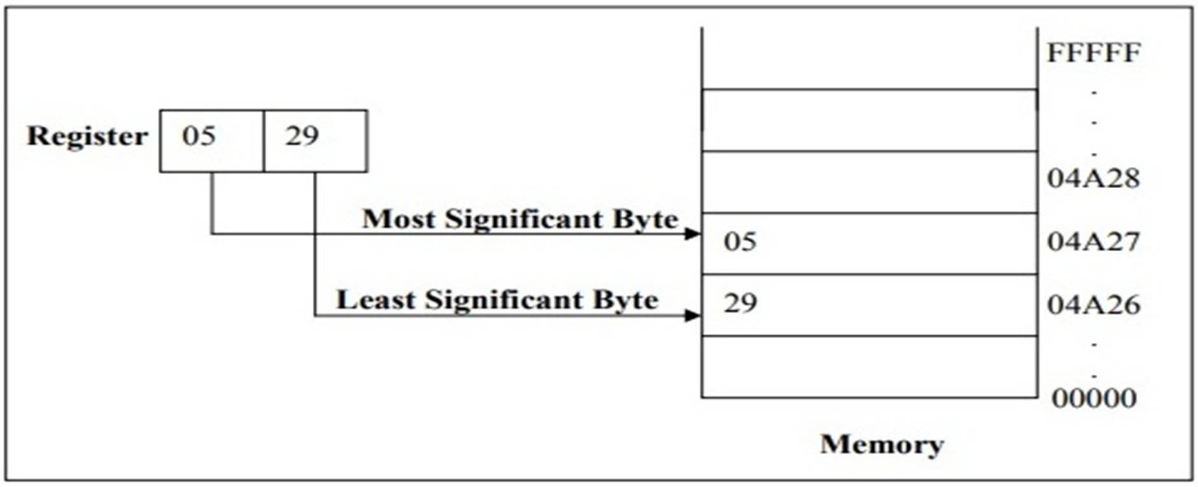
Chapter Two Addressing Data Memory

Addressing Data Memory

Depending on the model, the processor can access one or more bytes of memory at a time. Consider the Hexa value **(0529H)** which requires two bytes or one word of memory. It consist of high order **(most significant)** byte 05 and a low order **(least significant)** byte 29.

The processor store the data in memory in reverse byte sequence i.e. the low order byte in the low memory address and the high order byte in the high memory address. For example, the processor transfer the value **0529H** from a register into memory address **04A26 H** and **04A27H** like this:



The processor expects numeric data in memory to be in reverse byte sequence and processes the data accordingly, again reverses the bytes, restoring them to correctly in the register as hexa **0529H**. When programming in assembly language, you have to distinguish between the address of a memory location and its contents. In the above example the content of address **04A26H** is 29, and the content of address **04A27H** is 05.

There are two types of addressing schemes:

1. **<u>An Absolute Address</u>**, such as **<u>04A26H</u>**, is a 20 bit value that directly references a specific location.

2. **<u>A Segment Offset Address</u>**, combines the starting address of a segment with an offset value.

Segments and Addressing

Segments are special area defined in a program for containing the code, the data, and the stack. **<u>Segment Offset</u>** within **<u>a</u>** **<u>program</u>**, all memory locations within a segment are relative to the segment starting address. The distance in bytes from the segment address to another location within the segment is expressed as an **<u>offset</u>** (or displacement).

To reference any memory location in a segment, the processor combine the segment address in a segment register with the offset value of that location, that is, its distance in byte from the start of the segment.

