8086 Instruction Set and Assembly Language Programming

Microprocessor and Micro-Controller

Course Material – EC 403

By

Er. Mrs. Saraswati Saha,

Assistant Professor, ECE Department,

RCCIIT, Kolkata

wants to get data from the data segment using BP as the offset register in this instruction, the instruction should be modified as MOV AX, DS: [BP].

Table 14.2 shows the instructions that address memory segments other than the default ones.

Table 14.2 Instructions that include the segment override prefix

Instruction	Default segment	Accessed segment
MOV BX, ES:[BP]	SS	ES
MOV BX, SS:[DI]	DS	SS
MOV CX, ES:[BX]	DS	ES
MOV CX, ES:[SI]	DS	ES
MOV AX, CS:[BX]	DS	CS

14.3 INSTRUCTION SET OF 8086

The instructions of the 8086 are classified as data transfer, arithmetic, logical, flag manipulation, control transfer, shift/rotate, string, and machine control instructions.

14.3.1 Data Transfer Instructions

The data transfer instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LSS, LAHF, and SAHF. These instructions are discussed here in detail:

MOV The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number. The general format of the MOV instruction is MOV destination, source.

Example:

```
(a) MOV BL, 50H        ; Move immediate data 50H to BL.
(b) MOV CX, [BX]       ; Copy the word from the memory at [BX] to CX.
(c) MOV AX, CX         ; Copy the contents of CX to AX.
```

Note: [BX] indicates the memory location at the offset address specified by BX in the data segment.

PUSH The PUSH instruction is used to store the word in a register or a memory location into the stack, as explained in the stack addressing mode. SP is decremented by two after the execution of PUSH.

Example:

```
(a) PUSH CX            ; PUSH the content of CX into the stack.
(b) PUSH DS            ; PUSH the content of DS into the stack.
(c) PUSH [BX]          ; PUSH the word in the memory at [BX] into the stack.
```

POP The POP instruction copies the top word from the stack to a destination specified in the instruction. The destination can be a general-purpose register, a segment register, or a memory location. After the word is copied to the specified destination, SP is incremented by two.

Example:

```
(a) POP BX          ; Pop the content of BX from the stack.
(b) POP DS          ; Pop the content of DS from the stack.
(c) POP [SI]        ; Pop a word from the stack and store it in
                      the memory at [SI].
```

Note: [SI] indicates the memory location in the data segment at the offset address specified by SI.

XCHG The XCHG instruction exchanges the contents of a register with the contents of a memory location. It cannot exchange the contents of two memory locations directly. The source and destination must both be either words or bytes. The segment registers cannot be used in this instruction.

Example:

```
(a) XCHG AL, BL     ; Exchanges the content of AL and BL.
(b) XCHG CX, BX     ; Exchanges the content of CX and BX.
(c) XCHG AX, [BX]   ; Exchanges the content of AX with the
                      content of the memory at [BX].
```

XLAT The XLAT instruction is used to translate a byte in AL from one code to another code. The instruction replaces a byte in the AL register with a byte in the memory at [BX], which is one of the data items present in a look-up table.

Before XLAT is executed, the look-up table containing the desired codes must be put in the data segment and the offset address of the starting location of the look-up table is stored in BX. The code byte to be translated is put in AL. When XLAT is executed now, it adds the content of the AL with BX to find the offset address of the data in the look-up table. Further, the byte in that offset address will get copied to AL.

IN The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data is stored in AL and if a 16-bit port is read, the data is stored in AX. The IN instruction has two formats—fixed port and variable port.

In the *fixed port type* IN instruction, the 8-bit address of a port is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

Example:

```
IN AL, 80H          ; Input a byte from the port with address
                      80H to AL.
IN AX, 40H          ; Input a word from the port with address
                      40H to AX.
```

For the *variable port type* IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16-bit register, the port address can be any number between 0000H and FFFFH. Hence, we will be able to address up to 65,536 ports in this mode. The following example shows a part of a program having the IN instruction. The operations done when the instructions are executed are given in the corresponding comment fields.

