CSE 331L
(Microprocessor Interfacing & Embedded System (Lab))

HOMEWORK 04

Submitted by:

NAME: HOSNE ARA
ID: 1632267642
SECTION: 07

Submitted to:

ASIF AHMED NELOY

Assignment -04

8086 Instruction set

Data Transfer Instructions

MOV-MOV Destination, Source : 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 source and destination can not both be memory locations. They must both be of the same type (bytes or words). MOV instruction does not affect any flag.

→MOV CX,037AH — Put immediate number 037AH to CX

→MOV BL,[437AH] — copy byte in DS at offset 437AH to BL

→MOV AX, BX — Copy content of register BX to AX

→MOV DL,[BX] — Copy byte from memory at [BX] to DL

→ MOV DS, BX — Copy word from BX to DS register

→ MOV RESULT [BP], AX — Copy AX to two memory locations; AL to the first location, AH to the second; EA of the first memory location is sum of the displacement represented by RESULTS and content of BP. Physical address = EA+SS.

→ MOV ES: RESULTS [BP], AX — Same as the above instruction, but physical address = EA+ES, because of the segment override prefix ES.

LEA – LEA Register, Source: This instruction determines the offset of the variable on memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

→ LEA BX, PRICES — Load BX with offset of PRICE in DS

→ LEA BP, SS:STACK_TOP — Load BP with offset of STACK_TOP in SS

→ LEA CX, [BX][DI] — Load CX with EA = [BX]+[DI]

ARITHMETIC INSTRUCTIONS

ADD – ADD Destination, Source

ADC – ADC Destination, Source

These instructions add a number from some source to a number in some destination and put the result in the specified destination. The ADC also adds the states of the carry flag to the result. The source may be an immediate number, a register or a memory location. The destination may be a register or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type. If you want to add a byte to a word, you must copy the

byte to a word location and fill the upper byte of the word with 0's before adding. Flags affected: AF, CF, OF, SF, ZF.

→ ADD AL, 74H — Add immediate number 74H to content of AL. Result in AL.

→ ~~ADD~~ ADC CL, BL — Add content of BL plus carry states of content of CL

→ ADD DX, BX — Add content of BX to content of DX

→ ADD DX, [SI] — Add word from memory at offset [SI] in DS to content of DX

→ ADC AL, PRICES [BX] — Add byte from effective address PRICES [BX] plus carry states to content of AL

→ ADD AL, PRICES [BX] — Add content of memory at effective address PRICES [BX] to AL

SUB - SUB Destination, Source
SBB - SBB Destination, Source

These instructions subtract the number in some source from the number in some destination and put the result in the destination. The SBB instruction also subtracts the content of carry flag from the destination. The source may be an immediate number, a register or memory location. The destination can also be a register or a memory location. However, the source and the destination cannot both be memory location. The source and the destination must both be of the same type. If you want to subtract a byte from a word, you must first move the byte to a word location such as a 16-bit register and fill the upper byte of the word with 0's. Flags affected: AF, CF, OF, PF, SF, ZF.

→ SUB CX, BX — CX − BX ; Result in CX

→ SBB CH, AL — Subtract content of AL and content of CF from content of CH. Result in CH.

→ SUB AX, 3427H — Subtract immediate number 3427H from AX

→ SBB BX, [3427H] — Subtract word at displacement 3427H in DS and content of CF from BX

→ SUB PRICES [BX], 04H — Subtract 04 from byte at effective address PRICES [BX], if PRICES is declared with DB; Subtract 04 from word at effective address PRICES [BX], if it is declared with DW.

→ SBB CX, TABLE [BX] — Subtract word from effective address TABLE [BX] and Status of CF from CX.

→ SBB TABLE [BX], CX Subtract CX and States of CF from word in memory at effective address TABLE [BX].

MUL - MUL Source :

This instruction multiplies an unsigned byte in some source with an unsigned byte in AL register or an unsigned word in some source with an unsigned word in AX register. The source can be a register or a memory location. When a byte is multiplied by the content of AL, the result in put in AX. When a word is multiplied by the content of AX, the result is put in DX and AX registers. If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0's. AF, PF, SF and ZF are undefined after a MUL instruction If you want to multiply a byte with a word

you must first move the byte to a word location such as an extended register and fill the upper byte of the word with all 0's. You cannot use the CBW instruction for this, because the CBW instruction fills the upper byte with copies of the most significant bit of the lower byte.

