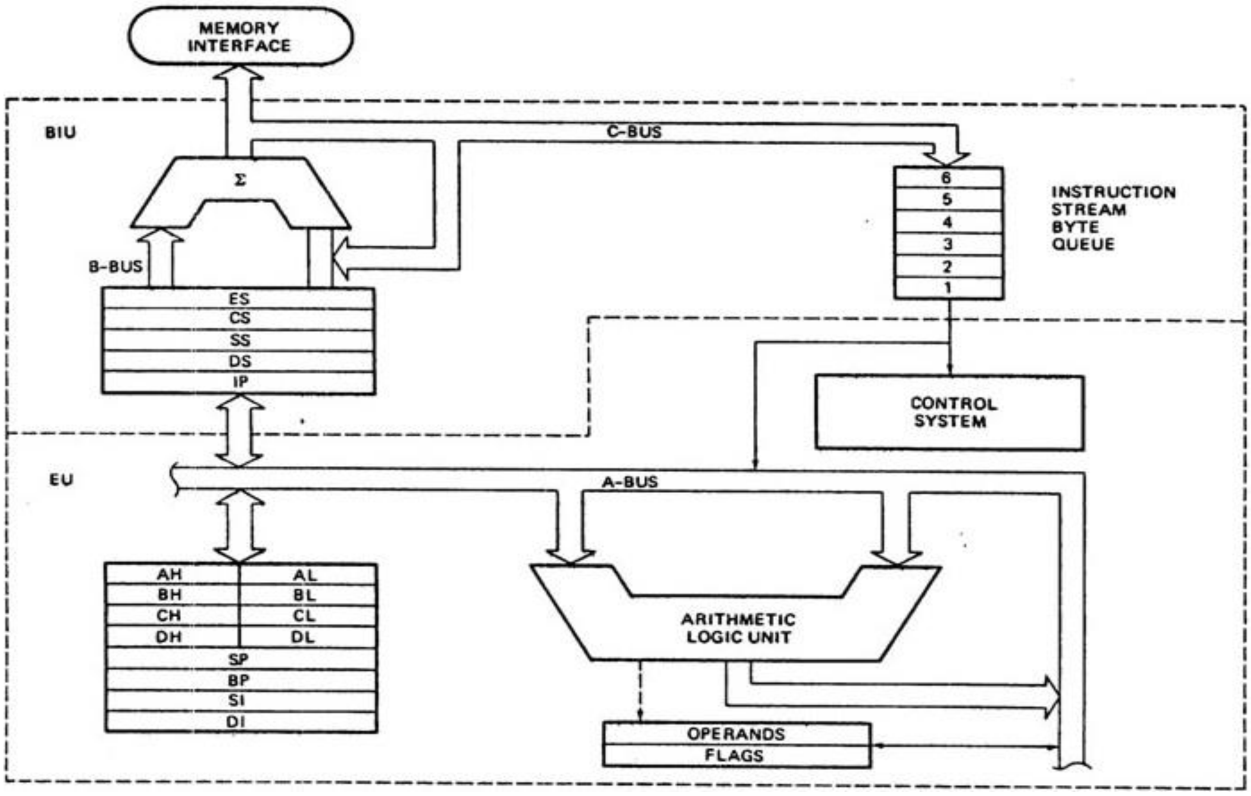
MICROPROCESSORS AND MICROCONTROLLERS

Course code: 18CS4DCMMC **Credits: 03**
Year:2020

Unit	Contents of the Unit
1	**INTRODUCTION, MICROPROCESSOR ARCHITECTURE – 1:** The Microprocessor and its Architecture: Internal Microprocessor Architecture, Real Mode Memory Addressing, Data Addressing Modes, Introduction to programming 8086: Data Transfer Instructions, Arithmetic Instructions: Addition, Subtraction and Comparison, Multiplication and Division Textbook 1:Chapter 2: Section 2.1,2.2 ,Chapter 3: Section 3.1, Chapter 4:4.2,4.3,4.4,4.5,4.7, Chapter 5:5.1,5.2
2	**INTERRUPTS AND PROGRAMMING**: Introduction to Interrupts, DOS Interrupts, BIOS Interrupts, Interrupt vector table, Program Control Instructions: The Jump Group, Controlling the Flow of the Program, Macros and Procedures, BCD and ASCII arithmetic instructions, Basic Logic Instructions. Textbook 1: Chapter 12: Section 12.1, Chapter 6: 6.1, 6.2, 6.3, 6.4, Chapter 8: Page 257 (Macros) , Chapter 5:Section 5.3, 5.4, 5.5
3	**I/O INTERFACE:** The Programmable Peripheral Interface 82C55, Architecture, Control Word Register, Modes of operation, Programmable Interval Timer 8254, Architecture, Control Word Register, Interfacing microprocessor to Logic Controller, keyboard, 7-segment display. Textbook 1: Chapter 11: Section 11.3, 11.4
4	**THE 8051 ARCHITECTURE & BASIC INSTRUCTION SET:** Overview of the 8051 Family, Inside the 8051, Introduction to 8051 Assembly Level Programming, 8051 Data Types and Directives, 8051 Flag bits and PSW register, 8051 Register Banks and Stack, Example Programs. Textbook 2:Chapter 1: Section 1.2, Chapter 2: Section 2.1, 2.2, 2.5, 2.6, 2.7
5	**INSTRUCTION SET AND PROGRAMMING 8051:** Arithmetic instructions, Logic and Compare instructions, Rotate instructions and data serialization, BCD, ASCII and other Application Programs, Loop and Jump instructions, CALL instructions, 8051 Addressing Modes: Immediate and Register Addressing modes, Accessing memory using various addressing modes. Textbook 2: Chapter 6: Section 6.1, 6.3, 6.4, 6.5 Chapter 3: Section 3.1, 3.2 Chapter 5: Section 5.1, 5.2

Module 1:

Internal Microprocessor Architecture:



The main features of 8086

It is a 16-bit Microprocessorμp).It’s ALU, internal registers works with 16bit binary word.
• 8086 has a 20 bit address bus can access up to 220= 1 MB memory locations.
• 8086 has a 16bit data bus. It can read or write data to a memory/port either 16bits or 8 bit at a time.
• It can support up to 64K I/O ports.
• It provides fourteen 16 -bit registers.
• Frequency range of 8086 is 6-10 MHz
• It has multiplexed address and data bus AD0- AD15 and A16 – A19.
• It can prefetchupto 6 instruction bytes from memory and queues them in order to speed up instruction execution.
• It requires +5V power supply.
• A 40 pin dual in line package.
• 8086 is designed to operate in two modes, Minimum mode and Maximum mode.

* 8086 has two blocks

i. Bus Interfacing Unit(BIU) and
ii. Execution Unit(EU).

BUS INTERFACE UNIT:
• It provides a full 16 bit bidirectional data bus and 20 bit address bus.
• The bus interface unit is responsible for performing all external bus operations.
• Specifically it has the following functions:
i. Instruction fetch,
ii. Instruction queuing,
iii. Operand fetch and storage,
iv. Address relocation and
v. Bus control.
• The BIU uses a mechanism known as an instruction stream queue to implement a*pipeline architecture*. This queue permits prefetch of up to six bytes of instruction code.These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, theBIU fetches two instruction bytes in a single memory cycle.
• After a byte is loaded at the input end of the queue, it automatically shifts up through the
FIFO to the empty location nearest the output.
• The EU accesses the queue from the output end. It reads one instruction byte after theother from the output of the queue. If the queue is full and the EU is not requesting accessto operand in memory.
• The BIU also contains a dedicated adder which is used to generate the 20bit physicaladdress that is output on the address bus. This address is formed by adding an appended16 bit segment address and a 16 bit offset address.
• For example: The physical address of the next instruction to be fetched is formed bycombining the current contents of the code segment CS register and the current contentsof the instruction pointer IP register.
• The BIU is also responsible for generating bus control signals such as those for memoryread or write and I/O read or write.

EXECUTION UNIT
• The Execution unit is responsible for decoding and executing all instructions.
• The EU extracts instructions from the top of the queue in the BIU, decodes them,generates operands if necessary, passes them to the BIU and requests it to perform theread or write bytes cycles to memory or I/O and perform the operation specified by theinstruction on the operands.
• During the execution of the instruction, the EU tests the status and control flags andupdates them based on the results of executing the instruction.
• If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted
to top of the queue.
• When the EU executes a branch or jump instruction, it transfers control to a location
corresponding to another set of sequential instructions.
• Whenever this happens, the BIU automatically resets the queue and then begins to fetch
instructions from this new location to refill the queue.

REGISTER ORGANIZATION OF 8086

The 8086 microprocessor has a total of fourteen registers that are accessible to the programmer.It is divided into four groups. They are:

1. Four General purpose registers
2. Four Index/Pointer registers
3. Four Segment registers
4. Two Other registers

General purpose registers:
Accumulator (AX) This register consists of two 8-bit registers AL and AH, which can becombined together and used as a 16-bit register AX. AL in this case contains the loworder byteof the word, and AH contains the high-order byte. Accumulator can be used for I/O operationsand string manipulation.

