     mul    bl
     add    al, '0'
     mov    [res], al
     mov    ecx,msg
     mov    edx, len
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    ecx,res
     mov    edx, 1
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    eax,1  ;system call number (sys_exit)
     int    0x80   ;call kernel
```



```
section .data
msg db "The result is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

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

```
The result is:
6
```

## The DIV/IDIV Instructions

**The division operation generates two elements - a quotient and a remainder. In case of multiplication, overflow does not occur because double-length registers are used to keep the product. However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs.**

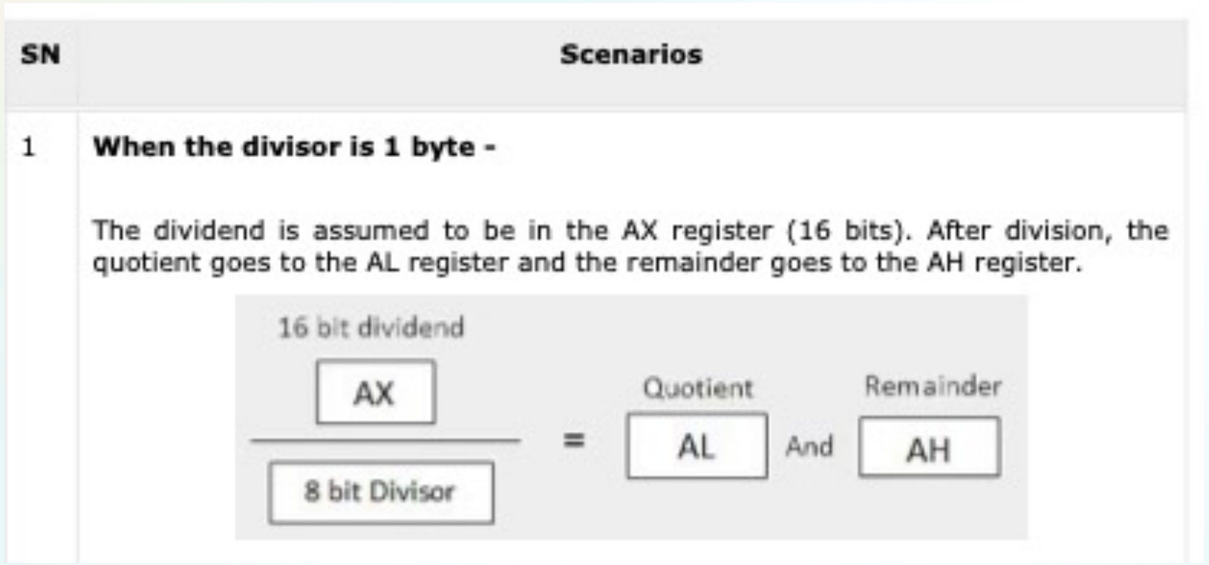
**The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data.**

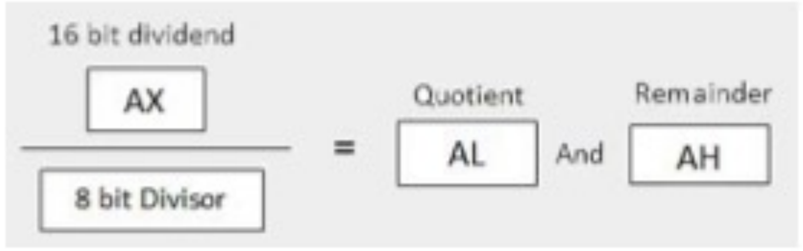
**The format for the DIV/IDIV instruction:**

```
DIV/IDIV divisor
```

