8086 INSTRUCTION SET

CLASSIFICATION OF INSTRUCTION SET OF 8086

1) **DATA TRANSFER INSTRUCTIONS**
   E.g.:: MOV, PUSH, POP
2) **ARITHMETIC INSTRUCTIONS**
   E.g.:: ADD, SUB, MUL
3) **LOGIC INSTRUCTIONS (BIT MANIPULATION INSTRUCTIONS)**
   E.g.:: AND, OR, XOR
4) **SHIFT INSTRUCTIONS & ROTATE INSTRUCTIONS**
   E.g.:: ROL, RCL, ROR, SHL
5) **PROGRAM EXECUTION AND TRANSFER INSTRUCTIONS (BRANCH INSTRUCTIONS)**
   E.g.:: JMP, CALL, JC
6) **ITERATION CONTROL INSTRUCTIONS (LOOP INSTRUCTIONS)**
   E.g.:: LOOP, LOOPZ, LOOPNE
7) **PROCESSOR CONTROL INSTRUCTIONS (INSTRUCTIONS OPERATING ON FLAGS)**
   E.g.:: STC, CLC, CMC
8) **EXTERNAL HARDWARE SYNCHRONIZATION INSTRUCTIONS**
   E.g.:: LOCK, ESC, WAIT
9) **INTERRUPT CONTROL INSTRUCTIONS**
   E.g.:: INT n, IRET, INTO (Interrupt on overflow)
10) **STRING INSTRUCTIONS**
    E.g.:: MOVSB, LODSB, STOSB

Data Transfer Instructions

1) **MOV Destination, Source**
Moves a byte/word **from** the **source to** the **destination** specified in the instruction.
Source: *Register, Memory Location, Immediate Number*
Destination: Register, Memory Location
Both, source and destination cannot be memory locations.

Eg: **MOV CX, 0037H**	; *CX ← 0037H*
MOV BL, [4000H]	; *BL ← DS:[4000H]*
MOV AX, BX	; *AX ← BX*
MOV DL, [BX]	; *DL ← DS:[BX]*
MOV DS, BX	; *DS ← BX*

2) **PUSH Source**
Push the **source** (word) **into** the **stack** and decrement the stack pointer by two.
The source MUST be a **WORD** (**16 bits**).
Source: *Register, Memory Location*

Eg: **PUSH CX**	; *SS:[SP-1] ← CH, SS:[SP-2] ← CL*
	; SP ← SP – 2
PUSH DS	; *SS:[SP-1, SP-2] ← DS*
	; SP ← SP – 2

3) **POP Destination**
POP a **word from** the **stack** into the given destination and increment the Stack Pointer by 2. The destination MUST be a **WORD** (16 bits).
Destination: *Register [EXCEPT CS], Memory Location*

Eg: **POP CX**	; *CH ← SS:[SP], CL ← SS:[SP+1]*
	; SP ← SP + 2
POP DS	; *DS ← SS:[SP, SP+1]*
	; SP ← SP + 2

Please Note: **MOV, PUSH, POP** are the ONLY instructions that use the Segment Registers as operands {except CS}.

4) **PUSHF**
Push value of **Flag Register into stack** and decrement the stack pointer by 2.
Eg: **PUSHF** ; *SS:[SP-1] ← Flag$_H$, SS:[SP-2] ← Flag$_L$, SP ← SP – 2*

5) **POPF**
POP a **word from** the **stack into** the **Flag register**.
Eg: **POPF** ; *Flag$_L$ ← SS:[SP], Flag$_H$ ← SS:[SP+1], SP ← SP + 2*

6) **XCHG Destination, Source**
Exchanges a byte/word between the **source and** the **destination** specified in the instruction.
Source: *Register, Memory Location*
Destination: Register, Memory Location
Even here, both operands cannot be memory locations.