Example:

```
MOV DX, 0FE50H        ; Initialize DX with the port address of FE50H.
IN AL, DX             ; Input a byte from the 8-bit port with port address
                        FE50H into AL.
IN AX, DX             ; Input a word from the 16-bit port with port address
                        FE50H into AX.
```

The drawback of the fixed port type IN instruction is that the port address cannot be changed once the program is stored in the ROM. The variable port type IN instruction has the advantage that the port address can be computed in the program during execution, and by loading it in DX, the corresponding port can be accessed using the IN instruction.

OUT The OUT instruction transfers a byte from AL or a word from AX to the specified port. Similar to the IN instruction, the OUT instruction has two forms—fixed port and variable port.

Examples for fixed port OUT instruction:

```
(a) OUT 48H, AL       ; Sends the content of AL to the port with address 48H.
(b) OUT 0F0H, AX      ; Sends the content of AX to the port with address F0H.
```

Examples for variable port OUT instruction:

The following example shows a part of a program having the OUT instruction.

```
MOV DX, 1234H         ; Load the port address 1234H in DX.
OUT DX, AL            ; Send the content of AL to the port with address 1234H.
OUT DX, AX            ; Send the content of AX to the port with address 1234H.
```

LEA (load effective address) The general format of the LEA instruction is LEA register, source. This instruction determines the offset address of the variable or memory location called the source and puts this offset address in the indicated 16-bit register.

Example:

```
(a) LEA BX, COST      ; Load BX with the offset address of COST in the data
                        segment, where COST is the name assigned to a memory
                        location in the data segment.
(b) LEA CX, [BX] [SI]; Load CX with the value equal to (BX) + (SI), where
                        (BX) and (SI) represent the content of BX and SI,
                        respectively.
```

LDS This instruction loads the register and DS with words from the memory. The general form of this instruction is LDS register, memory address of first word.

The LDS instruction copies a word from the memory location specified in the instruction into the register, and then copies a word from the next memory location into the DS register. LDS is useful in initializing the SI and DS registers at the start of a string before using one of the string instructions.

Example:

```
LDS SI, [2000H]        ; Copy the content of the memory at the offset address
                         2000H in the data segment to the lower-order byte of
                         SI, and the content of 2001H to the higher-order byte
                         of SI. Copy the content at the offset address 2002H
                         in the data segment to the lower-order byte of DS and
                         the content of 2003H to the higherorder byte of DS.
```

LES and LSS The LES and LSS instructions are similar to the LDS instructions, except that instead of the DS register, the ES and SS registers, respectively, are loaded, along with the register specified in the instruction.

LAHF This instruction copies the lower-order byte of the flag register into AH.

SAHF This instruction stores the content of AH in the lower-order byte of the flag register.

Except the SAHF and POPF instructions, no other data transfer instruction affects the flag register.

14.3.2 Arithmetic Instructions

The arithmetic instructions in the 8086 are used to perform addition, addition with carry, subtraction, subtraction with borrow, increment, decrement, negation (changing sign), comparison, multiplication, division, decimal-adjust after addition, decimal-adjust after subtraction, and processing of ASCII data. Let us now discuss each instruction in detail.

ADD The general format of the ADD instruction is ADD destination, source.

The data from the source and destination are added and the result is placed in the destination. The source may be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and destination cannot both be memory locations. The data from the source and destination must be of the same type (either bytes or words).

Example:

```
(a) ADD BL, 80H        ; Add the immediate data 80H to BL.
(b) ADD CX, 12B0H      ; Add the immediate data 12B0H to CX.
(c) ADD AX, CX         ; Add the content of AX and CX and store the result
                         in AX.
(d) ADD AL, [BX]       ; Add the content of AL and the byte from the memory at
                         [BX] and store the result in AL.
```

(e) `ADD CX, [SI]` ; Add the content of CX and the word from the memory at [SI] and store the result in CX.

(f) `ADD [BX], DL` ; Add the content of DL with the byte from the memory at [BX] and store the result in the memory at [BX].