Base register (BX): consists of two 8-bit registers BL and BH, which can be combinedtogether and used as a 16-bit register BX. BL in this case contains the low-order byte of theword, and BH contains the high-order byte. BX register usually contains a data pointer used forbased, based indexed or register indirect addressing.

Count register(CX): consists of two 8-bit registers CL and CH, which can be combinedtogether and used as a 16-bit register CX. When combined, CL register contains the loworderbyte of the word, and CH contains the high-order byte. Count register can be used in Loop,shift/rotate instructions and as a counter in string manipulation.

Data register (DX): consists of two 8-bit registers DL and DH, which can be combinedtogether and used as a 16-bit register DX. When combined, DL register contains the low orderbyte of the word, and DH contains the high-order byte. Data register can be used as a portnumber in I/O operations. In integer 32-bit multiply and divide instruction the DX registercontains high-order word of the initial or resulting number.

Index or Pointer Registers
These registers can also be called as Special Purpose registers.
Stack Pointer (SP) is a 16-bit register pointing to program stack, it is used to hold the addressof the top of stack. The stack is maintained as a LIFO with its bottom at the start of the stacksegment (specified by the SS segment register).Unlike the SP register, the BP can be used tospecify the offset of other program segments.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. It is usually used bysubroutines to locate variables that were passed on the stack by a calling program. BP register is
usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirectaddressing, as well as a source data address in string manipulation instructions. Used inconjunction with the DS register to point to data locations in the data segment.

Destination Index (DI) is a 16-bit register. Used in conjunction with the ES register in string operations. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions. In short, Destination Index and SI Source Index registers are used to hold address.

Segment Registers

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processorinstructions. The processor uses CS segment for all accesses to instructions referenced byinstruction pointer (IP) register. CS register cannot be changed directly. The CS register isautomatically updated during far jump, far call and far return instructions.

Stack segment (SS) is a 16-bit register containing address of 64KB segment with program stack.By default, the processor assumes that all data referenced by the stack pointer (SP) and basepointer (BP) registers is located in the stack segment. SS register can be changed directly usingPOP instruction.

Data segment (DS) is a 16-bit register containing address of 64KB segment with program data.
By default, the processor assumes that all data referenced by general registers (AX, BX, CX,DX) and index register (SI, DI) is located in the data segment. DS register can be changeddirectly using POP and LDS instructions.

Extra segment (ES) used to hold the starting address of Extra segment. Extra segment isprovided for programs that need to access a second data segment. Segment registers cannot beused in arithmetic operations.

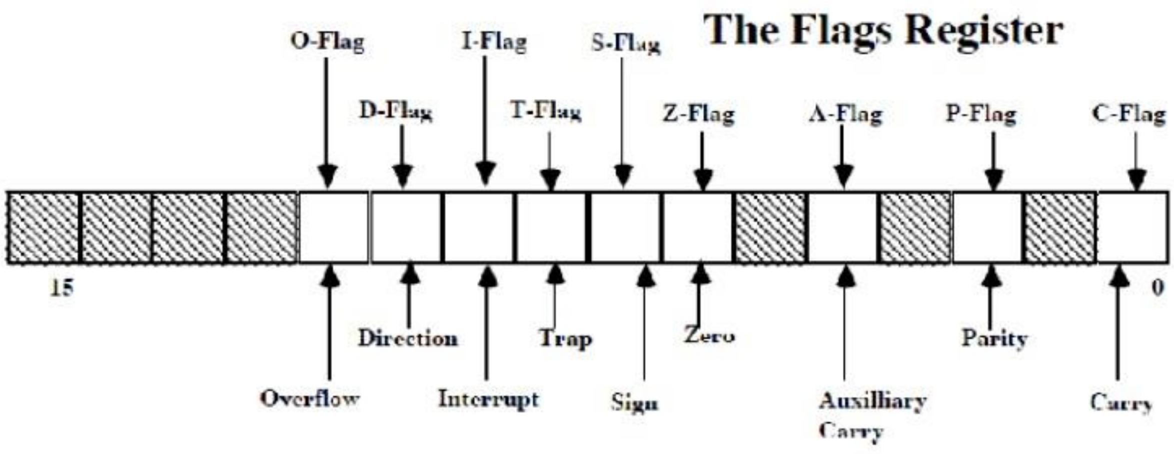
Other registers of 8086

1. **Instruction Pointer (IP)** is a 16-bit register. This is a crucially important register which is usedto control which instruction the CPU executes. The ip, or program counter, is used to store thememory location of the next instruction to be executed. The CPU checks the program counter toascertain which instruction to carry out next. It then updates the program counter to point to thenext instruction. Thus the program counter will always point to the next instruction to beexecuted.

2. **Flag Register** contains a group of status bits called flags that indicate the status of the CPU orthe result of arithmetic operations. There are two types of flags:

   1. The **status flags** which reflect the result of executing an instruction. The programmer cannotset/reset these flags directly.
   2. The **control flags** enable or disable certain CPU operations. The programmer can set/resetthese bits to control the CPU's operation.

Nine individual bits of the status register are used as control flags (3 of them) and status flags (6of them).The remaining 7 are not used.A flag can only take on the values 0 and 1. We say a flag is set if it has the value 1.The statusflags are used to record specific characteristics of arithmetic and of logical instructions.



Control Flags: There are three control flags
1. **The Direction Flag (D):** Affects the direction of moving data blocks by such instructions as MOVS, CMPS and SCAS. The flag values are 0 = up and 1 = down and can be set/reset by the STD (set D) and CLD (clear D) instructions.

2. **The Interrupt Flag (I):** Dictates whether or not system interrupts can occur. Interrupts areactions initiated by hardware block such as input devices that will interrupt the normal execution of programs. The flag values are 0 = disable interrupts or 1 = enable interrupts and can bemanipulated by the CLI (clear I) and STI (set I) instructions.

3. **The Trap Flag (T):** Determines whether or not the CPU is halted after the execution of each instruction. When this flag is set (i.e. = 1), the programmer can single step through his program to debug any errors. When this flag = 0 this feature is off. This flag can be set by the INT 3instruction.

Status Flags: There are six status flags

1. **The Carry Flag (C):** This flag is set when the result of an unsigned arithmetic operation istoo large to fit in the destination register. This happens when there is an end carry in an additionoperation or there an end borrows in a subtraction operation. A value of 1 = carry and 0 = nocarry.

2. **The Overflow Flag (O):** This flag is set when the result of a signed arithmetic operation is toolarge to fit in the destination register (i.e. when an overflow occurs). Overflow can occur whenadding two numbers with the same sign (i.e. both positive and both negative). A value of 1 =overflow and 0 = no overflow.

3. **The Sign Flag (S):** This flag is set when the result of an arithmetic or logic operation isnegative. This flag is a copy of the MSB of the result (i.e. the sign bit). A value of 1 meansnegative and 0 = positive.

4. **The Zero Flag (Z):** This flag is set when the result of an arithmetic or logic operation is equal to zero. A value of 1 means the result is zero and a value of 0 means the result is not zero.

5. **The Auxiliary Carry Flag (A):** This flag is set when an operation causes a carry from bit 3 to bit 4 (or a borrow from bit 4 to bit 3) of an operand. A value of 1 = carry and 0 = no carry.

6. **The Parity Flag (P):** This flags reflects the number of 1s in the result of an operation. If the number of 1s is even its value = 1 and if the number of 1s is odd then its value = 0.

Real Mode Memory Addressing

• Real-mode operation allows the microprocessor to address only first 1Mbyte of memory-space.
(The first 1M byte of memory is called the real memory, conventional memory or DOS memory system).

Segments & Offsets

• In real mode, a combination of a segment-address and an offset-address accesses a memory-location(Figure 2-3).

• The *segment-address* (located within one of the segment registers) defines the starting-address of any 64Kbyte memory-segment.

`The*offset-address*selects any location within the 64KB memory segment.

• Segments always have a length of 64KB.

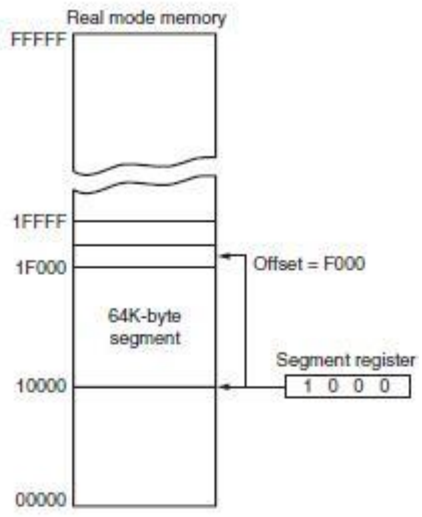
• Each segment-register is internally appended with a 0H on its rightmost end. This forms a 20-bit memory-address, allowing it to access the start of a segment. (For example, when a segment register contains 1200H, it addresses a 64Kbyte memory segment beginning at location 12000H).

