PAGE NO. 56

Chapter-3) Instruction set of 8086 µP

Instruction format: A machine level instruction format has one or more numbers of fields associated with it. The first field is called as operation code field or opcode field which indicate the type of operation to be performed by the CPU. The second field is called as operand field i.e data field on which CPU performs the operations specified by the instruction op-code.

8086 instruction set has 6 general formats of the instructions:

1] 1 byte instruction: This format is only 1 byte long and may have implicit data or register operand. The least significant 3 bits of the opcode are used to specify the register operand if any. Otherwise all the 8 bits form an opcode and the operand are implied.

2] Register to Register: Register to register instruction format is of two bytes long.

The first byte of the code indicate the operation code of the instruction i.e opcode and width of the operand specified by W. The second byte of instruction code indicates the register operand and R/M field. The registers specified in the REG field is one of the register operand. The R/M field indicate another operand which may be register or memory location.

offset → displacement

PAGE NO. 57

1st byte

D7	D6	D5	D4	D3	D2	D1	D0
Op-code							W

2nd byte

D7	D6	D5	D4	D3	D2	D1	D0
1	1	REG			R/M		

3] Register to/from memory with no displacement: In this type of instruction, the format of instruction is of 2 byte long. The first byte is same as in the register to register format but the second byte contains MOD field as shown in the figure

1st byte

D7	D6	D5	D4	D3	D2	D1	D0
Op-code							W

2nd byte

D7	D6	D5	D4	D3	D2	D1	D0
MOD		REG			R/M		

4] Register to / from memory with displacement: This type of instruction format includes 1 or 2 additional bytes for displacement along with 2-bytes format of register to / from memory

1st byte

D7	D6	D5	D4	D3	D2	D1	D0
Op-code							W

2nd byte

D7	D6	D5	D4	D3	D2	D1	D0
MOD		REG			R/M		

3rd byte

D7	D6	D5	D4	D3	D2	D1	D0
Lower byte displacement							

4th byte

D7	D6	D5	D4	D3	D2	D1	D0
Higher byte displacement							

5] Immidiate operand to register:

1st byte

D7	D6	D5	D4	D3	D2	D1	D0
Op-code							

2nd byte

D7	D6	D5	D4	D3	D2	D1	D0
11		Op-code				R/M	

3rd byte

D7	D6	D5	D4	D3	D2	D1	D0
Lower byte DATA							

4th byte

D7	D6	D5	D4	D3	D2	D1	D0
Higher byte ~~data~~ DATA							

In this type, the instruction format content the first byte as well as 3-bits from the second byte are used for op code. One or two bytes of immidiate data as shown below:

6] Immidiate operate to memory with 16-bit displacement:

1st byte

D7	D6	D5	D4	D3	D2	D1	D0
op-code							

2nd byte

D7	D6	D5	D4	D3	D2	D1	D0
MOD	Opcode				R/M		

3rd byte

D7 ... D0

Lower byte displacement

4th byte

D7 ... D0

Higher byte displacement

5th byte

D7 ... D0

Lower byte DATA

6th byte

D7 ... D0

Higher byte DATA

The length of this instruction format is of 5 byte or 6 byte format

The opcode have single bit indicators and the definitions and significance are given below:

- W-bit : Indicates the width of operand. If W=0, the operand is of 8-bit and if W=1, the operand is of 16-bit
- D-bit : It indicates one of the operand in register in case of two operand instruction. If D bit = 0, the registers specified by REG field is source operand and else it is a destination operand.
- S-bit: It is sign extension bit and is used with W-bit i.e either byte or word operation.
- V-bit : Used in case of rotate and shift instruction. If shift count is 1, this bit is set to 0, and if shift count is more than 1 this bit is set to 1
- Z-bit : Used by REP instruction to control the looping operation

REG code of different registers in the opcode bytes are assigned with different binary codes as shown below.

→ PTO

Table 3.1

W-bit	Register address bits	Registers (8/16 bit)	Segment register (2-bit code)	Segment register (16-bit)
0	000	AL	00	ES
0	001	CL	01	CS
0	010	DL	10	SS
0	011	BL	11	DS
0	100	AH		
0	101	CH		
0	110	DH		
0	111	BH		
1	000	AX		
1	001	CX		
1	010	DX		
1	011	BX		
1	100	SP		
1	101	BP		
1	110	SI		
1	111	DI		

Table 3.2

Operands	Memory operands			8/16 bit register operands	
MOD / R/M	without offset (00)	with 8-bit offset (01)	with 16-bit offset (10)	(11)	
				W=0	W=1
000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16	AL	AX
001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16	CL	CX
010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16	DL	DX
011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D16	BL	BX
100	(SI)	(SI)+D8	(SI)+D16	AH	SP
101	(DI)	(DI)+D8	(DI)+D16	CH	BP
110	D16 (direct address)	(BP)+D8	(BP)+D16	DH	SI
111	(BX)	(BX)+D8	(BX)+D16	BH	DI

PAGE NO. 63
DATE

First the addressing mode of instruction must be decided to find out the MOD and R/M fields of a particular instruction. The R/M and addressing mode row content indicates the R/M field and addressing mode specifies the MOD field.

When data is referred as an operand then DS is the default data segment register, CS is the default code segment for storing program codes, SS is the default stack segment register and ES is the default segment register for the destination data storage.

Addressing modes of 8086:

1) Immidiate Addressing Mode
2) Direct Addressing Mode
3) Register Addressing Mode
4) Register indirect Addressing Mode
5) Register relative Addressing Mode
6) Base indexed Addressing Mode
7) Relative base indexed Addressing Mode

Addressing modes is used to locate data or operands in memory, register or I/O. Hence the addressing mode of any instruction specifies the type of operand and the way they are accessed for executing an instruction.

1) Immidiate Addressing Mode :- In this mode, the immidiate data is the part of instruction and appears in the form of successive byte or bytes after the op-code bytes.

So immidiate data may be 8-bit [byte] or 16-bit [word] in length

Ex:- MOV AL, 46H — AL is loaded with 8-bit immidiate data 46H

MOV BX, 1234H — BX is loaded with 16-bit immidiate data 1234H

2) Direct Addressing Mode: In this mode, a 16 bits memory address (offset) of operand is directly specified in the instruction as a part of it. The offset or displacement may be either 8-bit or 16-bits

Examples:

MOV AL, [3000H]	AL will be loaded with content of memory location whose offset is 3000H from base address
AND AX, [8000H]	AX will be ANDed with contents of memory location whose offset is 8000H from base.

3) Register Addressing mode: In this mode, the data is stored in a register and it is referred using particular segment register i.e all registers except IP may be used on this mode

- Register may be source operand, destination operand or both
- Register may be 8 bit or 16-bit.

Examples:

MOV AX, CX	Copies the content of CX register to AX register
AND AL, BL	ANDing the content of BL with AL, store result in AL
ROR AL, CL	Rotate the contents of AL, CL times.

4) Register Indirect Addressing Mode: In this mode, the address of the memory location which contains data or operand is available in an indirect way using offset register such as BX, SI, DI register. The default segment register is either DS or ES depending the instruction used. If BP is used then SS is the default segment register.

Examples: MOV AX, [BX]	Copies the content of memory location whose offset is in BX register
SUB [SI], AL	Subtract the content of AL from the memory location whose offset is in SI register and store the result in same memory location

Indexed addressing mode: In this mode, the offset of operand is stored in any one of the index registers i.e SI and DI. DS and ES are the default segments for index registers SI and DI respectively depending on the instruction used. This mode allows the use of signed displacement

Examples: MOV BL, [SI]	copies the byte from memory location whose offset is in indexed register SI to AL
ADD AX, [DI+8]	Copies the word from memory location whose offset will be calculated by adding 8 with the content of DI register.

5) Register relative addressing mode: In this mode, the data is available at an effective address formed by adding 8-bits or 16-bits displacement with the content of any one of the registers such as BX, BP, SI and DI in the default DS and ES segment.