Specifying addresses

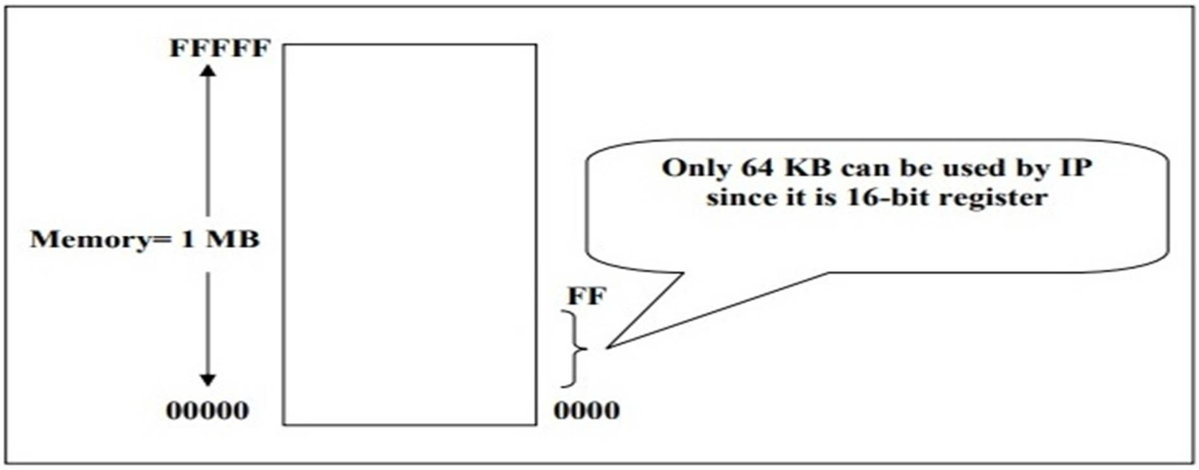
To represent a segment address and its relative offset we use the notation:

<u>Segment: offset</u>

Thus **<u>020A:1BCD</u>** denotes offset **<u>1BCDH</u>** from segment **<u>020AH</u>**. The actual address it refers to is obtained in the following way:

- Add zero to the right hand side of the segment address.
- Add to this the offset.

Hence the actual address referred to by 020A:1BCD is 03C6D.



Address Bus in the **8086** is 20 bits wide (20 lines) i.e. the processor can access memory of size 220 or 1048576 bytes **(1MB).**

Instruction Pointer = 16 bit register which means the processor can only address 0 – 216 (65535) bytes of memory. But we need to write instructions in any of the 1MB of memory. This can be solved by using memory segmentation., where each segment register is 16-bit (this 16-bit is the high 16-bit of Address Bus (A4- A19)) i.e. each of the segment registers represent the actual address after shifting the address 4-bit to get 20 bits.

Registers

Registers are 8, 16, or 32-bit high speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

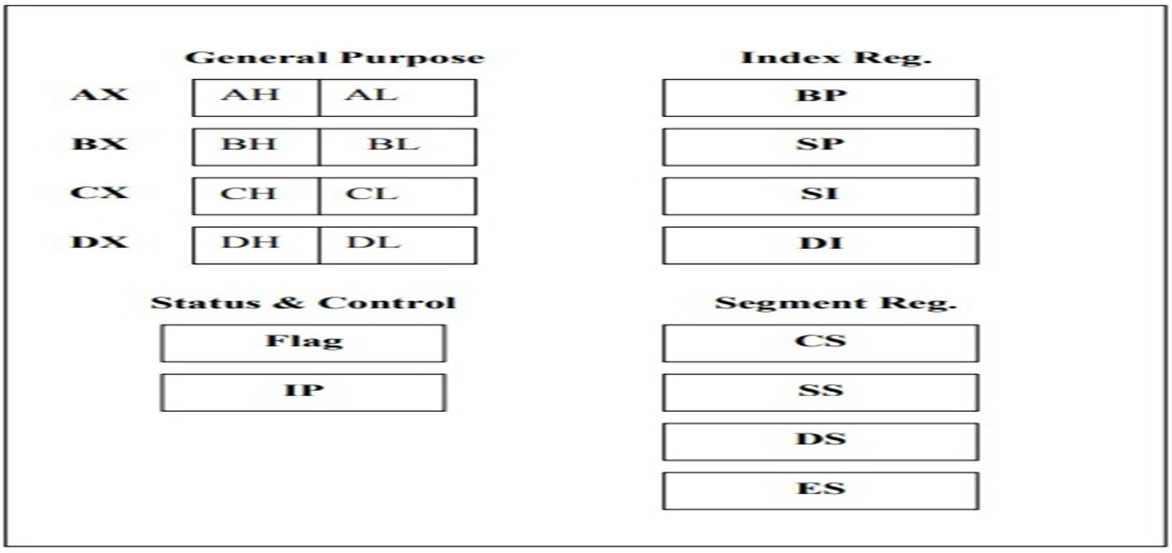


Figure 7: Intel 16-bit registers

The CPU has an internal data bus that is generally twice as wide as its external data bus.

Data Registers: The general purpose registers, are used for arithmetic and data movement. Each register can be addressed as either 16-bit or 8 bit value. Example, **AX** register is a 16-bit register, its upper 8-bit is called **AH**, and its lower 8-bit is called AL. Bit 0 in AL corresponds to bit 0 in **AX** and bit 0 in AH corresponds to bit 8 in AX. See Figure 8.