Eg: **XCHG CX, BX**	; *CX ←→ BX*
XCHG BL, CH	; *BL ←→ CH*

7) **XLATB / XLAT** (very important)
Move into AL, the **contents of** the **memory location** in Data Segment, **whose** effective **address** is **formed by** the **sum of BX and AL**.

```
Eg: XLAT                    ; AL ← DS:[BX + AL]
                            ; i.e. if DS = 1000H; BX = 0200H; AL = 03H
                            ; ∴ 10000    … DS × 16
                            ; +   0200   … BX
                            ; +     03   … AL
                            ; =10203 ∴ AL ← [10203H]
```

Note: the difference between XLAT and XLATB

In XLATB there is no operand in the instruction.
E.g.:: XLATB
It works in an implied mode and does exactly what is shown above.

In XLAT, we can specify the name of the look up table in the instruction
E.g.:: XLAT SevenSeg
This will do the translation form the look up table called SevenSeg.
In any case, the base address of the look up table must be given by BX.

8) LAHF
Loads AH with **lower byte** of the **Flag** Register.

9) SAHF
Stores the contents of **AH into** the **lower byte** of the **Flag** Register.

10)LEA register, source
Loads Effective Address (offset address) **of** the **source into** the **given register**.
Eg: **LEA BX, Total** ; *BX ← offset address of Total in Data Segment.*

11)LDS destination register, source
Loads the **destination register and DS** register **with offset** address **and segment address** specified by the **source**.
Eg: **LDS BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *DS ← {DS: [Total + 2], DS:[Total + 3]}*

12)LES destination register, source
Loads the **destination register and ES** register with the **offset address and** the **segment address** indirectly specified by the **source**.
Eg: **LES BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *ES ← {DS: [Total + 2], DS:[Total + 3]}*

I/O ADDRESSING MODES OF 8086 (5m – Important Question)

I/O addresses in 8086 can be either 8–bit or 16-bit

Direct Addressing Mode:

If we use **8-bit I/O address** we get a **range of 00H… FFH**.
This gives a total of **256 I/O ports.**
Here we use Direct addressing Mode, that is, the **I/O address is specified in the instruction.**

```
E.g.:: IN AL, 80H                  ; AL gets data from I/O port address 80H.
```

This is also called **Fixed Port Addressing**.

Indirect Addressing Mode:

If we use **16-bit I/O address** we get a **range of 0000H… FFFFH**.
This gives a total of **65536 I/O ports.**
Here we use Indirect addressing Mode, that is, the **I/O address is specified by DX register.**

```
E.g.::  MOV DX, 2000H
        IN AL, DX                  ; AL gets data from I/O port address 2000H given by DX.
```

This is also called **Variable Port Addressing**.

13) **IN destination register, source port**
Loads the **destination register with** the contents of the **I/O port** specified by the source.

Source: It is an I/O port address.
If the address is 8-bit it will be given in the instruction by **Direct addressing mode.**
If it is a 16 bit address it will be given by DX register using **Indirect addressing mode.**

Destination: It has to be some form of "A" register, in which we will get data from the I/O device.
If we are getting 8-bit data, it will be AL or AH register.
If we are getting 16-bit data, it will be AX register.

```
Eg: IN AL, 80H            ; AL gets 8-bit data from I/O port address 80H
    IN AX, 80H            ; AX gets 16-bit data from I/O port address 80H
    IN AL, DX             ; AL gets 8-bit data from I/O port address given by DX.
    IN AX, DX             ; AX gets 16-bit data from I/O port address given by DX.
```

14) **OUT destination port, source register**
Loads the destination I/O port with the contents of the source register.

```
Eg: OUT 80H, AL           ; I/O port 80H gets 8-bit data from AL
    OUT 80H, AX           ; I/O port 80H gets 16-bit data from AX
    OUT DX, AL            ; I/O port whose address is given by DX gets 8-bit data from AL
    OUT DX, AX            ; I/O port whose address is given by DX gets 16-bit data from AX
```

Segment Overriding

In every instruction, a particular segment register is accessed for the base address.
Eg: **MOV CL, [5000H]** *; CL← DS:[5000H] as Data Seg is accessed by default*

However, we can also **override the segment** as follows:
Eg: **MOV CL, CS:[5000H]** *; Here CL ← CS:[5000H], this is **Segment Overriding**.*

By default, the address 5000H would have been an offset for the data segment, BUT here we **override** it with the Code segment as shown above.

