Microprocessors Techniques I

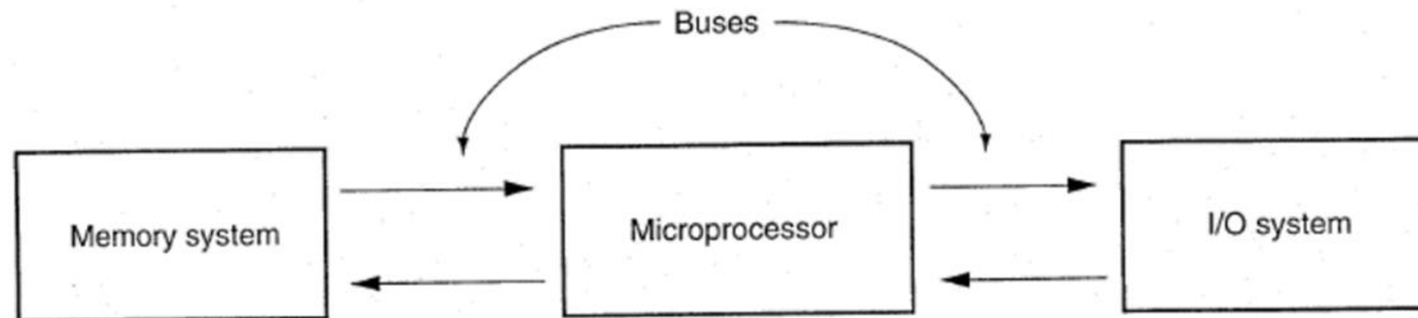
Introduction

A ***Microprocessor*** incorporates most or all of the functions of a computer's central processing unit (CPU) on a single integrated circuit. It is the part of the microcomputer that executes instructions of the program and processes data. It is responsible for performing all arithmetic operations and making the logical decisions initiated by the computer’s program. In addition to arithmetic and logic functions, the MPU controls overall system operation.

Microcomputer System

The hardware of a microcomputer system can be divided into four functional sections:

- ***Input unit***
- ***Microprocessing Unit***
- ***Memory Unit***
- ***Output Unit***.

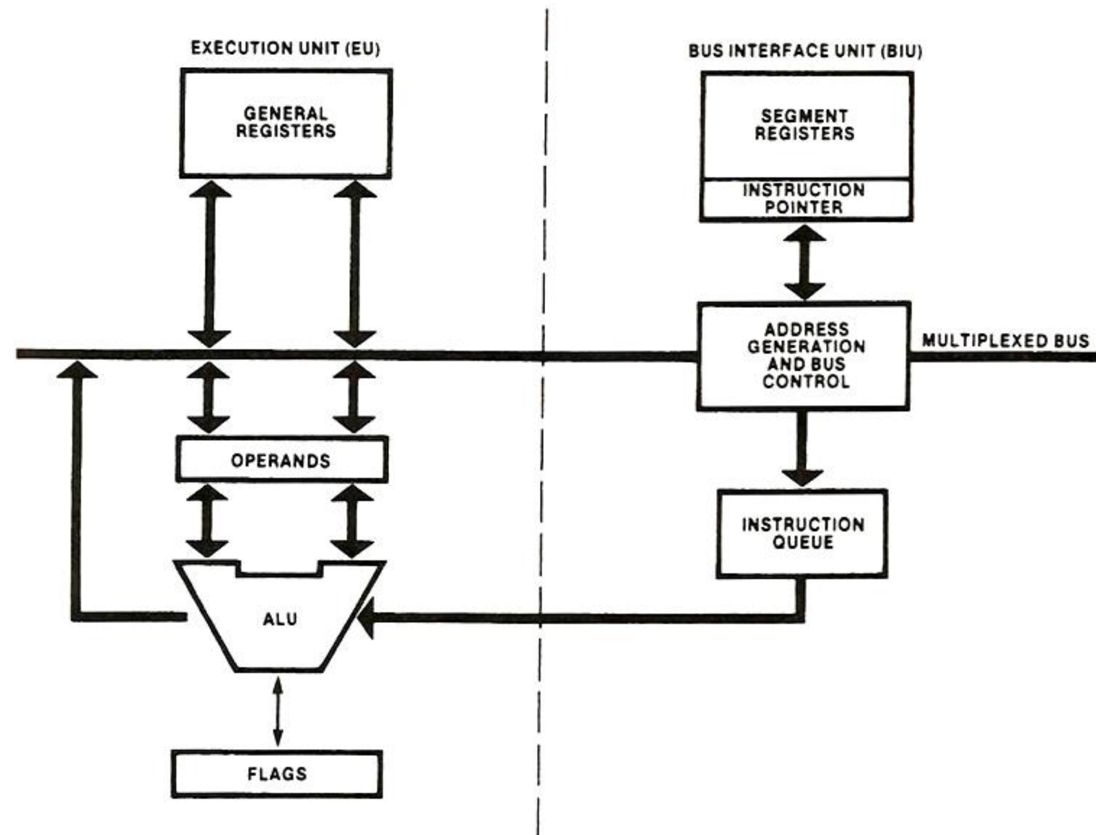


Software Architecture

The 8086 microprocessor employs parallel processing – that is, it is implemented with several simultaneously operating processing units. It contains two processing units: the ***Bus Interface Unit*** (BIU) and the ***Execution Unit*** (EU).

Each unit has dedicated functions and both operate at the same time. This parallel processing makes the fetch and execution of instructions independent operations. This results in efficient use of the system bus and higher performance for the microcomputer system.

Execution & Bus Interface Units



Bus Interface Unit

The BIU is responsible for performing all external bus operations, such as:

- Instruction fetching
- Reading and writing of data operands for memory
- Address generating
- Inputting or outputting data for input/output peripherals
- instruction queuing

These operations take place over the system bus. This bus includes 16-bit bidirectional data bus, a 20-bit address bus, and the signals needed to control transfer over the bus.

The BIU uses a mechanism known as ***instruction queue*** to implement a pipelined architecture. This queue permits the 8086 to prefetch up to 6 bytes of instruction code. Whenever the queue is not full, the BIU is free to look ahead in the program by prefetching the next sequential instructions.

Execution Unit

The EU is responsible for two tasks:

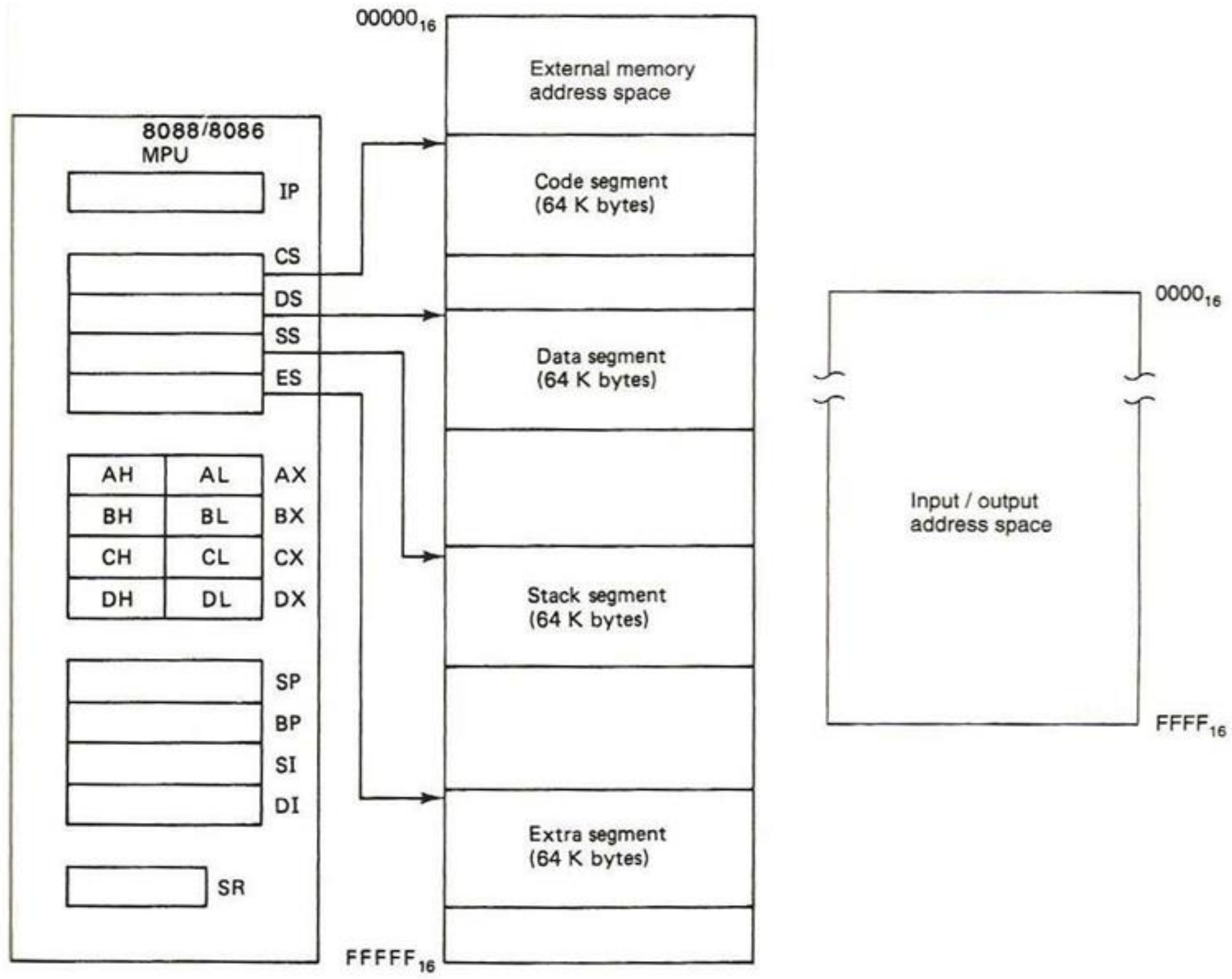
- Decoding instructions
- Executing instructions

It accesses instructions from the output end of the instruction queue and data from the general-purpose registers or memory. It reads one instruction byte after the other from the output of the queue, decodes them, generates data addresses if necessary, passes them to the BIU and requests it to perform the read or write operations to memory or I/O, and perform the operation specified by the instruction.