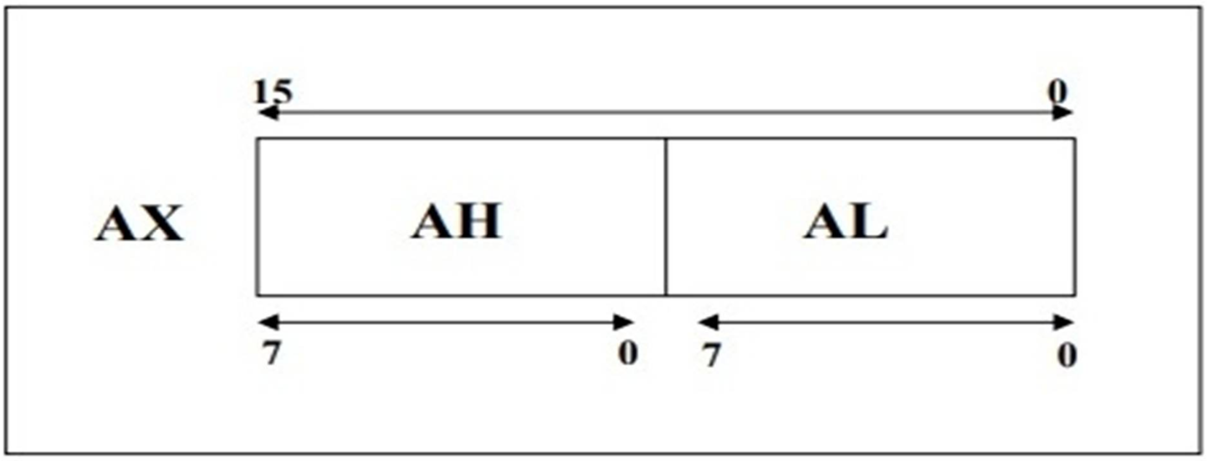


Figure 8: AX register

Instructions can address either 16-bit data register as AX, BX, CX, and DX or 8-bit register as AL, AH, BL, BH, CL, CH, Dl, and DH. If we move 126FH to AX then AL would immediately 6FH and AH = 12H.

General Purpose Register

Each general purpose register has special attributes:

- ❑ **AX (Accumulator):** AX is the accumulator register because it is favored by the CPU for arithmetic operations. Other operations are also slightly more efficient when performed using **AX**.
- ❑ **BX (Base):** the BX register can hold the address of a procedure or variable. Three other registers with this ability are **SI**, **DI** and **BP**. The **BX** register can also perform arithmetic and data movement.
- ❑ **CX (Counter):** the CX register acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement **CX**
- ❑ **DX (Data):** the DX register has a special role in multiply and divide operation. When multiplying for example **DX** hold the high 16 bit of the product.

Segment Registers

Segment Registers: the CPU contain four segment registers, used as base location for program instruction, and for the stack.

- ❖ **CS (Code Segment):** The code segment register holds the base location of all executable instructions (code) in a program.

- **DS (Data Segment):** the data segment register is the default base location for variables. The CPU calculates their location using the segment value in DS.
- **SS (Stack Segment):** the stack segment register contain the base location of the stack.
- **ES (Extra Segment):** The extra segment register is an additional base location for memory variables.

Index Registers

Index registers contain the offset of data and instructions. The term offset refers to the distance of a variable, label, or instruction from its base segment. The index registers are:

- ✓ **BP (Base Pointer):** the **BP** register contain an assumed offset from the stack segment register, as does the stack pointer. The base pointer register is often used by a subroutine to locate variables that were passed on the stack by a calling program.
- ✓ **SP (Stack Pointer):** the stack pointer register contain the offset of the top of the stack. The stack pointer and the stack segment register combine to form the complete address of the top of the stack.
- ✓ **SI (Source Index):** This register takes its name from the string movement instruction, in which the source string is pointed to by the source index register.
- ✓ **DI (Destination Index):** the **DI** register acts as the destination for string movement instruction

Status and Control Register

1. **IP (Instruction Pointer):** The instruction pointer register always contain the offset of the next instruction to be executed within the current code segment. The instruction pointer and the **code segment** register combine to form the complete address of the next instruction.
2. **The Flag Register:** is a special register with individual bit positions assigned to show the status of the CPU or the result of arithmetic operations. The Figure9 describe the **8086/8088** flags register:

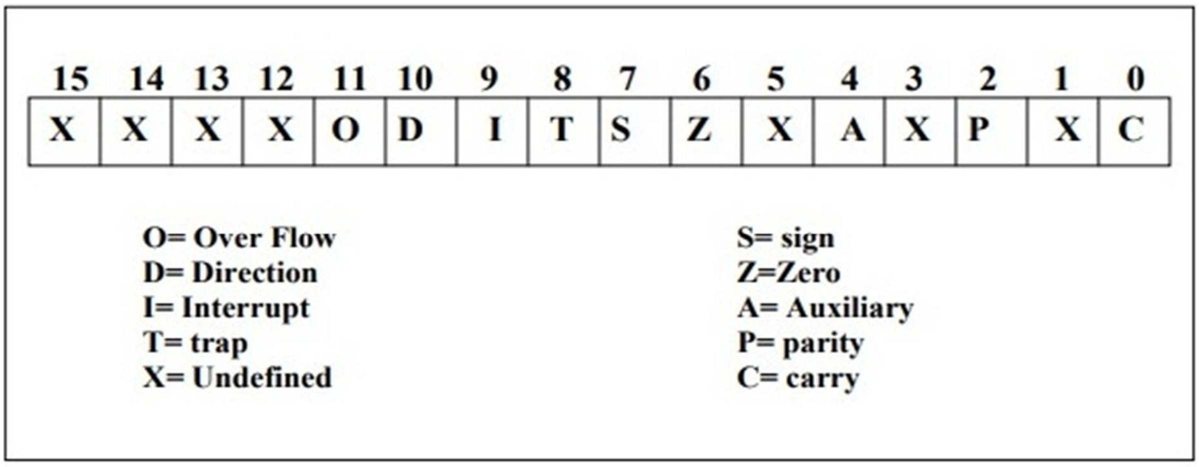


Figure 9: Flag Register

There Two Basic Types of Flags

There two basic types of flags: (control flags and status flags)

1. **Control Flags:** individual bits can be set in the flag register by the programmer to control the CPU operation , these are

- **The Direction Flag (DF):** affects block data transfer instructions, such as **MOVS**, **CMPS**, **SCAS**. The flag values are **0 = up** and **1 = down**.

- **The Interrupt flag (IF):** dictates whether or not a system interrupt can occur. Such as keyboard, disk drive, and the system clock timer. A program will sometimes briefly disable the interrupt when performing a critical operation that cannot be interrupted. The flag values are **1 = enable**, **0 = disable**.

- **The Trap flag (TF):** Determine whether or not the CPU is halted after each instruction. When this is set, a debugging program can let a programmer to enter single stepping (trace) through a program one instruction at a time. The flag values are **1 = on**, **0 = off**. The flag can be set by **INT 3** instruction.

2. Status Flags: The status flags reflect the outcomes of arithmetic and logical operations performed by the CPU, these are:

- **The Carry Flag (CF):** is set when the result of an unsigned arithmetic operation is too large to fit into the destination for example, if the sum of 71 and 99 where stored in the 8-bit register AL, the result cause the carry flag to be 1. The flag **values = 1 = carry**, **0 = no carry**.

- **The Overflow (OF):** is set when the result of a signed arithmetic operation is too wide (too many bits) to fit into destination. **1 = overflow**, **0 = no overflow.**

- **Sign Flag (SF):** is set when the result of an arithmetic of logical operation generates a negative result, **1= negative**, **0 = positive**.

- **Zero Flag (ZF):** is set when the result of an arithmetic of logical operation generates a result of zero, the flag is used primarily by jump or loop instructions to allow branching to a new location in a program based on the comparison of two values. The flag **value = 1 = zero**, **& 0 = not zero**.

- **Auxiliary Flag:** is set when an operation causes a carry from bit 3 to bit 4 (or borrow from bit 4 to bit 3) of an operand. The flag **value = 1 = carry**, **0 = no carry**.
- **Parity Flag:** reflect the number of 1 bits in the result of an operation. If there is an **even number** of bit, the parity is even. If there is an **odd number** of bits, parity is **odd**. This flag is used by the OS to verify memory integrity and by communication software to verify the correct transmission of data.

Instruction Execution and Addressing

An assembly language programmer writhe a program in **symbolic code** and uses the **assembler** to translate it into machine code as .EXE program. For program execution, the system looks only the machine code into memory.