Another example:

```
MOV BL, [BP]          ; BL ← SS:[BP] … Normal
MOV BL, DS:[BP]       ; BL ← DS:[BP] … Overriding
```

Arithmetic Instructions

1) **ADD/ADC destination, source**
Adds the source to the destination and stores the **result** back **in** the **destination**.
Source: Register, Memory Location, Immediate Number
Destination: Register
Both, source and destination have to be of the same size.
ADC also adds the carry into the result.

Eg: **ADD AL, 25H**	; *AL ← AL + 25H*
ADD BL, CL	; *BL ← BL + CL*
ADD BX, CX	; *BX ← BX + CX*
ADC BX, CX	; *BX ← BX + CX + Carry Flag*

2) **SUB/SBB destination, source**
It is similar to ADD/ADC except that it does subtraction.

3) **INC destination**
Adds "1" to the specified **destination**.
Destination: Register, Memory Location
Note: Carry Flag is NOT affected.

Eg: **INC AX**	; *AX ← AX + 1*
INC BL	; *BL ← BL + 1*
INC BYTE PTR [BX]	; *Increment the **byte** pointed by BX in the Data Segment*
	; *i.e. DS:[BX] ← DS;[BX] + 1*
INC WORD PTR [BX]	; *Increment **word** pointed by BX in the Data Segment*
	; *{DS:[BX], DS:[BX+1]}← {DS:[BX], DS:[BX+1]}+1*

4) **DEC destination**
It is similar to INC. Here also Carry Flag is NOT affected.

5) **MUL source**(**unsigned** 8/16-bit register)
If the **source** is **8-bit**, it is **multiplied with AL** and the **result** is stored in **AX** (AH–higher byte, AL–lower byte)
If the **source** is **16-bit**, it is **multiplied** with **AX** and the **result** is stored in **DX-AX** (DX–higher byte, AX–lower byte)
Source: Register, Memory Location
MUL affects AF, PF, SF and ZF.

Eg:**MUL BL**	; *AX ← AL × BL*
MUL BX	; *DX-AX ← AX × BX*
MUL BYTE PTR [BX]	; *AX ← AL × DS:[BX]*

6) **IMUL source**(**signed** 8/16-bit register)
Same as MUL except that the source is a SIGNED number.

7) **DIV source**(**unsigned** 8/16-bit register – divisor)
This instruction is used for **UNSIGNED** division.
Divides a **WORD by a BYTE**, OR a **DOUBLE WORD by a WORD.**
If the **divisor is 8-bit** then the **dividend is in AX** register.
After division, the **quotient is in AL** and the **Remainder in AH**.
If the **divisor is 16-bit** then the **dividend is in DX-AX** registers.
After division, the **quotient is in AX** and the **Remainder in DX**.

Source: Register, Memory Location
ALL flags are **undefined** after DIV instruction.

Eg: **DIV BL** ; *AX ÷ BL :- AL ← Quotient; AH ← Remainder*
DIV BX ; *{DX,AX} ÷ BX :- AX ← Quotient; DX ← Remainder*

Please Note: If the divisor is 0 or the result is too large to fit in AL (or AX for 16-bit divisor), them 8086 does a Type 0 interrupt (Divide Error).

8) **IDIV source**(**signed** 8/16-bit register – divisor)
Same as DIV except that it is used for **SIGNED** division.

9) **NEG destination**
This instruction forms the **2's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location
ALL condition flags are **updated**.
Eg: **Assume** AL= 0011 0101 = 35 H then
NEG AL ;*AL ← 1100 1011 = CBH. i.e. AL ← 2's Complement (AL)*

10) **CMP destination, source**
This instruction **compares the source with the destination**.
The source and the destination must be of the same size.
Comparison is **done by internally SUBTRACTING** the **SOURCE form DESTINATION**.
The result of this subtraction is NOT stored anywhere, instead the Flag bits are affected.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
ALL condition flags are **updated**.
Eg: **CMP BL, 55H** ; *BL compared with 55H i.e. BL – 55H.*
CMP CX, BX ; *CX compared with BX i.e. CX – BX.*

11) **CBW [Convert signed BYTE to signed WORD]**
This instruction **copies sign of** the byte in **AL into** all the bits of **AH**.
AH is then called *sign extension of AL.*
No Flags affected.

