National University of computer and emerging sciences

Computer Organization and Assembly Language (EL-2003)

Semester: Fall 2025
Section: BCS-3C
Course Instructor: Muhammad Owais

LAB # 03

Registers, Data and its Types, Definition statement & Assembly Instructions

Lab Objectives:

By the end of this lab, students will be able to:

1. Understand the role of registers as small, fast storage units in the CPU used for data manipulation during execution.
2. Learn how to define data by declaring variables or constants in memory for assembly programs.
3. Use data definition statements with directives like BYTE, WORD, and DWORD to reserve memory and specify data types.
4. Perform data initialization by assigning initial values to variables at the time of declaration.
5. Implement multiple initializations to define several values in a single statement, useful for arrays or sequences.
6. Initialize strings by declaring sequences of characters stored in memory.
7. Apply basic assembly language instructions such as MOV, ADD, and SUB for data transfer and arithmetic operations.
8. Develop and analyze a sample program demonstrating data definitions and fundamental instructions.

Basic Program Execution Registers

Introduction to Registers

A ***register*** is a high-speed storage location directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, loop counters are held in registers rather than variables. Figure 1 shows the basic program execution registers. There are **eight general-purpose registers**, **six segment registers**, a **processor status flags register (EFLAGS)**, and an **instruction pointer (EIP)**.

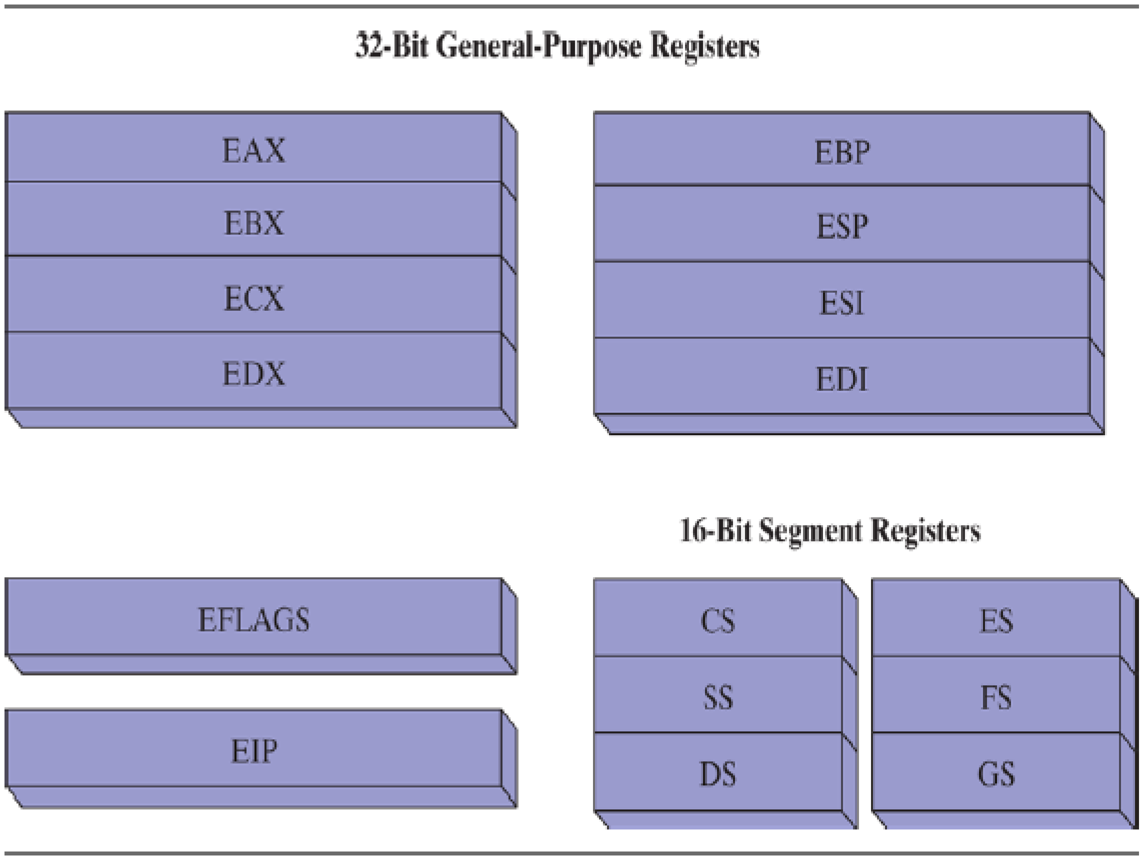


Figure 1: Basic program execution registers.