Software Model

The software model of 8086 includes 13 16-bit internal registers: the instruction pointer, four data registers, two pointer registers, two index registers, and four segment registers. In addition, there is status register with nine of its bits implemented as status & control flags.

The 8086 architecture implements independent memory and input/output spaces; the memory address space is 1048576 bytes (1Mbyte) in length and I/O address space is 65536 bytes (64Kbyte) in length.


00000₁₆
External memory
address space
8088/8086
MPU
IP
CS
DS
SS
ES
Code segment
(64 K bytes)
Data segment
(64 K bytes)
Stack segment
(64 K bytes)
Extra segment
(64 K bytes)
AH
AL
AX
BH
BL
BX
CH
CL
CX
DH
DL
DX
SP
BP
SI
DI
SR
FFFFF₁₆
0000₁₆
Input / output
address space
FFFF₁₆

Memory Segmentation

Even though the 8086 has a 1Mbyte address space, not all this memory is active at one time. Actually, the 1Mbytes of memory are partitioned into 64Kbyte (65,536) segments.

A ***segment*** represents an independently addressable unit of memory consisting of 64 K consecutive byte-wide storage locations. Each segment is assigned a base address that identifies its starting point (its lowest address byte-storage location).

Only four of these 64 Kbytes segments are active at a time: the code segment, stack segment, data segment, and extra segment.

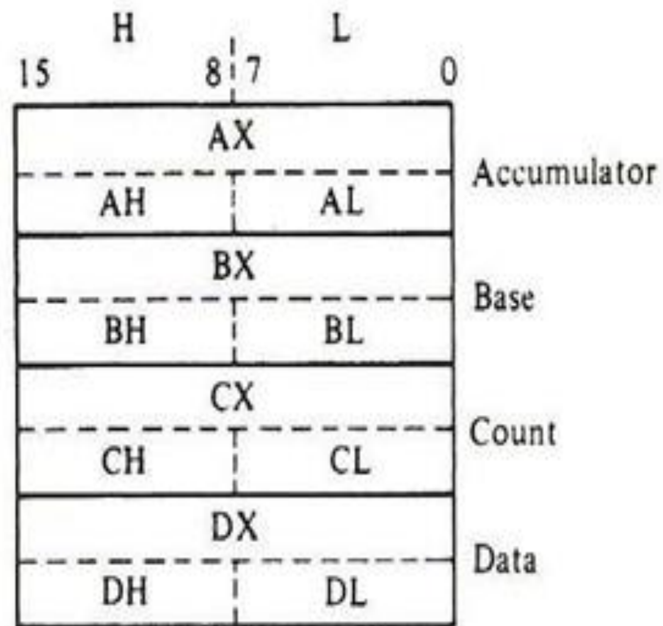
Internal Registers

- ***Segment Registers*** The segments of memory that are active are identified by the values of addresses held in 8086's four internal segment registers: CS (code segment), SS (stack segment), DS (data segment), and ES (extra segment). Each of these registers contains a 16-bit base address that points to the lowest addressed byte of the segment in memory.

- ***Instruction Pointer Register*** It is also 16-bit in length and identifies the location of the next word of instruction code to be fetched from the current code segment of memory. It contains the offset of the next word of instruction code instead of its actual address. Every time a word of code is fetched from memory, the 8086 updates the value in IP such that it points to the first byte of the next sequential word of code (IP incremented by 2).

Internal Registers

- ***Data Registers*** The 8086 has four general purpose data registers: the accumulator register (A), the base register (B), the count register (C), and the data register (D). These names imply special functions they are meant to perform.



Dedicated Registers Functions

During program execution they hold temporary values of frequently used intermediate results. The advantage of storing these data in internal registers instead of memory during processing is that they can be accessed much faster. Each of these registers can be accessed either as a whole 16-bit (for word data operations) or as two 8-bit registers (for byte wide data operations). When software places a new value in one byte of a register, for instance AL, the value in the other byte (AH), does not change.

Register	operation
AX	Word multiply, word divide, word I/O
AL	Byte multiply, byte divide, byte I/O, translate, decimal arithmetic
AH	Byte multiply, byte divide
BX	Translate
CX	String operations, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O

Internal Registers

- ***Pointer Registers*** They store offset addresses (the displacement of a storage location in memory from the segment base address in a segment register). Software uses the value held in a pointer register to access memory locations relative to the stack segment register. They are only accessed as words. The two pointer registers are: the stack pointer (SP) and base pointer (BP). The value in SP always represents the offset of the next stack location that is to be accessed (Top of Stack). BP also represents an offset relative to the SS. One common use of BP is to reference parameters that are passed to a subroutine by way of the stack.

- ***Indexed Registers*** They also store offset addresses from the data segment or extra segment registers. They are only accessed as words. For some operations, an operand that is to be processed may be located in memory instead of the internal registers. In this case, an index address is used to identify the location of the operand in memory. The source index (SI) register holds an offset address that identifies the location of a source operand, and the destination index (DI) register holds an offset for a destination operand.

Internal Registers

- ***Status Register*** (flag register) Is 16-bit register in which only nine of its bits are implemented. Six of these bits represent status flags: the carry flag (CF), parity flag (PF), auxiliary flag (AF), zero flag (ZF), sign flag (SF), and overflow flag (OF). The logic state of these status flags indicate conditions that are produced as the result of executing an instruction. The other three flags that provide control functions are: the direction flag (DF), interrupt flag (IF), and trap flag (TF). The instruction set of the 8086 includes instructions for saving, loading, or manipulation the flags.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

Status Flags

- ***The carry flag:*** CF is set if there is a carry-out or a borrow-in for the most significant bit of the result during the execution of an instruction. Otherwise CF is reset.
- ***The parity flag:*** PF is set if the result produced by the instruction has even parity (if it contains an even number of bits at the 1 logic level). If parity is odd, PF is reset.
- ***The auxiliary flag:*** AF is set if there is a carry-out from the low nibble into the high nibble or a borrow-in from the high nibble into the low nibble of the lower byte in a 16-bit word. Otherwise, AF is reset.
- ***The zero flag:*** ZF is set if the result produced by an instruction is zero. Otherwise, ZF is reset.
- ***The sign flag:*** The MSB of the result is copied into SF. Thus, SF is set if the result is a negative number or reset if it is positive.
- ***The overflow flag:*** When OF is set, it indicates that the signed result is out of range. If the result is not out of range, OF remains reset.

Control Flags

- ***The trap flag:*** If TF is set, the 8086 goes into the single-step mode of operation. When in the single-step mode, it executes an instruction and then jumps to a special service routine that may determine the effect of executing the instruction. This type of operation is very useful for debugging programs.

- ***The interrupt flag:*** For the 8086 to recognize *maskable interrupt requests* at its interrupt (INT) input, the IF flag must be set. When IF is reset, requests at INT are ignored and the maskable interrupt interface is disabled.

- ***The direction flag:*** The logic level of DF determines the direction in which string operations will occur. When set, the string instructions automatically decrement the address; therefore the string data transfers proceed from high address to low address. On the other hand, resetting DF causes the string address to be incremented (data transfers proceed from low address to high address).

Generating a Memory Address

A segment base and an offset describe a logical address; both are 16-bit quantities. However, the physical address that is used to access memory is 20-bits in length. The generation of the physical address involves combining a 16-bit offset value (located in IP, BX, SI, DI, SP, or BP) and a 16-bit segment base value (located in CS, DS, SS, or ES).

The segment base address represents the starting location of the 64 Kbyte segment in memory – that is, the lowest address byte in the segment. The offset identifies the distance in bytes that the storage location of interest resides from this starting address. Therefore, the lowest address byte in a segment has an offset of 0000_{16}, and the highest address byte has an offset of $FFFF_{16}$.