# The DIV/IDIV Instructions

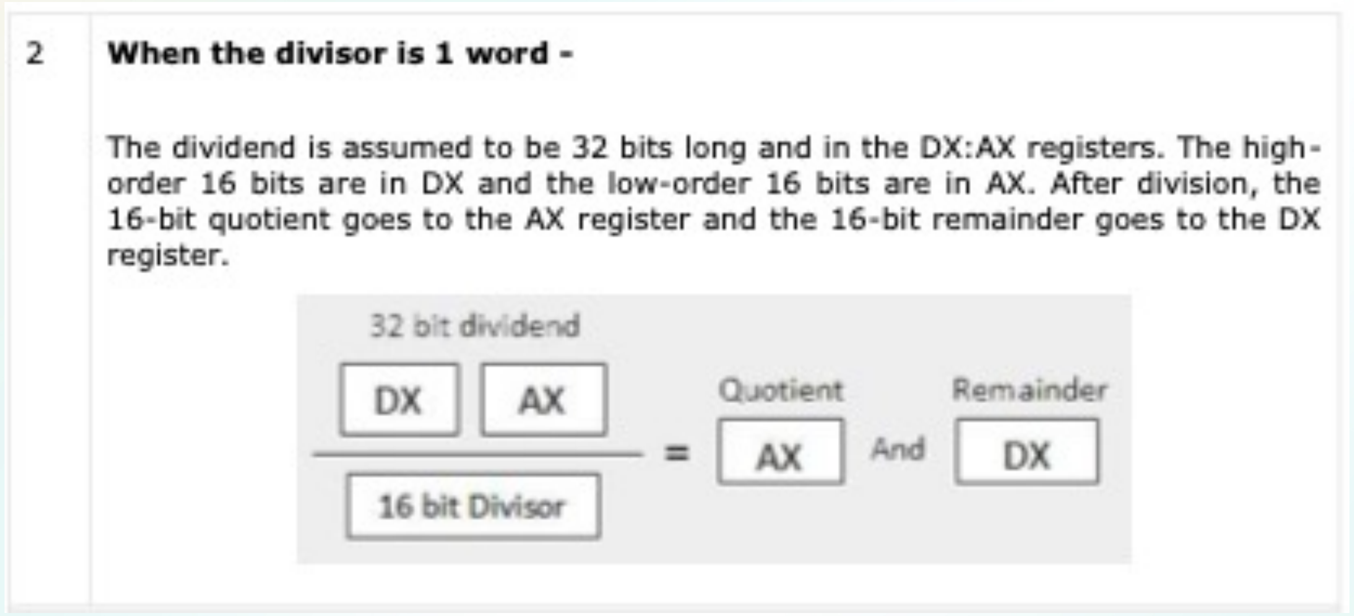
**The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size:**



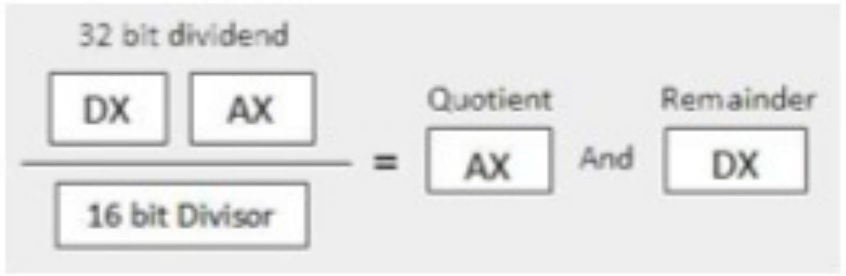
| SN | Scenarios |
|---|---|
| 1 | **When the divisor is 1 byte -** <br> The dividend is assumed to be in the AX register (16 bits). After division, the quotient goes to the AL register and the remainder goes to the AH register. <br>   16 bit dividend<br>AX<br>8 bit Divisor<br>=<br>Quotient<br>AL<br>And<br>Remainder<br>AH |

