Computer Science Dept. / 2nd year 1st Course (2024-2025)
Microprocessor
Dr.Haleema Essa

INSTRUCTION SET OF 8086

❖ DATA TRANSFER INSTRUCTIONS

1) 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 cannot 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 [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

2) XCHG – XCHG Destination, Source

The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location. It **cannot directly** exchange the content of two memory locations. The source and destination must both be of the same type (bytes or words). ***The segment registers cannot be used in this instruction***. This instruction does not affect any flag.

XCHG AX, DX Exchange word in AX with word in DX

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

XCHG AL, [BX] Exchange byte in AL with byte in memory at [BX]

3) LEA – LEA Register, Source

This instruction determines the offset of the memory location named as the source and puts this offset in the indicated 16-bit register.

LEA BX, 2435H Load BX with immediate 2435H

LEA BP, SS: STACK_TOP Load BP with offset of STACK_TOP

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

4) LDS – LDS Register, Memory address of the first word

This instruction loads new values into the specified register and into the DS register from four successive memory locations. The word from two memory locations is copied into the specified register and the word from the next two memory locations is copied into the DS registers.

LDS BX, [4326] Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.

5) LES – LES Register, Memory address of the first word

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES does not affect any flag.

LES BX, [789AH] Copy content of memory at displacement 789AH in DS to BL, content of 789BH to BH, content of memory at displacement 789CH and 789DH in DS is copied to ES register.

LES DI, [BX] Copy content of memory at offset [BX] and offset [BX]+1 in DS to DI register. Copy content of memory at offset [BX]+2 and [BX]+3 to ES register.

6) 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. 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 [BX] Decrement SP by 2, and copy word from memory in DS at [BX] to stack

7) POP – POP Destination

The 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 [BX] Copy a word from top of stack to memory in DS at [BX]; increment SP by 2.

8) PUSHF (PUSH FLAG REGISTER TO STACK)

The PUSHF instruction decrements the stack pointer by 2 and copies a word in the **Flag Register** to two memory locations in stack pointed to by the stack pointer. The stack segment register is not affected. This instruction does to affect any flag.

9) POPF (POP WORD FROM TOP OF STACK TO FLAG REGISTER)

The POPF instruction copies a word from two memory locations at the top of the stack to the flag register and increments the stack pointer by 2. The stack segment register and word on the stack are not affected. This instruction does to affect any flag.

❖ INPUT-OUTPUT INSTRUCTIONS

1) 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, OC8H Input a byte from port OC8H to AL

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

For the variable-port form of the 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, 0FF78H Initialize DX to point to port

IN AL, DX Input a byte from 8-bit port 0FF78H to AL

IN AX, DX Input a word from 16-bit port 0FF78H to AX

The variable-port IN instruction ***has advantage*** **that the port address can be computed or dynamically determined in the program**. Suppose, for example, that an 8086-based computer needs to input data from 10 terminals, each having its own port address. Instead of having a separate procedure to input data from each port, you can write one generalized input procedure and simply pass the address of the desired port to the procedure in DX.

2) 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

❖ ARITHMETIC INSTRUCTIONS

1) 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 status of the carry flag to the result.

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

ADC CL, BL Add content of BL plus carry status to content of CL, Result in CL

ADD DX, BX Add content of BX to content of DX, Result in DX

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

ADC AL, [BX] Add byte from memory at [BX] plus carry status to content of AL

2) 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.

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 [BX], 04H Subtract 04 from byte at [BX]

SBB CX, [BX] Subtract word at [BX] and status of CF from CX.

SBB [BX], CX Subtract CX and status of CF from word in memory at [BX].

3) MUL – MUL Source
IMUL – IMUL 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 (product) is put in **AX**. When a word is multiplied by the content of AX, the result is put in **DX and AX** registers.

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]

The **IMUL** instruction multiplies a **signed** byte from source with a signed byte in AL or a signed word from some source with a signed word in AX. The source can be a register or a memory location. When a byte from source is multiplied with content of AL, the signed result (product) will be put in AX. When a word from source is multiplied by AX, the result is put in DX and AX.

IMUL BH Multiply signed byte in AL with signed byte in BH; result in AX.

IMUL AX Multiply AX times AX; result in DX and AX

4) DIV – DIV Source
IDIV – IDIV 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**.

Note: 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.

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

The **IDIV** instruction is used to divide a **signed** word by a signed byte, or to divide a signed double word by a signed word.

IDIV BL Signed word in AX/signed byte in BL

IDIV BP Signed double word in DX and AX/signed word in BP

IDIV BYTE PTR [BX] AX / byte at offset [BX] in DS

5) 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 PRICES [BX] Increment element pointed to by [BX] in array PRICES.

Note: Increment a word if PRICES is declared as an array of words; Increment a byte if PRICES is declared as an array of bytes.

6) 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] and [BP+1] in SS.