• Because of the internally appended 0H, real-mode segments can begin only at a 16byte boundary in the memory. This 16-byte boundary is called a *paragraph.*

• Because a real-mode segment of memory is 64K in length, once the beginning address is known, the ending address is found by adding FFFFH.

• The offset-address is added to the start of the segment to address a memory location within the memory-segment. (For example, if the segment address is 1000H and the offset address is 2000H,the microprocessor addresses memory location 12000H).

• In the 80286, an extra 64K minus 16bytes of memory is addressable when the segment is FFFFH and the HIMEM.SYS driver for DOS is installed in the system. This area of memory is referred to as *high memory*.

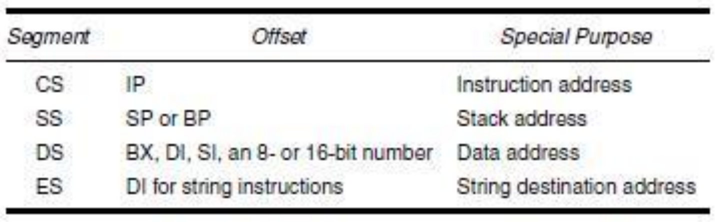


The real mode memory-addressing scheme using a segment address plus an offset

Default Segment & Offset Registers

• The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules define the segment-register and offset-register combination. For example, the CS register is always used with the IP to address the next instruction in a program.

• The CS register defines the start of the code-segment and the IP locates the next instruction within the code-segment. For example, if CS=1400H and IP=1200H, the microprocessor fetches its next instruction from memory location 14000H+1200H=15200H

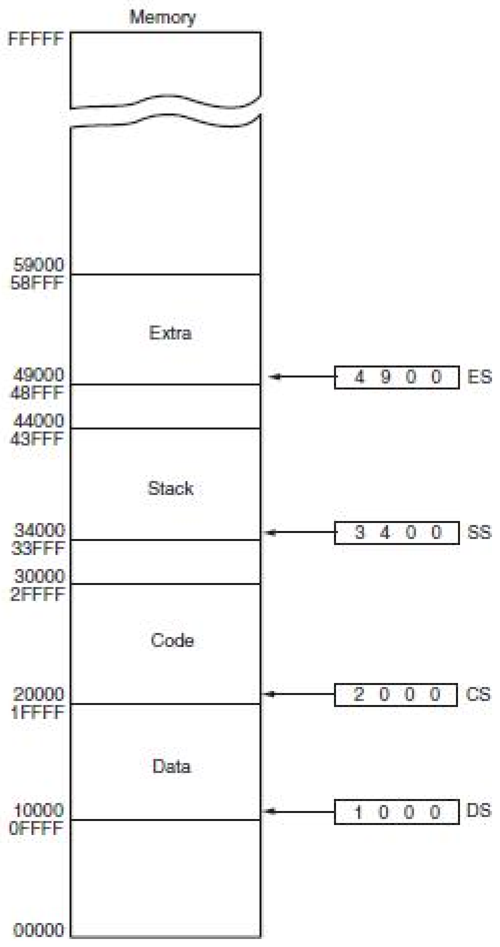


Segment	*Offset*	*Special Purpose*
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

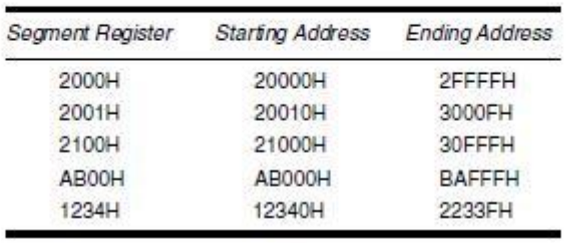
Default 16-bit segment and offset combinations.

Segment & Offset Addressing Scheme Allows Relocation

- This scheme allows both programs and data to be relocated in the memory .
- This also allows programs written to function in the real-mode to operate in a protected-mode system.
- Relocatable-program can be placed into any area of memory and executed without change.
- Relocatable-data can be placed in any area of memory and used without any change to the program.



F A memory system showing the placement of four memory segments



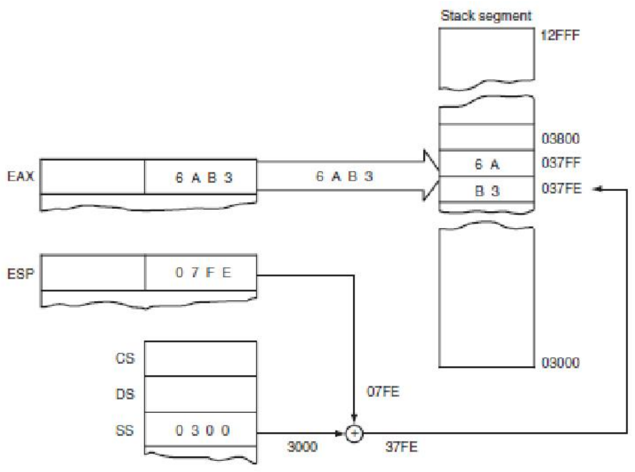
Segment Register	*Starting Address*	*Ending Address*
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

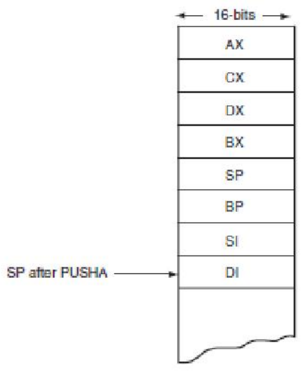
Example of real mode segment addresses

Data Transfer Instructions:

PUSH

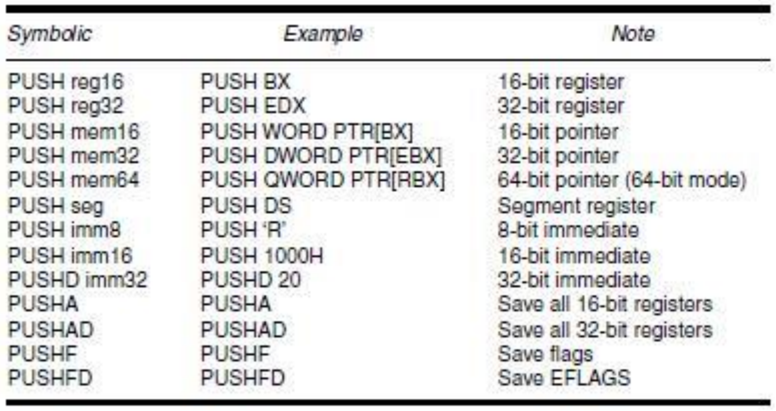
• In 8086, PUSH always transfers 2 bytes of data to the stack .
• Source of data may be
→any 16-bit register
→immediate-data
→any segment-register or
→any 2 bytes of memory-data
• Whenever data are pushed onto stack,
→first data byte moves to the stack-segment memory location addressed by SP-1
→second data byte moves to the stack-segment memory location addressed by SP-2
• After the data are pushed onto stack, SP is decrement by 2. (SP=SP-2)
• PUSHF(push flags) copies the contents of flag register to the stack.
• PUSHA(push all) copies the contents of the internal register-set(except the segment registers) to the stack in the following order: AX, CX, DX, BX, SP, BP, SI and DI .
• After all registers are pushed, SP is decremented by 16. (SP=SP-16)





(A)The effect of the PUSH AX instruction on ESP and stack memory locations 37FFH and 37FEH.This instruction is shown at the point after execution

(B) Operation of PUSHA



Symbolic	*Example*	*Note*
PUSH reg16	PUSH BX	16-bit register
PUSH reg32	PUSH EDX	32-bit register
PUSH mem16	PUSH WORD PTR[BX]	16-bit pointer
PUSH mem32	PUSH DWORD PTR[EBX]	32-bit pointer
PUSH mem64	PUSH QWORD PTR[RBX]	64-bit pointer (64-bit mode)
PUSH seg	PUSH DS	Segment register
PUSH imm8	PUSH 'R'	8-bit immediate
PUSH imm16	PUSH 1000H	16-bit immediate
PUSHD imm32	PUSHD 20	32-bit immediate
PUSHA	PUSHA	Save all 16-bit registers
PUSHAD	PUSHAD	Save all 32-bit registers
PUSHF	PUSHF	Save flags
PUSHFD	PUSHFD	Save EFLAGS

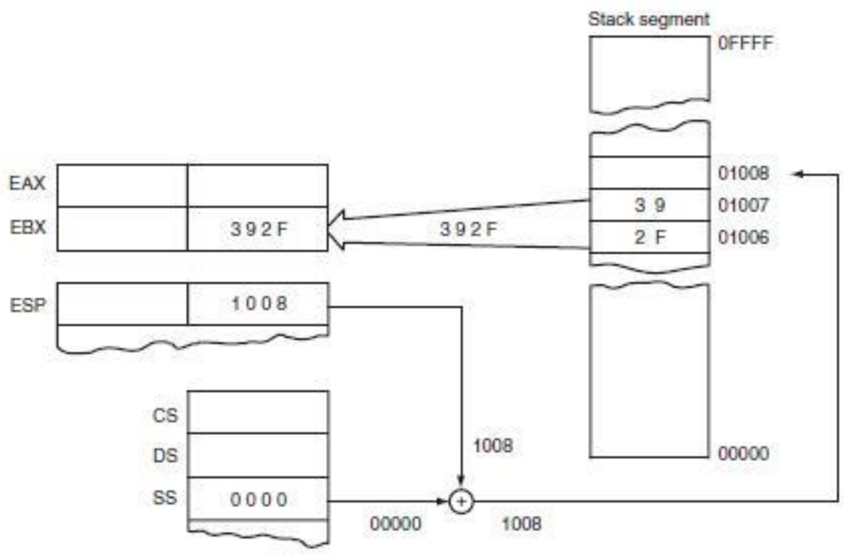
POP

• POP removes data from the stack &
places it into 16-bit register, segment-register or 16-bit memory-location .