Example: MOV AX, 50[BX] Copies the word from memory location whose offset will be calculated by adding 50H with the content of BX register.

6) Base indexed addressing mode: In this mode, the effective address of data is calculated by adding the content of a base register BX or BP to the content of an index register SI or DI with default segment DS or ES.

Example: MOV AX, [BX][SI]: Copies the word from the memory location whose offset is calculated by adding content of BX with the contents of SI

7) Relative base indexed addressing mode: In this mode, the offset address of data is calculated by adding the 8-bit or 16-bits displacement with the sum of base register BX or BP and SI or DI register in the default segment DS and ES

Examples: MOV AX, 50[BX][SI] Copies the word from the memory location whose offset is calculated by adding the 50H with the content of BX and SI i.e [50+BX+SI]

Instruction set of 8086:

1) Data transfer instruction group: The instruction of this group are used to transfer data from source to destination. Instruction like STORE, MOV, LOAD, EXCHANGE, INPUT, OUTPUT

2) Arithmetic & logical instruction group: These instructions performs arithmetical and logical operations like INCREMENT, DECREMENT, COMPARE, SCAN, etc.

3) Branch instruction group: The instructions of this group are used to transfer the program execution control to the address specified in the instruction such as CALL, JUMP and RETURN instruction

4) Loop instruction group: These instruction are used to perform unconditional and conditional loops for ex:- LOOP NZ and LOOP Z belongs to this category.

5) Machine control instruction group: These instruction controls the status of machines NOP, HALT, WAIT, LOCK are examples.

6) Flag manipulation instruction group: The instruction of these group which directly affect the flag register such as CLD, STD, CLI, STI

7) Shift and rotate instruction group: The instruction of this group is used to perform bitwise shifting or rotation in either direction with or without count in CX

8) String manipulation group: The instruction of this group are used to perform various string manipulation operation such as LOAD, MOV, SCAN, COMPARE, STORE

(I) Data copy / Transfer instruction:

(1) MOV destination, source

This instruction is used to transfer data from source i.e register/ memory location/immidiate data to destination i.e another register or memory location. The source of an instruction can be any one of the segment registers or other general or special purpose registers or a memory location and destination of instruction can be registered or memory location.

But in immidiate addressing mode segment register should not be the destination register.

Operation:

Destination ← Source

Examples:

MOV BX, 3456H	Immidiate addressing mode where 3456H is immidiate data copied to BX register by this instruction.
MOV AL, [3000H]	Direct Addressing mode where the data from memory location 3000H is copied to AL register by this instruction.
MOV AX, BX	Register addressing mode where the contents of BX register is copied to AX register by this instruction.
MOV AL, [SI]	Indirect addressing mode where the data from the memory location addressed by SI register by this instruction.
MOV AH, 50H [BX]	Base register relative addressing mode where the displacement 50H is added to BX register to get an effective address from which data is copied to AH register by this instruction.

MOV AL, [BX][SI] :	Base index addressing mode where the effective address is calculated by adding the contents of BX and SI registers and data is copied to AL register
MOV AL, 50H[BX][DI]	Relative base indexed addressing mode where the effective address is computed by adding the displacement 50H to the sum of the content of BX and DI register, and data is copied to AL registers.

(2) PUSH SOURCE

This instruction is used to store word from source onto the stack location. Stack pointer is decremented and then stores a word from source to the location in the stack segment where the stack pointer points. The source of the word must be a 16 bit general purpose register, segment register or 16 bit memory location. After decrementing the stack addresses, the higher byte copies to the higher address and lower byte copies to the lower address.

Operation:

SP ← SP - 2

SS:[SP] ← MSB of source

SS:[SP-1] ← LSB of source

Examples:

PUSH BX — Decrement SP by 2, copy BX to stack i.e content of BH register to higher address of stack and BL register to lower address of stack.

PUSH DS — Decrement SP by 2, copy DS to stack i.e higher byte of DS register to higher address of stack and lower byte of DS register to lower address of stack.

PUSH AL — Not allowed must push a word

PUSH [5000H] — Decrement SP by 2, copy word from memory locations to stack i.e content of 5000H to higher address of stack and 5001H to lower address of stack.

③ POP DESTINATION

The instruction stores a word from stack location pointed by the stack pointer to a destination specified in the instruction. The destination must be a 16 bit general purpose register, a segment register or 16 bit memory location. The stack pointer is automatically incremented by 2 during the execution of this instruction to point the next word on the stack and then the word is copied to the specified destination.

Operation:

LSB of destination ← SS:[SP]

MSB of destination ← SS:[SP+1]

SP ← SP+2

Examples:

POP DX — Copy a word from top of stack to DX, SP = SP+2 i.e content of [SP] to DL register and content of [SP+1] to DH register.

POP DS — Copy a word from top of the stack to DS register. SP = SP+2

POP [8000H] — Copy a word from top of the stack to memory locations 8000H and 8001H

④ XCHG destination, source:

This instruction exchanges the contents of the register with contents of another register or memory location. The instruction cannot directly exchange the contents of two memory locations. A memory location can be specified as the source or as the destination by any of 24 addressing modes. Source and destination must be both word or they must both be byte. The segment register can not be used in this instruction.

Operation: Destination ⟷ Source

Examples:

XCHG AX, BX — Exchange word in AX with word in BX

XCHG BL, CH — Exchange byte in BL with byte in CH

XCHG AX, [7000H] — Exchange the word in AX with memory i.e AH with the content of 7000H memory location and AL with the content of 7001H memory location

XCHG AL, NUM[BX] — Exchange the AL with byte in memory of EA = NUM[BX]

PAGE NO. 72

⑤ IN accumulator, port:

IN instruction copies data from a port to the destination which may be AL or AX i.e accumulator. The address of the port can be specified in the instruction directly or indirectly. For the fixed port type the 8 bit address of a port is specified directly in the instruction. For variable port type the 16 bit address of a port is specified in DX register only. DX register must always be loaded with the 16-bit port address before the IN instruction.

Operation:

AL ← [Port] for byte

AL ← [Port]
and AH ← [Port+1] for word

Examples:

In fixed port type

IN AL, 80H Input a byte from port whose address is 80H

IN Ax, 80H Input a word from port whose address is 80H

In variable port type

MOV DX, 8000H Initialize Dx to point port with port address

IN AL, DX Input a byte from 8 bit port whose address is in DX to AL

IN AX, DX Input a word from 16 bit port whose address is in DX to AX

PAGE NO. 73

⑥ OUT port, accumulator

OUT instruction copies a byte from AL or a word from AX to the specified port. The address of the port can be specified in the instruction directly or indirectly. For the fixed port type, the 8-bit address of a port is specified directly in the instruction. For variable port type, the 16 bit address of a port is specified in DX register only. So DX register must always be loaded with the 16-bit port address before the OUT instruction.

Operation:

[Port] ← AL for byte

[Port] ← AL [Port+1] ← AH for word

Examples:

In fixed port type

OUT 80H, AL Copy the contents of AL to port 80H

OUT 80H, AX Copy the contents of AX to port 80H

In variable port type

MOV DX, 6000H Initialize DX with 16-bit port address.

OUT DX, AL Copy the contents of AL to port

OUT DX, AX Copy the contents of AX to port

⑦ XLAT

The XLAT instruction replaces a byte in the AL register with the byte from the lookup table in memory. Before the execution of XLAT instruction the lookup table containing the values for the new code must be put in memory and the offset of the starting address of the lookup table must be loaded in BX. To point to a desired byte in the lookup table, the XLAT instruction adds the byte in AL to the offset of start of table in BX. It then copies the byte from the address pointed to by

PAGE NO. 74
DATE

BX+AL but into AL. XLAT changes no flags.