To obtain the physical address, the value in the segment register is shifted left by four bits (with its LSBs filled with zeros). The offset value is then added. The result of this addition is 20-bit physical address.

Software

- A microcomputer does not know how to process data, it must be told where to get data, what to do with the data, and where to put the results when it is done. These are the jobs of the ***software***.

- The sequence of commands to tell a microcomputer what to do is called a ***program***.

- Each command in a program is an ***instruction***.

- Programs must always be coded in ***machine language*** before they can be executed by the microprocessor.

- A program written in machine language is often referred to as ***machine code*** (instruction is encoded using 0s and 1s). It is almost impossible to write programs directly in machine language. For this reason, programs are normally written in 8086 assembly language or a high-level language such as C.

Software

- In ***assembly language***, each of the operations is described with alphanumeric symbols instead of 0s and 1s.
- An instruction can be divided into two parts: operation code (opcode) and operands.
- The ***opcode*** is the part of instruction that identifies the operation that is to be performed (add, subtract, move, …). Each opcode is assigned a unique letter combination called a ***mnemonic*** (ADD, SUB, MOV, …).
- ***Operands*** describe the data are to be processed as the microprocessor carries out the operation specified by the opcode. They identify whether the source and destination of the data are registers within the MPU or storage locations in data memory.
- The ***assembler*** is the program that translates the assembly language programs to an equivalent machine language program for execution by microprocessor.
- The ***compiler*** is the program that converts high-level language statements to machine code instructions.

Addressing Modes

An ***addressing mode*** is a method of specifying an operand. The 8086 is provided with various addressing modes to access operands categorized as following:

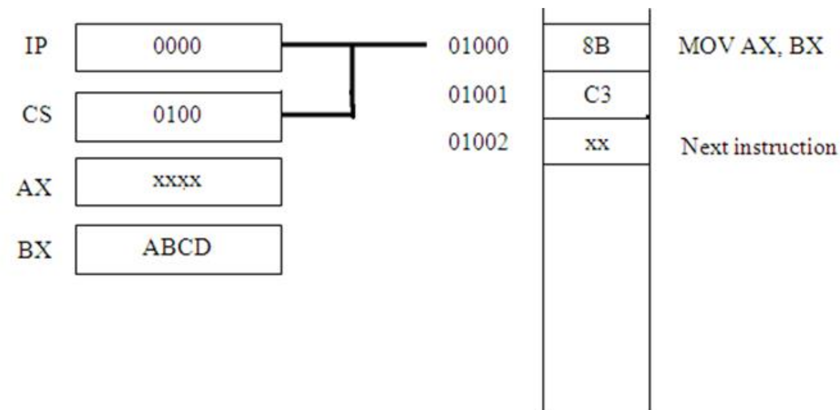
- Register operand addressing mode
- Immediate operand addressing mode
- Memory operand addressing modes
  - Direct addressing mode
  - Register indirect addressing mode
  - Based addressing mode
  - Indexed addressing mode
  - Based-indexed addressing mode

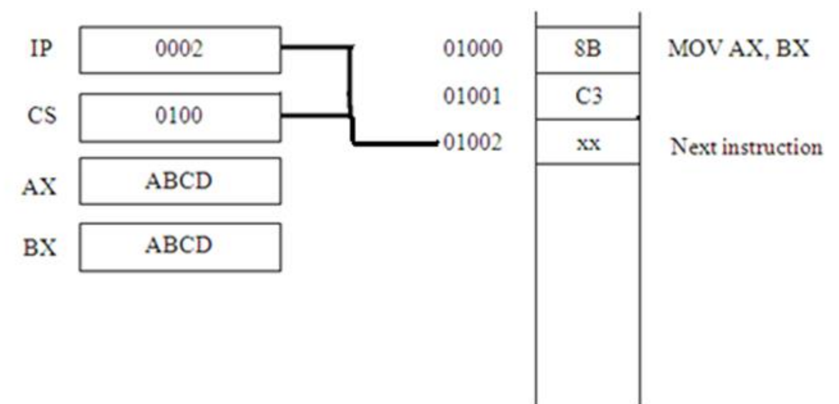
Register Operand Addressing Mode

The operand to be accessed is specified as residing in an internal register.

For example: **MOV AX, BX**

This stands for "move the content of BX (source operand) to AX (destination operand)". Assume the following values in registers and memory (on the left) just prior to execute the above instruction. The result of executing the instruction is that the content of BX (the value ABCD) is copied into AX as shown (on the right).



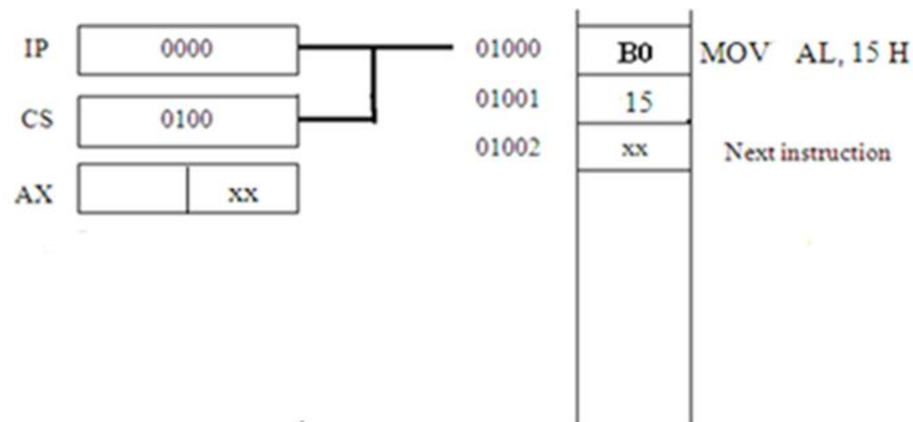


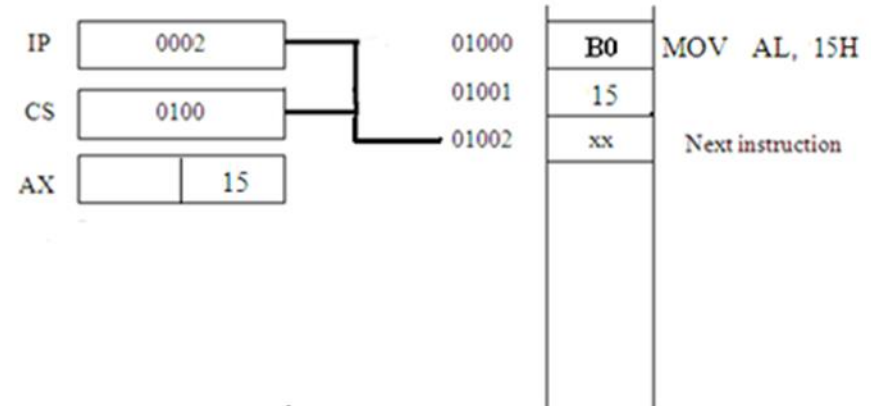
Immediate Operand Addressing Mode

The operand to be accessed is part of instruction. It can be 8-bit or 16-bit in length. This addressing mode can only be used to specify a source operand.

For example: **MOV AL, 15H**

The result produced by executing this instruction is that the immediate operand (15H) is loaded into the lower byte of accumulator (AL).