→ MUL BH — Multiply AL with BH; result in AX

→ MUL CX — Multiply AX with CX; result high word in DX, low word in AX

→ MUL BYTE PTR [BX] — Multiply AL with byte in DS pointed to by [BX]

→ MUL FACTOR [BX] — Multiply AL with byte at effective address FACTOR [BX], if it is declared as type byte with DB. Multiply AX with word at effective address FACTOR [BX], if it is declared as type word with DW.

```
→ MOV  AX, MCAND_16    Load 16-bit multiplicand
                       into AX
  MOV  CL, MPLIER_8    Load 8-bit multiplier
                       CL
  MOV  CH, 00H         Set upper byte of CX
                       to all 0's
  MUL  CX              AX times CX; 32-bit
                       result in DX and AX
```

DIV - DIV Source:

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word (32 bits) by a word. When a word is divided by a byte, the word must be in the AX register. The divisor can be in a register or a memory location. After the division, AL will contain the 8-bit quotient and AH will contain the 8-bit remainder. When a double word is divided by a word, the most significant word of the double word must be in DX and the least significant word of the double word

must be in AX. After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder. If an attempt is made to divide by 0 or if the quotient is too large to fit in the destination (greater than FFH/FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after a DIV instruction.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all 0's.

DIV BL	Divide word in AX by byte in BL; quotient in AL, remainder in AH.
DIV CX	Divide down word in DX and AX by word in CX; quotient in AX and remainder in DX.

DIV SCALE [BX] — AX/ if SCALE[BX] is of type byte ; or (DX and AX)/(word at effective address SCALE [BX] if SCALE[BX] is of type word.

INC - INC Destination :

The INC instruction adds 1 to a specified register or to a memory location. AF, OF, PF, SF and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result will be all 0's with no carry.

→ INC BL — Add 1 to contains of BL register

→ INC CX — Add 1 to contains of CX register

→ INC BYTE PTR [BX] — Increment byte in data segment at offset contained in BX.

→ INC WORD PTR [BX] — Increment the word at offset of [BX] and [BX+1] in the data segment.

→ INC TEMP

Increment byte or word named TEMP in the data segment. Increment byte if MAX_TEMP declared with DB. Increment word if MAX_TEMP is declared with DW.

→ INC PRICES[BX]

Increment element pointed to by [BX] in array PRICES. Increment a word if PRICES is declared as an array of words. Increment a byte if PRICES is declared as an array of bytes.

DEC-DEC Destination:

This instruction subtracts 1 from the destination word or byte. The destination can be a register or a memory location. AF, OF, SF, PF and ZF are updated but CF is not affected. This means that if an 8-bit destination containing 00H or a 16-bit destination containing 0000H is decremented, the result will be FFH or FFFFH with no carry (borrow).

→DEC CL	Subtract 1 from content of CL register
→DEC BP	Subtract 1 from content of BP register
→DEC BYTE PTR [BX]	Subtract 1 from byte at offset [BX in DS.
→DEC WORD PTR [BP]	Subtract 1 from a word at offset [BP] in SS.
→DEC COUNT	Subtract 1 from byte or word named COUNT in DS. Decrement a byte if COUNT is declared with a DB. Decrement a word if COUNT is declared with a DW.

DAA (DECIMAL ADJUST AFTER BCD ADDITION)

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA to work correctly. If the lower nibble in AL after an

addition is greater than 9 or AF was set by the addition, then the DAA instruction will add 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 was set by the addition or correction then the DAA instruction will add 60H to AL.

→ Let AL = 59 BCD, and BL = 35 BCD

```
ADD AL, BL
DAA
```

AL = 8EH; lower nibble > 9, add 06H to AL

AL = 94 BCD, CF = 0

→ Let AL = 88 BCD, and BL = 49 BCD

```
ADD AL, BL
DAA
```

AL = D1H; AF = 1, add 06H to AL

AL = D7H; upper nibble > 9, add 60H to AL

AL = 37 BCD, CF = 1

The DAA instruction updates AF, CF, SF, PF and ZF; but OF is undefined.

AAA (ASCII ADJUST FOR ADDITION)

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD.