Operation:
AL ← DS:[BX+AL]

Example:
.DATA
TABLE DB '0123456789ABCDEF'
CODE DB 11
.CODE
.
.
.
MOV BX, offset TABLE Point BX to the start of lookup table in DS
MOV AL, CODE
XLAT Replace code in AL with code from lookup table.
The content of AL will be OBH

(8) LEA 16 bit register, source

This instruction determines the offset of variable or memory location names as source and loads this offset in specified 16 bits register.

Operation:
16 bit register ← effective address

Example:
LEA BX, ARRAY Load BX with offset of variable ARRAY
LEA SI, LIST Load SI with offset of variable LIST
LEA CX, [BX][DI] Load CX with effective address by adding content of BX & DI

PAGE NO. 75
DATE

(9) LDS/LES 16 bit register, memory address of first word

These instructions copy a word from two consecutive memory locations into the registers specified in the instruction. It then copies a word from next two consecutive memory location into the DS register.

Operation:
For LDS instruction: 16 bit register ← [Memory address]
DS ← [Memory address+2]
For LES instruction: 16 bit register ← [Memory address]
ES ← [Memory address+2]

Examples:
LDS BX, [1234H] Copy the contents of memory location 1234H in BL, contents of 1235H to BH and the contents of 1236H and 1237H in DS register
LES BX, [1234H] Copy the contents of memory location 1234H in BL, contents of 1235H to BH and the contents of 1236H and 1237H in ES register.

(10) LAHF:
This instruction stores lower byte of flag register of 8086 to the AH register.
Examples: LAHF

(11) SAHF:
This instruction copies the content of AH register which is used to set or reset the flag in the lower byte of the flag register of 8086.
Examples: SAHF

(12) PUSHF:
* This instruction is used to store flag register on to stack.
* The stack pointer is decremented by two and stores the word in the flag register to the memory locations pointed by the stack register.

(13) Example: PUSHF

POPF: This instruction is used to store a word from the memory locations at the top of stack to the flag register and increment stack pointer by 2.

Example: POPF

Difference between MOV & LXI:
* MOV: This instruction copies data from source i.e registers / memory location / immidiate data to destination i.e another register / memory location.
* LXI: This is instruction is not available in 8086 instruction set and used to load immidiate 16 bit data to register pair like BC, DE, HL, in 8085.

(II) Arithmetic instructions: These instructions perform the arithmetic operations like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions.

(1) ADD/ADC destination, source: The ADD instruction adds a number from some source to a number from some destination. The ADC instruction adds the carry flag into the result.

The source may be an immidiate number, a register, or a memory location as specified by any 24 addressing modes as given in table 3.2 on Pg 62.

The destination may be a register or a memory location specified by any one of 24 addressing modes as given in table 3.2.

The source and destination must be of the same type and cannot both be memory locations.

Destination should not be an immidiate number.

Flags affected: OF, CF, PF, AF, SF, ZF

Operation:
destination ← source + destination ... for ADD
destination ← source + destination + CF ... for ADC

Examples:

ADD AL, 74H	Immidiate addressing mode instruction that adds the immidiate number 74H to AL and stores result in AL.
ADD AX, BX	Register addressing mode instruction that adds the contents of BX with AX and stores result in AX.
ADD AL, [6000H]	Direct addressing mode instruction that adds the contents of memory location 6000H with AL, stores result in AL.
ADD AL, [SI]	Register indirect addressing mode instruction that adds the contents of memory location pointed by SI index register with AL and store result in AL.
ADC AX, 1234H	Immidiate addressing mode instruction that adds the immidiate number 1234H to AX with carry and stores result in AL.
ADC AX, BX	Register addressing mode instruction that adds the content of BX to AX with carry and stores result in AX.

(2) SUB/SBB destination, source :- The SUB instruction is used to subtract the data in source from the data in destination and stores result in destination.

The SBB instruction is used to subtract the source operand and the borrow (CF), which may reflect from the result of the previous calculations, from the destination operand, and the result, is stored in destination operand.

Source must be a register or memory location or immidiate data and the destination must be a register or a memory location.

The destination operands should not be an immidiate data and the source and destination must be a register or memory location.

The destination operands should not be an immidiate data and the source and destination both should not be memory operands.

Flag affected : OF, CF, PF, AF, SF and ZF

Operation :

destination ← destination - source .. For SUB

destination ← destination - source - CF .. for SBB

Examples :

SUB AL, 74H — Immidiate addressing mode instruction that subtracts the immidiate number 74H from AL and stores result in AL.

SUB AX, BX — Register addressing mode instruction that subtracts the contents of BX from AX and stores result in AX.

SUB AL, [6000H] — Direct addressing mode instruction that subtracts the content of memory location 6000H from AL, stores result in AL.

SUB AL, [SI] — Register indirect addressing mode instruction that subtracts the contents of memory location pointed by SI index register from AL and store result in AL.

SBB AX, 1234H — Immidiate addressing mode instruction that subtracts the immidiate number 1234H and borrow from AX and stores result in AX.

SBB AX, BX — Register addressing mode instructions that subtracts the content of BX and borrow from Ax and stores result in AX.

(3) INC destination :- This instruction adds 1 to the indicated destination. The destination can be a register or memory location specified by any one of 24 ways given in Table 3.2 on pg 62. Immidiate data cannot be an operand of this instruction

Flag affected : OF, PF, AF, SF, ZF

Operation :

destination ← destination + 1

Examples :

INC AX — Increment the content of AX by 1

INC [2000H] — Increment the content of memory location 2000H by 1

INC TEMP — Increment the byte/word named as TEMP by 1

(4) DEC destination :- This instruction subtracts 1 from the indicated destination. The destination can be a register or memory location specified by any one of the 24 ways given in table in 3.2. Immidiate data cannot be an operand of this instruction

PAGE NO. 80

Flag affected: OF, PF, AF, SF, ZF

Operation:
destination ← destination − 1

Examples:
DEC AX Decrement content of AX by 1
DEC [2000H] Decrement content of memory location 2000H by 1
DEC TEMP Decrement the byte or word named as TEMP by 1

(5) CMP destination, source: The CMP instruction make a comparison between a byte/word from the specified source and byte/word from the specified destination. The source and destination can be an immidiate data, a register or a memory location specified by one of 24 ways given in table 3-2. However the source and destination cannot be both memory locations. The comparison is actually done by non-destructive subtraction of the source byte or word from the destination byte or word i.e the source and the destination will not be changed, but the flags will be set to indicate the results of the comparison.

Flag affected: OF, CF, PF, AF, SF, ZF

Operation:
(a) If destination > source then CF=0, ZF=0, SF=0
(b) If destination < source then CF=1, ZF=0, SF=1
(c) If destination = source then CF=0, ZF=1, SF=0

Examples:
CMP AL, 0FFH Compare AL with immidiate number FFH...
CMP AX, BX Compare AX with BX
CMP CX, COUNT Compare CX with COUNT

PAGE NO. 81

(6) DAA (Decimal adjust accumulator): DAA instruction is used to convert the result of the addition of two packed BCD numbers into a packed BCD number. DAA only works on AL register. So, DAA instruction must be used after the ADD/ADC instruction. The ADD/ADC instruction adds the two BCD numbers in hexadecimal format and DAA instruction convert this hexadecimal result into BCD result.

The working of DAA instruction is given below:
1] If the value of the lower nibble in AL accumulator is greater than 9 or if AF flag is set, then DAA instruction adds 6 to the lower nibble of AL accumulator.
2] If the value of higher nibble in AL accumulator is greater than 9 or if CF flag is set, the DAA instruction adds 6 to the higher nibble of AL accumulator.

Flag affected: CF, PF, AF, SF, ZF and OF is undefined.

Operation:
(a) If lower nibble of AL ≥ 9 or AF=1, then AL = AL + 06
(b) If higher nibble of AL ≥ 9 or CF=1, then AL = AL + 60
(c) If both above condition are satisfied, then AL = AL + 66

Example: (Refer textbook)

(7) DAS (Decimal adjust after subtraction): DAS instruction is used to convert the result of the previous subtraction of two packed BCD numbers to a packed BCD number. DAS instruction only works on AL register. So, DAS instruction must be used after the SUB/SBB instruction. The SUB/SBB instruction subtracts the two BCD numbers in hexadecimal format and DAS instruction convert this hexadecimal result to BCD result.