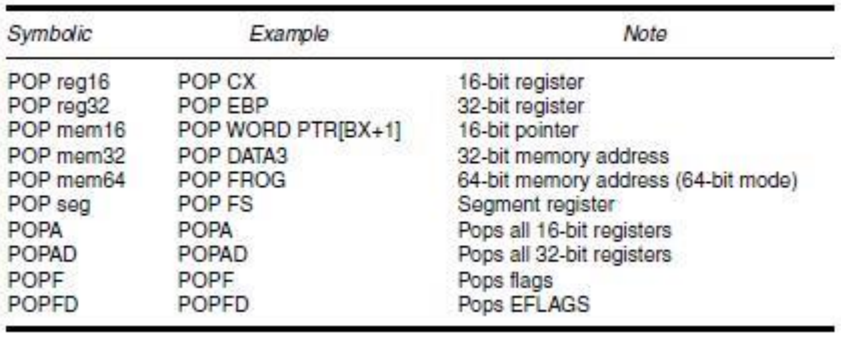
• POPF(pop flags)
→removes 2 bytes of data from the stack &
→places it into the flag-register

• POPA(pop all)
→removes 16 bytes of data from the stack &
→places them into the following registers (in the order: DI, SI, BP, SP, BX, DX, CX and AX).

• This instruction is not available as an immediate POP.



The POP BX instruction showing how data are removed from the stack. This instruction is shown after execution



Symbolic	Example	Note
POP reg16	POP CX	16-bit register
POP reg32	POP EBP	32-bit register
POP mem16	POP WORD PTR[BX+1]	16-bit pointer
POP mem32	POP DATA3	32-bit memory address
POP mem64	POP FROG	64-bit memory address (64-bit mode)
POP seg	POP FS	Segment register
POPA	POPA	Pops all 16-bit registers
POPAD	POPAD	Pops all 32-bit registers
POPF	POPF	Pops flags
POPFD	POPFD	Pops EFLAGS

LEA (Load-Effective Address)

• This loads a 16-bit register with the offset-address of the data specified by the operand

• LEA BX,[DI] ;this loads offset-address specified by [DI] into register BX.

Whereas MOV BX, [DI] ;this loads data stored at memory-location addressed by [DI] into BX.

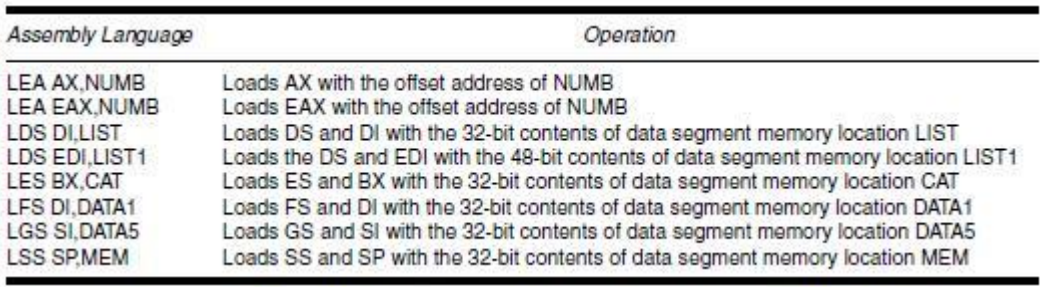
• The OFFSET directive performs the same function as an LEA instruction if the operand is a displacement. For example, the MOV BX, OFFSET LIST performs the same function as LEA BX, LIST.

• Why is the LEA instruction available if the OFFSET directive accomplishes the same task?

Ans: OFFSET only functions with simple operands such as LIST. It may not be used for an operand such as [DI], LIST [SI] and so on.

• OFFSET directive is more efficient than the LEA instruction for simple operands. Because it takes the microprocessor longer to execute the LEA BX, LIST instruction than the MOV BX, OFFSET LIST.

• The reason that the MOV BX, OFFSET LIST instruction executes faster is because the assembler calculates the offset address of LIST, whereas the microprocessor calculates the address for the LEA instruction.



Assembly Language	Operation
LEA AX,NUMB	Loads AX with the offset address of NUMB
LEA EAX,NUMB	Loads EAX with the offset address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST1	Loads the DS and EDI with the 48-bit contents of data segment memory location LIST1
LES BX,CAT	Loads ES and BX with the 32-bit contents of data segment memory location CAT
LFS DI,DATA1	Loads FS and DI with the 32-bit contents of data segment memory location DATA1
LGS SI,DATA5	Loads GS and SI with the 32-bit contents of data segment memory location DATA5
LSS SP,MEM	Loads SS and SP with the 32-bit contents of data segment memory location MEM

LDS, LES, LFS, LGS and LSS

• The LDS, LES, LFS, LGS and LSS instructions load

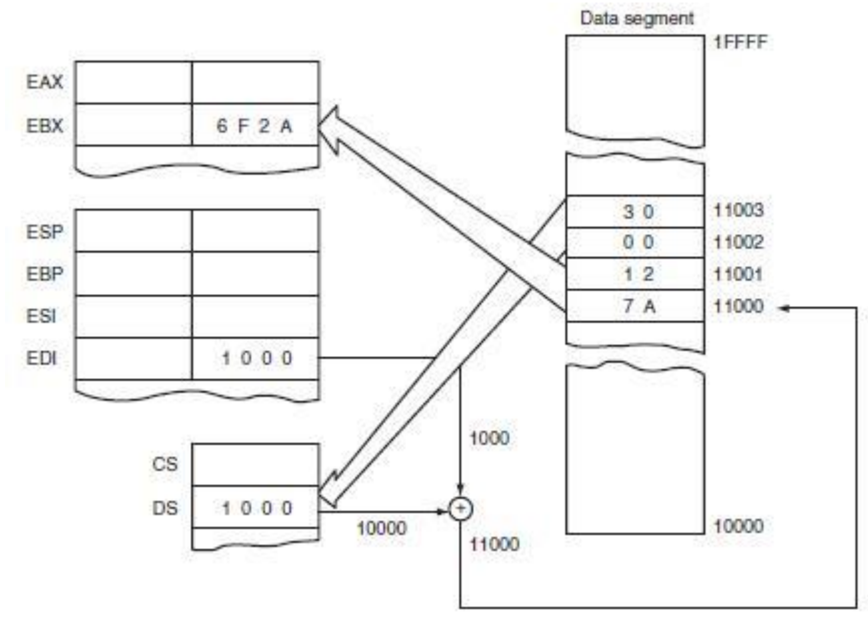
→any 16-bit register with an offset-address &

→DS, ES, FS, GS or SS with a segment-address.

• For example, LDS BX,[DI] ;this instruction transfers 32-bit number(addressed by DI in the segment) into the BX and DS registers .

• These instructions cannot use the register addressing-mode (MOD=11).

(These instructions use any of the memory-addressing modes to access a 32-bit section of memory that contains both the segment and offset address. The 32-bit section of memory contains a 16-bit offset and 16-bit segment address)



The LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H.This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.

String Data Transfers (LODS, STOS, MOVS, INS & OUTS)
The Direction flag

• D flag (located in flag-register) is used only with the string instructions.

• D flag selects the auto-increment(D=0) or the auto-decrement(D=1) operation for the DI and SI registers during string operations.

• CLD instruction clears the D flag(D=0) &
STD instruction sets the D flag(D=1).{.'. CLD instruction selects the auto-increment mode and STD selects the auto-decrement mode}

• Whenever a string instruction transfers a byte, DI and/or SI is incremented or decremented by 1. If a word is transferred, DI and/or SI is incremented or decremented by 2.

• For example, STOSB instruction uses the DI register to address a memory-location. When STOSB executes, only the DI register is incremented or decremented without affecting SI.

DI and SI

• The DI offset-address accesses data in the extra-segment for all string instructions that use it. The SI offset-address accesses data, by default, in the data-segment. (The DI segment assignment is always in the extra segment when a string instruction executes. This assignment cannot be changed)

• The reason that one pointer addresses data in the extra-segment and the other in the data-segment is so that the MOVS instruction can move 64KB of data from one segment of memory to another.

