ASSEMBLY - ADDRESSING MODES

Most assembly language instructions require operands to be processed. An operand address provides the location, where the data to be processed is stored. Some instructions do not require an operand, whereas some other instructions may require one, two, or three operands.

When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source. Source contains either the data to be delivered *immediateaddressing* or the address *inregisterormemory* of the data. Generally, the source data remains unaltered after the operation.

The three basic modes of addressing are −

- Register addressing
- Immediate addressing
- Memory addressing

Register Addressing

In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

For example,

```
MOV DX, TAX_RATE   ; Register in first operand
MOV COUNT, CX     ; Register in second operand
MOV EAX, EBX     ; Both the operands are in registers
```

As processing data between registers does not involve memory, it provides fastest processing of data.

Immediate Addressing

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

For example,

```
BYTE_VALUE  DB  150     ; A byte value is defined
WORD_VALUE  DW  300     ; A word value is defined
ADD  BYTE_VALUE, 65     ; An immediate operand 65 is added
MOV  AX, 45H            ; Immediate constant 45H is transferred to AX
```

Direct Memory Addressing

When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value. This offset value is also called **effective address**.

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

In direct memory addressing, one of the operands refers to a memory location and the other operand references a register.

For example,

```
ADD BYTE_VALUE, DL ; Adds the register in the memory location
MOV BX, WORD_VALUE ; Operand from the memory is added to register
```

Direct-Offset Addressing

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data −

```
BYTE_TABLE DB  14, 15, 22, 45       ; Tables of bytes
WORD_TABLE DW  134, 345, 564, 123   ; Tables of words
```

The following operations access data from the tables in the memory into registers −

```
MOV CL, BYTE_TABLE[2] ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2 ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3] ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3 ; Gets the 4th element of the WORD_TABLE
```

Indirect Memory Addressing

This addressing mode utilizes the computer's ability of *Segment:Offset* addressing. Generally, the base registers EBX, EBP $orBX$, BP and the index registers DI, SI, coded within square brackets for memory references, are used for this purpose.

Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register.

The following code snippet shows how to access different elements of the variable.

```
MY_TABLE TIMES 10 DW 0  ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE]     ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110          ; MY_TABLE[0] = 110
ADD EBX, 2              ; EBX = EBX +2
MOV [EBX], 123          ; MY_TABLE[1] = 123
```

The MOV Instruction

We have already used the MOV instruction that is used for moving data from one storage space to another. The MOV instruction takes two operands.

Syntax

The syntax of the MOV instruction is −

```
MOV  destination, source
```

The MOV instruction may have one of the following five forms −

```
MOV  register, register
MOV  register, immediate
MOV  memory, immediate
MOV  register, memory
MOV  memory, register
```

Please note that −

- Both the operands in MOV operation should be of same size
- The value of source operand remains unchanged

The MOV instruction causes ambiguity at times. For example, look at the statements −

```
MOV  EBX, [MY_TABLE]  ; Effective Address of MY_TABLE in EBX
MOV  [EBX], 110       ; MY_TABLE[0] = 110
```

It is not clear whether you want to move a byte equivalent or word equivalent of the number 110. In such cases, it is wise to use a **type specifier**.

Following table shows some of the common type specifiers −

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Example

The following program illustrates some of the concepts discussed above. It stores a name 'Zara Ali' in the data section of the memory, then changes its value to another name 'Nuha Ali' programmatically and displays both the names.

```
section .text
   global_start     ;must be declared for linker (ld)
_start:             ;tell linker entry point

   ;writing the name 'Zara Ali'
   mov edx,9        ;message length
   mov ecx, name    ;message to write
   mov ebx,1        ;file descriptor (stdout)
   mov eax,4        ;system call number (sys_write)
   int 0x80         ;call kernel

   mov [name],  dword 'Nuha'    ; Changed the name to Nuha Ali

   ;writing the name 'Nuha Ali'
   mov edx,8        ;message length
   mov ecx,name     ;message to write
   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
name db 'Zara Ali '
```

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

```
Zara Ali Nuha Ali
```