General-Purpose Registers

The general-purpose registers are primarily used for arithmetic and data movement. As shown in Figure 2, the lower 16 bits of the EAX register can be referenced by the name AX.

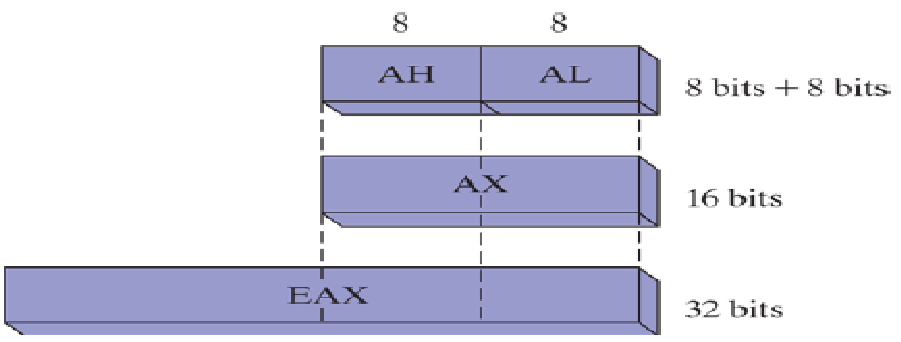
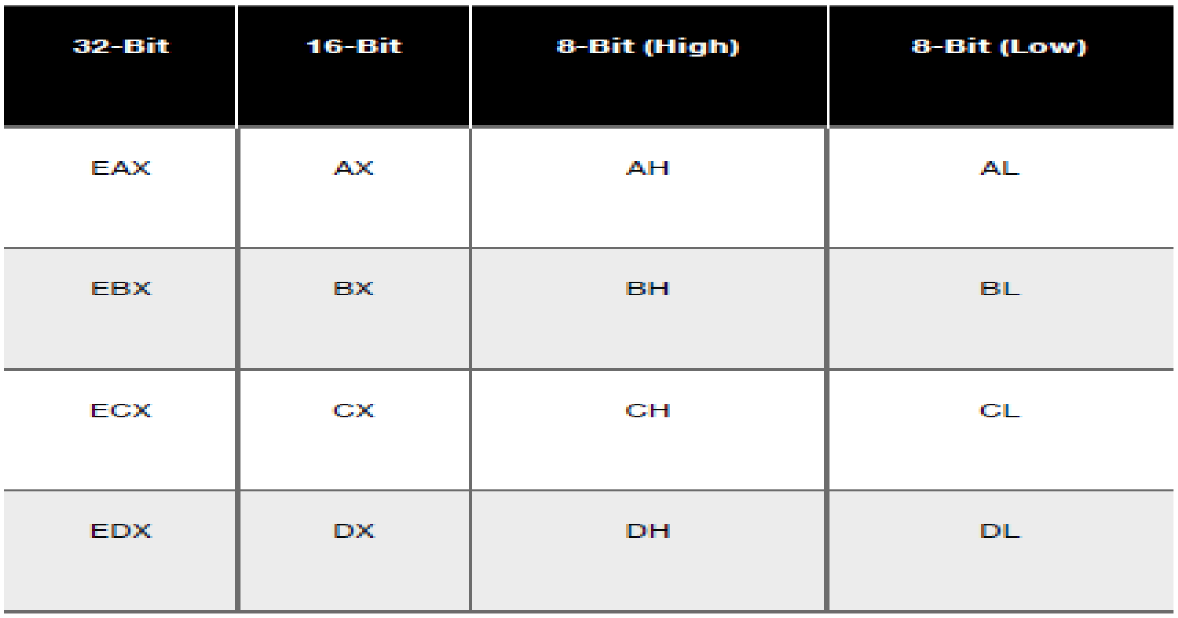