PAGE NO. 82

The working of DAS instruction is given below:

1] If the value of the lower nibble in AL accumulator is greater than 9 or if AC flag is set, the DAS instruction subtracts 6 to the lower nibble of AL accumulator.

2] If the value of the higher nibble in AL accumulator is greater than 9 or if CF flag is set, the DAS instruction subtracts 6 to the higher nibble of AL accumulator.

Flags affected: CF, PF, AF, SF, ZF and OF is undefined

Operation:

(a) If lower nibble of AL > 9 or AF = 1 then AL = AL - 06

(b) If higher nibble of AL > 9 or CF = 1 then AL = AL - 60

(c) If both above conditions are satisfied AL = AL - 66

Example: (Refer textbook)

(8) NEG destination: This instruction converts the number byte/word in a destination in the 2's complement and store result in the destination which may be a register or a memory location specified by any one of the addressing modes.

Flags affected: OF, CF, PF, AF, SF, ZF

Operation:

destination ← 2nd Complement of destination

Examples:

NEG AX Replace number in AX with its 2's complement

NEG BYTE PTR[BX] Replace byte at offset [BX] in DS with its 2's complement

MUL source: The instruction is used to multiply an unsigned byte from source with an unsigned byte in the AL register, or an unsigned word from source with an unsigned word in the AX register.

The source must be a any register or memory location.

PAGE NO. 83

When a byte is multiplied with the byte in AL, then the result is stored in AX because the result of multiplication of two 8-bit i-e bytes numbers is maximum 16 bits.

When a word is multiplied with the word in AX, then the MSW of result is stored in DX and LSW of word in AX register because the result of multiplication of two 16-bit numbers is maximum 32 bits.

If MSB or MSW of result is zero, then CF and OF both will be set.

Flag affected: OF, CF and PF, AF, SF, ZF are undefined

Operation:

(a) If source is byte then AX ← AL * unsigned 8 bit source

(b) If source is word then DX:AX ← AX * unsigned 16 bit source

Examples:

MUL BL Multiply AL by BL, result in AX

MUL CX Multiply AX by CX, result in DX:AX

MUL BYTE PTR [BX] Multiply AL by byte in DS pointed by [BX], result in AX

(10) IMUL source: This instruction is used to multiply a signed byte from source with the signed byte in the AL register during signed byte multiplication, and a signed word from source with a signed word in the AX register during signed word multiplication. The source must be register or memory location.

When a byte is multiplied with the byte in AL, then the result is stored in AX because the result of two 8-bit i-e bytes numbers is maximum 16 bits.

When a word is multiplied with the word in AX, then the MSB result is stored in DX and LSB in AX register because the result of two 16-bits i-e word numbers is maximum 32 bits.

If the magnitude of the product does not require all the bits of the destination, the unused bits are filled with the copies of the sign bit.

Flag affected: OF, CF and PF, AF, SF, ZF are undefined

Operation:

(a) If source is byte then AX ← AL * signed 8 bit source

(b) If source is word then DX : AX ← AX * signed 16 bit source

Examples:

IMUL BL — Multiply AL by BL, result in AX

IMUL CX — Multiply AX by CX, result in DX : AX

IMUL BYTE PTR [BX] — Multiply AL by byte in DS pointed by [BX], result in AX.

Example of multiplication of signed byte with signed word

MOV CX, multiplier — load signed word multiplier in CX

MOV AL, multiplicand — load signed byte multiplicand in AL

CBW — Extend sign of AL into AH

IMUL CX — Word multiplies, result in DX : AX

DIV source: This instruction divides an unsigned word by an unsigned byte during 16/8 division, and to divide unsigned double word i.e 32 bits by an unsigned word during 32/16 division. During the division of a word by a byte, the word (dividend) must be in the AX register and the byte (divisor) may be in any 8 bit register or memory location.

After the division operation 8 bit quotient will be available in AL register and 8 bit remainder will be available in AH register.

While dividing double word by word, the most significant word of the double word should be in DX and the least significant of the double word should be in AX.

After the division operation, 16 bit quotient will be available in AX register and 16 bit remainder in DX register.

If word or double word is divided by 0 or quotient is too large to fit in AL or AX i.e greater than FFH or FFFFH, the 8086 will automatically generate a type 0 interrupt i.e divide by 0 interrupt or divide overflow interrupt.

During division of double word by word, the dividend must be in DX : AX for double word or AX for word, but source of the divisor should be a word or byte register or a memory location.

During the division of a byte by a byte, we must first store dividend ~~word in AX~~ byte in AL and fill AH with all 0's for unsigned dividend. When we want to divide a word by a word in AX and fill DX with all 0's for unsigned dividend.

Flag affected: None and OF, CF, PF, AF, SF, ZF are undefined

Operations:

a) If source is byte then

AL ← AL / unsigned 8 bit source

AH ← AL MOD unsigned 8 bit source

b) If source is word then

AX ← DX : AX / unsigned 16 bit source

DX ← DX : AX MOD unsigned 16 bit source

Examples:

DIV BL — Divide word in AX by byte in BL, quotient in AL and remainder in AH

DIV CX — Divide word in Dx and AX by word in CX, quotient in AX and remainder in DX.

DIV NUM [BX] — Divide word in AX by byte in memory location pointed by [BX].

(12) IDIV source: This instruction divides an signed word by an signed byte during 16/8 division, and to divide signed double word i.e 32-bits by an signed word during 32/16 division. During the division of a word by a byte, the word(dividend) must be in AX register and a byte (divisor) may be in any 8 bit register or memory location.

After the division operation, 8 bit quotient will be available in AL register and 8 bit remainder will be available in AH register.

During the division of double word by word, the dividend must be in DX:AX for double word or AX for word, but source of the divisor should be a word or byte register or memory location.

During the division of a byte by a byte, we must first store dividend byte in AL and fill AH with all 0's for unsigned dividend.

If a word or double word is divided by 0 or the quotient is too large to fit in AL or AX i.e greater than FFH and or FFFFH, the 8086 will automatically generate a type 0 interrupt i.e divide by 0 interrupt or divide overflow interrupt.

For division, the dividend (numerator) must always be in AX:DX for word denominator and AX for byte denominator, but source of the divisor (denominator) can be a register or a memory location.

When we want to divide a byte by a byte, we must first store dividend byte in AL and fill all bits in AH with signed-bit of AL using CBW instruction.

When we want to divide a word by a word, we must first store dividend word in AX and fill all bits in DX with sign bit of AX using CWD instruction.

Flag affected: None & OF, CF, PF, AF, SF, ZF are undefined

Operation:

a) If source is byte then

AL ← AL/signed 8 bit source

AH ← AL MOD signed 8 bit source

b) If source is word then

AX ← DX:AX / signed 16 bit source

DX ← DX:AX MOD signed 16 bit source

Examples:

IDIV BL — Divide a signed word in AX by a signed byte in BL, quotient in AL and remainder in AH

IDIV CX — Divide a signed double word in CX, quotient in AX and remainder in DX

IDIV NUM[BX] — Divide a signed word in AX by a signed byte in memory location pointed by [BX]

(13) CBW: This instruction copies the sign of byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. The CBW operation must be done before performing division of a signed byte in the AL by another signed byte with IDIV instruction.

Operation:

AH ← Filled with 8th bit of AL i.e D7

Example:

AX = 00000000 10011011

CBW convert signed byte in AL to signed word in AX

AX = 11111111 10011011

(14) CWD: This instruction copies the sign bit of a word in AX to all the bits of the DX register. In other word, it extend the sign of AX into all of DX. The CWD operation must be done before performing division of a signed word in AX by another signed word with the IDIV instruction.