Every instruction consists of at least one **operation**, such as MOV, ADD. Depending on the operation, an instruction may also have one or more **operands** that reference the data the operation is to process.

The Basic Steps the Processor

The basic steps the processor takes in executing on instruction are:

1. **Fetch the next instruction** to be executed from memory and place it in the instruction queue.
2. **Decode the instruction** calculates addressed that reference memory, deliver data to the Arithmetic Logic Unit, and increment the instruction pointer **(IP)** register.

3. **<u>Execute the instruction</u>**, performs the request operation, store the result in a register or memory, and set flags such as zero or carry where required.

For an .EXE program the **<u>CS</u>** register provide the address of the beginning of a program code segment, and **<u>DS</u>** provide the address of the beginning of the data segment.

The **<u>CS</u>** contains instructions that are to be executed, where as the **<u>DS</u>** contain data that the instruction reference. The **<u>IP</u>** register indicates the offset address of the current instruction in the **<u>CS</u>** that is to be executed. An instruction operand indicates on offset address in the **<u>DS</u>** to be referenced.

Consider and example in which the program loader has determined that it is to be load on .EXE program into memory beginning at location **<u>05BE0H</u>**. The loader accordingly initialize CS with segment address **<u>05BE[0]H</u>** and **<u>IP</u>** with zero.

<u>CS:</u> **<u>IP</u>** together determine the address of the first instruction to execute **<u>05BE0H + 0000H = 05BE0H</u>**. In this way the first instruction in CS being execution, if the first instruction is two byte long, the processor increment **<u>IP</u>** by **<u>2</u>**, so that , the next instruction to be executed is **<u>05BE0H + 0002H = 05BE2H</u>**.

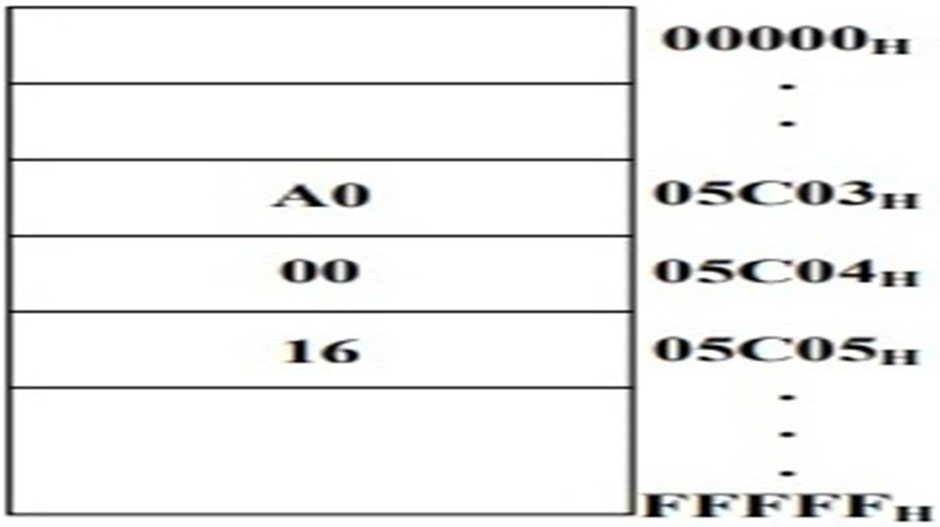
Assume the program continues executing, and **<u>IP</u>** contain the offset **<u>0023H.</u>** **<u>CS: IP</u>** now determine the address of the next instruction to execute, as follows:

CS address:	**05BE0H**
IP offset:	**0023H +**
Instruction address:	**05C03H**

EX: let's say that **MOV** instruction beginning at **0FC03H** copies the content of a byte in memory into the **AL** register. The byte is at offset **0016H** in the **DS**. Her are the machine code and the symbolic code for this operation.

Address	Symbolic Code	MIC code
0FC03	MOV AL, [0016]	A0 1600

Address **0FC03H** contain the first byte **(A0H)** of the MIC code instruction the processor is to access



The second and third byte contains the offset value in reversed byte sequence. In symbolic code, the operand **[0016]** in square brackets (an index operator) indicates an offset value to distinguish it from the actual storage address 16. Lest say that the program has initialized the **DS** register with **DS** address **05D1[0]H**. To access the data item, the processor determines its location from the segment address in **DS** + the offset **(0016H)** in the instruction. Operand become **DS** contain **0FD1[0]H**, the actual location of the reference data item is

DS:	**05D10H**	
Offset:	**0016H**	+
Address of data item:	**05D26H**	

Assume the address **05D26H** contain **4AH**, the processor now extract the **4AH** at address **05D26H** and copy it into **AL** register.