## The DIV/IDIV Instructions

**The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size:**

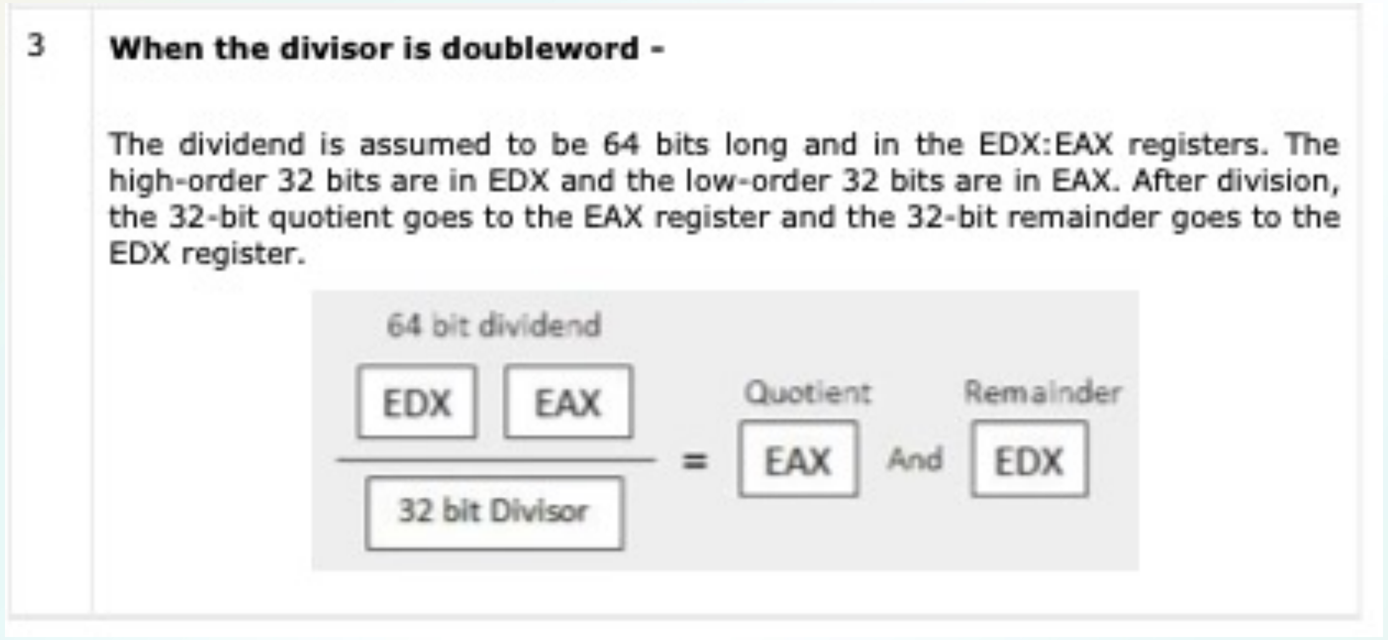


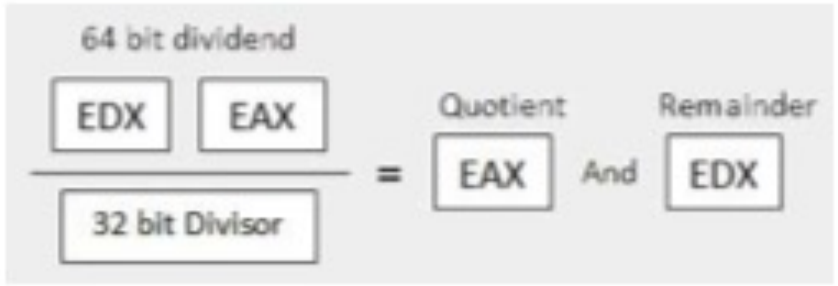
| | |
|---|---|
| 2 | **When the divisor is 1 word -** The dividend is assumed to be 32 bits long and in the DX:AX registers. The high-order 16 bits are in DX and the low-order 16 bits are in AX. After division, the 16-bit quotient goes to the AX register and the 16-bit remainder goes to the DX register. |



## The DIV/IDIV Instructions

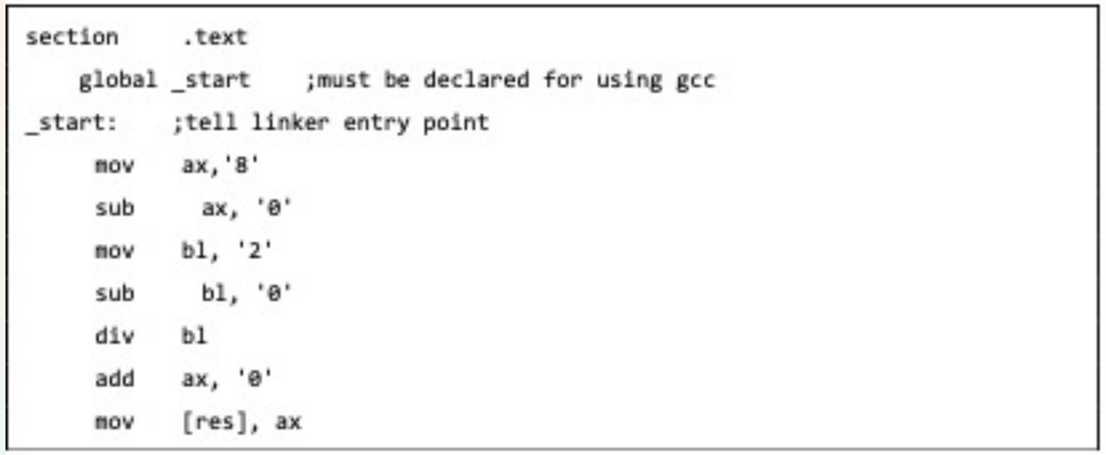
**The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size:**