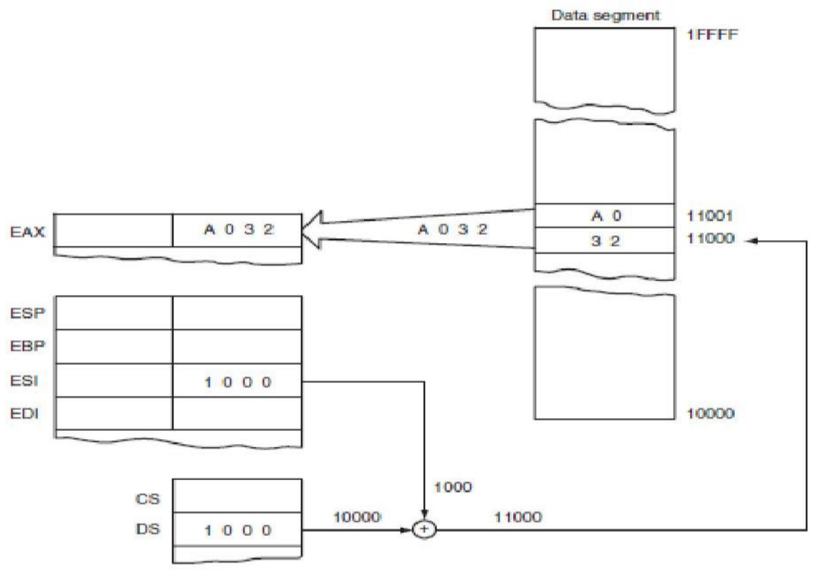
LODS(load string)

• This loads AL or AX with data stored at the data-segment offset-address indexed by SI(Table4-11).

• LODSB(loads a byte) instruction causes a byte to be loaded into AL
LODSW(loads a word) instruction causes a word to be loaded into AX (Figure 4-18).

• After loading AL with a bytes, SI is incremented if D=0 or decremented if D=1.

Assembly Language	*Operation*
LODSB	AL = DS:[SI]; SI = SI ± 1
LODSW	AX = DS:[SI]; SI = SI ± 2
LODSD	EAX = DS:[SI]; SI = SI ± 4
LODSQ	RAX = [RSI]; RSI = RSI ± 8 (64-bit mode)
LODS LIST	AL = DS:[SI]; SI = SI ± 1 (if LIST is a byte)
LODS DATA1	AX = DS:[SI]; SI = SI ± 2 (if DATA1 is a word)
LODS FROG	EAX = DS:[SI]; SI = SI ± 4 (if FROG is a doubleword)



The operation of the LODSW instruction if DS=1000H, D=0, 11000H=32, and 11001H=A0.This instruction is shown after AX is loaded from memory but before SI increments by 2.

STOS(store string)

- This stores AL or AX at the extra-segment memory-location addressed by DI .
- STOSB(stores a byte) instruction stores the byte in AL at the extra-segment memory-location addressed by DI STOSW(stores a word) instruction stores AX in extra-segment memory-location addressed by DI
- After the byte(AL) is stored, DI is incremented if D=0 or decremented if D=1.

STOS with a REP

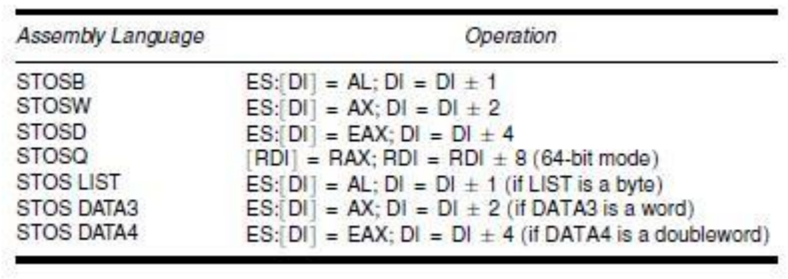
- The repeat prefix(REP) is added to any string data transfer instruction except the LODS instruction.('.' it doesn't make any sense to perform a repeated LODS operation)
- REP prefix causes CX to decrement by 1 each time the string instruction executes.

After CX decrement, the string instruction repeats.

If CX reaches 0, the instruction terminates and the program continues with the next sequential instruction.

(For example, if CX is loaded with 100 and a *REP STOSB* instruction executes, the microprocessor automatically repeats the STOSB instruction 100 times)

Forms of the STOS instruction



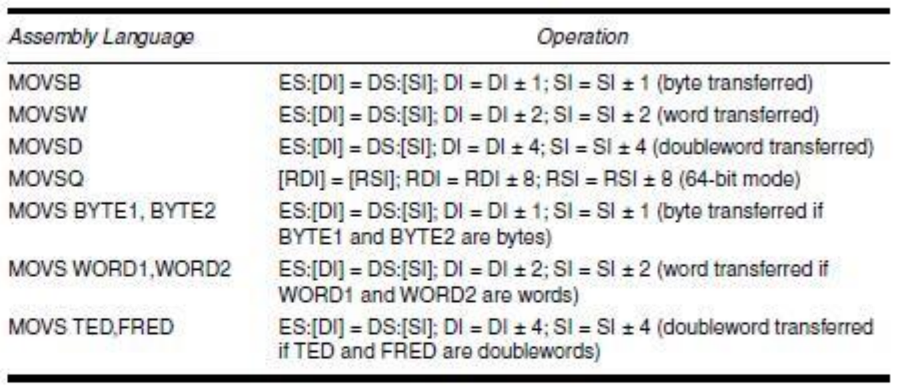
Assembly Language	*Operation*
STOSB	ES:[DI] = AL; DI = DI ± 1
STOSW	ES:[DI] = AX; DI = DI ± 2
STOSD	ES:[DI] = EAX; DI = DI ± 4
STOSQ	[RDI] = RAX; RDI = RDI ± 8 (64-bit mode)
STOS LIST	ES:[DI] = AL; DI = DI ± 1 (if LIST is a byte)
STOS DATA3	ES:[DI] = AX; DI = DI ± 2 (if DATA3 is a word)
STOS DATA4	ES:[DI] = EAX; DI = DI ± 4 (if DATA4 is a doubleword)

MOVS

- This transfers a bytes(or word) from the data-segment memory-location addressed by SI to the extra-segment memory-location addressed by DI (Table 4-14).

- SI/DI is incremented if D=0 or decremented if D=1.
- This is the only memory-to-memory transfer allowed in the 8086 microprocessor.

• The destination-operand must always be located in the extra-segment. While the source-operand located in the data-segment may be overridden so that another segment may be used.



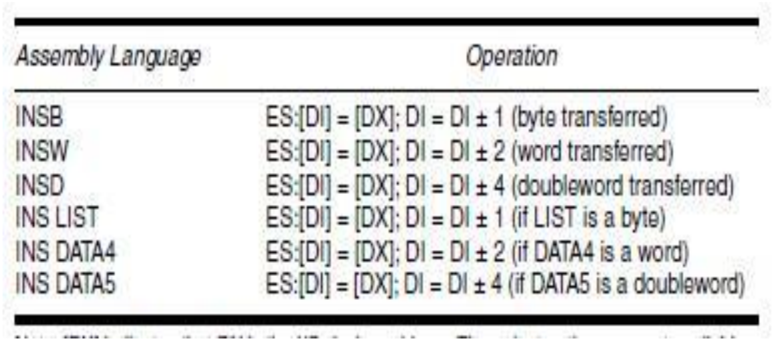
Assembly Language	Operation
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVSD	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred)
MOVSQ	[RDI] = [RSI]; RDI = RDI ± 8; RSI = RSI ± 8 (64-bit mode)
MOVS BYTE1, BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred if WORD1 and WORD2 are words)
MOVS TED,FRED	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred if TED and FRED are doublewords)

INS

• This transfers a byte (or word) of data from an I/O device into the extra-segment memory-location addressed by DI.
- I/O address is contained in the DX register .
- INSB inputs data from an 8-bit I/O device & stores it in byte-sized memory-location indexed by SI.

    INSW inputs 16-bit I/O data and stores it in a word-sized memory-location.

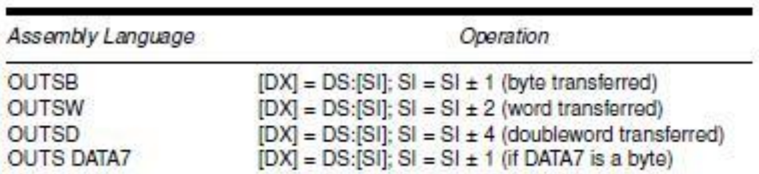
• This instruction can be repeated using the REP prefix, which allows an entire block of input-data to be stored inthe memory from an I/O device.



Assembly Language	Operation
INSB	ES:[DI] = [DX]; DI = DI ± 1 (byte transferred)
INSW	ES:[DI] = [DX]; DI = DI ± 2 (word transferred)
INSD	ES:[DI] = [DX]; DI = DI ± 4 (doubleword transferred)
INS LIST	ES:[DI] = [DX]; DI = DI ± 1 (if LIST is a byte)
INS DATA4	ES:[DI] = [DX]; DI = DI ± 2 (if DATA4 is a word)
INS DATA5	ES:[DI] = [DX]; DI = DI ± 4 (if DATA5 is a doubleword)

OUTS

- This transfers a byte (or word) of data from the data-segment memory-location address by SI to an I/O device.
- I/O device is addressed by DX register .