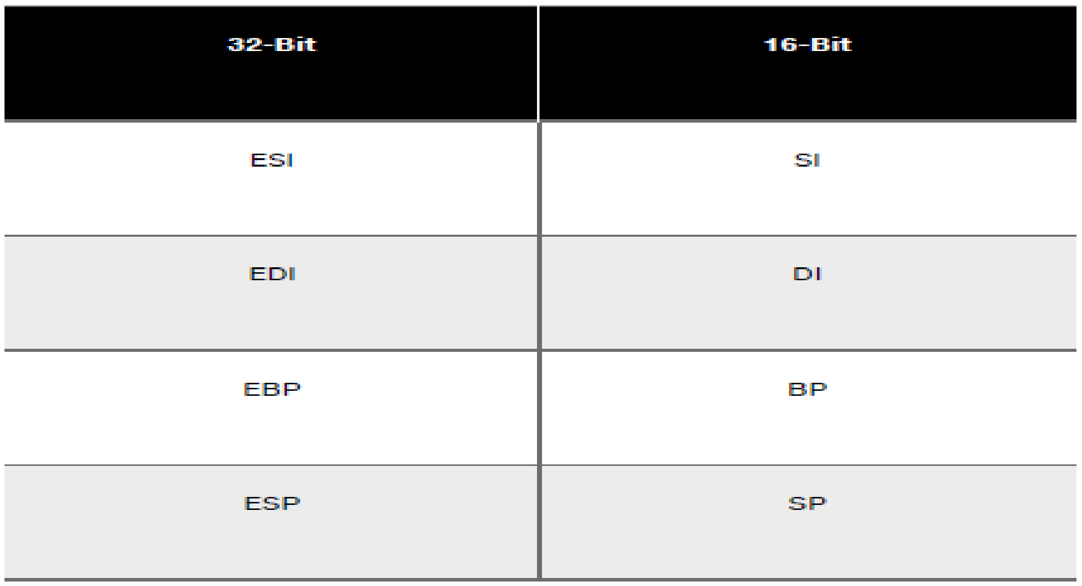
Figure 2: General-purpose registers.

Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named AH and an 8-bit lower half named AL. The same overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

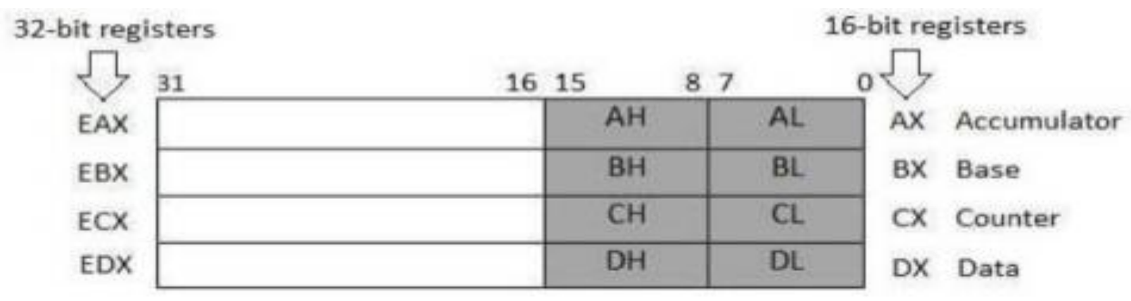


32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:



32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP



AX (Accumulator): It is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

BX (Base register): It could be used in indexed addressing / hold address of data.

CX (Counter register): The ECX, CX registers store the loop count in iterative operations.

DX (Data register): It is also used in input/output operations. It is also used with AX register along with DX for multiply and division operations involving large values. Holds data for output.

Segment Registers:

When a program load into the memory, its divide into the different segments/portions that contain data, code and stack. There are four types of segment registers.