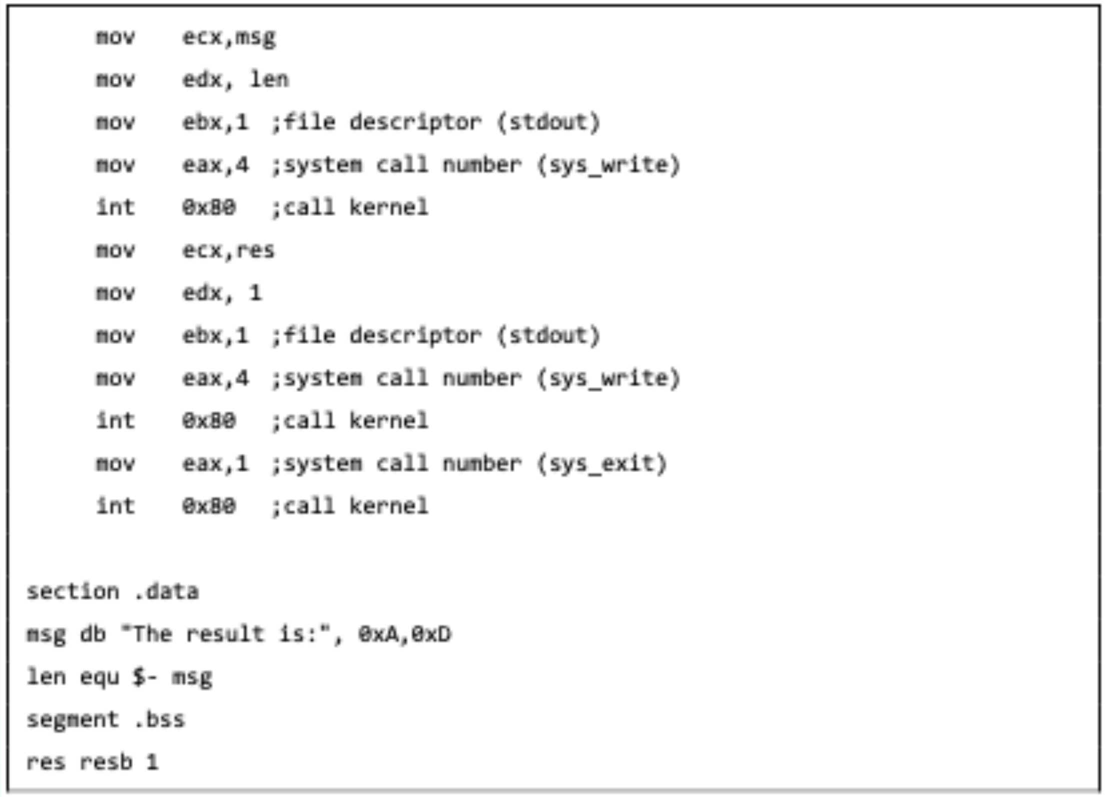
| | |
|---|---|
| 3 | **When the divisor is doubleword -** The dividend is assumed to be 64 bits long and in the EDX:EAX registers. The high-order 32 bits are in EDX and the low-order 32 bits are in EAX. After division, the 32-bit quotient goes to the EAX register and the 32-bit remainder goes to the EDX register.  64 bit dividend<br>EDX<br>EAX<br>32 bit Divisor<br>=<br>Quotient<br>EAX<br>And<br>Remainder<br>EDX |

## The DIV/IDIV Instructions

**The following example divides 8 with 2. The dividend 8 is stored in the 16-bit AX register and the divisor 2 is stored in the 8-bit BL register.**