An instruction may also access more than one byte at a time

EX: Suppose an instruction is to store the content of the **AX** register **(0248H)** in two adjacent byte in the **DS** beginning at offset **0016H**.

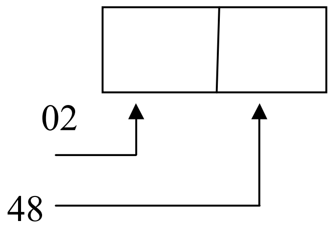
The symbolic code **MOV [0016], AX**

The processor stores the two byte in memory in revered byte sequence as

Content of AX:	02	48
Offset in DS:	0017	0016

Another instruction, **MOV AX, [0016]**, subsequently could retrieve these byte by copy them from memory back into **AX**.

The operation reverses (and corrects) the byte in **AX** as:



Number of Operands

Operands specify the value an instruction is to operate on, and where the result is to be stored. Instruction sets are classified by the number of operands used. An instruction may have no, one, two, or three operands.

1. **Three-Operand Instruction:** In instruction that have three operands, one of the operand specifies the destination as an address where the result is to be saved. The other two operands specify the source either as addresses of memory location or constants.

EX: **A=B+C**

ADD destination, source1, source2

ADD A,B,C

EX: **Y=(X+D)* (N+1)**

ADD T1, X,

D ADD T2, N, 1

Mul Y, T1, T2

2.Two operand instruction : In this type both operands specify sources. The first operand also specifies the destination address after the result is to be saved. The first operand must be an address in memory, but the second may be an address or a constant.

ADD destination, source

EX: A=B+C

MOV A, B

ADD A, C

EX: Y=(X+D)* (N+1)

```
MOV T1, X
ADD T1, D
MOV Y, N
ADD Y, 1
MUL Y, T1
```

3. <u>One Operand instruction:</u> Some computer have only one general purpose register, usually called on Acc. It is implied as one of the source operands and the destination operand in memory instruction the other source operand is specified in the instruction as location in memory.

ADD source

<u>LDA</u> source; copy value from memory to ACC.

<u>STA</u> destination; copy value from Acc into memory.

EX: A=B+C

```
LDA B
ADD C
STA A
```

EX: Y=(X+D)* (N+1)

```
LDA X
ADD D
STA T1
LDA N
ADD 1
MUL T1
STA Y
```

4. <u>Zero Operand instruction:</u> Some computers have arithmetic instruction in which all operands are implied, these zero operand instruction use a stack, a stack is a list structure in which all insertion and deletion occur at one end, the element on a stack may be removed only in the reverse of the order in which they were entered. The process of inserting an item is called **Pushing**, removing an item is called **<u>Popping</u>**.

Computers that use Zero operand instruction for arithmetic operations also use one operand **<u>PUSH</u>** and **<u>POP</u>** instruction to copy value between memory and the stack.

<u>PUSH source</u>; Push the value of the memory operand onto the Top of the stack.

<u>POP destination</u>; POP value from the Top of the stack and copy it into the memory operand.

EX: A=B+C

PUSH B

PUSH C

ADD;

POP A

Pop the two value of the stack, add them, and then push the sum back into the stack

EX: Y=(X+D)* (N+1)

```
 PUSH X
 PUSH D
 ADD
PUSH N
PUSH 1
ADD
MUL
POP Y
```

Assembly Language Instruction

Assembly Language Instruction Assembly language instructions are provided to describe each of the basic operations that can be performed by a microprocessor. They are written using **alphanumeric symbols** instead of the 0s and 1s of the microprocessor's machine code. Program written in assembly language are called **source code**. An assembly language description of this instruction is

```
ADD AX, BX
```

In tins example, the contents of BX and AX are added together and their sum is put in AX. Therefore, BX is considered to be the **source operand** and AX the **destination operand**.

Here is another example of an assembly language statement:

```
LOOP:    MOV AX, BX ;    COPY BX INTO AX
```

This instruction statement starts with the word LOOP. It is an address identifier for the instruction MOV AX, BX. This type of identifier is called a **label** or **tag**. The instruction is followed by "COPY BX INTO AX." This part of the statement is called **a comment**. Thus a general format for writing and assembly language statement is:

```
LABEL:     INSTRUCTION ;       COMMENT
```