<u>Eg: Assume</u>
AX = XXXX XXXX **1**001 0001
Then **CBW** gives
AX = **1111 1111 1**001 0001

12) **CWD [Convert signed WORD to signed DOUBLE WORD]**
This instruction **copies sign of** the **WORD** in **AX into** all the bits of **DX**.
DX is then called *sign extension of AX.*
No Flags affected.

<u>Eg: Assume</u>
AX = **1**000 0000 1001 0001
DX = XXXX XXXX XXXX XXXX
Then **CWD** gives
AX = **1**000 0000 1001 0001
DX = **1111 1111 1111 1111**

Note: Both CBW and CWD are used for Signed Numbers.

Decimal Adjust Instructions

13)DAA [Decimal Adjust for Addition]

It makes the **result** in **packed BCD** form **after** BCD **addition** is performed.
It works **ONLY** on **AL** register.
All Flags are updated; **OF** becomes **undefined** after this instruction.
For AL register ONLY
If $D_3 – D_0 > 9$ OR Auxiliary Carry Flag is set => ADD 06H to AL.
If $D_7 – D_4 > 9$ OR Carry Flag is set => ADD 60H to AL.

Assume AL = 14H
CL = 28H
Then **ADD AL, CL** gives
AL = 3CH
Now **DAA** gives
AL = 42 (06 is added to AL as C > 9)
If you notice, $(14)_{10} + (28)_{10} = (42)_{10}$

14)DAS [Decimal Adjust for Subtraction]

It makes the **result** in **packed BCD** form **after** BCD **subtraction** is performed.
It works **ONLY** on **AL** register.
All Flags are updated; **OF** becomes **undefined** after this instruction.
For AL register ONLY
If $D_3 – D_0 > 9$ OR Auxiliary Carry Flag is set => Subtract 06H from AL.
If $D_7 – D_4 > 9$ OR Carry Flag is set => Subtract 60H from AL.

Assume AL = 86H
CL = 57H
Then **SUB AL, CL** gives
AL = 2FH
Now **DAS** gives
AL = 29 (06 is subtracted from AL as F > 9)
If you notice, $(86)_{10} - (57)_{10} = (29)_{10}$

ASCII Adjust Instructions (for the AX register ONLY)

15)AAA [ASCII Adjust for Addition]

It makes the **result** in **unpacked BCD form**.
In **ASCII** Codes, **0 … 9** are represented as **30 … 39**.
When we **add ASCII Codes**, we need to **mask** the **higher byte** (Eg: 3 of 39).
This can be **avoided** if we **use AAA** instruction **after the addition** is performed.
AAA updates the **AF** and the **CF**; But **OF, PF, SF, ZF** are **undefined** after the instruction.
Eg: Assume
AL = 0011 0100 … ASCII 4.
CL = 0011 1000 … ASCII 8.
Then **ADD AL, CL** gives
AL = 01101100
i.e. AL = 6CH … it is the Incorrect temporary Result

Now **AAA** gives

AL = 0000 0010 … Unpacked BCD for 2.
Carry = 1 … this indicates that the answer is 12.

16)AAS [ASCII Adjust for Subtraction]

It makes the **result** in **unpacked BCD form**.
In **ASCII** Codes, **0 … 9** are represented as **30 … 39**.
When we **subtract ASCII Codes**, we need to **mask** the **higher byte** (Eg: 3 of 39).
This can be **avoided** if we **use AAS** instruction **after the subtraction** is performed.
AAS updates the **AF** and the **CF**; But **OF, PF, SF, ZF** are **undefined** after the instruction.

<u>Eg: Assume</u>

AL = 0011 1001 … ASCII 9.
CL = 0011 0101 … ASCII 5.

Then **SUB AL, CL** gives

AL = 0000 0100

i.e. AL = 04H

Now **AAS** gives

AL = 0000 0100 … Unpacked BCD for 4.
Carry = 0 … this indicates that the answer is 04.