Code Segment (CS): The CS register holds the base or starting address of all executable instructions.

Data Segment (DS): The DS register holds the base location / address of the memory variable.

Stack Segment (SS): The SS register holds the base location of the program stack.

Extra Segment (ES): The ES register is an extra base location for the memory variable.

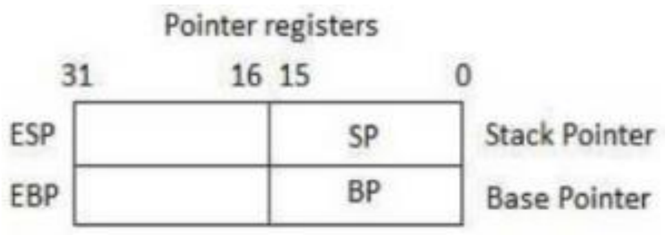
Pointer Registers

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers:

Instruction Pointer (IP): The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.

Stack Pointer (SP): The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.

Base Pointer (BP): The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

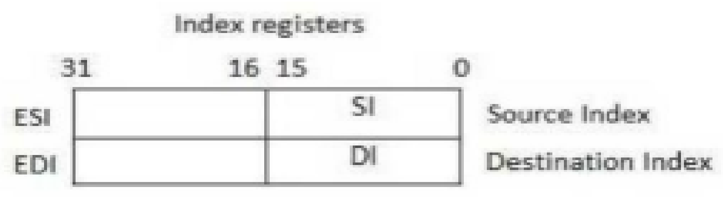


Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers.

Source Index (SI): It is used as source index for string operations.

Destination Index (DI): It is used as destination index for string operations.



Defining Data

Intrinsic Data Types:

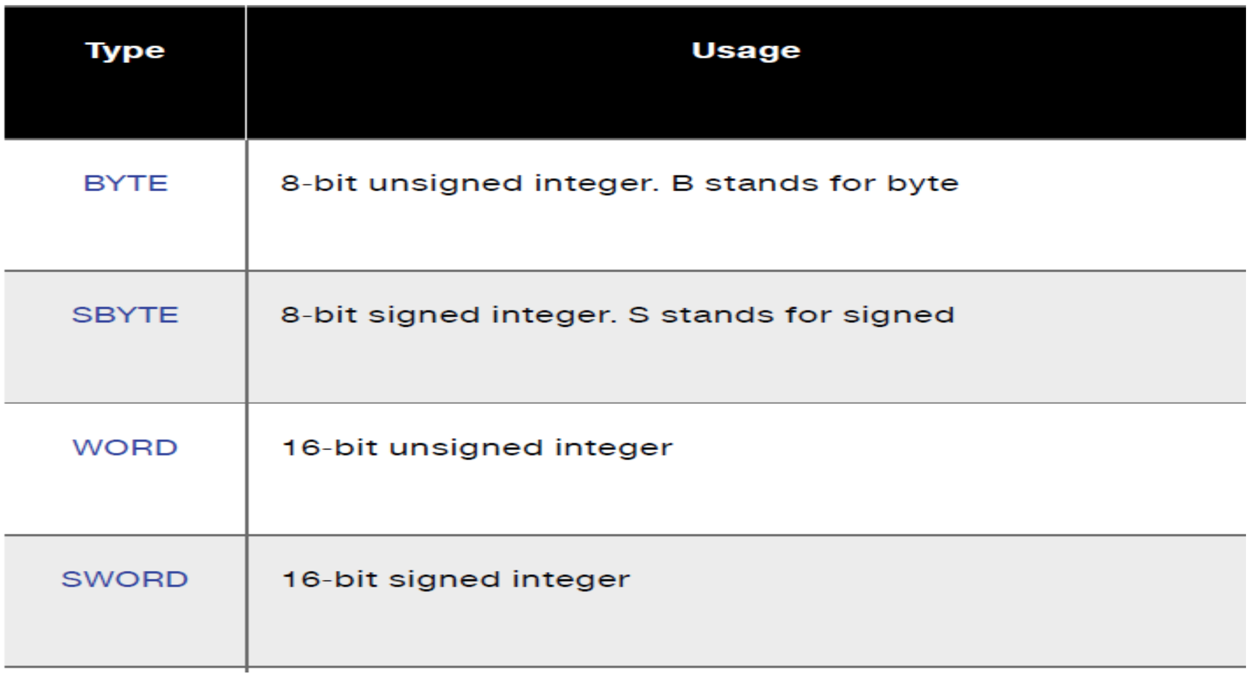
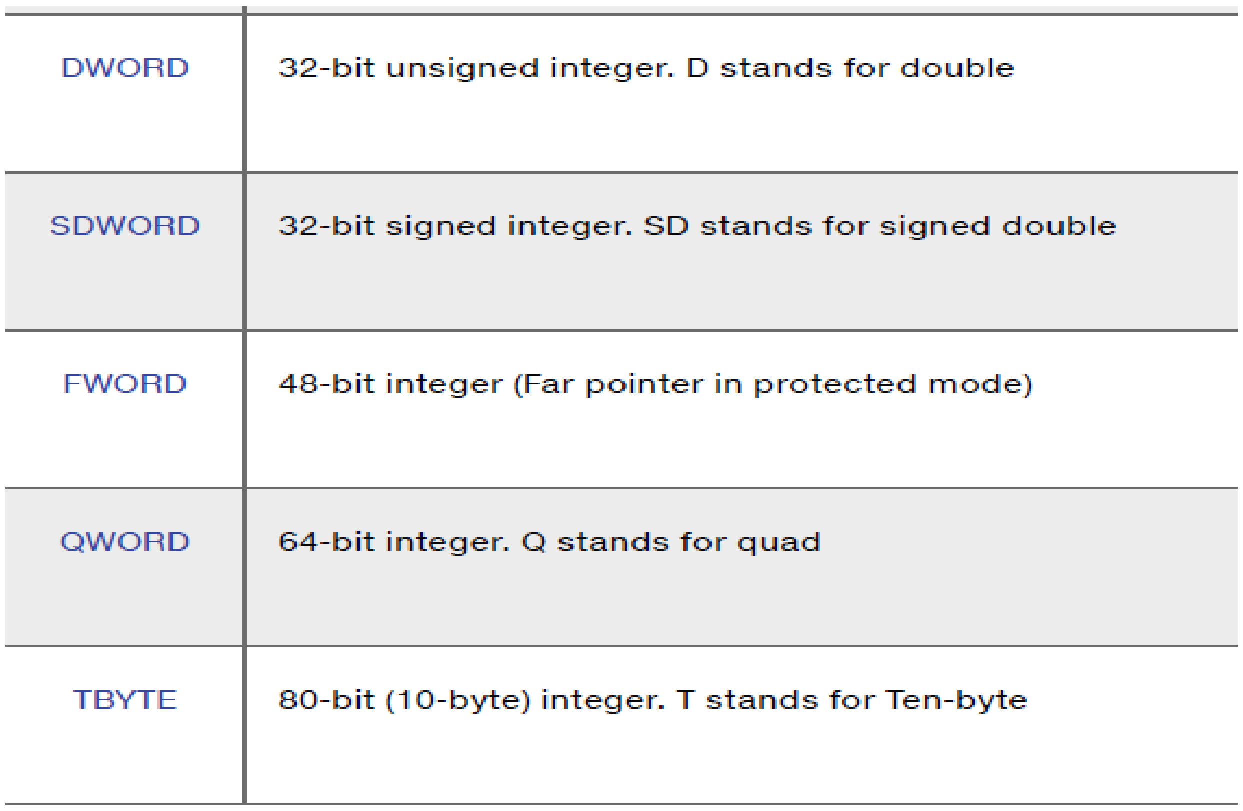
The assembler recognizes a basic set of intrinsic data types, which describe types in terms of their size (byte, word, doubleword, and so on), whether they are signed, and whether they are integers or reals. There's a fair amount of overlap in these types—for example, the DWORD type (32-bit, unsigned integer) is interchangeable with the SDWORD type (32-bit, signed integer). You might say programmers use SDWORD to communicate to readers that a value will contain a sign, but there is no enforcement by the assembler. The assembler only evaluates the sizes of operands. So, for example, you can only assign variables of type DWORD, SDWORD, or REAL4 to a 32-bit integer. In the given table contains a list of all the intrinsic data types. The notation IEEE in some of the table entries refers to standard real number formats published by the IEEE Computer Society.

