KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

ADDRESSING MODES OF 8086:

- The addressing modes describe the types of operands and the way they are accessed for executing an instruction.
- Addressing mode indicates a way of locating data or operands.
- According to the flow of instruction execution, the instructions may be categorized as
  - i) Sequential control flow instructions and
  - ii) Control transfer instructions

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logic, data transfer and processor control instructions are sequential control flow instructions.

The control transfer instructions, on the other hand, transfer control to some predefined address or the address somehow specified in the instruction, after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control flow instructions are:

1. Immediate: In this type of addressing, immediate data is a part of instruction and appears in the form of successive byte or bytes.

Ex: MOV AX, 0005H

In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. Direct: In the direct addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

Ex: MOV AX, [5000H]

Here, data resides in a memory location in the data segment, whose effective address may be completed using 5000H as the offset address and content of DS as segment address. **The effective address here, is 10H * DS + 5000H**.

3. Register: In register addressing mode, the data is stored in a register and is referred using the particular register. All the registers, except IP, may be used in this mode.

Ex: MOV BX, AX

4. Register Indirect: Sometimes, the address of the memory location, which contains data or operand, is determined in an indirect way, using the offset register. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI register. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

Ex: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset address is in BX. **The effective address of the data is given as 10H * DS+[BX].**

5. Indexed: In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers, SI and DI respectively. This is a special case of register indirect addressing mode.

Ex: MOV AX, [SI]

Here, data is available at an offset address stored in SI in DS. **The effective address, in this case, is computed as 10*DS+[SI].**

6. **Register Relative:** In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment.

Ex: MOV AX, 50H[BX]

Here, **the effective address is given as 10H *DS+50H+[BX]**

7. **Based Indexed:** The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Ex: MOV AX, [BX][SI]

Here, BX is the base register and SI is the index register. **The effective address is computed as 10H * DS + [BX] + [SI].**

8. **Relative Based Indexed:** The effective address is formed by adding an 8 or 16-bit displacement with the sum of the contents of any one of the base register (BX or BP) and any one of the index register, in a default segment.

Ex: MOV AX, 50H [BX] [SI]

Here, 50H is an immediate displacement, BX is base register and SI is an index register. **The effective address of data is computed as**

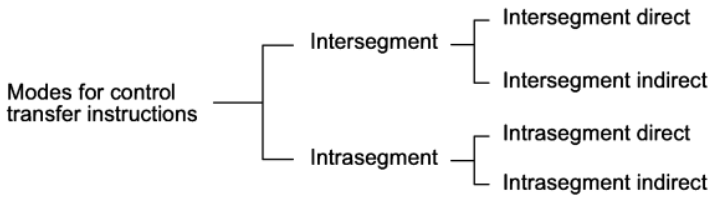
10H * DS + [BX] + [SI] + 50H

The addressing modes for control transfer instructions are:

For control transfer instructions, the addressing modes depend upon whether the destination is within the same segment or different one. It also depends upon the method of passing the destination address to the processor.

Basically, there are two addressing modes for the control transfer instructions, **intersegment** addressing and **intrasegment** addressing modes.

- If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode.
- If the destination location lies in the same segment, the mode is called intrasegment mode.



9. **Intrasegment Direct Mode:** In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an **immediate displacement valu**e. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

Ex: JMP 0080H

The effective address to which the control will be transferred is given by **10H*[CS] + [IP] + 0080H**

10. **Intrasegment Indirect Mode:** In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.
Ex: JMP [BX]

The effective address to which the control will be transferred is given by **10H*[CS] + [BX]**

11. **Intersegment Direct:** In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.
Ex: JMP 5000H:2000H

The effective address to which the control will be transferred is given by **10H*5000H + 2000H**

12. **Intersegment Indirect:** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e contents of a memory block containing four bytes, i.e IP (LSB), IP(MSB), CS(LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.
Ex: JMP [2000H]

INSTRUCTION SET OF 8086

The 8086 instructions are classified into the following main types:

1) Data copy/transfer instructions
2) Arithmetic instructions
3) Logical instructions
4) Shift and rotate instructions
5) Branch instructions
6) Loop instructions
7) Machine control instructions
8) Flag manipulation instructions
9) String instructions

1) Data copy/transfer instructions:

1. **MOV: Move** 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).