Operation:
DX ← filled with 16th bit of AX i.e D15

Example:
DX = 00000000 00000000 AX = 11110000 11000111
CWD convert signed word in AX to signed double word in DX:AX
Result after execution of CWD
DX = 11111111 11111111 AX = 11110000 11000111

(15) AAA (ASCII adjust after addition): This instruction can be used to convert the contents of the AL register to unpacked BCD result. The higher nibble of the AL register is filled with zeros. This instruction should be executed after the ADD instruction and the result is placed in the AL register. Before execution of AAA instruction, AH should be loaded with 0.

The working of AAA instruction, AH is as follows:

Flag affected: AF and CF
Operation: 1] Clear the high order nibble of AL i.e AL = AL AND 0FH
2] IF lowes nibble of AL>9 or AF = 1 then,
a) AL = AL+6
b) AH = AH+1
c) AF = CF = 1
d) AL = AL AND 0FH

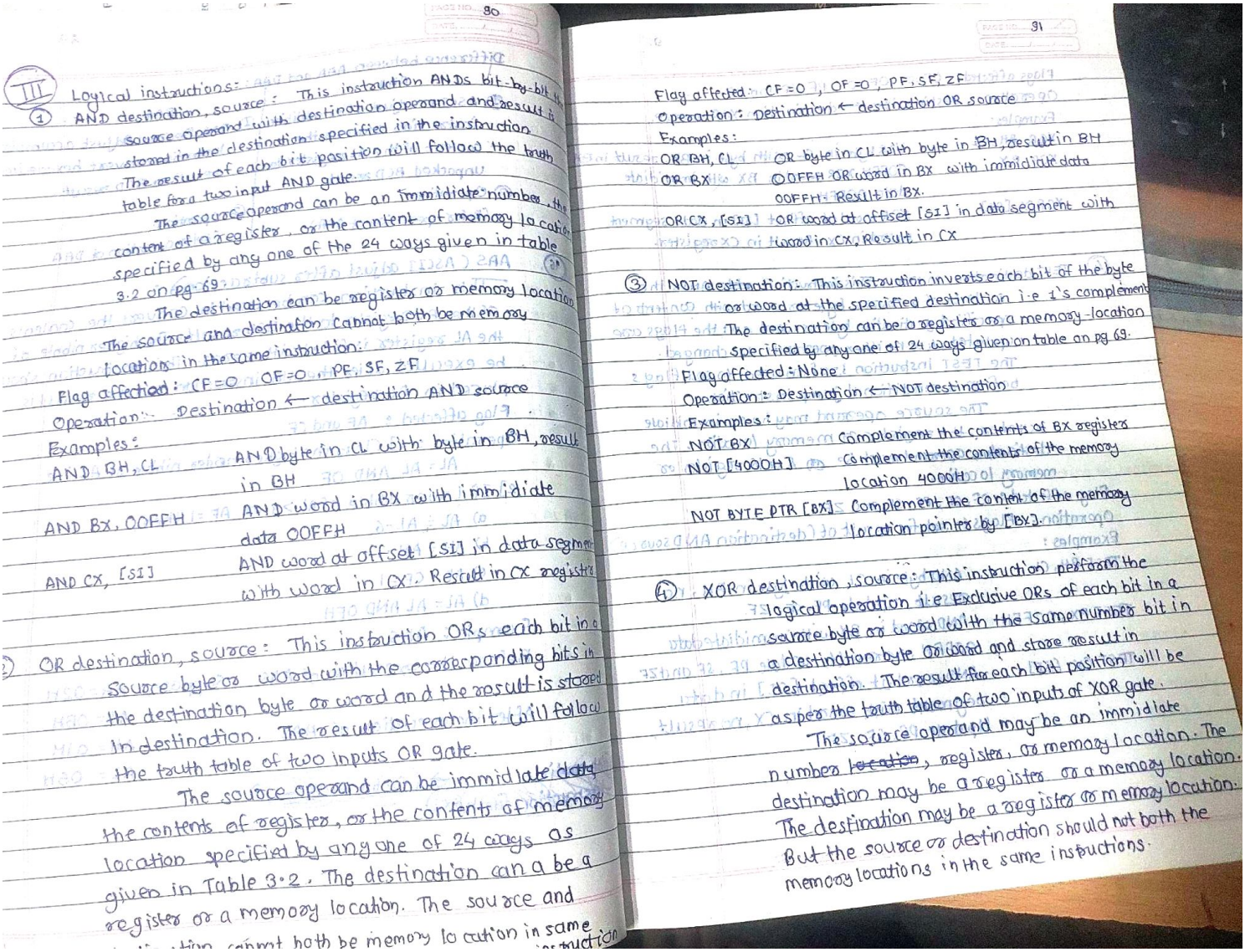
Example: AAA
Before the execution of AAA, Suppose AH = 00H
AL = 0BH
After the execution of AAA AH = 01H
AL = 01H

Difference between AAA and DAA:

AAA	DAA
① ASCII adjust after addition instruction converts the content of the AL register to unpacked BCD result	① DAA Decimal Adjust accumulator instruction convert hexadecimal result to BCD result.
② Operation: (Refer operation of AAA)	② Operation: (Refer operation of DAA

(16) AAS (ASCII adjust after subtraction): This instruction can be used to convert the contents of the AL register to the BCD result. The higher nibble of the AL register is filled with zeros. This instruction should be executed after the SUB instruction and the result is placed in AL register.
Flag affected: AF and CF
Operation: 1] Clear the higher nibble of AL i.e AL = AL AND 0FH
2] IF lower nibble of AL>9 or AF = 1 then,
a) AL = AL - 6
b) AH = AH - 1
c) AF = CF = 1
d) AL = AL AND 0FH
Example: AAS
Before the execution of AAS Suppose, AH = 02H
AL = 0BH
After the execution of AAS AH = 01H
AL = 05H

Arithmetic Instruction Continued on P.90



III Logical instructions:

① AND destination, source: This instruction ANDs bit-by-bit the source operand with destination operand and result is stored in the destination specified in the instruction. The result of each bit position will follow the truth table for a two input AND gate.

The source operand can be an immidiate number, the content of a register, or the content of memory location specified by any one of the 24 ways given in table 3.2 on pg 69.

The destination can be register or memory location. The source and destination cannot both be memory locations in the same instruction.

Flag affected: CF=0 OF=0, PF, SF, ZF

Operation: Destination ← destination AND source

Examples:

AND BH, CL — AND byte in CL with byte in BH, result in BH

AND BX, 00FFH — AND word in BX with immidiate data 00FFH

AND CX, [SI] — AND word at offset [SI] in data segment with word in CX. Result in CX register

② OR destination, source: This instruction ORs each bit in a source byte or word with the corresponding bits in the destination byte or word and the result is stored in destination. The result of each bit will follow the truth table of two inputs OR gate.

The source operand can be immidiate data the contents of register, or the contents of memory location specified by any one of 24 ways as given in Table 3.2. The destination can a be a register or a memory location. The source and

Flag affected: CF=0, OF=0, PF, SF, ZF

Operation: Destination ← destination OR source

Examples:

OR BH, CL — OR byte in CL with byte in BH, result in BH

OR BX, 00FFH — OR word in BX with immidiate data 00FFH. Result in BX.

OR CX, [SI] — OR word at offset [SI] in data segment with word in CX. Result in CX

③ NOT destination: This instruction inverts each bit of the byte or word at the specified destination i.e 1's complement. The destination can be a register or a memory location specified by any one of 24 ways given on table on pg 69.

Flag affected: None

Operation: Destination ← NOT destination

Examples:

NOT BX — Complement the contents of BX register

NOT [4000H] — Complement the contents of the memory location 4000H

NOT BYTE PTR [BX] — Complement the content of the memory location pointed by [BX].

④ XOR destination, source: This instruction perform the logical operation i.e Exclusive ORs of each bit in a source byte or word with the same number bit in a destination byte or word and store result in destination. The result for each bit position will be as per the truth table of two inputs of XOR gate.