Data Definition Statement:

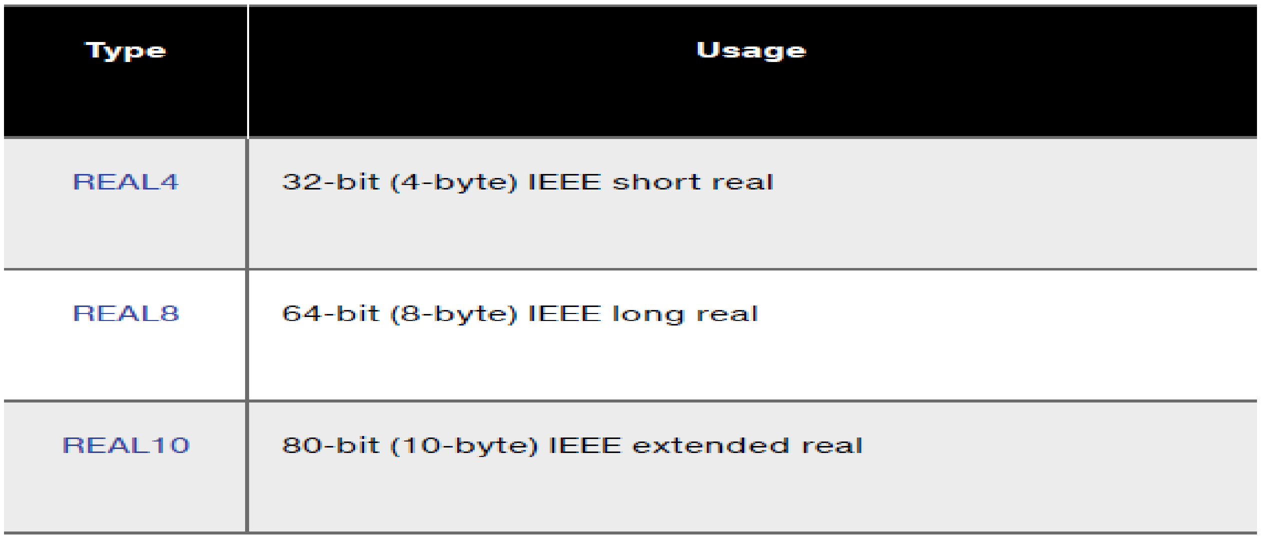
A data definition statement sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types. A data definition has the following syntax:

[name] directive initializer [,initializer]...

Initializer:
At least one initializer is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas. For integer data types, initializer is an integer constant or expression matching the size of the variable's type, such as BYTE or WORD. If you prefer to leave the variable uninitialized (assigned a random value), the ? symbol can be used as the initializer.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte



Type	Usage
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Examples:

value1 **BYTE** 'A'	; character constant
value2 **BYTE** 0	; smallest unsigned byte
value3 **BYTE** 255	; largest unsigned byte
value4 **SBYTE** −128	; smallest signed byte
value5 **SBYTE** +127	; largest signed byte
greeting1 **BYTE** "Good afternoon”, 0	; String constant
greeting2 **BYTE** 'Good night' ,0	; String constant
list **BYTE** 10,20,30,40	; Multiple initializers

Note: A question mark (?) initializer leaves the variable uninitialized, implying it will be assigned a value at runtime:

value6 **BYTE** ?