17)AAM [BCD Adjust After Multiplication]

Before we multiply two ASCII digits, we mask their upper 4 bits.
Thus we have two unpacked BCD operands.
After the two unpacked BCD operands are multiplied, the AAM instruction converts this result into unpacked BCD form in the AX register.
AAS updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.

<u>Eg: Assume</u>

AL = 0000 1001 … unpacked BCD 9.
CL = 0000 0101 … unpacked BCD 5.

Then **MUL CL** gives

AX = 0000 0000 0010 1101 = 002DH.

Now **AAM** gives

AX = 0000 0100 0000 0101 = 0405H.
This is 45 in the unpacked BCD form.

18)AAD [Binary Adjust before Division]

This instruction converts the unpacked BCD digits in AH and Al into a Packed BCD in AL.
AAD updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.

<u>Eg: Assume</u>

CL = 07H.
AH = 04.
AL = 03.
$\therefore$ AX = 0403H … unpacked BCD for $(43)_{10}$

Then **AAD** gives

AX = 002BH … i.e. $(43)_{10}$

Now **DIV CL** gives (divide AX by unpacked BCD in CL)

AL = Quotient = 06 … unpacked BCD
AH = Remainder = 01 … unpacked BCD

LOGICAL INSTRUCTIONS [BIT MANIPULATION INSTRUCTIONS]

1) **NOT destination**
This instruction forms the **1's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location. **No Flags affected**.
Eg: **Assume** AL= 0011 0101
NOT AL ; *AL ← 1100 1010 … i.e. AL = 1's Complement (AL)*

2) **AND destination, source**
This instruction **logically ANDs** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **AND BL, CL** ; *BL ← BL AND CL*

3) **OR destination, source**
This instruction **logically Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **OR BL, CL** ; *BL ← BL OR CL*

4) **XOR destination, source**
This instruction **logically X-Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **XOR BL, CL** ; *BL ← BL XOR CL*

5) **TEST destination, source**
This instruction **logically ANDs** the source with the destination **BUT** the **RESULT** is **NOT STORED ANYWHERE**. **ONLY** the **FLAG** bits are **AFFECTED**.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **TEST BL, CL** ; *BL AND CL; result not stored; Flags affected.*
*Note: Don't forget this instruction because it will be used later in **multiprocessor systems**!*

SHIFT INSTRUCTIONS

1) **SAL/SHL destination, count**
LEFT-Shifts the **bits of destination**.
MSB shifted **into** the **CARRY**.
LSB gets a **0**.

Bits are shifted 'count' number of times.
If count = 1, it is directly specified in the instruction.
If count > 1, it has to be given using CL Register.

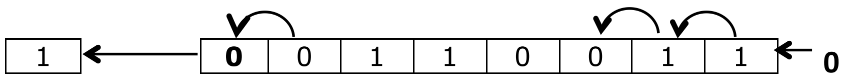
Destination: Register, Memory Location.

Eg: **SAL BL, 1** *; Left-Shift BL bits, once.*

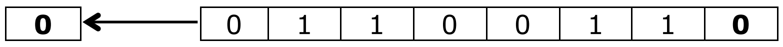
Assume:

Before Operation: BL = 0011 0011 and CF = 1

Carry **Destination**



After Operation: BL = 0110 0110 and CF = 0



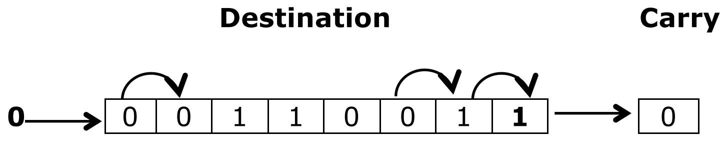
More examples:
MOV CL, 05H *; Load number of shifts in CL register.*
SAL BL, CL *; Left-Shift BL bits CL (5) number of times.*

2) **SHR destination, count**
RIGHT-Shifts the bits of destination.
MSB gets a **0** (∴ Sign is lost).
LSB shifted **into** the **CARRY**.