The flags AF, CF, OF, PF, SF, and ZF are affected by the execution of the ADD instruction.

ADC This instruction adds the data in the source and destination with the content of the carry flag and stores the result in the destination. The general format of this instruction is ADC destination, source.

All the rules specified for ADD are applicable to the ADC instruction.

SUB The general form of the subtract (SUB) instruction is SUB destination, source. It subtracts the number in the source from the number in the destination and stores the result in the destination. Like the ADD instruction, the source may be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and destination cannot both be memory locations. The data from the source and destination must be of the same type (either bytes or words).

For subtraction, the carry flag (CF) functions as the borrow flag. If the result is negative after subtraction, CF is set. Otherwise, it is reset. The flags AF, CF, OF, PF, SF, and ZF are affected by the SUB instruction.

Example:

(a) `SUB AL, BL` ; Subtract BL from AL and store the result in AL.

(b) `SUB CX, BX` ; Subtract BX from CX and store the result in CX.

(c) `SUB BX, [DI]` ; Subtract the word in the memory at [DI] from BX and store the result in BX.

(d) `SUB [BP], DL` ; Subtract DL from the byte in the memory at [BP] and store the result in the memory at [BP].

SBB Subtract with borrow—The general form of this instruction is SBB destination, source. The SBB instruction subtracts the content of the source and the carry flag from the content of the destination and stores the result in the destination. The rules for the source and the destination are same as that for the SUB instruction. AF, CF, OF, PF, SF, and ZF are affected by this instruction.

INC The increment (INC) instruction adds 1 to the content of a specified register or a memory location. The data incremented may be a byte or word. While the carry flag is not affected by this instruction, the flags AF, OF, PF, SF, and ZF are affected.

Example:

(a) `INC CL` ; Increment the content of CL by 1.

(b) `INC AX` ; Increment the content of AX by 1.

(c) `INC BYTE PTR [BX]` ; Increment the byte in the memory at [BX] by 1.

(d) `INC WORD PTR [SI]` ; Increment the word in the memory at [SI] by 1.

In these examples, the terms BYTE PTR and WORD PTR are assembler directives, which are used to specify the type of data (byte or word) to be incremented in the memory.

DEC The decrement (DEC) instruction subtracts 1 from the content of a specified register or memory location. The data decremented may be a byte or a word. CF is not affected, but AF, OF, PF, SF, and ZF flags are affected by this instruction.

NEG The negate (NEG) instruction replaces the byte or word in the specified register or memory location by its 2's complement (i.e., changes the sign of the data). The CF, AF, SF, PF, ZF, and OF flags are affected by this instruction.

Example:

```
(a) NEG AL          ; Take 2's complement of the data in AL and store it
                      in AL.
(b) NEG CX          ; Take 2's complement of the data in CX and store it
                      in CX.
(c) NEG BYTE PTR [BX] ; Take 2's complement of the byte in the memory at
                      [BX] and store the result in the same place.
(d) NEG WORD PTR [SI] ; Take 2's complement of the word in the memory at
                      [SI] and store the result in the same place.
```

CMP The general form of the compare (CMP) instruction is CMP destination, source. This instruction compares a byte or word in the source with a byte or word in the destination and affects only the flags, according to the result. The content of the source and destination are not affected by the execution of this instruction. The comparison is done by subtracting the content of the source from that of the destination. The AF, OF, SF, ZF, PF, and CF flags are affected by the instruction. The rules for the source and destination are the same as those for the SUB instruction.

Example:

After the instruction CMP AX, DX is executed, the status of CF, ZF, and SF will be as follows:

	CF	ZF	SF
If AX = DX	0	1	0
If AX > DX	0	0	0
If AX < DX	1	0	1

MUL The multiply (MUL) instruction is used for multiplying two unsigned bytes or words. The general form of the MUL instruction is MUL source. The source can be a byte or a word from a register or memory location, which is considered as the multiplier. The multiplicand is taken by default from AL and AX for byte and word type data, respectively. The result of multiplication is stored in AX and DX–AX (i.e., the most significant word of the result in DX and the least significant word of the result in AX) for byte and word type data, respectively. (*Note*: Multiplying two 8-bit data gives a 16-bit result and multiplying two 16-bit data gives a 32-bit result.)