The source operand may be an immidiate number, register, or memory location. The destination may be a register or a memory location. The destination may be a register or memory location. But the source or destination should not both the memory locations in the same instructions.

Flags affected: CF=0, OF=0, PF, SF and ZF.
Operation: Destination ← destination XOR source.
Examples:
XOR BH, CL — XOR byte in CL with byte in BH, result in BH
XOR BX, 00FFH — XOR word in BX with immidiate data 00FFH
XOR CX, [SI] — XOR word at offset [SI] in data segment with word in CX. Result in CX register.

④ TEST destination, source: This instruction ANDs the contents of a source byte or word with contents of specified destination byte or word and the flags are updated, but neither operands are changed. The TEST instruction is often used to set flag before a conditional jump instruction.
The source operand may be an immidiate number, the register or memory location. The destination operand must be an 8 or 16 register or memory location.
Flag affected: CF, OF, PF, SF and ZF
Operation: Flags ← set for result of (destination AND source)
Examples:
TEST BH, CL — AND byte in CL with byte in BH, no result, update PF, SF, ZF
TEST BX, 00FFH — AND word in BX with immidiate data 00FFH, no result, update PF, SF and ZF
TEST CX, [SI] — AND word at offset [SI] in data segment with word in CX, no result, Update PF, SF, ZF.

Difference between AND and TEST

AND	TEST
① Destructive AND instruction means destination is modified after the execution of instructions	① Non destructive TEST instruction means destination is not modified after the execution of instructions
② Flag affected: CF=0, OF=0, PF, SF, ZF.	② Flag affected: CF, OF, PF, SF and ZF.
③ Operation: Destination ← destination AND source	③ Operation: Flags ← set for result of (destination AND source)

(IV) Control Transfer or Branching instruction:
The control transfer or branching instructions are used to change path or flow of execution of the program to new address specified in the instruction directly or indirectly.
When this type of instruction is executed, the contents of CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow or path of execution is going to be transferred.
Depending on the addressing modes specified in the table 3.2, the CS may or may not modify. There are two types of control transfer or branching instructions as given as follows:
1] Unconditional control transfer or branching instruction
2] Conditional control transfer or branching instruction

1) Unconditional control transfer or branching instructions:

① CALL a procedure: The CALL instruction is used to transfer the program control to the sub-program or subroutine. There are two basic types of CALLs, near and far.

A near CALL is a call to a procedure, which is in the same code segment as CALL instruction. When 8086 executes the near CALL instruction, the stack pointer is decremented by 2 and copies the offset i.e IP of the next instruction after the CALL instruction on stack. This offset value is used to transfer back the program control to the calling program after the execution of subroutine or procedure. Then 8086 loads the offset of the first instruction of the procedure into the IP and RET instruction at the end of the procedure will return execution to the instruction after the CALL instruction in calling program by copying the offset value stored back to IP.

A far call is used to call a procedure which resides in another code segment from that which contain call instructions. When CPU 8086 executes the far call instruction the stack pointer is decremented by two and copies the content of CS register onto the stack. Again stack pointer is decremented by two, and copies the content of IP i.e offset of the next instruction after the CALL instruction onto the stack.

Finally it loads CS with the segment base address of the code segment that contains procedure and IP with the offset of the first instruction of the procedure in that segment. A RET instruction at the end of procedure will return execution to the next instruction of the calling program by restoring the same value of CS and IP from the stack.

Operation:

1) IF NEAR CALL, then SP ← SP-2 Save IP on stack
   IP ← Address of procedure
2) IF FAR CALL, then SP ← SP-2
   SP ← CS i.e save CS on stack
   CS ← New segment base address of the called procedure
   SP ← SP-2
   SP ← IP i.e save IP on stack
   IP ← New offset address of the called procedure

Examples:

CALL DELAY — Direct within the segment that calls the procedure of the name DELAY

CALL BX — Indirect within the segment where BX contains the offset of the first instruction of the procedure and replace the contents of the IP with the contents of the BX register.

CALL FAR PTR SHOW — Direct to another segment i.e far or inter segment, SHOW is the name of procedure and must be declared FAR with SHOW PROC FAR at its start. → The assembler will determine the code segment base for the segment which contains procedure and the offset of start of the procedure in that segment.

② RET instruction: The RET instruction is used to return the program execution control from the procedure to the next instruction immidiate after the CALL instruction in the calling program

If the procedure is a near procedure, then return will be done by restoring the value of IP with the word from the stack top. The word from the stack top is offset of the next instruction in the calling program that was pushed on the stack by call instruction. So stack pointer is incremented by two and return address is popped from the stack to IP.

If the procedure is far procedure i.e in a different code segment from the CALL instruction which calls it, the IP will be replaced by the word from stack top which is nothing but the offset of the next instruction after the CALL instruction of the CALL instruction.

Again the stack pointer is incremented by two; the CS is replaced with the current stack top which is the segment base address of the segment where CALL instruction resides. After the replacement of CS, again the stack pointer is incremented by two.

Operation:

1] For NEAR Return then IP ← content of top of stack
SP ← SP+2

2] For FAR Return then IP ← contents of top of stack
SP ← SP+2
CS ← Contents of top of stack
SP ← SP+2

③ INT N [N=Type of interrupt]: This instruction causes the 8086 to call a far procedure in a manner similar to the way in which the 8086 respond to an interrupt signal on its INTR or NMI inputs. The term type refers to the number between 0 to 255 that identifies the interrupt.

When the 8086 execute an INT type instruction, it will perform following operation:

a) Decrement the stack pointer by 2 and push the flags onto the stack.
b) Decrement the stack pointer by 2 again and push contents of the CS.
c) Decrement the stack pointer by 2 again and push the offset of the next instruction after the INT type instruction on the stack.
d) Get the new value for IP from an absolute memory address of 4 times the type specified in the instruction. For ex:- for an INT 8 instruction, the new IP will be read from address 00020H.
e) Get a new value for CS from an absolute memory address of 4 times the type specified in the instruction plus 2.
f) Reset both the IF and the TF flags, other flags are not affected by this instruction.

④ INTO instruction [Interrupt on overflow]:

If the overflow flag is set, this instruction will generate type 4 interrupt and causes the 8086 to do an indirect far call a procedure which is written by the user to handle the overflow condition.

When 8086 executes an INTO instruction, it will perform the following operation:

a) Decrement the stack pointer by 2, and push the flags on the stack.
b) Decrement the stack pointer by 2, and push CS on the stack
c) Decrement the stack pointer by 2 again and push IP contains the offset of the next instruction after an INTO instruction on the stack.

d) Get the new value for IP from an absolute memory address 00010H in IVT (Interrupt vector table).
e) Get a new value for CS from an absolute memory address 00012H in IVT.
f) Reset the TF and the IF, other flags are not affected by this instruction.