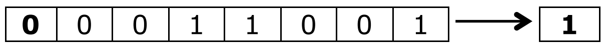
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1, it has to be given using CL register.
Eg: **SHR BL, 1** *; Right-Shift BL bits, once.*

Assume:

Before Operation: BL = 0011 0011 and CF = 0



After Operation: BL = 00011 1001 and CF = 1

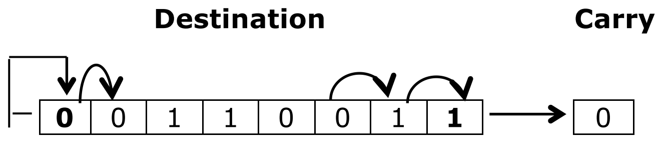


3) **SAR destination, count**
RIGHT-Shifts the bits of destination.
MSB placed **in MSB itself** (∴ Sign is preserved).
LSB shifted **into** the **CARRY**.

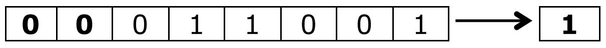
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1 it has to be given using CL register.
Destination: Register, Memory Location

Eg: **SAR BL, 1** *; Right-Shift BL bits, once.*

Assume:
Before Operation: BL = 0011 0011 and CF = 0



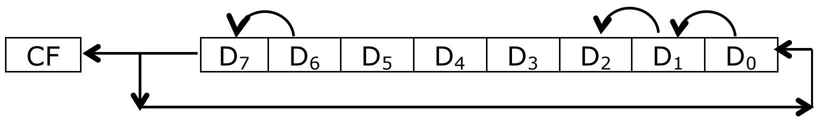
After Operation: BL = 0001 1001 and CF = 1



ROTATE INSTRUCTIONS

1) **ROL destination, count**
LEFT-Shifts the bits of destination.
MSB shifted **into** the **CARRY**.
MSB also goes **to LSB**.
Bits are shifted 'count' number of times.
  If count = 1, it is directly specified in the instruction.
  If count > 1, it has to be loaded in the CL register, and CL gives the count in the instruction.
Destination: Register, Memory Location
Eg: **ROL BL, 1** *; Left-Shift BL bits once.*

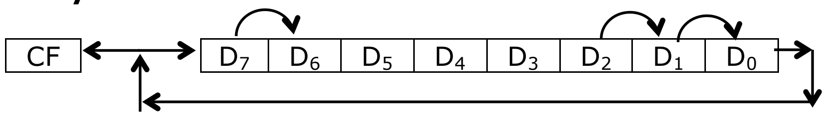
Carry **Destination**



More examples:
```
MOV CL, 05H        ; Load number of shifts in CL register.
ROL BL, CL         ; Left-Shift BL bits CL (5) number of times.
```

2) **ROR destination, count**
RIGHT-Shifts the bits of destination.
LSB shifted **into** the **CARRY**.
LSB also goes **to MSB**.
Bits are shifted 'count' number of times.
  If count = 1, it is directly specified in the instruction.
  If count > 1, it has to be loaded in the CL register, and CL gives the count in the instruction.
Eg:
  ROR BL, 1 *; Right-Shift BL bits once.*

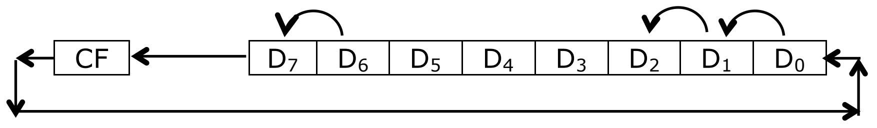
Carry **Destination**



3) RCL destination, count
LEFT-Shifts the bits of destination.
MSB shifted **into** the Carry Flag **(CF)**.
CF goes **to LSB**.
Bits are shifted 'count' number of times.
If count = 1, it is directly specified in the instruction.
If count > 1, it has to be loaded in the CL register, and CL is specified as the count in the instruction.
Destination: Register, Memory Location