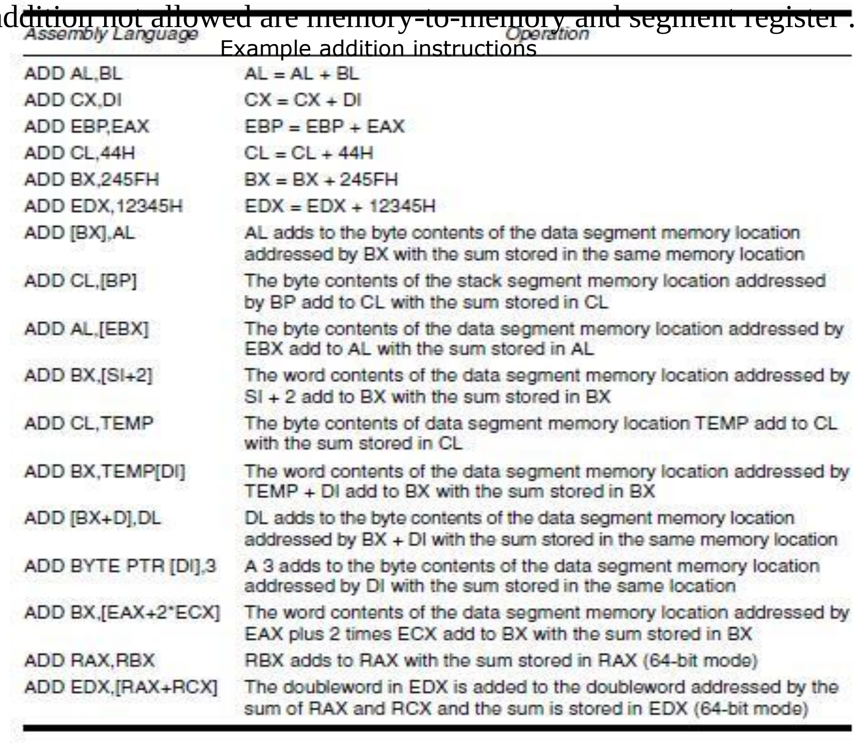
Assembly Language	Operation
OUTSB	[DX] = DS:[SI]; SI = SI ± 1 (byte transferred)
OUTSW	[DX] = DS:[SI]; SI = SI ± 2 (word transferred)
OUTSD	[DX] = DS:[SI]; SI = SI ± 4 (doubleword transferred)
OUTS DATA7	[DX] = DS:[SI]; SI = SI ± 1 (if DATA7 is a byte)

ARITHMETIC AND COMPARE INSTRUCTIONS

ADD(Addition)

• This is used to add byte/word of data of 2 registers or a register & a memory-location

• The only types of addition not allowed are memory-to-memory and segment register .



Example addition instructions

Assembly Language	Operation
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL
ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location
ADD BYTE PTR [DI],3	A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location
ADD BX,[EAX+2*ECX]	The word contents of the data segment memory location addressed by EAX plus 2 times ECX add to BX with the sum stored in BX
ADD RAX,RBX	RBX adds to RAX with the sum stored in RAX (64-bit mode)
ADD EDX,[RAX+RCX]	The doubleword in EDX is added to the doubleword addressed by the sum of RAX and RCX and the sum is stored in EDX (64-bit mode)

Register Addition

• Whenever arithmetic and logic instructions execute, the contents of the flag-register changes.

• Any ADD instruction modifies contents of sign, zero, carry, auxiliary, carry, parity& overflow flagsProgram to compute AX=BX+CX+DX.

```
ADD AX,BX
ADD AX,CX
ADD AX,DX
```

Immediate Addition

Immediate addition is employed whenever constant-data are added Program to add immediate data

```
MOV DL,12H
ADD DL,33H
```

Memory-to-Register Addition

Program to add two consecutive bytes of data(stored at the data segment offset locations NUMB and NUMB+1) to the AL register.

```
LEA DI,NUMB  ;address NUMB
MOV AL,0       ;clear sum
ADD AL,[DI]    ;add NMB
ADD AL,[DI+1]  ;add NUMB+1
```

Array Addition

Program to add the contents of array element at indices 3, 5 and 7

```
MOV AL,0                    ;clear sum
                           ;address element
MOV SI,3                   3
ADD AL,ARRAY[SI]           ;add element 3
ADD
AL,ARRAY[SI+2]            ;add element 5
ADD
AL,ARRAY[SI+4]            ;add element 7
```

INC(Increment Addition)

• This adds 1 to any register or memory-location, except a segment-register.

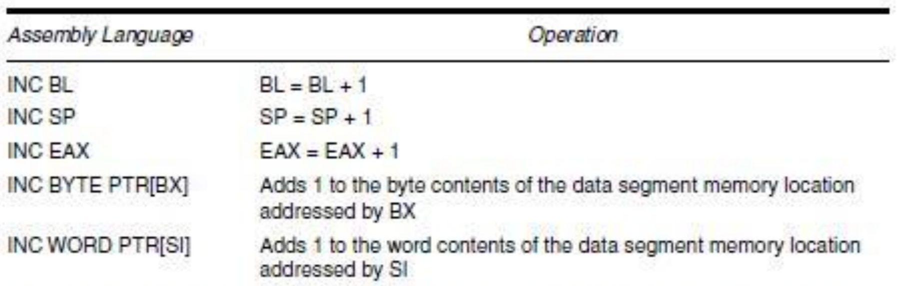
• With indirect memory increments, the size of the data must be described by using the BYTE PTR, WORD PTR directives.

The reason is that the assembler cannot determine if, for example, INC [DI] instruction is a byte or word sized increment.

The INC BYTE PTR[DI] instruction clearly indicates byte sized memory data.

Using INC instruction, program to add two consecutive bytes of data(stored at the data segment offset locations NUMB and NUMB+1) to the AL register.

```
LEA
DI,NUMB        ;address
MOV AL,0       ;clear sum
ADD
AL,[DI]        ;add NUMB
INC DI         ;increment DI
ADD
AL,[DI]        ;add NUMB+1
```



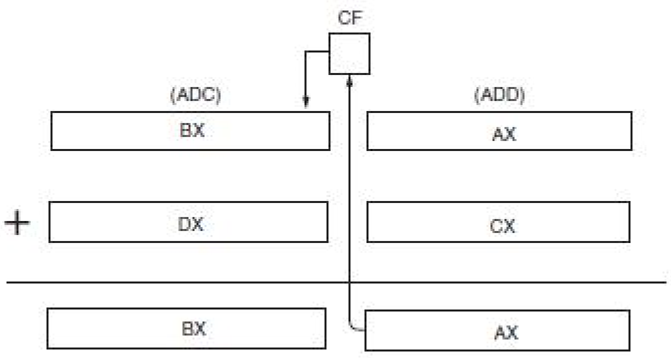
Assembly Language	Operation
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI

ADC(Addition with Carry)

• This adds the bit in the carry-flag(C) to the operand-data .

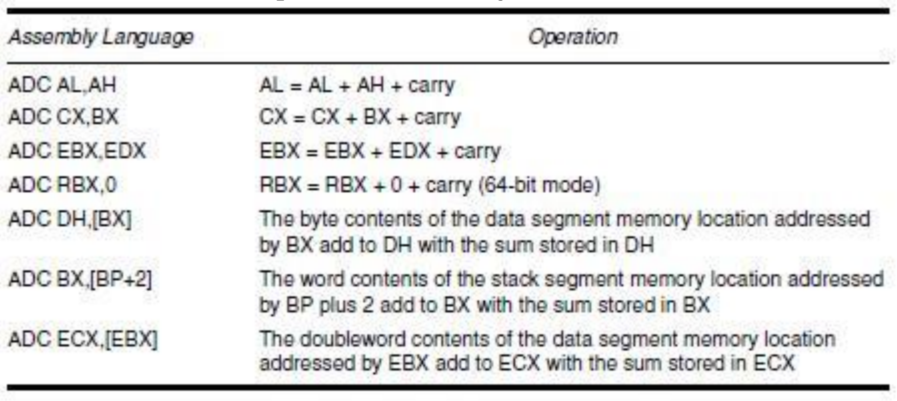
• This mainly appears in software that adds numbers that are wider than 16 bits in the 8086. Example 5-7: To add the 32-bit number in BX and AX to the 32-bit number in DX and CX

```
ADD AX,CX
ADC BX,DX
```



ADC showing how the carry flag(C) links the two 16-bit additions into one 32-bit addition

Example add-with-carry instructions



Assembly Language	Operation
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)
ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with the sum stored in DH
ADC BX,[BP+2]	The word contents of the stack segment memory location addressed by BP plus 2 add to BX with the sum stored in BX
ADC ECX,[EBX]	The doubleword contents of the data segment memory location addressed by EBX add to ECX with the sum stored in ECX

XADD(Exchange & Add)

• This instruction is used in 80486-Core2 microprocessors.