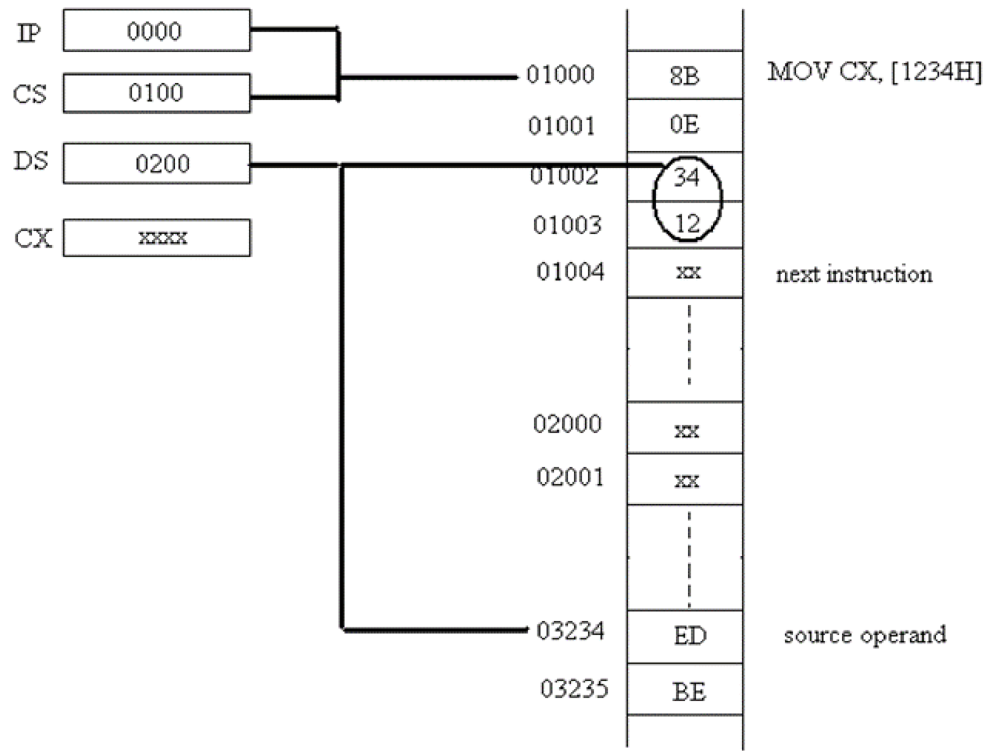


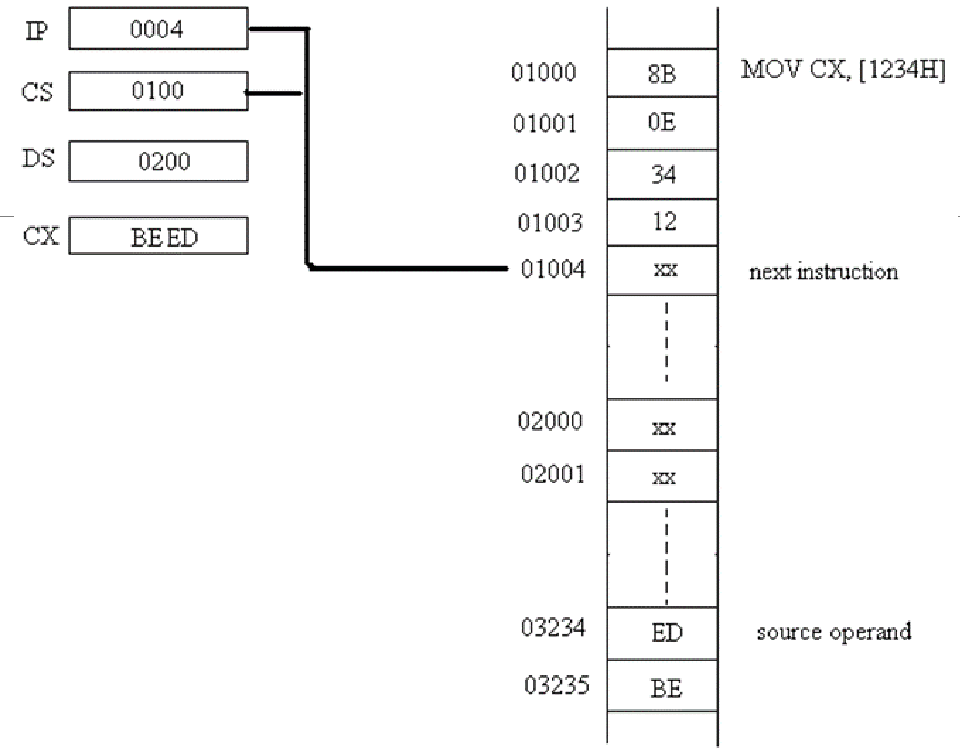
Memory Operand Addressing Modes

- ***Direct addressing mode*** the instruction op-code is followed by effective address.

For example: **MOV CX, [1234H]**

This stands for “move the content of memory location with offset 1234H in the current data segment to CX”. Assume the following values in registers and memory just prior to execute the above instruction





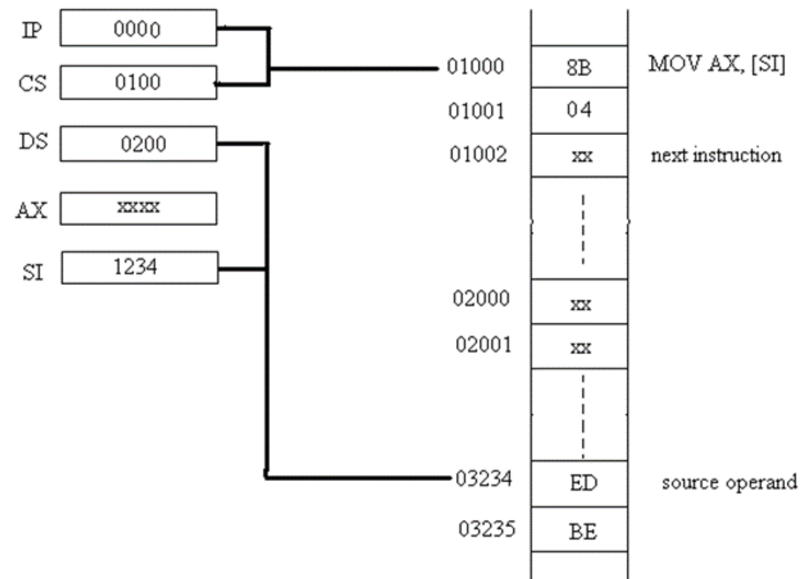
As the instruction is executed, the 8086 combines 1234H with 0200H to get physical address of source operand. Then it reads the word of data starting at this address (BEEDH) and loads it to the CX register.

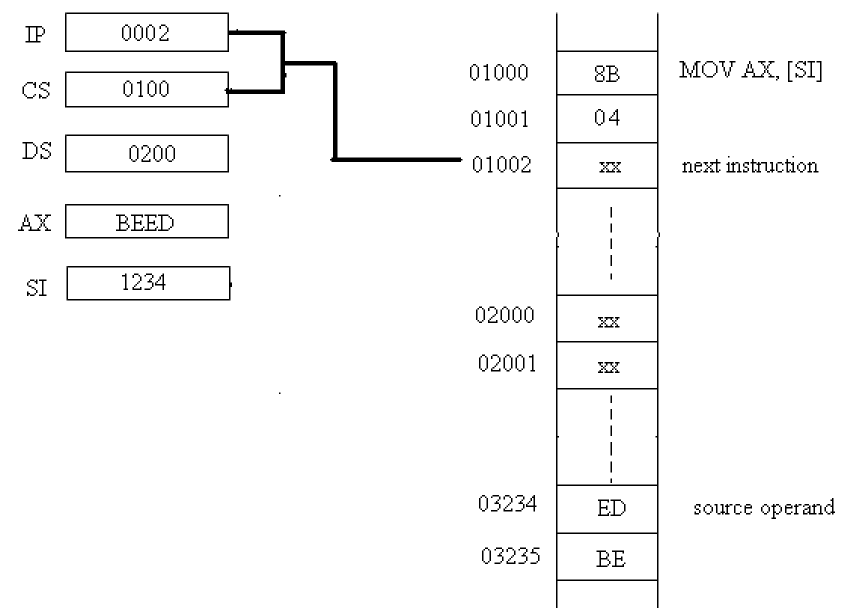
Memory Operand Addressing Modes

- ***Register indirect addressing mode*** the effective address resides in either a base register (BX or BP) or an index register (SI or DI).

For example: **MOV AX, [SI]**

Executing this instruction moves the content of location whose offset is in register SI into AX.





Memory Operand Addressing Modes

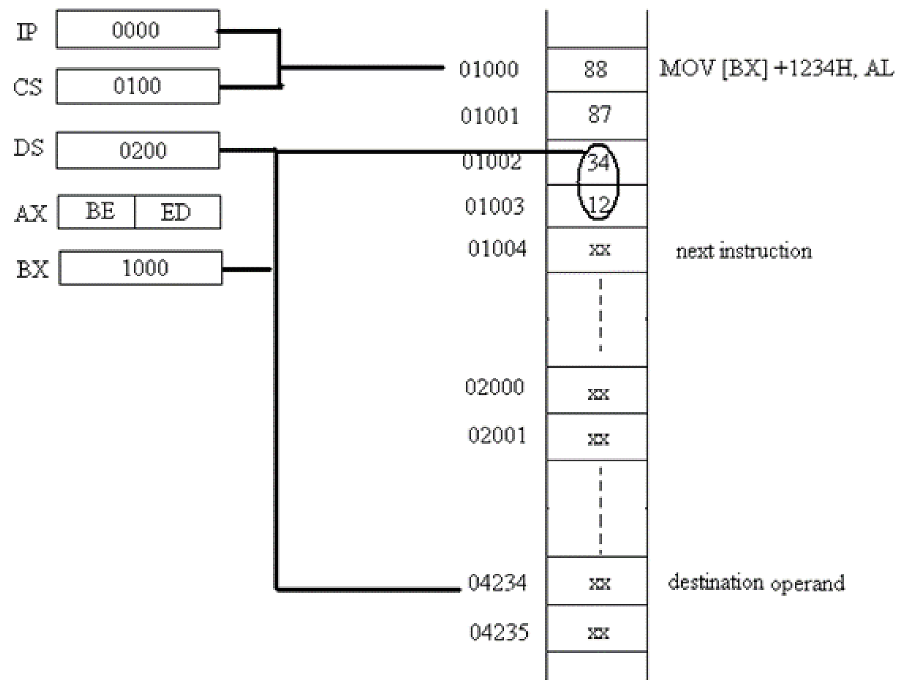
- ***Based addressing mode*** the effective address of the operand is obtained by adding a direct displacement to the content of either BX or BP register.