Eg:

- 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

2. **PUSH: Push to stack** The PUSH instruction decrements the stack pointer (SP) 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 a 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 be destroyed by a procedure. This instruction does not affect any flag.

Eg:

- ➢ PUSH BX — Decrement SP by 2, copy BX to stack.
- ➢ PUSH DS — Decrement SP by 2, and copy DS to stack.
- ➢ PUSH BL — Illegal; must push a word
- ➢ PUSH 20H [BX] — Decrement SP by 2, and copy the word from memory in DS at effective address = 20H + [BX] to stack

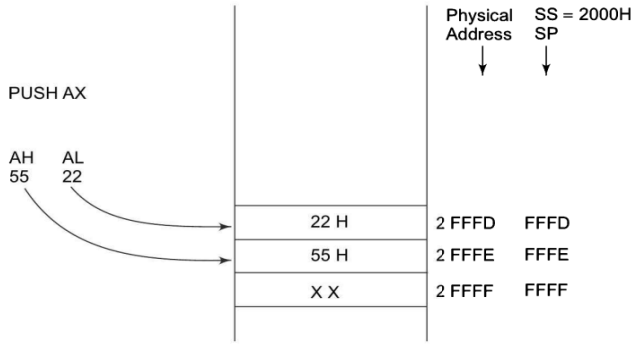
Pushing data to the stack:

Eg: MOV AX , 5522H
PUSH AX

Assume now the effective address (EA) = 10H*DS+SP = 10H*2000H + FFFFH = 2FFFF.

To push the data in AX to stack,

1. SP is decremented by 1 such that EA = 2FFFE.
2. Push the content in AH (upper byte) to 2FFFE.
3. Then again SP is decremented by 1 such that EA = 2FFFD.
4. Push the content in AL (lower byte) to 2FFFD.



Pushing Data to Stack Memory

3. **POP: Pop from stack** 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.

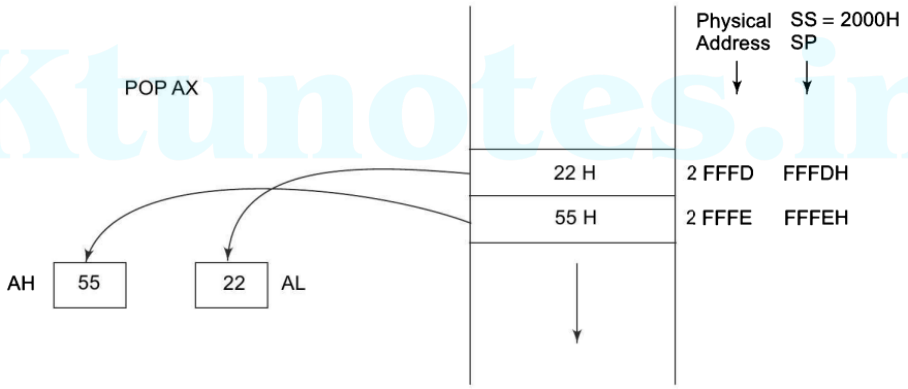
Pushing register content from the stack:

Eg: PUSH AX

Assume now the effective address (EA) = 10H*DS+SP = 10H*2000H + FFFDH = 2FFFD.

To push the data in AX to stack,

1. Contents of stack top memory location 2FFFD are stored in AL.
2. Then SP is incremented by 1 such that EA = 2FFFE.
3. Now the contents of the memory location pointed by SP are stored in AH.
4. Then SP is again incremented by 1 such that EA = 2FFFF.



Popping Register Contents from Stack Memory

4. **XCHG: Exchange** The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location(s). It cannot directly exchange the content of two memory locations. The source and destination must both be of the same type (bytes or words).