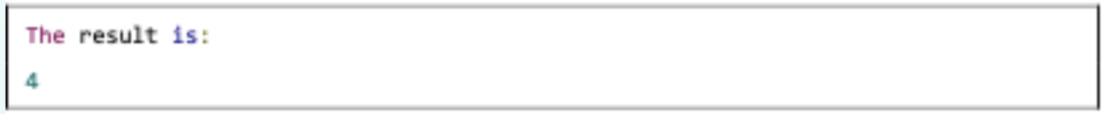


```
section    .text
   global _start    ;must be declared for using gcc
_start:    ;tell linker entry point
     mov    ax,'8'
     sub      ax, '0'
     mov    bl, '2'
     sub      bl, '0'
     div    bl
     add    ax, '0'
     mov    [res], ax
```



```
     mov    ecx,msg
     mov    edx, len
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    ecx,res
     mov    edx, 1
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    eax,1  ;system call number (sys_exit)
     int    0x80   ;call kernel

section .data
msg db "The result is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

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



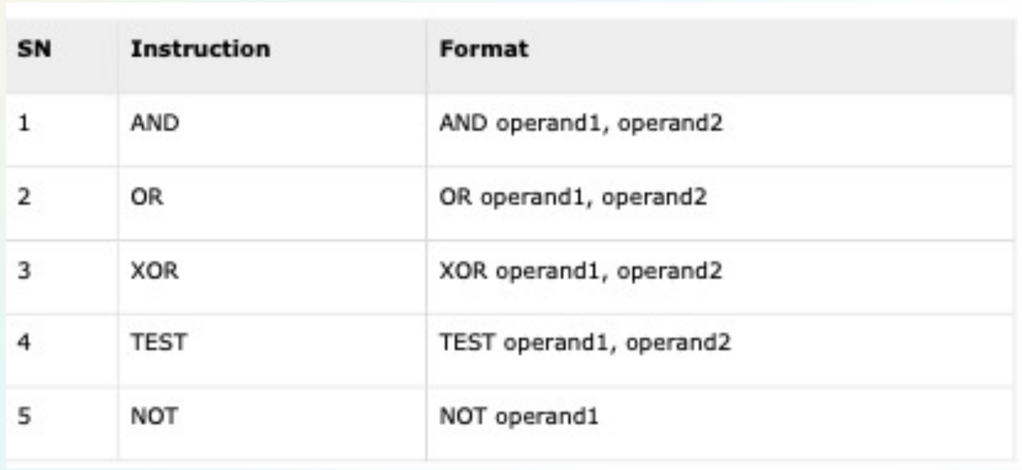
```
The result is:
4
```

FEU ALABANG FEU DILIMAN FEU TECH

# 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:

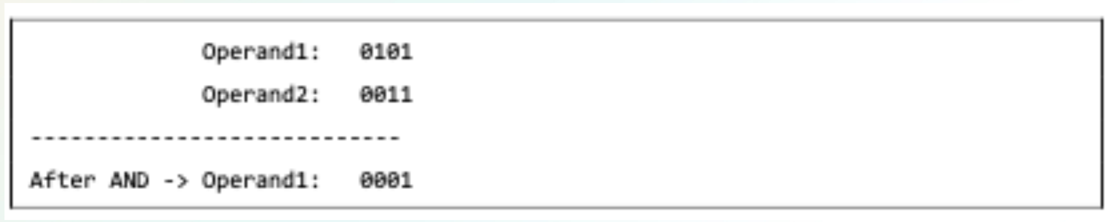


| SN | Instruction | Format |
|---|---|---|
| 1 | AND | AND operand1, operand2 |
| 2 | OR | OR operand1, operand2 |
| 3 | XOR | XOR operand1, operand2 |
| 4 | TEST | TEST operand1, operand2 |
| 5 | NOT | NOT operand1 |

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.

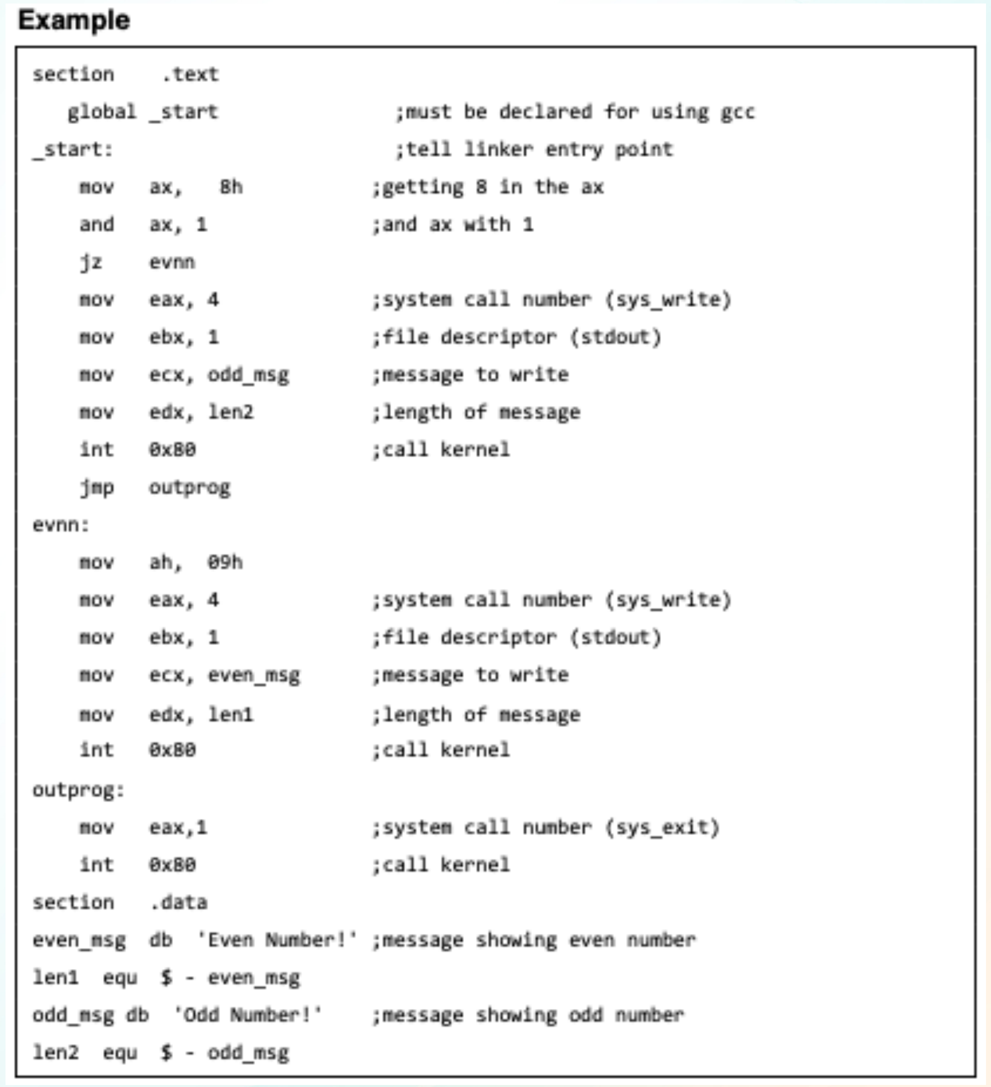
## The AND Instruction

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:

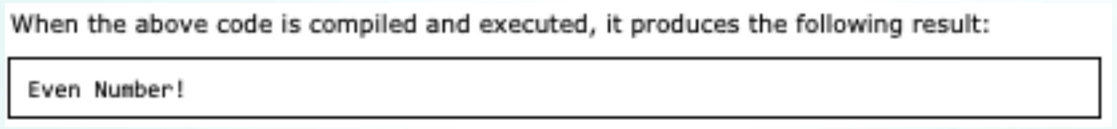
```
AND   AL, 01H      ; ANDing with 0000 0001
JZ     EVEN_NUMBER
```

# The AND Instruction



**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
```