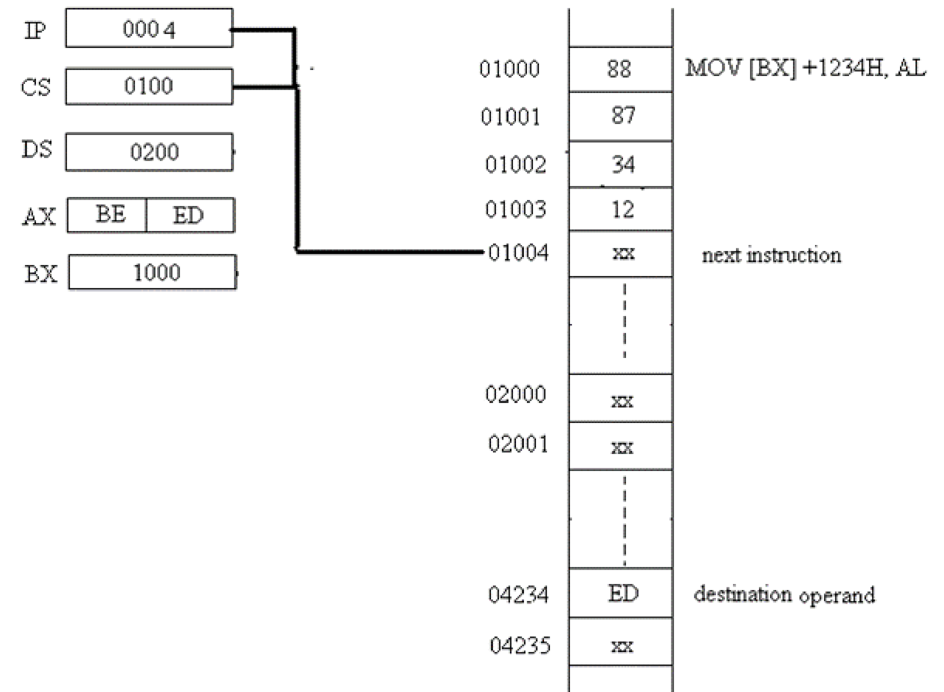
For example: **MOV [BX] + 1234H, AL**

The physical address of the destination operand is obtained from the contents of DS, BX, and the direct displacement then the content of AL is written into this location.

If BP is used instead of BX, the calculation of the physical address is performed using the content of the stack segment register instead of data segment.

Memory Operand Addressing Modes





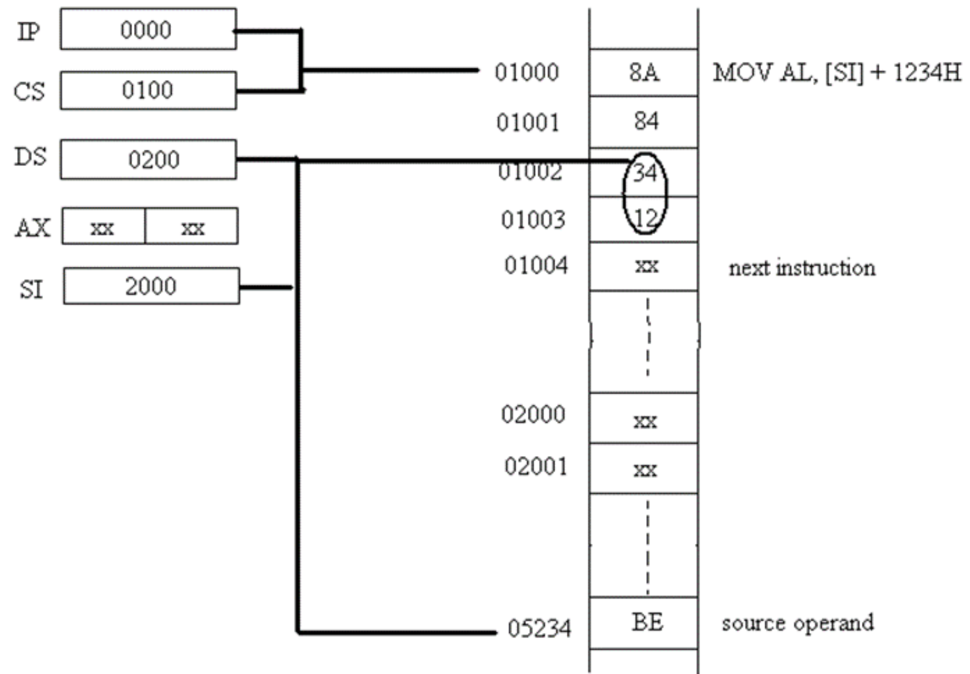
Memory Operand Addressing Modes

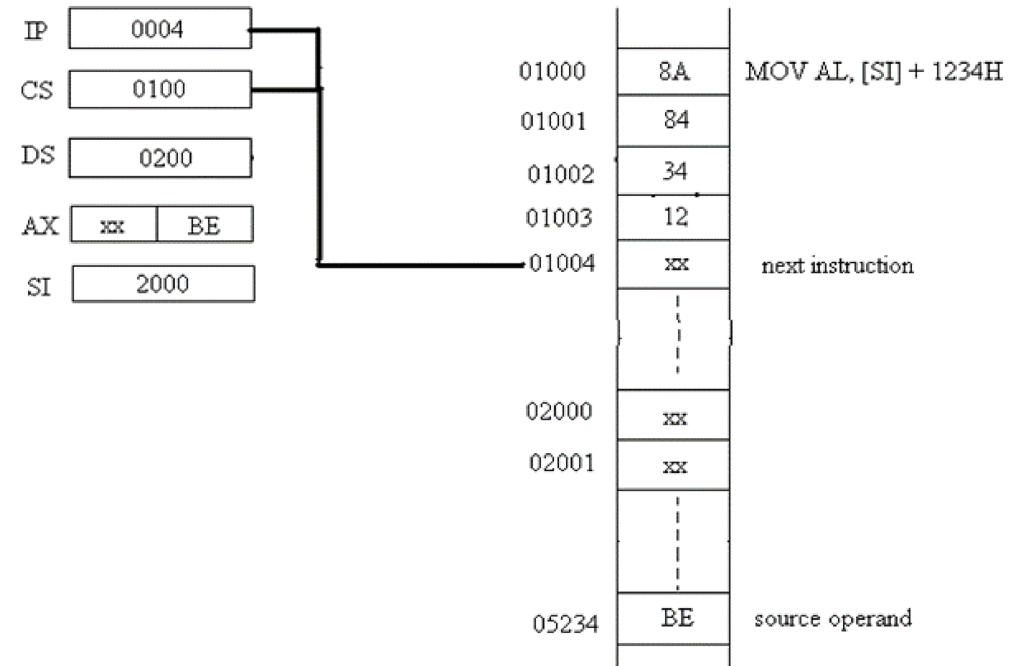
- ***Indexed addressing mode*** the physical address is obtained from the value in a segment register, an index register (SI or DI), and a displacement.

For example: **MOV AL, [SI] + 1234H**

First the physical address of the source operand is calculated from the contents of DS, SI, and the direct displacement. The byte stored at this location is read into AL.