→ Let AL = 0011 0101 (ASCII 5) and BL = 0011 1001 (ASCII 9)

ADD AL, BL

AAA

AL = 0110 1110 (6EH, which is incorrect BCD)

AL = 0000 0100 (unpacked BCD 4)

CF = 1 indicates answer is in decimal.

The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.

LOGICAL INSTRUCTIONS

AND-AND Destination, Source:

This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0 after AND. PF, SF and ZF are updated by the AND instruction. AF is undefined. PF has meaning only for an 8-bit operand.

→ AND CX, [SI] — AND word in DS at offset [SI] with word in CX register; Result in CX register.

→ AND BH, CL — AND byte in CL with byte in BH; Result in BH

→ AND BX, 00FFH — 00FFH Masks upper byte, leaves lower byte unchanged.

OR-OR Destination, Source :

This instruction ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after OR. PF, SF and ZF are updated by the OR instruction. AF is undefined. PF has meaning only for an 8-bit operand.

→ OR AH, CL CL ORed with AH, result in AH, CL not changed

→OR BP, SI — SI ORed with BP, result in BP, SI not changed.

→OR SI, BP — BP ORed with SI, result in SI, BP not changed.

→OR BL, 80H — BL ORed with immediate number 80H; sets MSB of BL to 1

→OR CX, TABLE[SI] — CX ORed with word from effective address TABLE[SI]; Content of memory is not changed.

XOR-OR Destination, source:

This instruction Exclusive-ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register or the content of a memory location. The destination can be a register or a memory location. The source

and destination cannot both be memory locations. CF and OF are both 0 after XOR. PF, SF and ZF are updated. PF has meaning only for an 8-bit operand. AF is undefined.

→XOR CL, BH — Byte in BH exclusive-ORed with byte in CL. Result in CL. BH not changed.

→XOR BP, DI — Word in DI exclusive-ORed with word in BP. Result in BP. DI not changed.

→XOR WORD PTR [BX], 00FFH — Exclusive-OR immediate number 00FFH with word at offset [BX] in the data segment. Result in memory location [BX]

CMP - CMP Destination, Source:

This instruction compares a byte/word in the specified source with a byte/word in the specified destination. The source can be an immediate number, a register or a memory location. The destination can be a register or a memory location. However, the source and the

destination cannot both be memory locations. The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed but the flags are set to indicate the result of the comparison. AF, OF, SF, ZF, PF and CF are updated by the CMP instruction. For the instruction CMP CX, BX, the values of CF, ZF and SF will be as follows:

	CF	ZF	SF	
CX = BX	0	1	0	Result of subtraction is 0
CX > BX	0	0	0	No borrow required, so
CX < BX	1	0	1	CF = 0

Subtraction requires borrow, so CF = 1

→ CMP AL, 01H
Compare immediate number 01H with byte in AL

→ CMP BH, CL
Compare byte in CL with byte in BH

→ CMP CX, TEMP
Compare word in DS at displacement TEMP with word at CX

→ CMP PRICES [BX], 49H Compare immediate number 49H with byte at offset [BX] in array PRICES

TEST-TEST Destination, Source:

This instruction ANDs the byte/word in the specified source with the byte/word in the specified destination. Flags are updated but neither operand is changed. The test instruction is often used to set flags before a conditional jump instruction.

The source can be an immediate number, the content of a register or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0's after TEST. PS, SF and ZF will be updated to show the results of the destination. AF is be undefined.

→ TEST AL, BH AND BH with AL. No result stored; Update PF, SF, ZF.

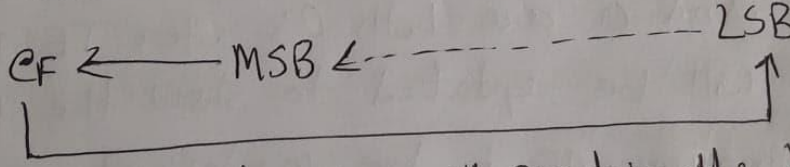
→ TEST CX,0001H — AND CX with immediate number 0001H ; No result stored ; Update PF, SF, ZF

→ TEST BP, [BX] [DI] — AND word are offset [BX] [DI] in DS with word in BP. No result stored. Update PF, SF and ZF

ROTATE AND SHIFT INSTRUCTIONS

RCL - RCL Destination, Count :

This instruction rotates all the bits in a specified word or byte some number of bit positions to the left. The operation circular because the MSB of the operand is rotated into the carry flag and the bit in the carry flag is rotated around into LSB of the operand.