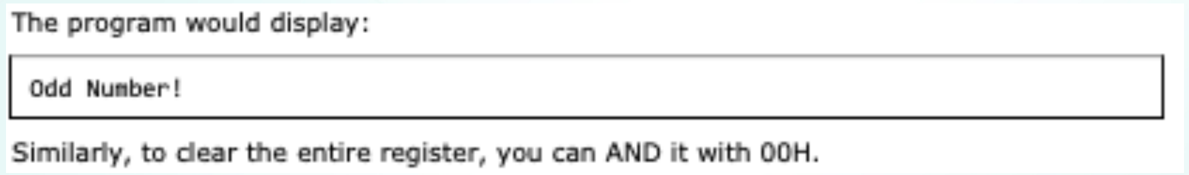
When the above code is compiled and executed, it produces the following result:

```
Even Number!
```

# The AND Instruction

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

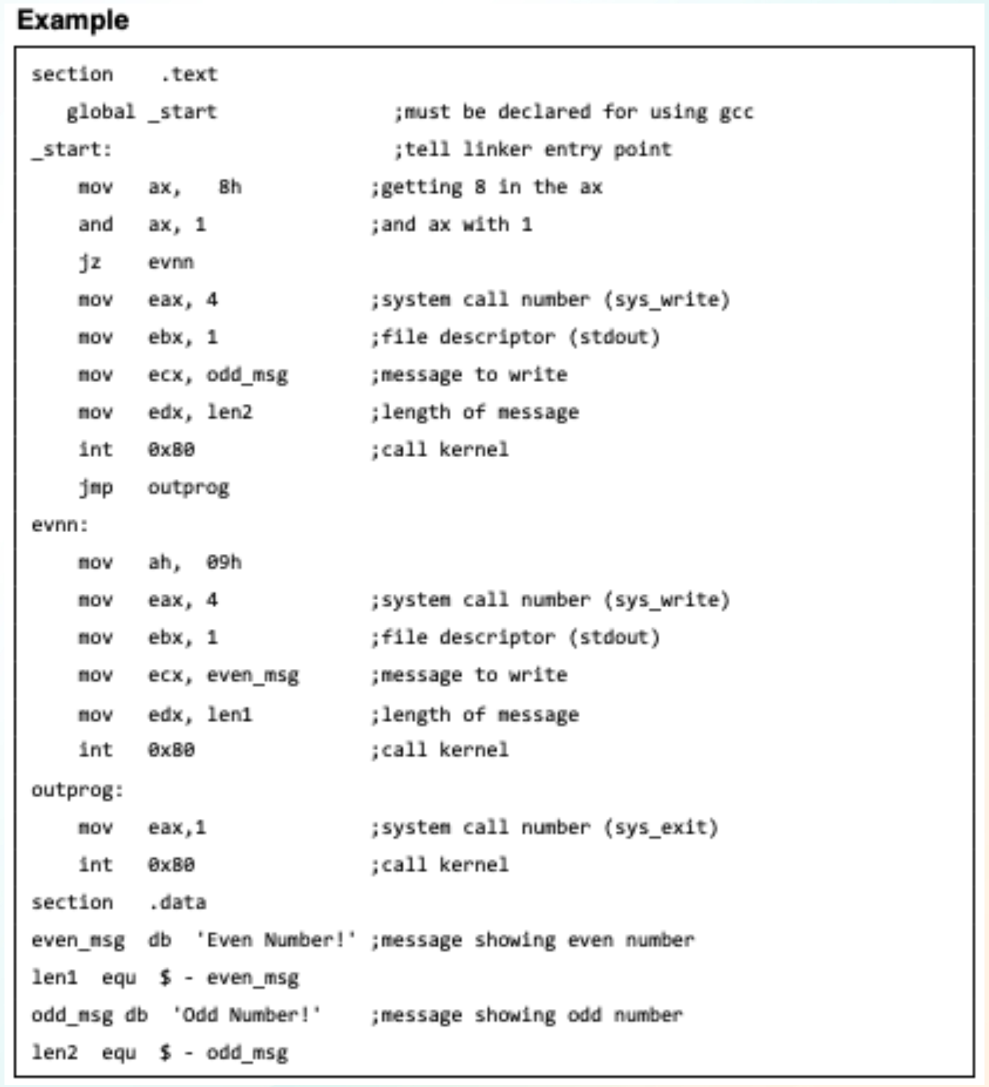
```
mov ax, 9h ; getting 9 in the ax
```