Eg:

- XCHG AX, DX — Exchange word in AX with the word in DX
- XCHG BL, CH — Exchange byte in BL with a byte in CH
- XCHG [5000H], AX — Exchange data between AX and memory location [5000H] in the data segment.

- ➢ XCHG AL, 20H[BX] Exchange byte in AL with a byte in memory
- ➢ at EA = 20H [BX] in DS.

5. **IN: Input the 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.

   Eg:

   - ➢ IN AL, OC8H Input a byte from port OC8H to AL
   - ➢ IN AX, 34H Input a word from port 34H to AX

   For a 16-bit port address, the port address is loaded into the DX register before the IN instruction.

   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

6. **OUT: Output to the port** The OUT instruction copies a byte from AL or a word from AX to the specified port.

   Eg:

   - ➢ OUT 3BH, AL Copy the content of AL to port 3BH
   - ➢ OUT 2CH, AX Copy the content of AX to port 2C

   For a 16-bit port address, the port address is loaded into the DX register before the OUT instruction.

   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

7. **XLAT: Translate** The XLATB instruction is used to translate a byte from one code (8 bits or less) to another code (8 bits or less). The instruction replaces a byte in the AL register with a byte pointed to by BX in a **lookup table (LUT**) in the memory.

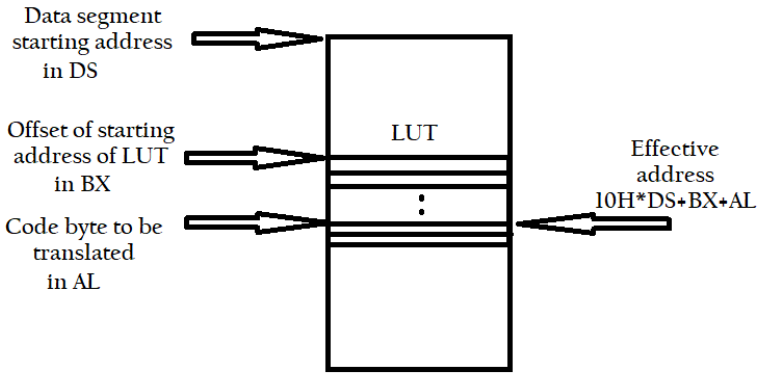
   Before the XLATB instruction can be executed, the lookup table containing the values for a new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX. The code byte to be translated is put in AL.

   The XLAT instruction adds the byte in AL to the offset of the start of the table in BX. It then copies the byte from the address pointed to by (BX + AL) back into AL.

   Eg:

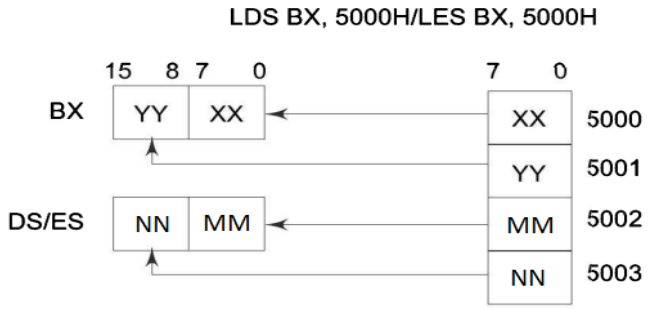
   MOV AX, 5000H Address of the data segment containing LUT is loaded
   MOV DS, AX in DS register.
   MOV BX, 2000H The offset of starting address of LUT is loaded in BX

MOV AL, 02H — The code byte to be translated is put in AL.
XLAT — Find the equivalent code (ie the content of effective address 10H*DS+BX+AL) and store it in AL.