DUP Operator:

The DUP operator allocates storage for multiple data items, using an integer expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data:

Examples:

BYTE 20 DUP(0)	; 20 bytes, all equal to zero
BYTE 20 DUP(?)	; 20 bytes, uninitialized

BYTE 4 DUP("STACK")   ; 20 bytes: "STACKSTACKSTACKSTACK"

Operand Types:

As x86 instruction formats:

```
[label:] mnemonic [operands][ ; comment ]
```

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

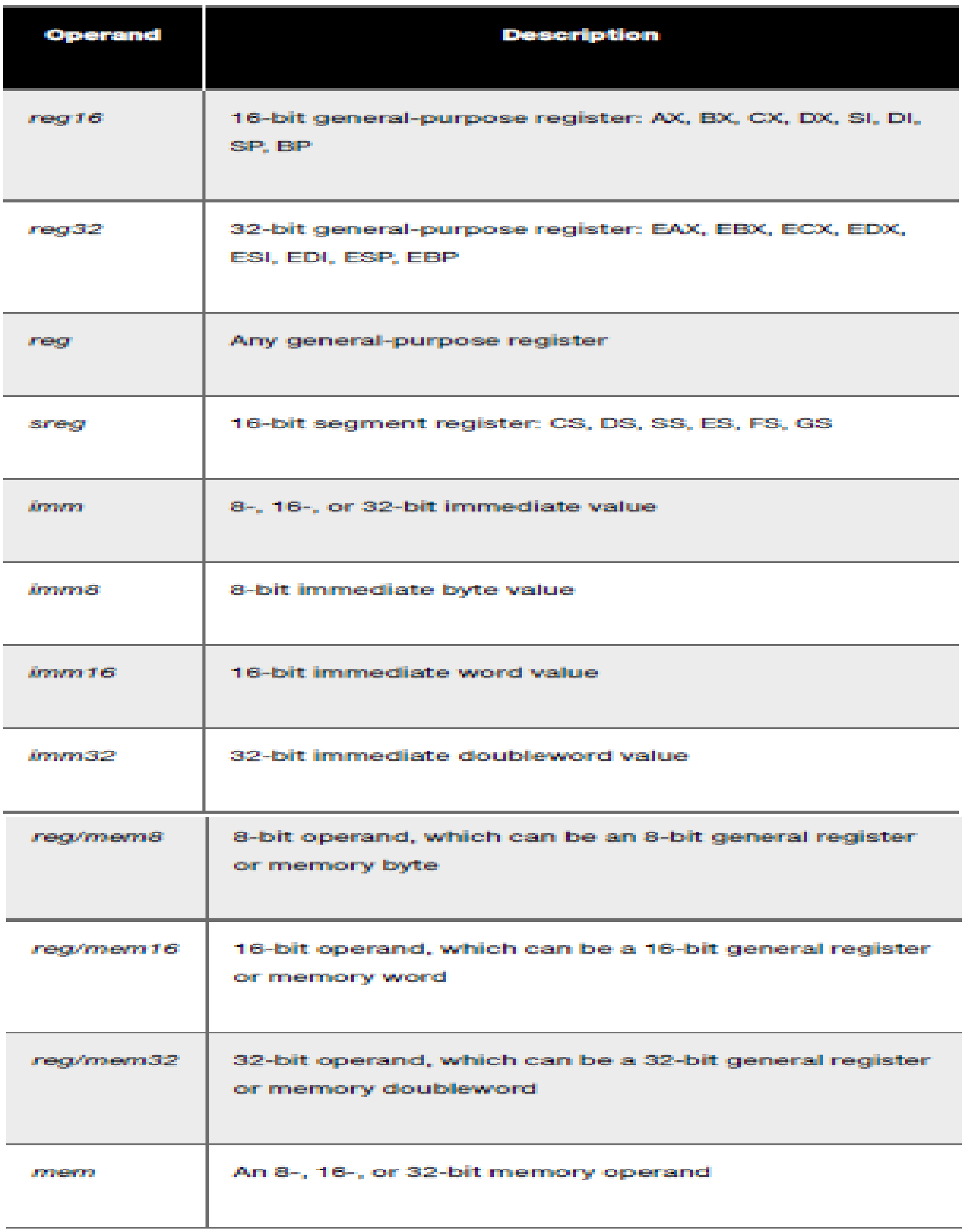
```
mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]
```