The program would display:

```
Odd Number!
```

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



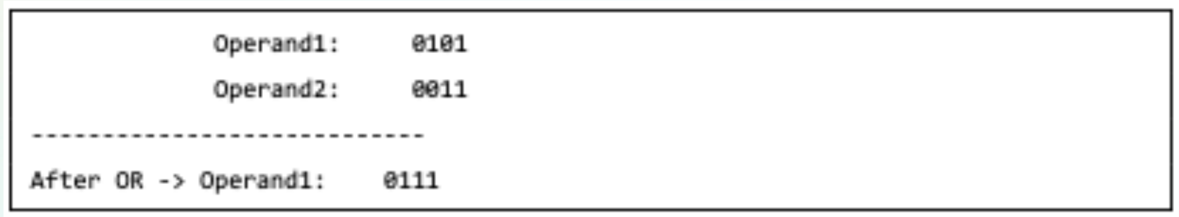
**Example**

```
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
```

# 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
```

FEU ALABANG FEU DILIMAN FEU TECH

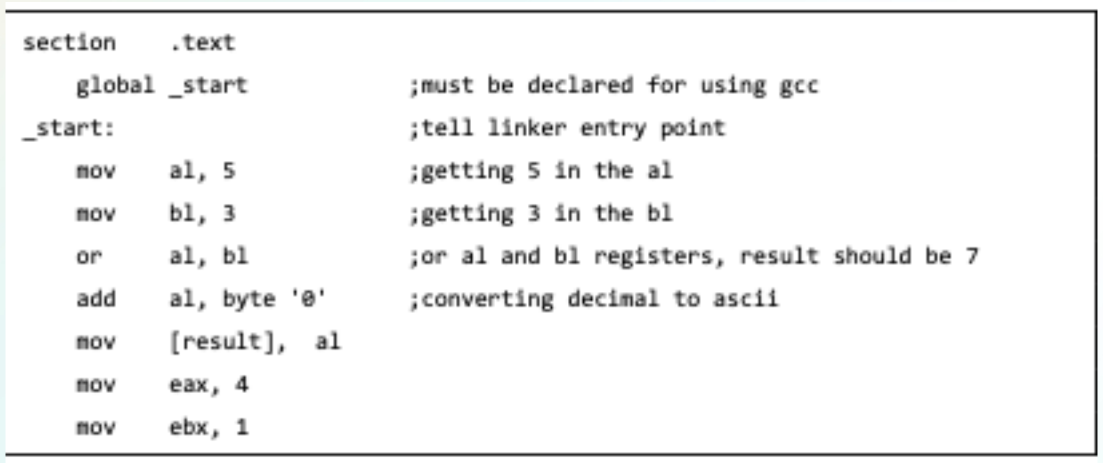
Technology Driven by Innovation

## The OR Instruction

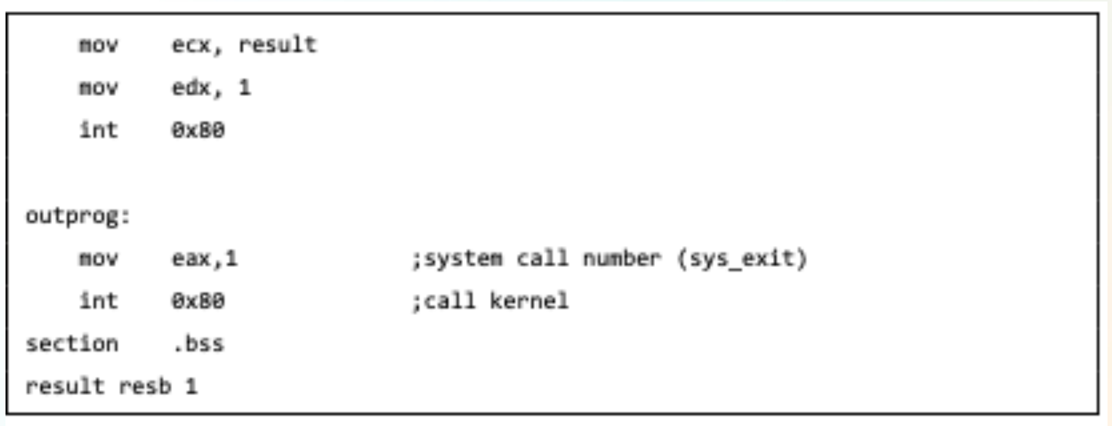
**The following example demonstrates the OR instruction. Let us store the value 5 and 3 in the AL and the BL registers, respectively, then the instruction,**