Memory Operand Addressing Modes





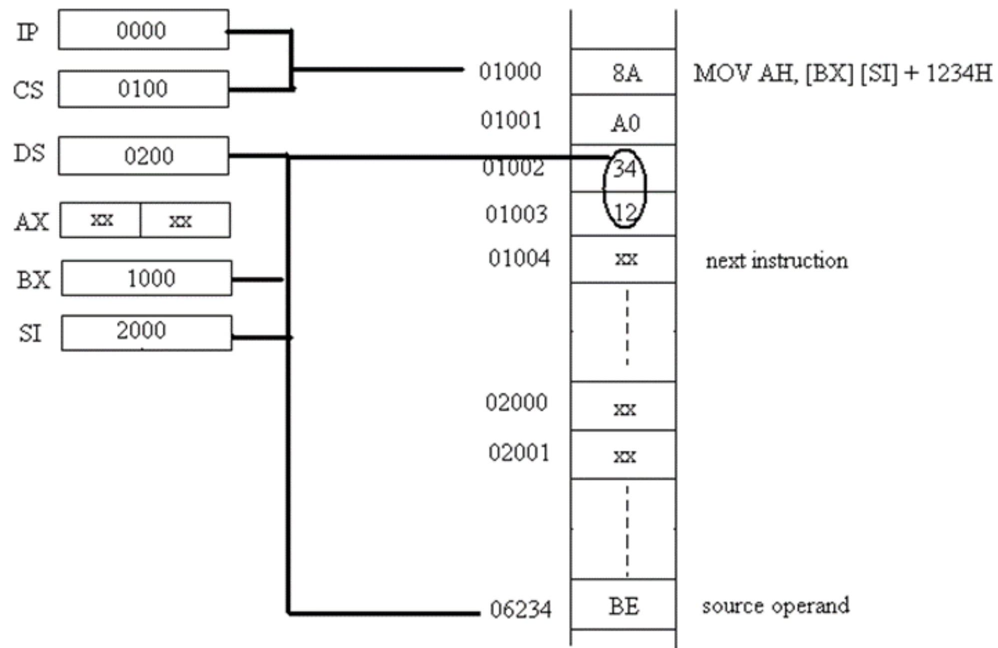
Memory Operand Addressing Modes

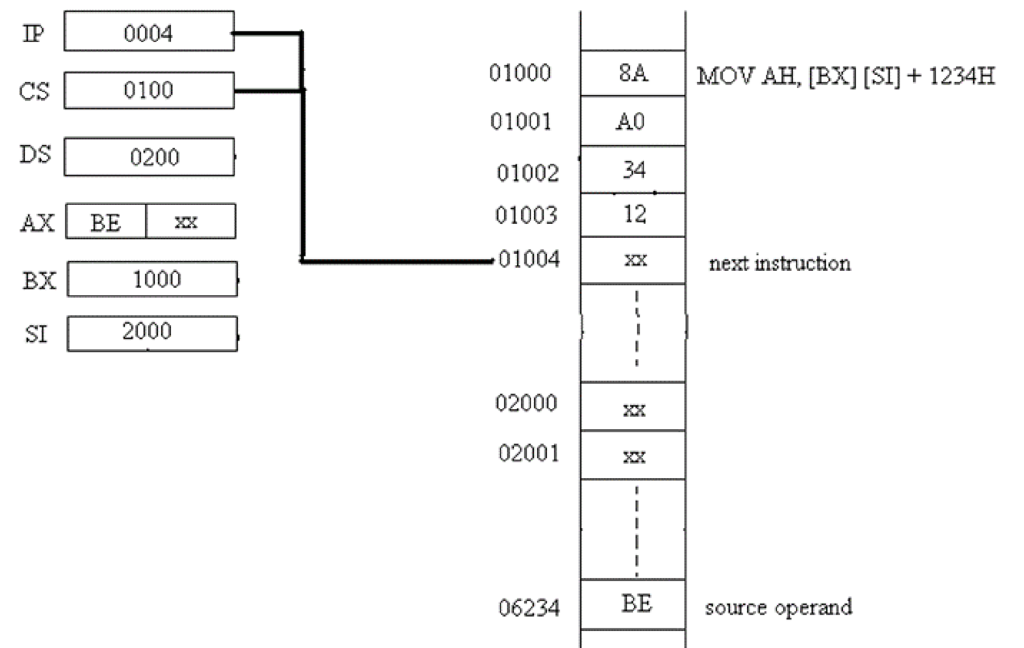
- ***Based-indexed addressing mode*** combines the based and indexed addressing modes.

For example: **MOV AH, [BX] [SI] + 1234H**

The physical address of the source operand is computed from the current content of DS, BX, SI, and the direct displacement. Execution of the instruction causes the value stored at this location to be read into AH.

Memory Operand Addressing Modes





Arithmetic Instructions

Addition Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
ADD	Addition	ADD D, S	(S) + (D)→ (D) Carry→ (CF)	OF, SF, ZF, AF,PF, CF
ADC	Add with carry	ADC D, S	(S) + (D) + (CF) →(D) Carry→(CF)	OF, SF, ZF, AF,PF, CF
INC	Increment by 1	INC D	(D) + 1→ (D)	OF, SF, ZF, AF,PF
AAA	ASCII adjust for addition	AAA		AF, CF (OF, SF, ZF, PFundefined)
DAA	Decimal adjust for addition	DAA		(SF, ZF, AF, PF, CF, OF undefined)

Arithmetic Instructions

Subtraction Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D, S	(D) - (S) ⟶ (D) Borrow ⟶ (CF)	OF, SF, ZF, AF,PF, CF
SBB	Subtract with borrow	SBB D, S	(D) - (S) - (CF) ⟶ (D) Borrow ⟶ (CF)	OF, SF, ZF, AF,PF, CF
DEC	Decrement by 1	DEC D	(D) - 1 ⟶ (D)	OF, SF, ZF, AF,PF
NEG	Negate	NEG D	0 – (D) ⟶ (D) 1 ⟶ (CF)	OF, SF, ZF, AF,PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF,AF, PF, CF (OF undefined)
AAS	ASCII adjust for subtraction	AAS		AF, CF (OF, SF, ZF, PF undefined)

Arithmetic Instructions

Multiplication and Division Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
MUL	Multiplication (unsigned)	MUL S	(AL).(S8) → (AX) (AX).(S16) → (DX),(AX)	OF, CF (SF, ZF, AF, PF undefined)
DIV	Division (unsigned)	DIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16) → (DX)	(OF, SF, ZF, AF, PF, CF undefined)
IMUL	Integer multiplication (signed)	IMUL S	(AL).(S8) → (AX) (AX).(S16) → (DX),(AX)	OF, CF (SF, ZF, AF, PF undefined)
IDIV	Integer division (signed)	IDIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16) → (DX)	(OF, SF, ZF, AF, PF, CF undefined)

Arithmetic Instructions

Multiplication and Division Instructions (continued)

Mnemonic	Meaning	Format	Operation	Flags affected
AAM	Adjust AL for multiplication	AAM	Q((AL)/10) → (AH) R((AL)/10) → (AL)	SF, ZF, PF (OF, AF, CF undefined)
AAD	Adjust AX for division	AAD	(AH).10 + (AL) → (AL) 00 → (AH)	SF, ZF, PF (OF, AF, CF undefined)
CBW	Convert byte to word	CBW	(MSBit of AL) → (All bits of AH)	None
CWD	Convert word to double word	CWD	(MSBit of AX) → (All bits of DX)	None

Logic Instructions

The 8086 has instructions for performing the logic operations AND, OR, exclusive-OR, and NOT bit-wise on byte-wide and word-wide data.

Mnemonic	Meaning	Format	Operation	Flags affected
AND	Logical AND	AND D, S	$(S) \cdot (D) \rightarrow (D)$	OF, SF, ZF, PF, CF (AF undefined)
OR	Logical OR	OR D, S	$(S) + (D) \rightarrow (D)$	OF, SF, ZF, PF, CF (AF undefined)
XOR	Logical Exclusive-OR	XOR D, S	$(S) \oplus (D) \rightarrow (D)$	OF, SF, ZF, PF, CF (AF undefined)
NOT	Logical NOT	NOT D	$(\overline{D}) \rightarrow (D)$	None

Arithmetic Instructions

- ***CBW Instruction*** extends the sign of the dividend to fill AX register (the sign extension does not change the value for the data).

- ***CWD Instruction*** extends the sign of the dividend to fill DX register (the sign extension does not change the value for the data).

The last two instructions are used to allow division of 8-bit dividend in AL by 8-bit divisor and 16-bit dividend in AX by 16-bit divisor

Shift Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
SAL/SHL	Shift arithmetic left/ Shift logical left	SAL/SHL D, Count	Shift the (D) left by the number of bit positions equal to Count and fill the vacated bits positions on the right with zeros	CF, PF, SF, ZF (AF undefined) (OF undefined if count ≠ 1)
SHR	Shift logical right	SHR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bits positions on the left with zeros	CF, PF, SF, ZF (AF undefined) (OF undefined if count ≠ 1)
SAR	Shift arithmetic right	SAR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bits positions on the left with the original MSBit	CF, PF, SF, ZF (AF undefined) (OF undefined if count ≠ 1)

Logic Instructions

- ***AND Instruction*** causes the content of source operand to be ANDed with the contents of destination operand. The result is reflected by the new content of destination. AND instruction is used to clear certain bit(s) of a byte or word.

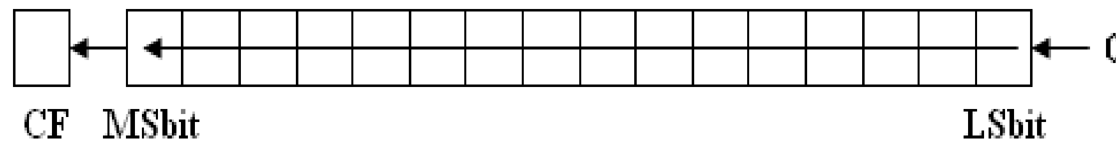
- ***OR Instruction*** used to set bit(s) in register or a storage location in memory.

- ***XOR Instruction*** used to reverse the logic level of bit(s) in register or a storage location in memory (toggling the bit).

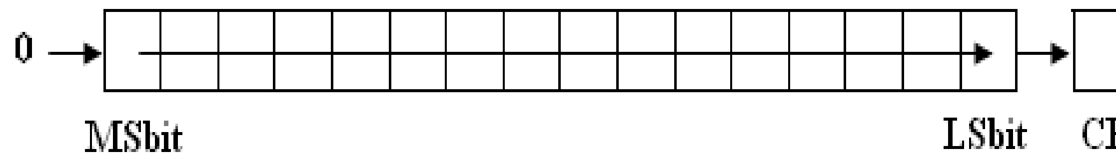
- ***NOT Instruction*** used to obtain the 1's complement of destination operand (internal register or a location in memory).