7) 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 **06H** to the lower nibble in AL. If the result in the **upper nibble** of AL in 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 AL = 8EH; lower nibble > 9, add 06H to AL

DAA AL = 94 BCD, CF = 0

Let AL = 88 BCD, and BL = 49 BCD

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

DAA 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.

8) DAS (DECIMAL ADJUST AFTER BCD SUBTRACTION)

This instruction is used after subtracting one packed BCD number from another packed BCD number, to make sure the result is correct packed BCD. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is **greater than 9** or the **AF was set** by the subtraction, then the DAS instruction will **subtract 06** from the lower nibble AL. If the result in the upper

nibble is now **greater than 9** or if the **carry flag was set**, the DAS instruction will **subtract 60** from AL.

Let AL = 86 BCD, and BH = 57 BCD

SUB AL, BH AL = 2FH; lower nibble > 9, subtract 06H from AL

DAS AL = 29 BCD, CF = 0

Let AL = 49 BCD, and BH = 72 BCD

SUB AL, BH AL = D7H; upper nibble > 9, subtract 60H from AL

DAS AL = 77 BCD, CF = 1 (borrow is needed) The DAS instruction updates AF, CF, SF, PF, and ZF; but OF is undefined.

9) CBW (CONVERT SIGNED BYTE TO SIGNED WORD)

This instruction ***copies the sign bit of the byte in AL to all the bits in AH***. AH is then said to be the **sign extension** of AL. CBW does not affect any flag.

Let AX = 00000000 10011011 (-155 decimal)

CBW Convert signed byte in AL to signed word in AX (AX = 11111111 10011011) (-155 decimal)

10) CWD (CONVERT SIGNED WORD TO SIGNED DOUBLE WORD)

This instruction copies the sign bit of a word in AX to all the bits of the DX register. In other words, it extends the sign of AX into all of DX. CWD affects no flags.

Let DX = 00000000 00000000, and AX = 11110000 11000111 (-3897 decimal from signed 2's complement)

CWD Convert signed word in AX to signed double word in DX:AX (DX = 11111111 11111111 AX = 11110000 11000111) (-3897 decimal)

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

AAA AL = 0000 0100 (unpacked BCD 4) CF = 1 indicates answer is 14 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.

12) AAM (BCD ADJUST AFTER MULTIPLY)

Before you can multiply two ASCII digits, you must first mask the upper 4 bit of each. This leaves unpacked BCD (one BCD digit per byte) in each byte. After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX. AAM works only after the multiplication of two unpacked BCD bytes, and ***it works only the operand in AL***. AAM updates PF, SF and ZF but AF; CF and OF are left undefined.

Let AL = 00000101 (unpacked BCD 5), and BH = 00001001 (unpacked BCD 9)

MUL BH AL x BH: AX = 00000000 00101101 = 002DH

AAM AX = 00000100 00000101 = 0405H (unpacked BCD for 45) **(AH = AL / 10 and AL = remainder)**

❖ STRING MANIPULATION INSTRUCTIONS

1) MOVS – MOVS Destination String Name, Source String Name
MOVSB – MOVSB Destination String Name, Source String Name
MOVSW – MOVSW Destination String Name, Source String Name

This instruction **copies a byte or a word from location in the data segment to a location in the extra segment**. ***The offset of the source in the data segment must be in the SI register***. ***The offset of the destination in the extra segment must be in the DI register***. For multiple-byte or multiple-word moves, the number of elements to be moved is put in the **CX** register so that it can function ***as a counter***. After the byte or a word is moved, SI and DI are automatically adjusted to point to the next source element and the next destination element. If DF is **0**, then SI and DI will **incremented by 1** after a byte move and by 2 after a word move. If DF is **1**, then **SI and DI will be decremented by 1** after a byte move and by 2 after a word move. MOVS does not affect any flag.

When using the MOVS instruction, you must in some way tell the assembler whether you want to ***move a string as bytes or as word***. There are two ways to do this. The first way is to indicate the name of the source and destination strings in the instruction, as, for example; MOVS DEST, SRC. In the second way, the assembler will code the instruction for a byte / word move by adding a “B” or a “W” to the MOVS mnemonic. **MOVSB** says move a string as bytes; **MOVSW** says move a string as words.

MOV SI, OFFSET SOURCE Load offset of start of source string in DS into SI
MOV DI, OFFSET DESTINATION Load offset of start of destination string in ES into DI
CLD Clear DF to auto increment SI and DI after move
MOV CX, 04H Load length of string into CX as counter
REP MOVSB Move string byte until CX = 0

2) LODS / LODSB / LODSW (LOAD STRING BYTE INTO AL OR STRING WORD INTO AX)

This instruction ***copies a byte from a string location pointed to by SI to AL***, or a ***word from a string location pointed to by SI to AX***. If DF is 0, SI will be automatically incremented (by 1 for a byte string, and 2 for a word string) to point to the next element of the string; If DF is 1. SI will be automatically decremented (by 1 for a byte string, and 2 for a word string) to point to the previous element of the string. LODS does not affect any flag.