• This adds the source to the destination and stores the sum in the destination.

• The difference is that after the addition takes place, the original value of the destination is copied into the source-operand. For example, if BL=12H and DL=02H,

XADD BL,DL instruction executes,

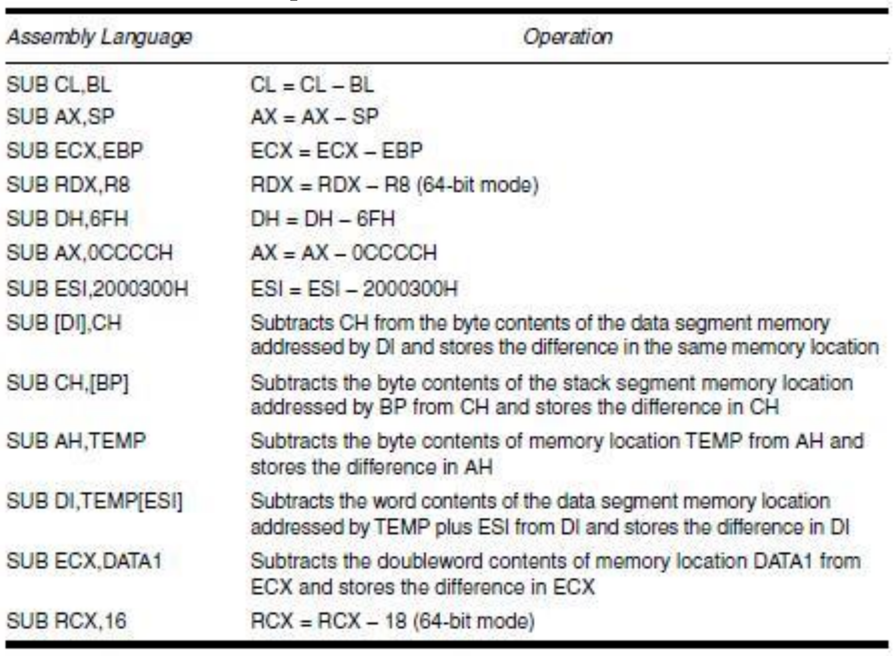
BL register contains the sum 14H and DL becomes 12H

SUB(Subtraction)

• Only types of subtraction not allowed are memory-to-memory and segment register subtractions.

• This affects the flag bits .

Example subtraction instructions



Assembly Language	Operation
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
SUB RDX,R8	RDX = RDX – R8 (64-bit mode)
SUB DH,6FH	DH = DH – 6FH
SUB AX,0CCCCH	AX = AX – 0CCCCH
SUB ESI,2000300H	ESI = ESI – 2000300H
SUB [DI],CH	Subtracts CH from the byte contents of the data segment memory addressed by DI and stores the difference in the same memory location
SUB CH,[BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH
SUB AH,TEMP	Subtracts the byte contents of memory location TEMP from AH and stores the difference in AH
SUB DI,TEMP[ESI]	Subtracts the word contents of the data segment memory location addressed by TEMP plus ESI from DI and stores the difference in DI
SUB ECX,DATA1	Subtracts the doubleword contents of memory location DATA1 from ECX and stores the difference in ECX
SUB RCX,16	RCX = RCX – 18 (64-bit mode)

Register Subtraction

To subtract the 16-bit contents of registers CX and DX from the contents of register BX

```
SUB BX,CX
SUB BX,DX
```

Immediate Subtraction

To subtract 44H from 22H

```
MOV CH,22H
SUB CH,44H
```

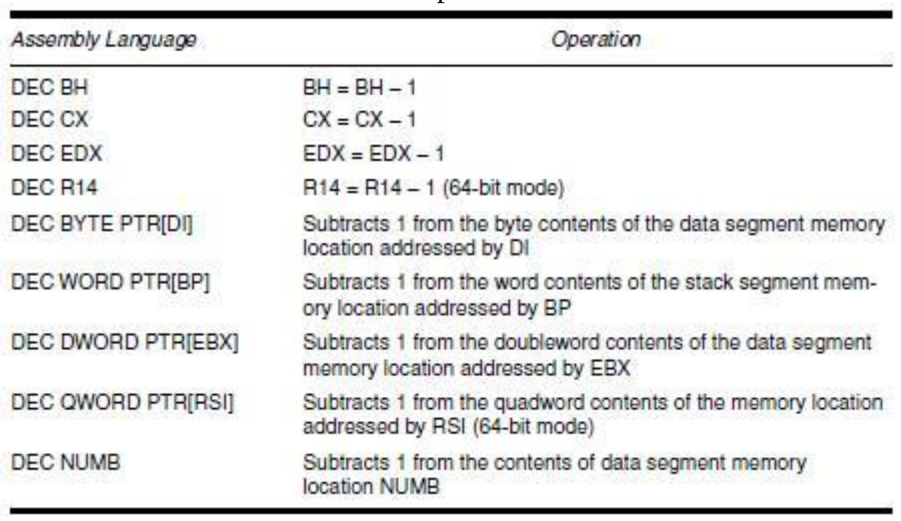
After the subtraction, the difference(0DEH) moves into the CH register. The flags change as follows for this subtraction:

Z=0(result not zero) C=1(borrow)
A=1(half borrow)
S=1(result negative)
P=1(even parity)
O=0(no overflow)

DEC(Decrement Subtraction)

- This subtracts 1 from a register or the contents of a memory-location .
- The decrement indirect memory-data instructions require BYTE PTR or WORD PTR because the assembler cannot distinguish a byte from a word when an index-register addresses memory.
- For example, DEC [SI] is unclear because the assembler cannot determine whether the location addressed by SI is a byte or word.
- Using DEC BYTE PTR[SI],DEC WORD PTR[DI] tells the size of the data to the assembler.

Table 5-5: Example decrement instructions



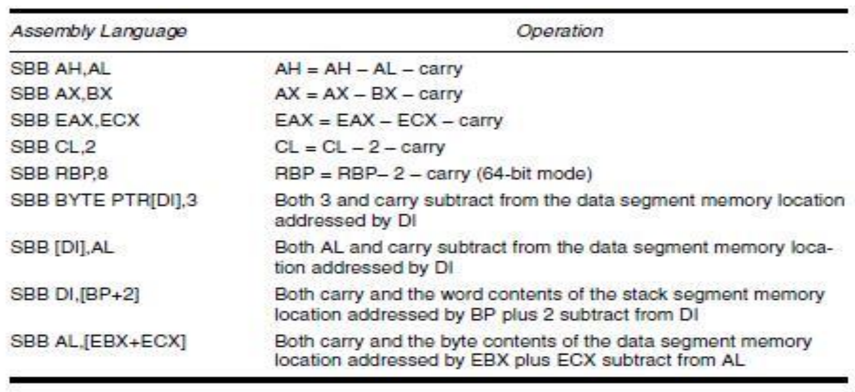
Assembly Language	Operation
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
DEC R14	R14 = R14 – 1 (64-bit mode)
DEC BYTE PTR[DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR[BP]	Subtracts 1 from the word contents of the stack segment memory location addressed by BP
DEC DWORD PTR[EBX]	Subtracts 1 from the doubleword contents of the data segment memory location addressed by EBX
DEC QWORD PTR[RSI]	Subtracts 1 from the quadword contents of the memory location addressed by RSI (64-bit mode)
DEC NUMB	Subtracts 1 from the contents of data segment memory location NUMB

Subtraction-with-Borrow(SBB)

• This functions as a regular subtraction except that the carry flag(which holds the borrow) also subtracts from thedifference .

Program to subtract BX-AX from SI-DI

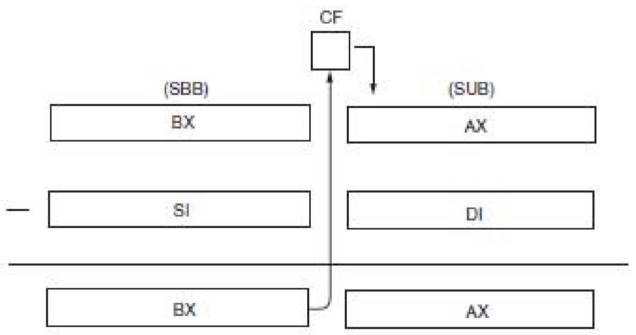
```
SUB AX,DI
SBB BX,SI
```



Assembly Language	Operation
SBB AH,AL	AH = AH – AL – carry
SBB AX,BX	AX = AX – BX – carry
SBB EAX,ECX	EAX = EAX – ECX – carry
SBB CL,2	CL = CL – 2 – carry
SBB RBP,8	RBP = RBP– 2 – carry (64-bit mode)
SBB BYTE PTR[DI],3	Both 3 and carry subtract from the data segment memory location addressed by DI
SBB [DI],AL	Both AL and carry subtract from the data segment memory location addressed by DI
SBB DI,[BP+2]	Both carry and the word contents of the stack segment memory location addressed by BP plus 2 subtract from DI
SBB AL,[EBX+ECX]	Both carry and the byte contents of the data segment memory location addressed by EBX plus ECX subtract from AL