Example:

```
MUL CH              ; Multiply AL and CH and store the result in AX.
MUL BX              ; Multiply AX and BX and store the result
                      in DX-AX.
MUL BYTE PTR [BX]   ; Multiply AL with the byte in the memory at
                      [BX] and store the result in DX-AX.
```

If the most significant byte of the 16-bit result is 00H or the most significant word of a 32-bit result is 0000H, both CF and OF will be 0. Checking these flags allows us to decide whether the leading 0s in the result have to be discarded or not. The AF, PF, SF, and ZF flags are undefined (i.e., a random number is stored in these bits) after the execution of the MUL instruction.

IMUL The IMUL instruction is used for multiplying the signed byte or word in register or memory location with AL or AX, and store the result in AX or DX–AX respectively. If the magnitude of the result does not require all the bits of the destination, the unused bits are filled with copies of the sign bit.

If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0s or all 1s), CF and OF will both be 0. Otherwise, both will be 1. AF, PF, SF, and ZF are undefined after IMUL.

To multiply a signed byte by a signed word, the byte is moved into a word location and the upper byte of the word is filled with copies of the sign bit. If the byte is moved into AL, using the CBW (convert byte to word) instruction, the sign bit in AL is extended into all the bits of AH. Thus, AX contains the 16-bit sign-extended word.

Example:

```
IMUL BL             ; Multiply AL with BL and store the result
                      in AX.
IMUL AX             ; Multiply AX and AX and store the result
                      in DX-AX.
IMUL BYTE PTR [BX]  ; Multiply AL with the byte from the memory
                      at [BX] and store the result in AX.
IMUL WORD PTR [SI]  ; Multiply AX with the word from the memory
                      at [SI] and store the result in DX-AX.
```

DIV The divide (DIV) instruction is used for dividing unsigned data. The general form of the DIV instruction is DIV source, where 'source' is the divisor. It can be a byte or word in a register or memory location. The dividend is taken by default from AX and DX–AX for byte and word type data division, respectively. Table 14.3 shows the complete details of the DIV instruction.

Table 14.3 Details of DIV instruction

Dividend (bits)	Divisor (bits)	Quotient (bits)	Remainder (bits)
AX (16)	Source (8)	AL (8)	AH (8)
DX–AX (32)	Source (16)	AX (16)	DX (16)

If an attempt is made to divide by 0 or if the quotient is too large to fit in AL or AX (i.e., if the result is greater than FFH in 8-bit division or FFFFH in 16-bit division), the 8086 automatically generates a type 0 interrupt. All flags are undefined after a DIV instruction.

Example:

(a) `DIV DL` ; Divide the word in AX by the byte in DL. The quotient is stored in AL and the remainder in AH.

(b) `DIV CX` ; Divide the double word (32 bits) in DX-AX by the word in CX. The quotient is stored in AX and the remainder in DX.

(c) `DIV BYTE PTR [BX]` ; Divide the word in AX by the byte from the memory at [BX]. The quotient is stored in AL and the remainder in AH.

IDIV The IDIV instruction is used for dividing signed data. The general form and the rules for the IDIV instruction are same as those for the DIV instruction. The quotient is a signed number and the sign of the remainder is the same as the sign of the dividend.

To divide a signed byte by a signed byte, the dividend byte is put in AL and using the CBW (convert byte to word) instruction, the sign bit of the data in AL is extended to AH. Thus, the byte in AL is converted to a signed word in AX. To divide a signed word by a signed word, the dividend byte is put in AX and using the CWD (convert word to double word) instruction, the sign bit of the data in AX is extended to DX. Thus, the word in AX is converted to a signed double word in DX–AX.