8. **LEA: Load effective address** This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register.
   Eg:
   LEA BX, L1 — Load BX with the offset of label L1 in the current data segment.
   LEA SI, L1[BX] — Load SI with Effective Address = L1 + [BX]

9. **LDS/LES: Load pointer to DS/ES** This instruction loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as the source in the instruction.
   Eg:



10. **LAHF: Load AH from the lower byte of flag** The LAHF instruction copies the lower byte of the 8086 flag register to the AH register.
11. **SAHF: Store AH to the lower byte of flag register** The SAHF instruction replaces the lower byte of the 8086 flag register with a byte from the AH register.
12. **PUSHF: Push flags to stack** The PUSHF instruction decrements the stack pointer by 2 and copies a word in the flag register to two memory locations in the stack pointed to by the stack pointer.

13. **POPF: Pop flag from the stack** 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.

2) Arithmetic instructions:

1. **ADD - Add**

   This instruction adds a number from some source to a number in some destination and put the result in the specified destination. 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.

   Eg:

ADD AL, 74H	Add the immediate number 74H to the content of AL. Result in AL
ADD DX, BX	Add content of BX to the content of DX
ADD DX, [SI]	Add word from memory at offset [SI] in DS to the content of DX

2. **ADC - Add with carry**

   This instruction adds 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.

   Eg:

ADC CL, BL	Add the content of BL plus carry status to the content of CL

3. **SUB - Subtract**

   These instructions subtract the number in some source from the number in some destination and put the result in the destination. The source may be an immediate number, a register, or a memory location. The destination can also be a register or a memory location. However, the source and the destination cannot both be a memory location.

   Eg:

SUB CX, BX	CX – BX; Result in CX
SUB AX, 3427H	Subtract immediate number 3427H from AX; result in AX

4. **SBB - Subtract with borrow**

   This instruction subtracts 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 the carry flag from the destination.

   Eg:

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

5. **MUL - multiply**

This instruction multiplies an unsigned byte in some source with an unsigned byte in the AL register or an unsigned word in some source with an unsigned word in the 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.

Eg:

MUL BH	Multiply AL with BH; result in AX
MUL CX	Multiply AX with CX; higher word of the result in DX, lower word of the result in AX

6. **IMUL - signed multiply**

This instruction multiplies a signed byte from the 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 the source is multiplied by the content of AL, the signed result (product) will be put in AX. When a word from the source is multiplied by AX, the result is put in DX and AX. If the magnitude of the product does not require all the bits of the destination, the unused byte/word will be filled with copies of the sign bit (0 - positive, 1 - negative).

Eg:

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
IMUL CX	Multiply CX with AX; Result in DX and AX

7. **DIV - Divide**

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.

Eg:

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

8. **IDIV - Signed division**

This instruction is used to divide a signed word by a signed byte, or to divide a signed double word by a signed word.

Eg:

IDIV BL	Signed word in AX/signed byte in BL
IDIV BX	Signed double word in DX and AX/signed word in BX

9. INC - increment

The INC instruction adds 1 to a specified register or to a memory location.

Eg:

INC BL	Add 1 to the content of BL register
INC CX	Add 1 to the content of CX register

10. DEC - decrement

This instruction subtracts 1 from the destination word or byte.

Eg:

DEC CL	Subtract 1 from the content of CL register
DEC BX	Subtract 1 from the content of BX register

11. 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.

Eg 1:

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

Eg 2:

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; Answer is 137

12. 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 the correct packed BCD.
The result of the subtraction must be in AL for DAS to work correctly.

- If the lower nibble in AL after subtraction is greater than 9 or the AF was set by the subtraction, then the DAS instruction will subtract 6 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.

Eg 1:
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

Eg 2:
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)

13. 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. After the addition, the AAA instruction is used to make sure the result is the correct **unpacked BCD**.

Eg:
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

14. AAS - ASCII Adjust for Subtraction

Numerical data coming into a computer from a terminal is usually in an ASCII code. In this code the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to subtract the ASCII codes for two decimal digits. The AAS instruction is then used to make sure the result is the correct **unpacked BCD**.