For multi-bit rotates, CF will contain the bit most recently rotated out of the MSB.

The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting

a 1 in the count position of the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction. RCL affects only CF and OF. OF will be a 1 after a single bit RCL if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

→ RCL DX, 1 — Word in DX 1 bit left, MSB to CF, CF to LSB

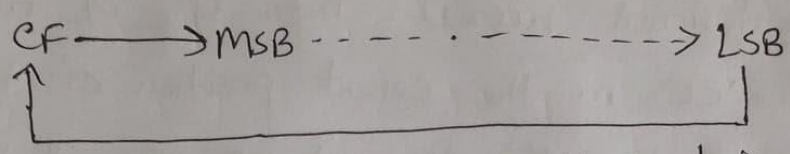
→ MOV CL, 4 — Load the number of bit positions to rotate into CL

RCL SUM[BX], CL — Rotate byte or word at effective address SUM [BX] 4 bits left. Original bit 4 now in CF, original CF now in bit 3.

RCR - RCR Destination, Count:

This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation circular because the LSB of the operand is rotated into the carry

flag and the bit in the carry flag is rotate around into MSB of the operand.



For multi-bit rotate, CF will contain the bit most recently rotated out of the LSB.

The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

RCR affects only CF and OF. OF will be a 1 after a single bit RCR if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

→RCR BX, 1 Word in BX right 1 bit, CF to MSB, LSB to CF

→MOV CL, 4 Load CL for rotating 4 bit position

RCR BYTE PTR[BX], 4 Rotate the byte at offset [BX] in DS 4 bit positions right CF = Original bit 3, Bit 4 — Original CF.

SAL-SAL Destination, Count

SHL-SHL Destination, count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB operation, a 0 is put in the LSB position. The MSB will be shifted into CF. In the case of multi-bit shift, CF will contain the bit most recently shifted out from the MSB. Bits shifted into CF previously will be lost.

CF ← MSB ← ---------- LSB ← 0

The destination operand can be a byte or a word. It can be in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register and put "CL" in the count position of the instruction.

The flags are affected as follow: CF contains the bit most recently shifted out from MSB. For a count of one, OF will be 1 if CF and the current MSB are not the same. For multiple-bit shifts, OF is undefined. SF and ZF will be updated to reflect the condition of the destination. PF will have meaning only for an operand in AL. AF is undefined.

→ SAL BX, 1	Shift word in BX 1 bit position left, 0 in LSB
→ MOV CL, 02h SAL BP, CL	Load desired number of shifts in CL Shift word in BP left CL bit positions, 0 in LSBs
→ SAL BYTE PTR [BX], 1	Shift byte in DX at offset [BX] 1 bit position left, 0 in LSB

SAR - SAR Destination, Count:

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. In other words, the sign bit is copied into the MSB. The LSB will be shifted into CF. In the case of multiple-bit shift, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.

MSB ⟶ MSB - - - - - - - - - ⟶ LSB ⟶ CF

The destination operand can be a byte or a word. It can be in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more ~~than~~ than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected as follow: CF contains the bit most recently shifted in from LSB. For a count of one, OF will be 1 if the two MSBs are not the same. After a multi- bit SAR, OF will be 0. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8-bit destination. AF will be undefined after SAR.

→ SAR DX, 1	Shift word in DI one bit position right, new MSB = old MSB
→ MOV CL, 02H	Load desired number of shifts in CL
SAR WORD PTR [BP], CL	Shift word at offset [BP] in stack segment right by two bit positions, the two MSBs are now copies of original LSB

SHR - SHR Destination, Count:

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position a 0 is put in its place. The bit shifted out of the LSB position goes to CF. In the case of multi-bit shifts, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.

0 ⟶ MSB - - - - - - - - - → LSB ⟶ CF

The destination operand can be a byte or a word in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction

→SHR BP,1 Shift word in BP one bit position right, 0 in MSB

→MOV CL,03H
SHR BYTE PTR [BX] Load desired number of shifts into CL
Shift byte in DS at offset [BX] 3 bits right; 0's in 3 MSBs.

TRANSFER-OF-CONTROL INSTRUCTIONS

JMP (UNCONDITIONAL JUMP TO SPECIFIED DESTINATIO

This instruction will fetch the next instructio from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as the JMP instructi then only the instruction pointer will be changed to get the destination location. This

is referred to as a near jump. If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register content will be changed to get the destination location. This referred to as a far jump.