If an attempt is made to divide by 0 or if the quotient is too large or too small to fit in AL and AX for 8- and 16-bit division, respectively (i.e., either the result is greater than the decimal value +127 in 8-bit division or the decimal value +32,767 in 16-bit division, or the result is less than the decimal value −128 in 8-bit division or the decimal value −32,767 in 16-bit division), the 8086 automatically generates a type 0 interrupt. All flags are undefined after a DIV instruction.

DAA Decimal adjust AL after BCD addition—This instruction is used to get the result of addition of two packed BCD numbers (in a packed BCD number two decimal digits are represented as eight bits) as a BCD number. The result of addition must be in AL for DAA to work correctly. If the lower nibble (four bits) in AL is greater than 9 after addition or if the AF flag is set by the addition, the DAA instruction adds 6 to the lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or if the carry flag is set by the addition, the DAA instruction adds 60H to AL.

Example:

(a) Let AL = 01011000 = 58 BCD
CL = 00110101 = 35 BCD

As the 8086 fetches instruction bytes from the memory, the coprocessor catches these bytes from the data bus and puts them in a queue. However, the coprocessor treats all the normal 8086 instructions as NOP instructions. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction.

WAIT When this instruction is executed, the 8086 checks the status of its $\overline{TEST}$ input pin and if the $\overline{TEST}$ input is high, it enters an idle condition during which it does not do any processing. The 8086 remains in this state until the 8086's $\overline{TEST}$ input pin is made low or an interrupt signal is received on the INTR or NMI pins. If a valid interrupt occurs while the 8086 is in this idle state, it returns to the idle state after the interrupt service routine is executed. The WAIT instruction does not affect flags. It is used to synchronize the 8086 with external hardware such as the 8087 coprocessor.

14.4 8086 ASSEMBLY LANGUAGE PROGRAMMING

A few assembly language programming examples, similar to the assembly language programming of the 8085, are given in this section. These programs can be converted into machine language programs and executed in an 8086-based system, either by manually finding the opcode for each instruction or by using an assembler. As finding the opcode of each instruction of the 8086 manually is time consuming, the line assembler or assembler is normally used. The *line assembler* converts each mnemonic of an instruction immediately into an opcode as it is entered in the system and this type of assembler is used in microprocessor trainer kits. The line assembler is stored in any one of the ROM type memories in the trainer kit. The assembler needs a personal computer for generating the opcodes of an assembly language program. The generated opcodes can be downloaded to the microprocessor-based system such as the microprocessor trainer kit or the microprocessor-based prototype hardware through the serial or parallel port of the computer.

There are many assemblers such as Microsoft Macro Assembler (MASM), Turbo Assembler (TASM), and DOS Assembler, which are used to convert the 8086 assembly language program into machine language program. While using these assemblers, the assembly language program is written using assembler directives. Assembler directives are commands to the assembler to indicate the size of a variable (either byte or word), number of bytes or words to be reserved in the memory, value of a constant, name of a segment, etc., in a program. Assembler directives are not converted directly into opcode, but are used to generate the proper opcode of an instruction. The use of Microsoft's assembler is discussed in this section.

14.4.1 Writing 8086 PROGRAMS

In general, assembler is used for generating the opcode of the 8086 assembly language program and there exists different manufacturers of the 8086 assembler

8086 assemblers, the digit 0 has to be included in the data or address as digit if the data or address starts with any one of the hex-digit from A to F. ition, the symbol 'H' present in the data or address need not be entered in gram even though it is mentioned here in the following programs. The mer has to go through the assembler manual to know these details.

le 14.1:

program to add a word type data located at the offset address 0800H (least ant byte) and 0801H (most significant byte) in the segment address 3000H er word type data located at the offset address 0700H (least significant and 0701H (most significant byte) in the same segment. Store the result offset address 0900H and 0901H in the same segment. Store the carry ed in the addition in the same segment at the offset address 0902H.