Eg 1:
Let AL = 00111001 (39H or ASCII 9), and BL = 00110101 (35H or ASCII 5)

SUB AL, BL	AL = 00000100 (BCD 04), and CF = 0

AAS	AL = 00000100 (BCD 04), and CF = 0 (no borrow required)

Eg 2:
Let AL = 00110101 (35H or ASCII 5), and BL = 00111001 (39H or ASCII 9)

SUB AL, BL	AL = 11111100 (– 4 in 2's complement form), and CF = 1
AAS	AL = 00000100 (BCD 04), and CF = 1 (borrow required)

15. AAM - ASCII Adjust after Multiplication

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 with the operand in AL.

Eg:
Let AL = 05H, and BH = 09H

MUL BH	AL x BH: AX = 002DH
AAM	AX = 0405H (unpacked BCD for 45)

16. AAD - ASCII Adjust before division

AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the BCD division, AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder.

Eg:
Let AX = 0607 (unpacked BCD for 67 decimal), and CH = 09H

AAD	AX = 0043 (43H = 67 decimal)
DIV CH	AL = 07; AH = 04;

17. 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.

Eg:
Let AL = 10011011 (–155 decimal)

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

18. 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.

Eg:
Let AX = 11110000 11000111 (–3897 decimal)

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

19. NEG - Negate

Finds the 2's complement of the operand and stores in the same operand.

Eg:
MOV BL,01H
NEG BL BL = FFH (2's complement of 01H)

20. CMP-Compare

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 results of the comparison.

Eg: 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 CF = 0
CX < BX	1	0	1	Subtraction requires borrow, so CF = 1

3) Logical instructions:

1. AND

This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The source and the destination cannot both be memory locations.

Eg:
MOV AL,0FH
MOV AH,05H
AND AL,AH Result is AL = 05H; AH has not changed

2. **OR**

   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.

   Eg:

```
MOV AL,0FH
MOV AH,05H
OR AL,AH            Result is AL = 0FH; AH has not changed
```

3. **XOR**

   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.

   Eg:

```
MOV AL,0FH
MOV AH,05H
OR AL,AH            Result is AL = 0AH; AH has not changed
```

4. **NOT**

   The NOT instruction inverts each bit (forms the 1's complement) of a byte or word in the specified destination. The destination can be a register or a memory location.

   Eg:

```
MOV BX,0FH
NOT BX              Result is BX = F0H (1's complement of 0FH)
```

5. **TEST**

   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.

   Eg:

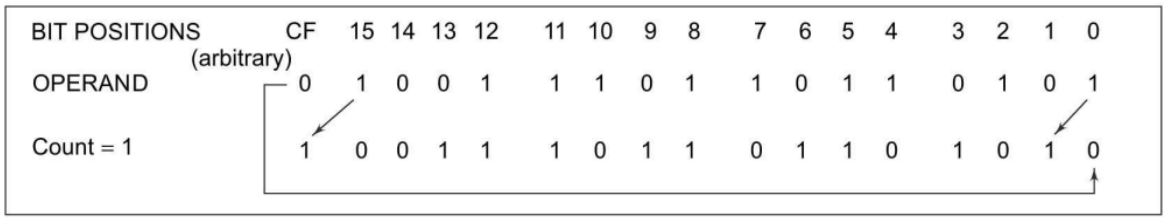
```
TEST AL, BH         AND BH with AL. No result stored; Update PF, SF, ZF
```

4) Rotate and shift instructions:

1. **RCL - Rotate Left through carry**

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

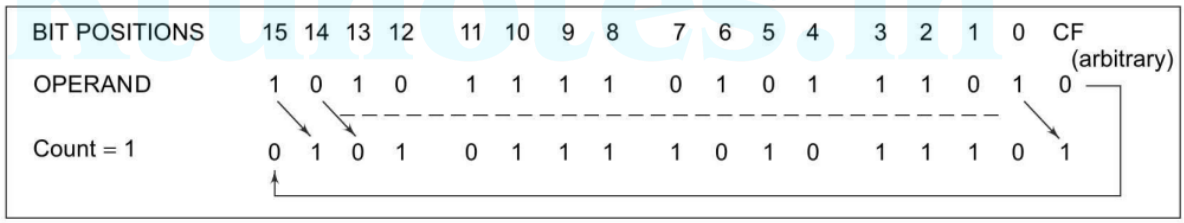
Eg:
MOV AX,9DB5H
RCL AX,01



2. **RCR - Rotate Right through carry**

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

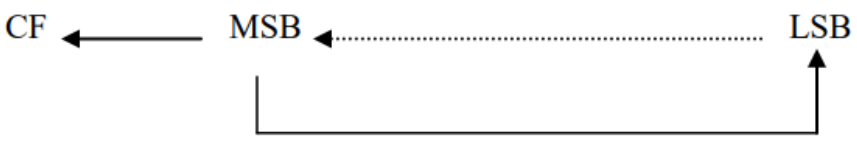
Eg:
MOV AX,AF5DH
RCR AX,01



3. **ROL - Rotate Left**

This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF.

Eg:
ROL AX, 1 Rotate the word in AX by 1 bit position left, MSB to LSB and CF



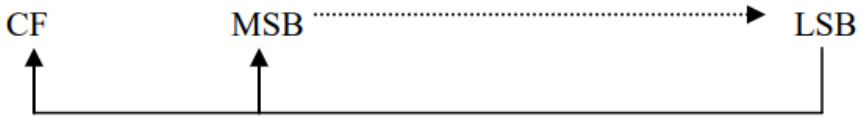
4. **ROR - Rotate Right**

This instruction rotates all the bits in a specified word or byte some number of bit positions to right. The operation is desired as a rotate rather than shift, because the bit

moved out of the LSB is rotated around into the MSB. The data bit moved out of the LSB is also copied into CF.

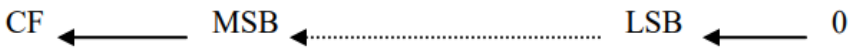
Eg:

ROR BL, 1 Rotate all bits in BL right by 1 bit position; LSB to MSB and to CF



5. **SAL - Shift Arithmetic Left**
6. **SHL - Shift Logical Left**

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 a multi-bit shift, CF will contain the bit most recently shifted out from the MSB. Bits shifted into CF previously will be lost.

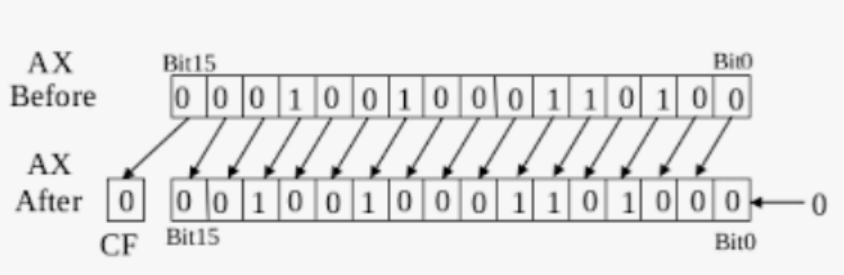


Eg:

SAL BX, 1 Shift word in BX 1 bit position left, 0 in LSB

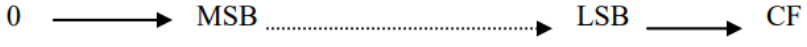
MOV CL, 02h Load desired number of shifts in CL

SAL BX, CL Shift word in BX left CL bit positions, 0 in LSBs



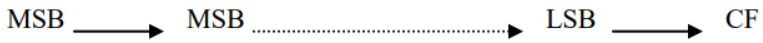
7. **SHR - Shift Arithmetic Right**

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.



8. **SHR - Shift Logical Right**

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.



5) String manipulation instructions:

1. **MOVS/MOVSB/MOVSW - Move string byte/word**

   This instruction copies a byte or a word from a 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 Page 19 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 be 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.

   Eg:

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/word**

   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.

   Eg:

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

3. **STOS / STOSB / STOSW - Store string byte/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 the next or previous element of the string. If DF is cleared, then DI will automatically be 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.