→ JMP CONTINUE

This instruction fetches the next instruction from address at label CONTINUE. If the label is in the same segment. If the label is another segment then IP and CS will be replaced with value coded in part of the instruction.

→ JMP BX

This instruction replaces the content of IP with the content of BX. BX must first be loaded with the offset of the destination

instruction in CS. This is a near jump. It is also referred to as an indirect jump because the new value of IP comes from a register rather than from the instruction itself, as in a direct jump.

→JMP WORD PTR [BX]

This instruction replaces IP with word from a memory location pointed ~~by~~ to by BX in DX. This is an indirect near jump.

→JMP DWORD PTR [SI]

This instruction replaces IP with word pointed ~~by~~ to by SI in DS. It replaces CS with a word pointed by SI + 2 in DS. This is an indirect far jump.

JBE/JNA (JUMP IF BELOW OR EQUAL/JUMP IF NOT ABOVE)

If, after a compare or some other instruction which affect flags, either the zero flag or the carry flag is 1, this instruction will cause execution to jump to a label

given in the instruction. If CF and ZF are both 0, the instruction will have no effect on program execution.

→ CMP AX, 4371H — Compare (AX - 4371H)
JBE NEXT — Jump to label NEXT if AX is below or equal to 4371H

→ CMP AX, 4371H — Compare (AX - 4371H)
JNA NEXT — Jump to label NEXT if AX not above 4371H

JG/JNLE (JUMP IF GREATER/JUMP IF NOT LESS THAN OR EQUAL

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the zero flag is 0 and the carry flag is the same as the overflow flag.

→ CMP BL, 39H — Compare by subtracting 39H from BL
JG NEXT — Jump to label NEXT if BL more positive than 39H

→ CMP BL, 39H
JNLE NEXT

Compare by subtracting 39H from BL
Jump to label NEXT if BL is not less than or equal to 39H

JL/JNGE (JUMP IF LESS THAN/JUMP IF NOT GREATER THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the sign flag is not equal to the overflow flag.

→ CMP BL, 39H
JL AGAIN

Compare by subtracting 39H from BL
Jump to label AGAIN if BL more negative than 39H

→ CMP BL, 39H
JNGE AGAIN

Compare by subtracting 39 H from BL
Jump to label AGAIN if BL not more positive than or equal to 39H.

JLE / JNG (JUMP IF LESS THAN OR EQUAL / JUMP IF NOT GREATER)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the zero flag is set, or if the sign flag not equal to the overflow flag.

→ CMP BL, 39H
JLE NEXT

Compare by subtracting 39H from BL
Jump to label NEXT if BL more negative than or equal to 39H

→ CMP BL, 39H
JNG NEXT

Compare by subtracting 39H from BL
Jump to label NEXT if BL not more positive than 39H.

JE/JZ (JUMP IF EQUAL / JUMP IF ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is set, then this instruction will cause a jump to the label given in the instruction.

```
→CMP BX,DX      Compare (BX-DX)
 JE DONE        Jump to DONE if BX = DX
→IN AL,30H      Read data from port 8FH
 SUB AL,30H     Subtract the minimum value.
 JZ START       Jump to label START if the
                result of subtraction is 0
```

JNE/JNZ (JUMP NOT EQUAL/JUMP IF NOT ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is 0, then this instruction will cause a jump to the label given in the instruction.

```
→IN AL,0F8H     Read data value from port
 CMP AL,72      Compare (AL-72)
 JNE NEXT       Jump to label NEXT if AL≠72
→ADD AX,0002H   Add count factor 0002H to AX
 DEC BX         Decrement BX
 JNZ NEXT       Jump to label NEXT if BX≠0
```

STACK RELATED INSTRUCTIONS

PUSH - PUSH source:

The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment to which the stack pointer points. The source of the word can be general purpose register, segment register or memory. The stack segment register and the stack pointer must be initialized before this instruction can be used. PUSH can be used to save data on the stack so that it will not destroyed by a procedure. This instruction does not affect any flag.