(5) IRET instruction: The IRET instruction is used at the end of the interrupt service procedure to return the execution to the interrupt program. During the execution of this instruction, the 8086 copies the saved value of IP from the stack to IP, the saved value of CS from the stack to CS and saved value of flags back to the flag register.
Function performed by this instruction are:
a) IP is popped from the stack then SP ← SP+2
b) CS is popped from the stack then SP ← SP+2
c) Flag register is popped from the stack then SP ← SP+2

(6) JMP destination address [Jump unconditional]:
This instruction unconditionally transfers the control of execution to the specified address using an 16-bit displacement or CS:IP. No flags are affected or checked by this instruction.
If the target of JMP is in the same code segment, it requires only the IP to be changed to transfer control to the target location. Such a jump is called as intra-segment jump or near jump.
If the target of JMP is in the different code segment from that containing the instruction JMP, then IP and CS will be changed to transfer control to the target location. Such a jump is called as far or inter-segment j

PAGE NO. 99
DATE

Difference between Inter and Intra Segment Jump:

Inter Segment Jump	Intra Segment Jump
① Intersegment jumps can transfer control to a instructions in a different code segment.	① Intrasegment jumps are always between instructions in the same code segment.
② It is called as FAR jump	② It is called as NEAR jump
③ Requires csand IP to be changed to transfer control to the target location	③ Requires only IP to be changed to transfer control to the target location
④ Ex: JMP ADRS32 ;direct intersegment	④ Ex: JMP DISP16 ;direct intrasegment

Examples:
JMP DOWN
JMP FAR PTR SKIP

Conditional Jump instructions:
The conditional JMP instruction transfers the control at the target location if some specified condition is satisfied. Conditional JMP instructions are normally used after compare instruction or arithmetic or logical instructions. The different types of conditional JMP instructions are given in table:

	Instruction	Function	Syntax
1)	JCXZ	Jump if CX register is zero	JCXZ label
2)	LOOP	CX=CX-1, jump if CX≠zero	LOOP label
3)	LOOPE/LOOPZ	CX=CX-1, jump if CX≠zero and ZF=1	LOOPE label LOOPZ label
4)	LOOPNE/LOOPNZ	CX=CX-1, jump if CX≠zero and ZF=1	LOOPNE label LOOPNE label
5)	JA	Jump is above [CF=0 and ZF=0]	JA label

	Instruction	Function	Syntax
6)	JNBE	Jump if not below or equal [CF=0 and ZF=0]	JNBE label
7)	JAE/ JNB/ JNC	Jump if above or equal [CF=0] / Jump if not below [CF=0] / Jump if no carry [CF=0]	JAE label / JNB label / JNC label
8)	JB	Jump if below [CF=1]	JB label
9)	JNAE/ JC	Jump if not above or equal [CF=1] / Jump if carry [CF=1]	JNAE label / JC label
10)	JE/ JZ	Jump if equal [ZF=1] / Jump if zero [ZF=1]	JE label / JZ label
11)	JG	Jump if greater SF=OF and ZF=0 after signed mathematics	JG label
12)	JNLE	Jump if not less or equal SF=OF and ZF=0 after signed mathematics	JNLE label
13)	JGE/ ~~JNL~~ JNL	Jump if greater or equal SF=OF after signed math / Jump if not less SF=OF after signed math	JGE label / JNL label
14)	JNGE/ JL	Jump if not greater or equal SF≠OF after signed math / Jump if less than SF≠OF after signed math	JNGE label / JL label
15)	JLE/ JNG	Jump is less than or equal SF≠OF and ZF=1 After signed math / Jump if not greater SF≠OF and ZF=1 after signed math	JLE label / JNG label
5)	JNE/ JNZ	Jump if not equal ZF=0 / Jump if not zero ZF=0	JNE label / JNZ label
7)	JNO	Jump if not overflow OF=0	JNO label

	Instruction	Function	Syntax
18)	JNP	Jump if not parity PF=0	JNP label
19)	JPO	Jump if parity ODD PF=0	JPO label
19)	JNS	Jump if not sign SF=0	JNS label
20)	JO	Jump if overflow OF=1	JO label
21)	JP JPE	Jump if parity EVEN PF=1 / Jump if parity equal PF=1	JP label / JPE label
22)	JS	Jump if sign SF=1	JS label

Comparison of JUMP and CALL instructions:

~~JUMP~~ JMP	CALL
① A JMP instruction permanently changes the IP for NEAR JMP and CS:IP for FAR JMP	① A CALL instruction store IP for near CALL and CS:IP for far call on the stack so that the original program execution sequence can be resumed.
② Does not requires RET instruction	② RET instruction is required to return to calling program
③ Conditional JMP instructions are available which ~~can~~ transfer program control to the target location if some specified condition is satisfied.	③ No such conditional CALL instructions are available
④ Stack is not used by JMP instruction	④ Stack is used by CALL instruction

Comparison of JNC and JMP instructions in 8086:

JNC 2000H	JMP 2000H
① Conditional branching instruction	① Unconditional branching
② CF flag checked by this instruction but CF flag is unaffected	② No flags are affected or checked by this instruction.
③ If CF=0, the program control transfer to the memory location 2000H	③ The program control is transfer to 2000H without checking the condition of any flags.
④ Conditional JNC instruction is normally used after compare instruction or arithmetic or logical instructions	④ Unconditional JMP instruction is used when program control to be transfer any part of the program after any instruction.

Difference between JMP and JNC:

* JMP: The program control is transfer to 2000H without checking the condition of any flags.
* JNC: If CF=0, the program control transfer to the specified memory location

Machine Control Instructions:

① HLT: Halt

The instruction HLT causes processor to enter into halt state. The CPU stops fetching and executing of instructions. The CPU can be brought out of the halt state with the occurrence of any one of the following events:

1] Interrupt signal on INTR pin
2] Interrupt signal on NMI pin
3] Reset signal on RESET pin.

② NOP: No operation:

This instruction is used to add wait-state of three clock cycles and during these three clock cycles, the CPU does not perform any operation. This instruction can be used to add delay loop in the program and delaying operation before proceeding to read or write from the port.

③ WAIT

The instruction WAIT causes processor to enter in an ideal state or wait state and continues to remain in that the processor receives state until one of the following signal.

1] signal on processor TEST pin
2] A valid interrupt on INTR pin
3] A valid interrupt on NMI pin

This signal is used to synchronise with other external hardware such as math co-processor 8087.

④ LOCK:

This instruction prevents other processors to take the control of shared resources. In multiprocessor system, the individual processors have their own local buses and memory and then processor are connected together by a system bus to access the shared system resources such as disk drives or memory or DMA. The LOCK instruction is used as a prefix to the critical instruction that has to be executed.

Example:
LOCK IN AL, 80H

(VI) Flag manipulation instructions: This type of instructions are used to change status of flags in flag register such as CF, DF, IF

① CLC: Clear carry:
This instructions clears the carry flag. CF ← 0

② CMC: Complement carry:
This instructions complement the carry flag. CF ← ~CF

③ STC: Set Carry:
This instruction set the carry flag. CF ← 1

④ CLD: Clear direction flag:
This instruction clears the direction flag. DF ← 0

⑤ STD: Set direction flag:
This instruction sets the direction flag. DF ← 1

⑥ CLI: Clear interrupt flag:
This instruction clears the interrupt flag. IF ← 0

⑦ STI: Set interrupt flag:
This instruction sets the interrupt flag. IF ← 1

(VII) String manipulation instructions: A string is a contiguous block of bytes or words and can used to hold any type of data or information that will fit into bytes or words. There are number of operation performed with string. The 8086 microprocessor supports string instructions for string movement, scan, comparison, load and store.

① MOVS: Move String
MOVSB: Move string Byte
MOVSW: Move String Word
The instruction MOVS transfers a byte or a word from the source string to the destination string. The source string must be in the data segment and destination string must be in extra segment.

The offset of the source byte or a word must be placed in SI register, which is represented as DS:SI and the offset of the destination byte or word must be in DI register, which is represented as ES:DI. On the execution of this instruction, SI and DI register are automatically incremented by one to point next element of source and destination.

If direction flag is reset (DF=0), the register SI and DI will be incremented by one for byte move and incremented by two for word move.

If direction flag is set (DF=1), the register SI and DI will be decremented by one for byte move and decremented by two for word move.

The DS:SI and ES:DI register must be loaded prior to the execution of MOVS instruction. Another way to move a byte or word string is by using implicit instruction MOVSB and MOVSW. The MOVSB is used to transfer byte from source to destination and MOVSW is used to transfer word from source to destination.

In multiple byte or word moves, the count must be loaded in CX register which functions are counter.