4. **CMPS /CMPSB / CMPSW - Compare string byte/word**

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.

Eg:

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

6) Control transfer or branching instructions:

i. Unconditional branch instructions

1. CALL - The CALL instruction is used to transfer execution to a subprogram or a procedure. 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 the CALL instruction.
- A far call is a call to a procedure, which is in a different segment from the one that contains the CALL instruction.

2. RET - The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure.

3. INT N - Type N interrupt

The term type in the instruction format refers to a number between 0 and 255, which identify the interrupt.

4. INTO - Interrupt on Overflow

If the overflow flag (OF) is set, this instruction causes the 8086 to do an indirect far call to a procedure where you write code to handle the overflow condition.

5. JMP - Unconditional jump to specified location

This instruction will fetch the next instruction from the location specified in the instruction. If the destination is in the same code segment as the JMP instruction, 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.

6. IRET - Return from ISR

The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program. To do this return, the 8086 copies the saved value of IP from the stack to IP, the stored value of CS from the stack to CS, and the stored value of the flags back to the flag register.

7. LOOP - Loop uncondionally

This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP.

ii. Conditional branch instructions

	Mnemonic	*Displacement*	*Operation*
1.	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1.
2.	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0.
3.	JS	Label	Transfer execution control to address 'Label', if SF=1.
4.	JNS	Label	Transfer execution control to address 'Label', if SF=0.
5.	JO	Label	Transfer execution control to address 'Label', if OF=1.
6.	JNO	Label	Transfer execution control to address 'Label', if OF=0.
7.	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1.
8.	JNP	Label	Transfer execution control to address 'Label', if PF=0.
9.	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1.
10.	JNB/JAE/JNC	Label	Transfer execution control to address 'Label', if CF=0.
11.	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1.
12.	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0.
13.	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1.
14.	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0.
15.	JLE/JNC	Label	Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or at least any one of SF and OF is 1(Both SF and OF are not 0).

1. JE / JZ - JUMP IF EQUAL / JUMP IF ZERO
2. JNE / JNZ - JUMP NOT EQUAL / JUMP IF NOT ZERO
3. JS - JUMP IF SIGNED / JUMP IF NEGATIVE

4. JNS - JUMP IF NOT SIGNED / JUMP IF POSITIVE
5. JO - JUMP IF OVERFLOW
6. JNO - JUMP IF NO OVERFLOW
7. JP / JPE - JUMP IF PARITY / JUMP IF PARITY EVEN
8. JNP / JPO - JUMP IF NO PARITY / JUMP IF PARITY ODD
9. JB / JC / JNAE - JUMP IF BELOW / JUMP IF CARRY / JUMP IF NOT ABOVE OR EQUAL
10. JBE / JNA - JUMP IF BELOW OR EQUAL / JUMP IF NOT ABOVE
11. JA / JNBE - JUMP IF ABOVE / JUMP IF NOT BELOW OR EQUAL
12. JL / JNGE - JUMP IF LESS THAN / JUMP IF NOT GREATER THAN OR EQUAL
13. JGE / JNL - JUMP IF GREATER THAN OR EQUAL / JUMP IF NOT LESS THAN
14. JLE / JNG - JUMP IF LESS THAN OR EQUAL / JUMP IF NOT GREATER
15. JG / JNLE - JUMP IF GREATER / JUMP IF NOT LESS THAN OR EQUAL

Conditional Loop Instructions

Mnemonic	*Displacement*	*Operation*
LOOPZ/LOOPE (Loop while ZF = 1; equal)	Label	Loop through a sequence of instructions from 'Label' while ZF=1 and CX= 0.
LOOPNZ/LOOPENE (Loop while ZF = 0; not equal)	Label	Loop through a sequence of instructions from 'Label' while ZF=0 and CX≠ 0.