There are three basic types of operands:

- **Immediate operand** — uses a numeric or character literal expression
- **Register operand** — uses a named CPU register
- **Memory operand** — references a memory location

  Given Table describes the standard operand types. It uses a simple notation for operands (in 32-bit mode) freely adapted from the Intel manuals. We will use it from this point on to describe the syntax of individual instructions.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL



Operand	Description
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

MOV Instruction:

The MOV instruction copies data from a source operand to a destination operand. Known as a data transfer instruction, It is used in virtually every program. Its basic format shows that the first operand is the destination and the second operand is the source:

```
MOV destination,source
```

The destination operand's contents change, but the source operand is unchanged. The right to left movement of data is similar to the assignment statement in C++ or Java:

```
dest = source;
```

In nearly all assembly language instructions, the left-hand operand is the destination and the right-hand operand is the source. MOV is very flexible in its use of operands, as long as the following rules are observed:

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer register (IP, EIP, or RIP) cannot be a destination operand.

Here is a list of the standard MOV instruction formats:

```
MOV  reg,reg
MOV  mem,reg
MOV  reg,mem
MOV  mem,imm
MOV  reg,imm
```

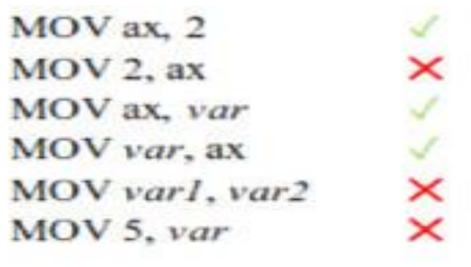
Example:
MOV bx, 2
MOV ax, cx

Example:
'A' has ASCII code 65D (01000001B, 41H)

The following MOV instructions stores it in register BX:

MOV bx, 65d
MOV bx, 41h
MOV bx, 01000001b
MOV bx, 'A'
All of the above are equivalent.

Examples:
The following examples demonstrate compatibility between operands used with MOV instruction:



MOV ax, 2	✓
MOV 2, ax	✗
MOV ax, *var*	✓
MOV *var*, ax	✓
MOV *var1*, *var2*	✗
MOV 5, *var*	✗

ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. Source is unchanged by the operation, and the sum is stored in the destination operand
Syntax:
ADD dest, source

SUB Instruction

The SUB instruction subtracts a source operand from a destination operand.
Syntax:
SUB dest, source

Sample Program:

```
TITLE Add and Subtract (AddSub.asm)
; This program adds and subtracts 32-bit integers.
INCLUDE Irvine32.inc
.code
main PROC
mov eax,10000h        ; EAX = 10000h
add eax,40000h        ; EAX = 50000h
sub eax,20000h        ; EAX = 30000h
call DumpRegs         ; display registers
exit
main ENDP
END main
```

LAB TASKS

Task # 01:

In the .data section, declare the following variables:

- A signed BYTE variable snum initialized with **-100**.
- An unsigned BYTE variable unum initialized with **200**.
- An uninitialized BYTE variable holder.

Task # 02:

Define a string variable called message initialized with "Hello World".

Task # 03:

Declare a variable named `num1` initialized with 500.
Declare another variable named `num2` initialized with 200.
Write assembly instructions to:

1. Load both into registers.
2. Add them.
3. Store the result in EAX.

Task # 04:

Write assembly code to load the following immediate values into the register EBX:

- Decimal value **90**
- Hexadecimal value **5Ah**
- Binary value **01011010b**
- ASCII character ‘B’

All four should load the same value in EBX.

Task # 05:

Write a small assembly program to compute:

EAX = 30000h + 60000h – 10000h

Task # 06:

In the .data section:

Create an array named marks of **8 bytes**, each initialized with 5.

Reserve **5 words** (2 bytes each) uninitialized.