→PUSH BX	Decrement SP by 2, copy BX to stack.
→ PUSH DS	Decrement SP by 2, copy DS to stack.
→ PUSH BL	Illegal ; must push a word
→ PUSH TABLE [BX]	Decrement SP by 2 and copy word from memory in DS at EA = TABLE + [BX] to stack

POP – POP Destination:

POP instruction copies a word from the stack location pointed to by the stack pointer to a destination specified in the instruction. The destination can be a general-purpose register, a segment register or a memory location. The data in the stack is not changed. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2 to point to the next word on the stack. The POP instruction does not affect any flag.

→ POP DX — copy a word from top of stack to DX; increment SP by 2

→ POP DS — copy a word from top of stack to DS; increment SP by 2

→ POP TABLE[BX] — Copy a word from top of stack to memory in DS with EA = TABLE + [BX]; increment SP by 2.

INPUT-OUTPUT INSTRUCTIONS

IN - IN Accumulator ; Port :

The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX.

The IN instruction has two possible formats, fixed port and variable port. For fixed port type, 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.

→ IN AL, 0C8H Input a byte from port 0C8H to AL

→ IN AX, 34H Input a word from port 34H to AX

For the variable-port ~~form~~ form of 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. Therefore, up to 65,536

ports are addressable in this mode.

→ MOV DX, OFF78H — Initialize DX to point to port
IN AL, DX — Inplet a byte from 8-bit port OFF78H to AL
IN AX, DX — Inplet a word from 16-bit port OFF78H to AX

The variable-port IN instruction has advantage that the port address can be computed or dynamically determined in the program.
The IN instruction does not change any flag.

OUT-OUT Port, Accumulator:

The OUT instruction copies a byte from AL or a word from AX to the specified port. The OUT instruction has two possible forms, fixed port and variable port.
For the fixed port form, the 8-bit port address is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

→OUT 3BH, AL Copy the content of AL to port 3BH

→OUT 2CH, AX Copy the content of AX to port 2CH

For variable port form of the OUT instruction, the content of AL or AX will be copied to the port at an address contained in DX. Therefore, the DX register must be loaded with the desired port address before this form of the OUT instruction is used.

→MOV DX, 0FFF8H Load desired port address in DX

OUT DX, AL Copy content of AL to port FFF8H

OUT DX, AX Copy content of AX to port FFF8H

The OUT instruction does not affect any flag.

8086 ASSEMBLER DIRECTIVES

ENDS (END SEGMENT)

This directive is used with the name of a segment to indicate the end of that logical segment.

→ CODE SEGMENT — Start of logical segment containing code instruction statements.
CODE ENDS — End of segment named CODE.

END (END PROCEDURE)

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

DW (DEFINE WORD)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER, and initialized with the value 437AH when the program is loaded into memory to be run.

→ WORDS DW 1234H, 3456H

Declare an array of 2 words and initialize them with the specified values.

→ STORAGE DW 100 DUP(0)

Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named as STORAGE.

→ STORAGE DW 100 DUP(?)

Reserve 100 word of storage in memory and give it the name STORAGE, but leave the words un-initialized.

PROC (PROCEDURE)

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term near or the term far is used to specify the type of the procedure. The statement DIVIDE PROC FAR, for example, identifies the start of a procedure named DIVIDE and tells the assembler that the procedure is far. The PROC directive is used with the ENDP directive to "bracket" a procedure.

ENDP (END PROCEDURE)

The directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The directive together with the procedure directive, PROC is used to "bracket" a procedure.

```
→ SQUARE_ROOT PROC      Start of procedure.
  SQUARE_ROOT ENDP      End of procedure.
```

LABEL

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to be keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term that specifies the type you want to associate with that name. If the label is going to be used as the destination for a jump or a call then the label must be specified as type near or type far. If the label is going to be used to reference a data item then the label must be specified as type byte, type word or type double word. Here's how we use the LABEL directive for a jump address.

→ENTRY_POINT LABEL FAR Can jump to here from another segment
NEXT: MOV AL,BL Can not do a far jump directly to a label with a colon

The following example shows how we use the label directive for a data reference.

→STACK_SEG SEGMENT STACK
DW 100 DUP (0) Set aside 100 words for stack.
STACK_TOP LABEL WORD Give name to next location after last word in stack.
STACK_SEG ENDS

To initialize stack pointer, use MOV SP, OFFSET STACK_TOP.

INCLUDE (INCLUDE SOURCE CODE FROM FILE)

This directive is used to tell the assembler to insert a block of source code from the named file into the current source module.