Memory location	Mnemonics	Remarks/Function
	MOV AX, 3000H	
	MOV DS, AX	; Initialize DS with the value 3000H
	MOV AX, [800H]	; Move the first data word to AX
	ADD AX, [700H]	; Add AX with the second data word
	MOV [900H], AX	; Store AX at the offset addresses 900H and 901H
	JC CARRY	; If carry =1, jump to CARRY
	MOV [902H], 00H	; If there is no carry, store 00H at the offset address 902H
	JMP END	; Jump to END
CARRY:	MOV [902H], 01H	; Store 01H at the offset address 902H
END:	HLT	; Stop

initialize a segment register with a value, the value is first loaded in one the general-purpose registers such as AX or BX. It is then moved to the ent register. In this example, AX is used to load 3000H into DS. metimes, instead of using the HLT instruction at the end, the software rupt instruction (INT) may be used to return the control to the monitor gram after executing the program.

le 14.2:

program to subtract the byte content of the memory location 3000H: from the byte of the memory location 4000H: 5000H and store the result at on 2000H: 3000H. Assume the input data and the result lie between –128 127, and the negative numbers are represented in 2's complement from. 3000H:4000H represents the segment address of 3000H and the offset 4000H in that segment)

Solution

Memory location	Mnemonics	Remarks/Function
	MOV BX, 3000H	
	MOV DS, BX	; Initialize DS with the segment address 3000H
	MOV CL, [4000H]	; Get the subtrahend from the offset address 4000H
	MOV BX, 4000H	
	MOV DS, BX	; Initialize DS with the segment address 4000H
	MOV AL, [5000H]	; Get the minuend at the offset address 5000H to AL
	SUB AL, CL	; AL <= AL - CL
	MOV BX, 2000H	
	MOV DS, BX	; Initialize DS with the segment address 2000H
	MOV [3000H], AL	; Store AL at the offset address 3000H
	HLT	; Stop

Note: After the execution of the program, if the result is positive, its MSB and carry are 0. If the result is negative, it is represented in 2's complement form and MSB and carry are 1.

Example 14.3:

Write a program to move a word string 200 bytes (i.e., 100 words) long from the offset address 1000H to the offset address 3000H in the segment 5000H.

Solution

Memory location	Mnemonics	Remarks/Function
	MOV AX, 5000H	
	MOV DS, AX	; Initialize DS with the segment address 5000H
	MOV ES, AX	; Initialize ES with segment address 5000H
	MOV SI, 1000H	; Initialize SI with the offset address of the source (i.e., 1000H)
	MOV DI, 3000H	; Initialize DI with offset address of the destination (i.e., 3000H)
	MOV CX, 100	; Initialize CX with the number of words in the string (decimal value of 100 or 64H)

(Contd)

Memory location	Mnemonics	Remarks/Function
	CLD	; Clear the D flag for auto–increment mode
	REP MOVSW	; Execute MOVSW instruction repeatedly CX times
	HLT	; Stop

Note:

(i) The MOVSB instruction can be used in this program instead of MOVSW, but CX must be loaded with the value 200.

(ii) As D is 0, every time MOVSW is executed, the SI and DI registers are incremented by 2, to point the next word in the string.

Example 14.4:

Write a program to find the smallest word in an array of 100 words stored sequentially in the memory, starting at the offset address 1000H in the segment address 5000H. Store the result at the offset address 2000H in the same segment.

Solution

Memory location	Mnemonics	Remarks/Function
	MOV CX, 99	; Initialize CX with the number of comparisons (=100 – 1)
	MOV AX, 5000H	
	MOV DS, AX	; Initialize DS with the segment address 5000H
	MOV SI, 1000H	; Initialize SI with the offset address 1000H
	MOV AX, [SI]	; Move the first word to AX
START:	INC SI	
	INC SI	; Increment SI twice to point the next word
	CMP AX, [SI]	; Compare the next word with the word in AX
	JC REPEAT	; If AX is smaller, jump to REPEAT
	MOV AX, [SI]	; Replace the word in AX with the smaller word
REPEAT :	LOOP START	; Repeat the operation from START
	MOV [2000H], AX	; Store the smallest number in AX at the offset address 2000H
	HLT	; Stop

Example 14.5:

Write a program to find the number of positive and negative data items in an array of 100 bytes of data stored from the memory location 3000H:4000H. Store the