Subtraction-with-borrow showing how the carry flag propagates the borrow

Example subtraction-with-borrow instructions

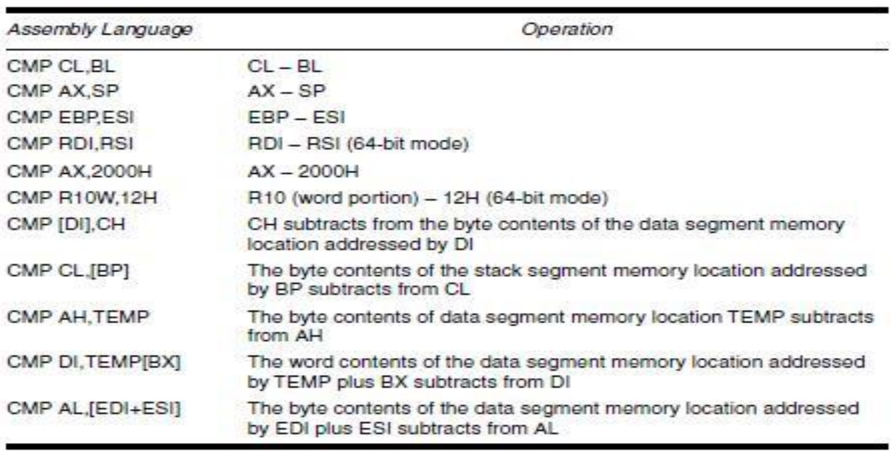


CMP(Comparison)

• This is a subtraction that changes only the flag
its,destination-operand never changes

• This is normally followed by a conditional jump instruction (which tests condition of the flag-bits)

• Only types of comparison not allowed are memory-to-memory and segment register comparisons



Assembly Language	*Operation*
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP RDI,RSI	RDI – RSI (64-bit mode)
CMP AX,2000H	AX – 2000H
CMP R10W,12H	R10 (word portion) – 12H (64-bit mode)
CMP [DI],CH	CH subtracts from the byte contents of the data segment memory location addressed by DI
CMP CL,[BP]	The byte contents of the stack segment memory location addressed by BP subtracts from CL
CMP AH,TEMP	The byte contents of data segment memory location TEMP subtracts from AH
CMP DI,TEMP[BX]	The word contents of the data segment memory location addressed by TEMP plus BX subtracts from DI
CMP AL,[EDI+ESI]	The byte contents of the data segment memory location addressed by EDI plus ESI subtracts from AL

CMPXCHG(Compare & Exchange)

• This is used only in 80486-Core2 microprocessor.

• This compares the destination-operand with the accumulator(AX).

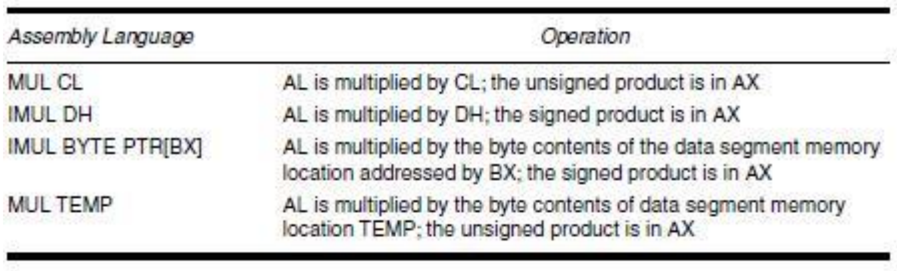
• If they are equal, the source-operand is copied into the destination;
if they are not equal, the destination-operand is copied into the accumulator.

• For example,*CMPXCHG CX, DX;*this instruction first compares the contents of CX with

AX.If CX=AX, DX is copied into AX;
otherwise CX is copied into AX.

8-bit Multiplication(MUL)

• The multiplicand is always in the AL register.
  The multiplier can be any 8-bit register or any memory location .
• The product after a multiplication is always a double-width product(AX).
• Immediate multiplication is not allowed.
• This contains one operand because
  it always multiplies the operand times the contents of register AL.
    For example, *MUL BL* ;this instruction multiplies the unsigned-contents of AL by the unsigned-contents of BL.
• For signed multiplication(IMUL),
  the product is in binary-form, if positive &
    in 2's complement forms if negative.



Assembly Language	*Operation*
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

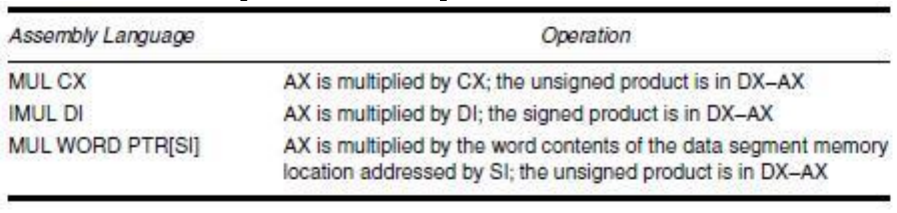
16-bit Multiplication

• AX contains the multiplicand while any 16-bit general purpose register contains the multiplier.
• The 32-bit product is stored in DX-AX.
  DX always contains the most significant 16-bits of the product, and
    AX contains the least significant 16 bits.

A Special Immediate 16-bit Multiplication

• This instruction is available in 80286-Core2 microprocessors.
• This contains 3 operands.
  →First operand is 16-bit destination-register;
  →Second operand is a register or memory-location that contains the 16-bit multiplicand &
  →Third operand is either 8-bit or 16-bit immediate-data used as the multiplier
• Example: IMUL CX,DX,12H ;this instruction multiplies 12H times DX and leaves a 16-bit signed product in CX.

Example 16-bit multiplication instructions



Assembly Language	*Operation*
MUL CX	AX is multiplied by CX; the unsigned product is in DX–AX
IMUL DI	AX is multiplied by DI; the signed product is in DX–AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX–AX

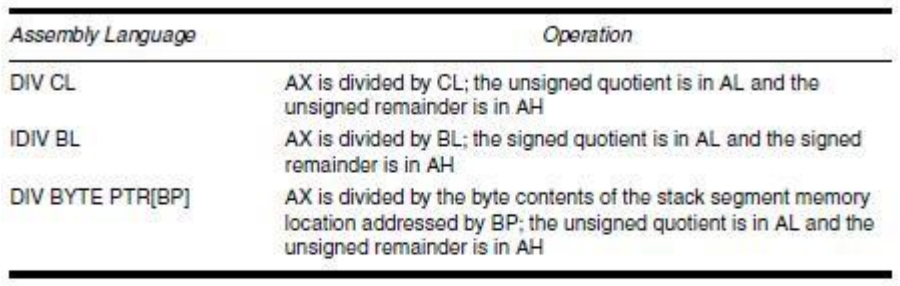
8-bit Division(DIV)

• AX register is used to store the dividend that is divided by the contents of any 8-bit register or memory-location

• After division, quotient appears in AL and
remainder appears in AH.

• For a signed division, quotient is positive or negative;
the remainder always assumes the sign of the dividend and is always an integer. For example, if AX=0010(+16) and BL=0FDH(-3) and
IDIV BL instruction executes AX=01FBH (This represents a quotient of -5(AL) with a remainder of 1(AH))

Example 8-bit division instructions



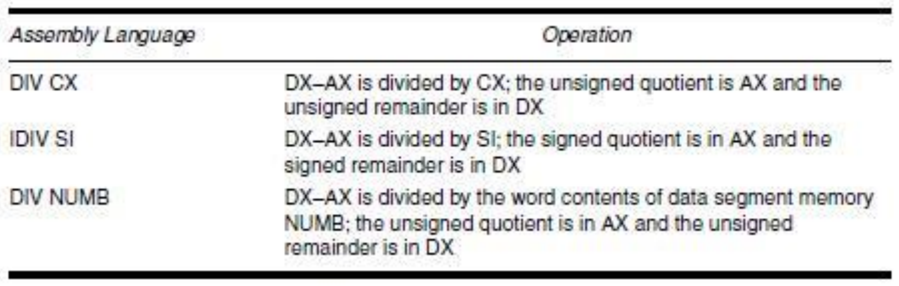
Assembly Language	*Operation*
DIV CL	AX is divided by CL; the unsigned quotient is in AL and the unsigned remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the signed remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP; the unsigned quotient is in AL and the unsigned remainder is in AH

16-Bit Division

• DX-AX register is used to store the dividend that is divided by the contents of any 16-bit register or memory-location .

• After division, the quotient appears in AX and the remainder appears in DX.

Example 16-bit division instructions



Assembly Language	*Operation*
DIV CX	DX–AX is divided by CX; the unsigned quotient is AX and the unsigned remainder is in DX
IDIV SI	DX–AX is divided by SI; the signed quotient is in AX and the signed remainder is in DX
DIV NUMB	DX–AX is divided by the word contents of data segment memory NUMB; the unsigned quotient is in AX and the unsigned remainder is in DX