```
OR AL, BL
```

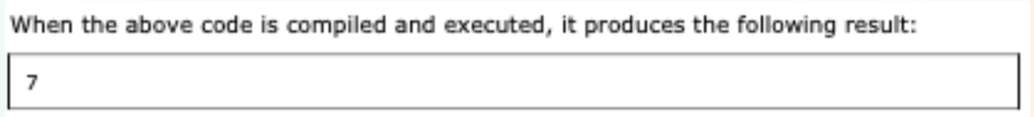
**should store 7 in the AL register:**



```
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
```

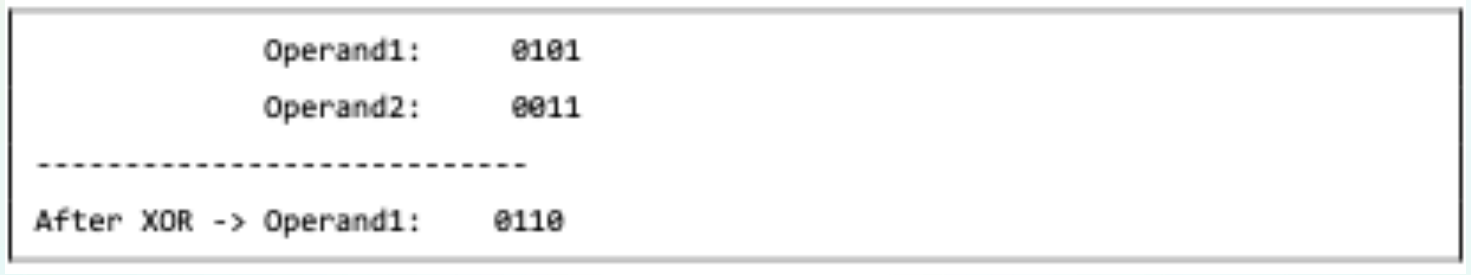
FEU ALABANG FEU DILIMAN FEU TECH

Technology Driven by Innovation

## The XOR Instruction

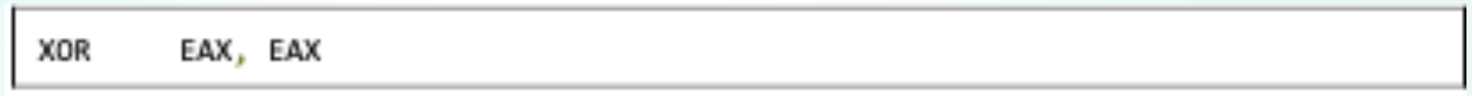
The XOR instruction implements the bitwise XOR operation. The XOR operation sets the resultant bit to 1, if and only if the bits from the operands are different. If the bits from the operands are same (both 0 or both 1), the resultant bit is cleared to 0.

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.

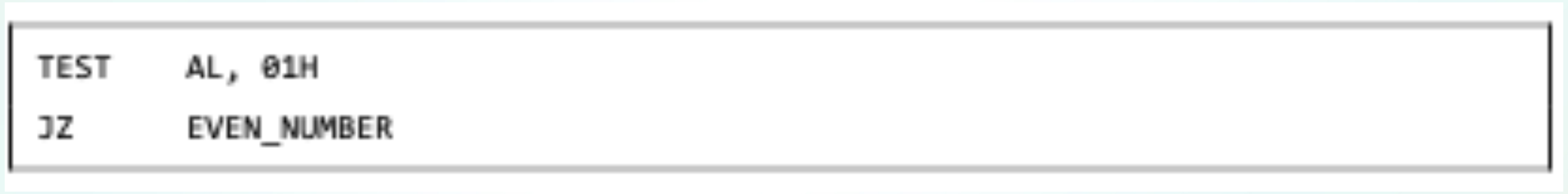


```
XOR     EAX, EAX
```

## 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.

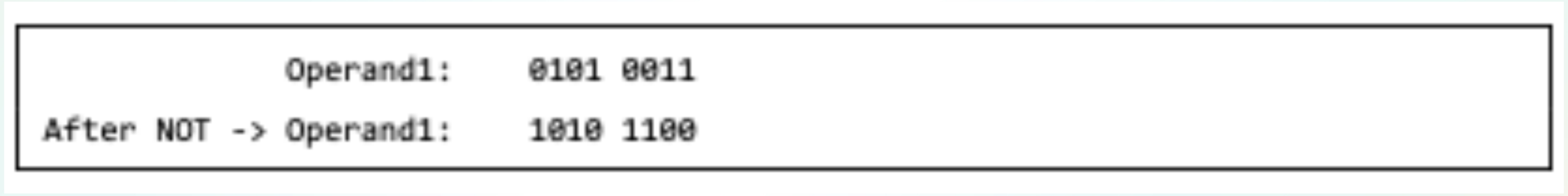


```
TEST    AL, 01H
JZ      EVEN_NUMBER
```

## 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
```