Shift Instructions

- ***SAL/SHL Instruction*** shifts the destination operand (register or storage location in memory) to the left by number of bits specified by count. The count can be either 1 for 1-bit shift, or the value in CL for more than 1-bit shift. The vacated LSBit locations is filled with zero and the last bit shifted out of the MSBit is saved in CF.

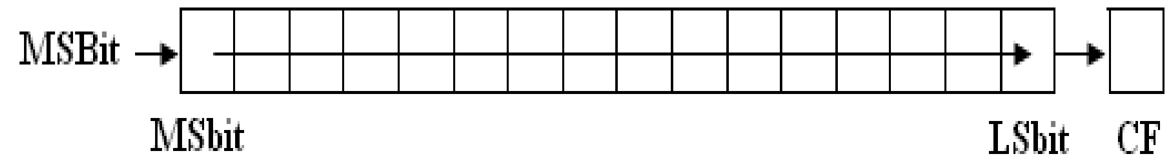


- ***SHR Instruction*** shifts the destination operand (register or storage location in memory) to the right by number of bits specified by count. The vacated MSBit locations is filled with zero and the last bit shifted out of the LSBit is saved in CF.



Shift Instructions

- ***SAR Instruction*** shifts the destination operand (register or storage location in memory) to the right by number of bits specified by count. The vacated MSBit locations is filled with the original MSBit and the last bit shifted out of the LSBit is saved in CF.



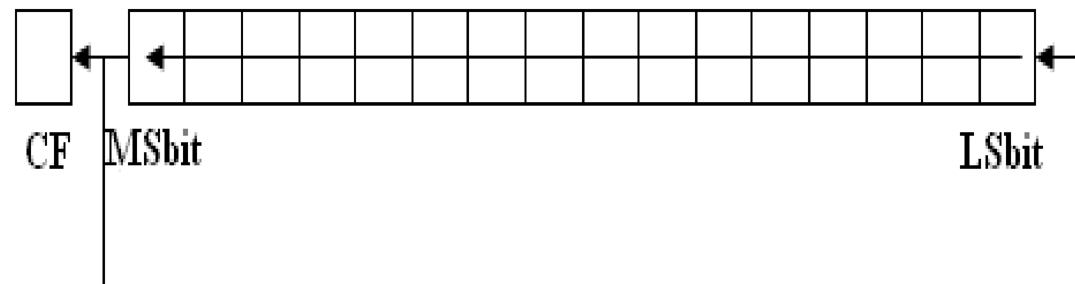
Rotate Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
ROL	Rotate left	ROL D, Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the left most bit goes back into the right most bit position	CF (OF undefined if count ≠ 1)
ROR	Rotate right	ROR D, Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the right most bit goes into the left most bit position	CF (OF undefined if count ≠ 1)
RCL	Rotate left through carry	RCL D, Count	Same as ROL except carry is attached to (D) for rotation	CF (OF undefined if count ≠ 1)
RCR	Rotate right through carry	RCR D, Count	Same as ROR except carry is attached to (D) for rotation	CF (OF undefined if count ≠ 1)

Rotate Instructions

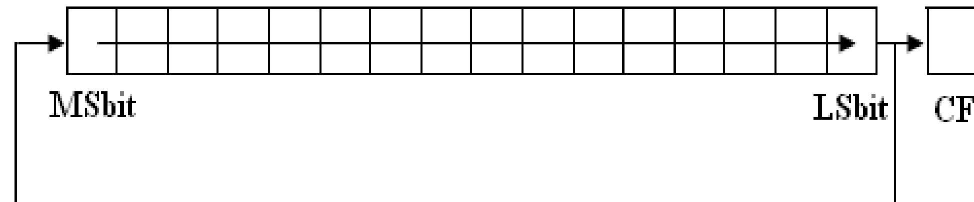
The rotate instructions are similar to the shift instructions; they perform many of the same programming functions such as alignment of data and isolation of a bit of data.

❖ ***ROL Instruction*** rotates the destination operand (register or storage location in memory) to the left by number of bits specified by count. The bits moved out at MSBit are not lost; instead they are reloaded at the other end.

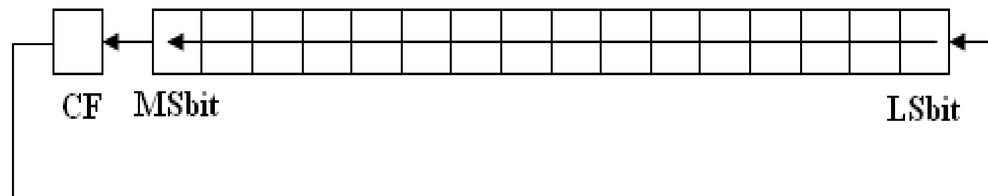


Rotate Instructions

- ***ROR Instruction*** rotates the destination operand (register or storage location in memory) to the right by number of bits specified by count. The bits moved out at LSBit are reloaded at the other end.

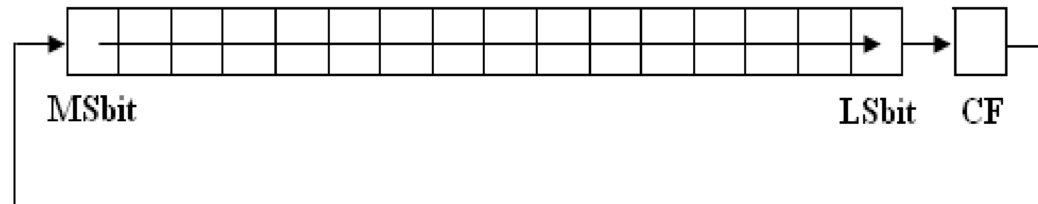


- ***RCL Instruction*** rotates the destination operand (register or storage location in memory) to the left through the carry flag by number of bits specified by count. The bits moved out at MSBit are reloaded at the other end.



Rotate Instructions

- ***RCR Instruction*** rotates the destination operand (register or storage location in memory) to the right through the carry flag by number of bits specified by count. The bits moved out at LSBit are reloaded at the other end.



Flag-Control Instructions

The instruction set includes a group of instructions that, when executed, directly affect the state of the flags.

Mnemonic	Meaning	Operation	Flags affected
LAHF	Load AH from flags	(AH) ← (Flags)	None
SAHF	Store AH into flags	(Flags) ← (AH)	SF, ZF, AF, PF, CF
CLC	Clear carry flag	(CF) ← 0	CF
STC	Set carry flag	(CF) ← 1	CF
CMC	Complement carry flag	(CF) ← $(\overline{CF})$	CF
CLI	Clear interrupt flag	(IF) ← 0	IF
STI	Set interrupt flag	(IF) ← 1	IF

Flag-Control Instructions

- ***LAHF Instruction*** loads the flags into AH to read them. The format of the flags information in AH is as shown below (bits 1, 3, and 5) are not used.

AH

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
SF	ZF	x	AF	x	PF	x	CF

- ***SAHF Instruction*** stores AH into flags to change them, used to start an operation with certain flags set or reset. The format in AH is same as above.

Flag-Control Instructions

❖ ***CLC, STC, CMC Instructions*** used to manipulate the carry flag, they permit CF to be cleared, set, or complemented, respectively.

❖ ***CLI, STI Instructions*** used to manipulate the interrupt flag. CLI instruction clears the interrupt flag (IF = 0, disables the interrupt interface). STI instruction sets the interrupt flag (IF = 1, microprocessor is enabled to accept interrupts).

Compare Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
CMP	Compare	CMP D, S	(D) - (S) is used in setting or resetting the flags	CF, AF, OF, PF,SF, ZF

Jump Instructions

conditional jump instruction set are shown in the table below. Each of these instructions tests for the presence or absence of certain status conditions. For some of the instructions, two different mnemonics can be used (this improves the program readability).

Mnemonic	Meaning	Condition
JA	Above	CF = 0 and ZF = 0
JAE	Above or equal	CF = 0
JB	Below	CF = 1
JBE	Below or equal	CF = 1 or ZF = 1
JC	Carry	CF = 1
JCXZ	CX register is zero	(CF or ZF) = 0
JE	Equal	ZF = 1

Jump Instructions

Mnemonic	Meaning	Condition
JG	Greater	ZF = 0 and SF= OF
JGE	Greater or equal	SF = OF
JL	Less	(SF xor OF) = 1
JLE	Less or equal	((SF xor OF)or ZF) = 1
JNA	Not above	CF = 1 or ZF = 1
JNAE	Not above nor equal	CF = 1
JNB	Not below	CF = 0
JNBE	Not below nor equal	CF = 0 and ZF = 0
JNC	Not carry	CF = 0
JNE	Not equal	ZF = 0

Jump Instructions

Mnemonic	Meaning	Condition
JNG	Not greater	((SF xor OF)or ZF) = 1
JNGE	Not greater nor equal	(SF xor OF) = 1
JNL	Not less	SF = OF
JNLE	Not less nor equal	ZF = 0 and SF= OF
JNO	Not overflow	OF = 0
JNP	Not parity	PF = 0
JNS	Not sign	SF = 0
JNZ	Not zero	ZF = 0
JO	Overflow	OF = 1
JP	Parity	PF = 1