CLD Clear direction flag so that SI is auto- incremented
MOV SI, OFFSET SOURCE Point SI to start of string
LODS SOURCE Copy a byte or a word from string to AL or AX

Note: The assembler uses the name of the string to determine whether the string is of type bye or type word. Instead of using the string name to do this, you can use the mnemonic LODSB to tell the assembler that the string is type byte or the mnemonic LODSW to tell the assembler that the string is of type word.

3) STOS / STOSB / STOSW (STORE STRING BYTE OR STRING WORD)

This instruction ***copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI***. In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to next or previous element of the string. If DF is cleared, then DI will automatically incremented by 1 for a byte string and by 2 for a word string. If DI is set, DI will be automatically decremented by 1 for a byte string and by 2 for a word string. STOS does not affect any flag.

MOV DI, OFFSET TARGET
STOS TARGET

Note: The assembler uses the string name to determine whether the string is of type byte or type word. If it is a byte string, then string byte is replaced with content of AL. If it is a word string, then string word is replaced with content of AX.

```
MOV DI, OFFSET TARGET
STOSB
```

"B" added to STOSB mnemonic tells assembler to replace byte in string with byte from AL. STOSW would tell assembler directly to replace a word in the string with a word from AX.

4) CMPS / CMPSB / CMPSW (COMPARE STRING BYTES OR STRING WORDS)

This instruction can be used to compare a byte / word in one string with a byte / word in another string. ***SI is used to hold the offset of the byte or word in the source string***, and ***DI is used to hold the offset of the byte or word in the destination string***. ***The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison***, but the two operands are not affected. After the comparison, SI and DI will automatically be incremented or decremented to point to the next or previous element in the two strings. If DF is set, then SI and DI will automatically be decremented by 1 for a byte string and by 2 for a word string. If DF is reset, then SI and DI will automatically be incremented by 1 for byte strings and by 2 for word strings. The string pointed to by SI must be in the data segment. The string pointed to by DI must be in the extra segment.

- If dest string > source string then
  CF =0, ZF = 0, SF = 0
- If dest string < source string then
  CF =1, ZF = 0, SF = 1
- If dest string = source string then
  CF =0, ZF = 1, SF = 0

The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.

```
MOV SI, OFFSET FIRST    Point SI to source string
MOV DI, OFFSET SECOND   Point DI to destination string
CLD                     DF cleared, SI and DI will auto-increment after compare
```

MOV CX, 100 Put number of string elements in CX
REPE CMPSB Repeat the comparison of string bytes until end of string or until compared bytes are not equal

CX functions as a counter, which the REPE prefix will cause CX to be decremented after each compare. The **B** attached to CMPS tells the assembler that the strings are of type byte. If you want to tell the assembler that strings are of type word, write the instruction as **CMPSW**. The REPE CMPSW instruction will cause the pointers in SI and DI to be incremented by **2** after each compare, if the direction flag is set.

5) SCAS / SCASB / SCASW (SCAN A STRING BYTE OR A STRING WORD)

SCAS ***compares a byte in AL or a word in AX with a byte or a word in ES pointed to by DI***. Therefore, the string to be scanned must be in the extra segment, and DI must contain the offset of the byte or the word to be compared. If DF is cleared, then DI will be incremented by 1 for byte strings and by 2 for word strings. If DF is set, then DI will be decremented by 1 for byte strings and by 2 for word strings. SCAS affects AF, CF, OF, PF, SF, and ZF, but it does not change either the operand in AL (AX) or the operand in the string.

- If byte in AL or word in AX > dest string byte or word then
  CF =0, ZF = 0, SF = 0
- If byte in AL or word in AX < dest string byte or word then
  CF =1, ZF = 0, SF = 1
- If byte in AL or word in AX = dest string byte or word then
  CF =0, ZF = 1, SF = 0

The following program segment scans a text string of 80 characters for a carriage return, 0DH, and puts the offset of string into DI:

MOV DI, OFFSET STRING
MOV AL, 0DH Byte to be scanned for into AL
MOV CX, 80 CX used as element counter
CLD Clear DF, so that DI auto increments
REPNE SCAS String Compare byte in string with byte in AL

6) REP / REPE / REPZ / REPNE / REPNZ (PREFIX) (REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)

REP is a prefix, which is written before one of the string instructions. It will cause the **CX register to be decremented** and the **string instruction to be repeated until CX = 0**. The instruction *REP MOVSB*, for example, will continue to copy string bytes until the number of bytes loaded into CX has been copied.

REPE and REPZ are two mnemonics for the same prefix. They stand for repeat if equal and repeat if zero, respectively. They are often used with the **Compare String instruction or with the Scan String instruction**. They will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero. In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words not equal.

REPE CMPSB Compare string bytes until end of string or until string bytes not equal.

REPNE and REPNZ are also two mnemonics for the same prefix. They stand for repeat if not equal and repeat if not zero, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are not equal (ZF = 0) and CX is not yet counted down to zero.

REPNE SCASW Scan a string of word until a word in the string matches the word in AX or until all of the string has been scanned.