7) Flag manipulation & process control instructions:

Flag Manipulation Instructions

CLC	-	Clear carry flag
CMC	-	Complement carry flag
STC	-	Set carry flag
CLD	-	Clear direction flag
STD	-	Set direction flag
CLI	-	Clear interrupt flag
STI	-	Set interrupt flag

Machine Control Instructions

WAIT	-	Wait for Test input pin to go low
HLT	-	Halt the processor
NOP	-	No operation
ESC	-	Escape to external device like NDP (numeric co-processor)
LOCK	-	Bus lock instruction prefix.

ASSEMBLER DIRECTIVES & OPERATORS OF 8086

DIRECTIVES direct the assembler to correctly interpret the program to code it appropriately.
OPERATORS perform the arithmetic and logical tasks.

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment.
For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to "bracket" a logical segment containing code of data.

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.

ASSUME

The ASSUME directive is used tell the assembler the name of the logical segment it should use for a specified segment.

- The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE.
- The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

PRICES DB 49H, 98H, 29H	Declare array of 3 bytes named PRICES and initialize them with specified values.

DD (DEFINE DOUBLE WORD)

The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word.

DQ (DEFINE QUADWORD)

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory.

DT (DEFINE TEN BYTES)

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory.

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.

WORDS DW 1234H, 3456H	Declare an array of 2 words named WORDS and initialize them with the specified values.

EQU (EQUATE)

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name.

```
CONTROL EQU 30H
MOV AL, CONTROL
```


LENGTH

LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array.

MOV CX, LENGTH STRING1	Determine the number of elements in STRING1 and load it into CX.

OFFSET

OFFSET is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it.

MOV BX, OFFSET PRICES	Determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and will load this value into BX.

PTR (POINTER)

The PTR operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear.

INC BYTE PTR [BX]	Tells the assembler that we want to increment the byte pointed to by BX.

EVEN (ALIGN ON EVEN MEMORY ADDRESS)

The EVEN directive tells the assembler to increment the location counter to the next even address, if it is not already at an even address. A NOP instruction will be inserted in the location incremented over.

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.

DIVIDE PROC FAR	Identifies the start of a procedure named DIVIDE and tells the assembler that the procedure is far (in a segment with different from the one that contains the instructions which calls the 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.

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

ORG (ORIGIN)

The ORG directive allows you to set the location counter to a desired value at any point in the program.

ORG 2000H	Tells the assembler to set the location counter to 2000H

NAME

The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written.

LABEL

The LABEL directive is used to give a name to the current value in the location counter. It assigns alternate name and type to a memory location.

ENTRY_POINT LABEL FAR	Label with name ENTRY_POINT of type FAR when used in JUMP or CALL
DVAL LABEL WORD	Label with name DVAL of type WORD when used For data reference

EXTRN

The EXTRN directive is used to tell the assembler that the name or labels following the directive are in some other assembly module.

EXTRN DIVIDE: FAR	Tells the assembler that DIVIDE is a label of type FAR in another assembler module.

PUBLIC

Large programs are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared PUBLIC in the module in which it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules.

PUBLIC DIVISOR, DIVIDEND	makes the two variables DIVISOR and DIVIDEND available to other assembly modules

SHORT

The SHORT operator is used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction in the program. The destination must in the range of –128 bytes to +127 bytes from the address of the instruction after the jump.

JMP SHORT NEARBY_LABEL

TYPE

The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually determines the number of bytes in the type of the variable. For a byte-type variable, the assembler will give a value of 1, for a word-type variable, the assembler will give a value of 2, and for a double word-type variable, it will give a value of 4.

GLOBAL (DECLARE SYMBOLS AS PUBLIC OR EXTRN)

The GLOBAL directive can be used in place of a PUBLIC directive or in place of an EXTRN directive.

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.

'+ & -' Operators represent addition and subtraction respectively and are typically used to add or subtract displacements to base or index registers or stack or base pointers.

Eg:

```
MOV AL, [ SI +2 ]
MOV DX, [ BX – 5 ]
MOV BX, [ OFFSET LABEL + 10 H ]
MOV AX, [ BX + 9I ]
```