Jump Instructions

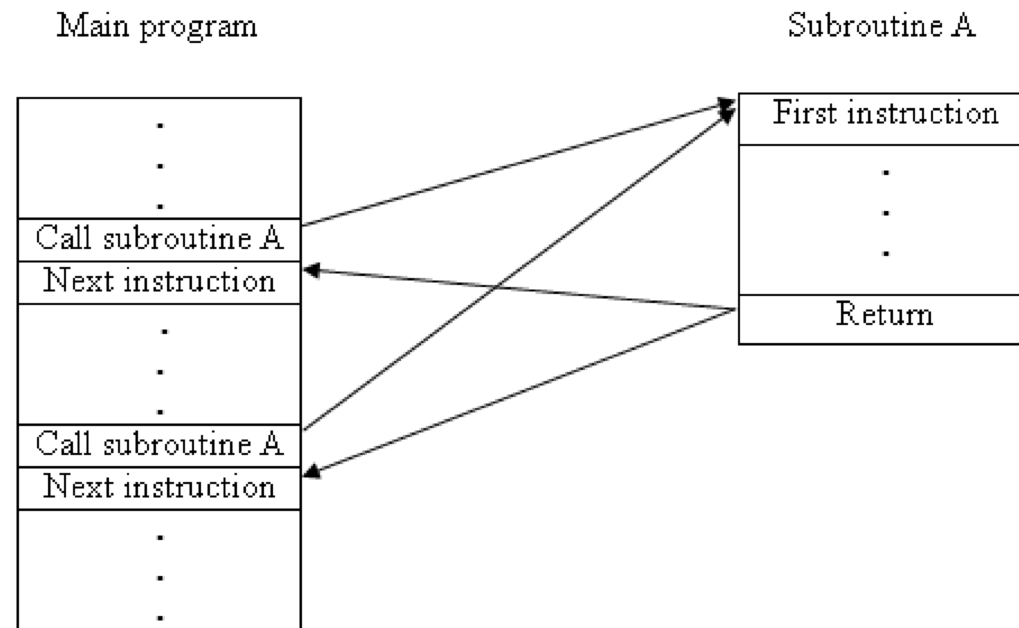
Mnemonic	Meaning	Condition
JPE	Parity even	PF = 1
JPO	Parity odd	PF = 0
JS	Sign	SF = 1
JZ	Zero	ZF = 1

Above and ***below*** are used to describe the comparison of unsigned numbers; while ***less*** and ***greater*** to describe comparison of signed numbers. For example, the number ABCDH is above the number 1234H if they are considered to be unsigned numbers. On the other hand, ABCDH is less than 1234H if they are treated as signed numbers.

When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions. When unsigned numbers are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions.

Subroutines Instructions

A subroutine is a special segment of program that can be called for execution from any point in a program. The subroutine is written to provide a function that must be performed at various points in the main program.



Execution Sequencing of a Program That Includes Subroutine Calling

Subroutines Instructions

There are instructions provided to transfer control from the main program to a subroutine and return control back to the main program.

Mnemonic	Meaning	Format	Operation	Flags affected
CALL	Subroutine call	CALL operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved to stack.	None
RET	Return	RET or RET operand	Return to the main program by restoring IP (and CS for far-proc). If operand is present, it is added to the contents of SP.	None
PUSH	Push word onto stack	PUSH S	((SP)) ⟵ (S) (SP) ⟵ (SP)-2	None
POP	Pop word off stack	POP D	(D) ⟵ ((SP)) (SP) ⟵ (SP)+2	None
PUSHF	Push flags onto stack	PUSHF	((SP)) ⟵ (Flags) (SP) ⟵ (SP)-2	None
POPF	Pop flags off stack	POPF	(Flags) ⟵ ((SP)) (SP) ⟵ (SP)+2	OF, DF, IF, TF, SF, ZF, AF, PF, CF

Subroutines Instructions

- ***CALL Instruction*** provides the mechanism to call a subroutine into operation by modifying either the value of IP or IP and CS to branch to a subroutine. The operand initiates either an intrasegment or intersegment call.

- ***Intrasegment call*** causes the content of IP to be saved on the stack and a new 16-bit value to be loaded into IP. The operands can be *Near-proc*, *Memptr 16*, or *Regptr 16*.

- ***Near-proc*** the 16-bit immediate operand is loaded to IP. CALL 1234H
- ***Memptr 16*** the content of a memory location (word) specified by the operand is loaded into IP. CALL [BX]
- ***Regptr 16*** the content of a register is loaded into IP. CALL BX

Subroutines Instructions

- ***Intersegment call*** permits the subroutine to reside in another code segment. The contents of CS and IP are saved on the stack, and then new values are loaded to them. The operands can be *Far-proc, Memptr 32*.

- ***Far-proc*** a 32-bit immediate operand is loaded into IP and CS. CALL 1234:5678H

- ***Memptr 32*** the pointer for the subroutine is stored as four consecutive bytes in data memory. The first word of memory is loaded into IP; the second word of memory is loaded into CS.

  CALL DWORD PTR [DI]

- ***RET Instruction*** returns control to the main program. It causes the value of IP or both IP and CS that were saved on the stack to be returned back to their corresponding registers. Program control is returned to the instruction that immediately follows the call instruction.

Subroutines Instructions

- ***PUSH Instruction*** used to save parameters on the stack. These data correspond to registers and memory locations that are used by the subroutine. In this way, their original contents are kept intact in the stack segment during the execution of the subroutine.

- ***POP Instruction*** used to retrieve parameters from the stack. Before a return to the main program takes place, the parameters are restored.

The operands for push and pop instructions can be a general-purpose register, a segment register (excluding CS), or a storage location in memory.

- ***PUSHF Instruction*** saves the content of flag register on the top of stack.

- ***POPF Instruction*** returns the flags from the top of stack to the flag register.

Loops Instructions

The 8086 microprocessor has instructions specifically designed for implementing loop operations. These instructions can be used in place of certain conditional jump instructions and give the programmer a simpler way of writing loop sequences.

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP short-label	(CX) $\leftarrow$ (CX) - 1 Jump is initiated to location defined by short-label if (CX) ≠ 0; otherwise, execute next sequential instruction
LOOPE/ LOOPZ	Loop while equal/ Loop while zero	LOOPE/ LOOPZ short-label	(CX) $\leftarrow$ (CX) - 1 Jump to location defined by short-label if (CX) ≠ 0 and ZF = 1; otherwise, execute next sequential instruction
LOOPNE/ LOOPNZ	Loop while not equal/ Loop while not zero	LOOPNE/ LOOPNZ short-label	(CX) $\leftarrow$ (CX) - 1 Jump to location defined by short-label if (CX) ≠ 0 and ZF = 0; otherwise, execute next sequential instruction

Strings Instructions

A string is a series of data words (or bytes) that reside in consecutive memory locations. The string instructions of 8086 permit programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory, scanning a string of data elements stored in memory to look for a specific value, comparing the elements of two strings in order to determine whether they are the same or different, and initializing a group of consecutive memory locations. These operations must be repeated to handle a string of more than one element.

Strings Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
MOVS	Move string	MOVSB/ MOVSW	((ES)0+(DI)) ← ((DS)0+(SI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None
CMPS	Compare string	CMPSB/ CMPSW	Set flags as per ((DS)0+(SI)) - ((ES)0+(DI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
SCAS	Scan string	SCASB/ SCASW	Set flags as per (AL or AX) - ((ES)0+(DI)) (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
LODS	Load string	LODSB/ LODSW	(AL or AX) ← ((DS)0+(SI)) (SI) ← (SI) ± 1 or 2	None
STOS	Store string	STOSB/ STOSW	((ES)0+(DI)) ← (AL or AX) (DI) ← (DI) ± 1 or 2	None
CLD	Clear DF	CLD	(DF) ← 0	DF
STD	Set DF	STD	(DF) ← 1	DF

Strings Instructions

- ***MOVSB/ MOVSW Instructions*** an element of the string specified by SI register with respect to DS is moved to the location specified by DI register with respect to ES. The move can be performed on a byte or a word of data. After the move is complete, the content of both SI and DI are automatically incremented or decremented by 1 for a byte move and by two for a word move. The address pointers in SI and DI increment or decrement depending on how the direction flag (DF) is set.

- ***CLD Instruction*** clears DF, this selects auto-increment mode in string operations so that each time a string operation is performed, SI and/ or DI are incremented by 1 if byte data are processed and by 2 if word data are processed.

- ***STD Instruction*** sets DF, this selects auto-decrement mode in string operations so that each time a string operation is performed, SI and/ or DI are decremented by 1 if byte data are processed and by 2 if word data are processed.

Strings Instructions

- ***LODSB/ LODSW Instructions*** LODSB loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element; SI is incremented or decremented by 1 after loading. LODSW indicates that the word-string element at physical address derived from DS and SI is to be loaded into AX. Then the index in SI is automatically incremented or decremented by 2.

- ***STOSB/ STOSW Instructions*** stores a byte from AL or a word from AX into a string location in memory. ES and DI are used to form the address of storage location in memory.