Operation:
ES:[DI] ← DS:[SI]
If byte movement
For DF=0 SI ← SI+1 and DI ← DI+1
For DF=1 SI ← SI-1 and DI ← DI-1
If word movement
For DF=0 SI ← SI+2 and DI ← DI+2
For DF=1 SI ← SI-2 and DI ← DI-2

Example:
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET S-STRING
MOV DI, OFFSET D-STRING
A] MOVS S-STRING, D-STRING
B] MOVSB
C] MOVSW

② LODS : Load string
LODSB : Load string byte
LODSW : Load string word
The instruction LODS transfer a byte or a word from the source string pointed by SI in DS to AL for byte or AX for word. On execution of a string instruction, SI is automatically updated to point next element of the source string.
If DF=0, the register SI will be incremented by 1 for byte and incremented by 2 for word. If DF=1, the register SI will be decremented by 1 for byte and decremented by 2 for word.
In the instruction LODS, the source must be explicitly declared with either DB or with DW

Another way to load a byte or word string is by using implicit instruction LODSB and LODSW. The instruction LODSB is used to transfer a string byte from source to AL and LODSW is used to transfer a string word from source to AX.
In multiple byte or word loads, the count must be loaded in CX register which functions as a counter.

Operation:
If byte movement:
AL ← DS:[SI]
For DF=0 SI ← SI+1
For DF=1 SI ← SI-1
If word movement:
AX ← DS:[SI]
For DF=0 SI ← SI+2
For DF=1 SI ← SI-2

Example:
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET S-STRING
A] LODS S-STRING
B] LODSB
C] LODSW

③ STOS : Store string
STOSB : Store string byte
STOSW : store string word
The instruction STOS transfer a byte or a word from the AL for byte and AX for word to destination string pointed by DI in ES. On execution of a string instruction, DI is automatically updated to point next element of the source string.

If DF=0. The register DI will be incremented by 1 for byte and increament by 2 for word. If DF=1, the register DI will be ~~increment~~ decremented by 1 for byte and decrement by 2 for word.

In the instruction STOS, the source must be explicitly declared either with DB or with DW. Another way to store a byte or word is by using implicit instruction STOSB and STOSW.

The STOSB is used to transfer a byte from AL to destination and the STOSW is used to transfer a word from from AX to destination.

In multiple byte or word loads, the count must be loaded in CX register which functions as counter.

Operation:
If byte movement
ES: [DI] ← AL
For DF=0 DI ← DI+1
For DF=1 DI ← DI-1
If word movement
ES: [DI] ← AX
For DF=0 DI ← DI+2
For DF=1 DI ← DI-2

Example:
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV DI, OFFSET D-STRING
A] STOS D-STRING
B] STOSB
C] STOSW

④ CMPS : Compare string
CMPSB : Compare string byte
CMPSW : Compare string word
The instruction CMPS compares a byte or a word in the source string with a byte or word in destination string. The source must be in data segment and destination must be in extra segment.

The offset of the source byte or a word must be placed in SI register, which is represented as DS:SI and the offset of the destination byte or a word must be in DI register which is represented as ES:DI.

On the execution of this instruction, SI and DI registers are automatically incremented by 1 to point the next element of source and destination.

If DF=0, the SI and DI registers will be incremented by 1 for byte compare and incremented by 2 for word compare.

If DF=1, the SI and DI registers will be decremented by 1 for byte compare and decremented by 2 for word compare.

The DS:SI and ES:DI register must be loaded prior to execution of CMPS instruction.

In the instruction CMPS, the source must be explicitly declared either with DB or DW. Another way to compare a byte or word string is by using implicit instruction CMPSB and CMPSW. The instruction CMPSB is used to compare a byte in source with destination and the instruction CMPSW is used to compare a word in source destination.

In multiple byte or word compare, the count must be loaded in Cx register which functions as counter.

Flags modified :- AF, CF, OF, PF, SF, ZF

Operations performed by the instructions :

1) If [destination string byte/word > source string byte/word] then CF=0, ZF=0, SF=0

2) If [destination string byte/word < source string byte/word] then CF=1, ZF=0, SF=1

3) If [destination string byte/word = source string byte/word] then CF=0, ZF=1, SF=0

For byte comparison:

1] If DF=0 then SI ← SI+1 DI ← DI+1

2] If DF=1 then SI ← SI-1 DI ← DI-1

For word comparison:

1] If DF=0 then SI ← SI+2 DI ← DI+2

2] If DF=1 then SI ← SI-2 DI ← DI-2

Examples:

```
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET S-STRING
MOV DI, OFFSET D-STRING
```

A] CMPS S-STRING, D-STRING

B] CMPSB

C] CMPSW

⑤ SCAS : scan string

SCASB : scan string byte

SCASW : scan string word.

The instruction SCAS scans a string byte or a word with a byte in AL and word AX. The destination string is pointed by DI in ES.

On execution of string instruction, DI is automatically updated to point next element of the source string.

If DF=0, the DI register will be incremented by 1 for byte and incremented by 2 for word. If DF=1, the DI register will be decremented by 1 for byte and decremented by 2 for word.

In the instruction, CMPS the source must be explicitly declared either with DB or with DW. Another way to scan a byte or word in a string is by using implicit instruction SCASB and SCASW. The SCASB is used to scan a byte in a string and SCASW is used to scan a word in a string.

In multiple byte or word scan, the count must be loaded in CX register which functions as counter.

Flags modified :- AF, CF, OF, PF, SF, ZF

Operations performed by this instruction:

1) [If byte in AL or word in AX > destination string byte or word] Then CF=0, ZF=0, SF=0

2) [If byte in AL or word in AX < destination string byte or word] Then CF=1, ZF=0, SF=1

3) [If byte in AL or word in AX = destination string byte or word] Then CF=0, ZF=1, SF=0

For byte scan:

If DF=0 then DI ← DI+1

If DF=1 then DI ← DI-1

For word scan:

If DF=0 then DI ← DI+2

If DF=1 then DI ← DI-2

Examples:

```
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV DI, OFFSET D-STRING
MOV AL, 'V'
```

A] SCAS D-STRING
B] SCASB
C] SCASW

⑥ REP : Repeat
The instruction is used as a prefix instruction with the string instructions and interpreted as a "repeat while not end of the string" [CX not equal to zero]. Count for repeat must be loaded in CX register.

Operation:
While CX ≠ 0
1] Execute string instruction
2] CX ← CX - 1

Examples:

```
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV CX, length of string
MOV SI, OFFSET S-STRING
MOV DI, OFFSET D-STRING
REP CMPSB or CMPSW
```

⑦ REPE : Repeat while equal

⑧ REPZ : Repeat while zero
The instruction is used as a prefix instruction with the string instructions and interpreted as a "repeat while not end of the string and string equal" [CX not equal to zero and ZF = 1]. Count must be loaded in CX register. Assembler generates same machine code for the instruction REPZ and REPE.

Operation:
While CX ≠ 0 and ZF = 1
1) Execute string instruction
2) CX ← CX - 1

Examples:

```
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
    MOV CX, length of string
    MOV SI, OFFSET S-STRING
    MOV DI, OFFSET D-STRING
    REPE CMPSB or CMPSW
    JE STR-EQU
NOT EQU : MOV AX, 01H
    :
STR-EQU : MOV AX, 00H
    :
```

⑨ REPNE : Repeat while not equal
REPNZ : Repeat while not zero
The instruction is used as a prefix instruction with the string instructions and interpreted as a "repeat while not end of the string and string not equal" [CX not equal to zero and ZF = 0]. Count for repeat must be loaded in CX register. Assembler generates same machine code for REPNE and REPNZ

Operation:
While CX ≠ 0 and ZF = 0
1) Execute string instruction
2) CX ← CX - 1

Examples:

```
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
CLD
MOV CX, length of string
MOV SI, OFFSET S-STRING
MOV DI, OFFSET D-STRING
REPE CMPSB or CMPSW
JNE NOT EQU
STR-EQU: MOV AX, 01H
:
NOT-EQU: MOV AX, 00H
```

AAM AAD SHL/SAL SABM SHR ROR
ROL RCR RCL

Difference between RCR & RCL

-(Refer textbook)