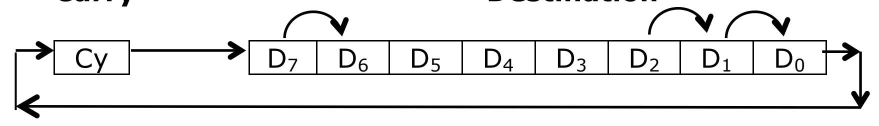
Eg: **RCL BL, 1** *; Left-Shift BL bits once.*

Carry **Destination**



4) RCR destination, count
RIGHT-Shifts the bits of destination.
LSB shifted **into** the **CF**.
CF goes **to MSB**.
Bits are shifted 'count' number of times.
If count = 1, it is directly specified in the instruction.
If count > 1, it has to be loaded in the CL register, and CL is specified as the count in the instruction.
Destination: Register, Memory Location
Eg:
RCR BL, 1 *; Right-Shift BL bits once.*

Carry **Destination**



More examples:
MOV CL, 05H *; Load number of shifts in CL register.*
RCR BL, CL *; Right-Shift BL bits CL (5) number of times.*

PROGRAM EXECUTION AND TRANSFER INSTRUCTIONS

These instructions cause a branch in the program sequence.
There are 2 main types of branching:

i. Near branch
ii. Far Branch

i. **Near Branch**
This is an **Intra-Segment Branch** i.e. the branch is to a new location within the current segment only.
Thus, **only** the value of **IP needs to be changed**.
If the Near Branch is in the **range of –128 to 127**, then it is called as a **Short Branch**.

ii. **Far Branch**
This is an **Inter-Segment Branch** i.e. the branch is to a new location in a different segment.
Thus, the values of **CS and IP need to be changed**.

JMP (Unconditional Jump)

INTRA-Segment (NEAR) JUMP

The Jump address is specified in two ways:

1) INTRA-Segment Direct Jump
The new Branch location is specified directly in the instruction
The new address is calculated by **adding** the 8 or16-bit **displacement** to the IP.
The CS does not change.
A +ve displacement means that the Jump is ahead (forward) in the program.
A -ve displacement means that the Jump is behind (backward) in the program.
It is also called as *Relative Jump*.
Eg: **JMP Prev** ; *IP ← offset address of "Prev".*
JMP Next ; *IP ← offset address of "Next".*

2) INTRA-Segment Indirect Jump
The New Branch address is specified indirectly through a **register** or a **memory location**.
The value in the IP is **replaced** with the new value.
The CS does not change.
Eg: **JMP WORD PTR [BX]** ; *IP ← {DS:[BX], DS: [BX+1]}*

INTER-Segment (FAR) JUMP

The Jump address is specified in two ways:

3) INTER-Segment Direct Jump
The new Branch location is **specified directly** in the instruction
Both **CS and IP get new values**, as this is an inter-segment jump.
Eg: **Assume NextSeg** is a label pointing to an instruction in a **different segment.**
JMP NextSeg ; *CS and IP get the value from the label NextSeg.*

4) INTER-Segment Indirect Jump
The new Branch location is **specified indirectly** through a **register** or a **memory location.**
Both **CS and IP get new values**, as this is an inter-segment jump.
Eg:**JMP DWORD PTR [BX]** ; *IP ← {DS:[BX], DS: [BX+1]},*
; *CS ← {DS:[BX+2], DS:[BX+3]}*

JCondition (Conditional Jump)

This is a conditional branch instruction.
If condition is **TRUE**, then it is **similar to** an **INTRA-Segment Direct Jump.**
If condition is FALSE, then branch does not take place and the next sequential instruction is executed**.**
The destination must be in the range of –128 to 127 from the address of the instruction (i.e. **ONLY SHORT Jump**).
Eg: **JNC Next** ; *Jump to Next If Carry Flag is not set (CF = 0).*
The various conditional jump instructions are as follows:

Mnemonic	Description	Jump Condition
Common Operations		
JC	Carry	**CF = 1**
JNC	Not Carry	CF = 0
JE/JZ	Equal or Zero	**ZF = 1**
JNE/JNZ	Not Equal or Not Zero	ZF = 0
JP/JPE	Parity or Parity Even	**PF = 1**
JNP/JPO	Not Parity or Parity Odd	PF = 0
Signed Operations		
JO	Overflow	**OF = 1**
JNO	Not Overflow	OF = 0
JS	Sign	**SF = 1**
JNS	Not Sign	SF = 0
JL/JNGE	Less	**(SF Ex-Or OF) = 1**
JGE/JNL	Greater or Equal	(SF Ex-Or OF) = 0
JLE/JNG	Less or Equal	**((SF Ex-Or OF) + ZF) = 1**
JG/JNLE	Greater	((SF Ex-Or OF) + ZF) = 0
Unsigned Operations		
JB/JNAE	Below	**CF = 1**
JAE/JNB	Above or Equal	CF = 0
JBE/JNA	Below or Equal	**(CF Ex-Or ZF) = 1**
JA/JNBE	Above	(CF Ex-Or ZF) = 0

CALL (Unconditional CALL)

CALL is an instruction that transfers the program control to a sub-routine, with the intention of coming back to the main program.
Thus, in CALL 8086 **saves the address of the next instruction into the stack** before branching to the sub-routine.
At the end of the subroutine, control transfers back to the main program using the return address from the stack.
There are two types of CALL: Near CALL and Far CALL.

INTRA-Segment (NEAR) CALL

The **new subroutine** called must be **in the same segment** (hence intra-segment).
The CALL **address** can be **specified directly** in the instruction **OR indirectly** through Registers or Memory Locations.
The following sequence is executed for a NEAR CALL:

i. 8086 will **PUSH Current IP** into the Stack.
ii. **Decrement SP by 2**.
iii. **New value** loaded **into IP**.

iv. **Control transferred** to a subroutine within the same segment.

```
Eg:   CALL subAdd            ; {SS:[SP-1], SS:[SP-2]} ← IP, SP ← SP – 2,
                             ; IP ← New Offset Address of subAdd.
```

INTER-Segment (FAR) CALL

The **new subroutine** called is in **another segment** (hence inter-segment).
Here CS and IP both get new values.
The CALL address can be specified directly OR through Registers or Memory Locations.
The following sequence is executed for a Far CALL:

i. **PUSH CS** into the Stack.
ii. **Decrement SP** by 2.
iii. **PUSH IP** into the Stack.
iv. **Decrement SP** by 2.
v. **Load CS** with new segment address.
vi. **Load IP** with new offset address.
vii. **Control transferred** to a subroutine in the new segment.

```
Eg:   CALL subAdd     ; {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP – 2,
                      ; {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP – 2,
                      ; CS ← New Segment Address of subAdd,
                      ; IP ← New Offset Address of subAdd.
```

There is **NO PROVISION for Conditional CALL**.

RET --- Return instruction

RET instruction causes the control to return to the main program from the subroutine.

Intrasegment-RET

```
    Eg: RET           ; IP ← SS:[SP], SS:[SP+1]
                      ; SP ← SP + 2
        RET n         ; IP ← SS:[SP], SS:[SP+1]
                      ; SP ← SP + 2 + n
```

Intersegment-RET

```
    Eg: RET           ; IP ← SS:[SP], SS:[SP+1]
                      ; CS ← SS:[SP+2], SS:[SP+3]
                      ; SP ← SP + 4
        RET n         ; IP ← SS:[SP], SS:[SP+1]
                      ; CS ← SS:[SP+2], SS:[SP+3]
                      ; SP ← SP + 4 + n
```

Please Note: The programmer writes the intra-seg and Inter-seg RET instructions in the same way. It is the assembler, which distinguishes between the two and puts the right opcode.

Differentiate between

	JMP INSTRUCTION	**CALL INSTRUCTION**
1	JMP instruction is used to **jump to a new location** in the program and continue	Call instruction is used to **invoke a subroutine, execute it and then return** to the main program.
2	A jump simply **puts the branch address into IP**.	A call **first stores the return address into the stack** and then loads the branch address into IP.
3	In 8086 Jumps can be either **unconditional or conditional**.	In 8086, Calls are only **unconditional**.
4	Does **not use the stack**	**Uses the stack**
5	Does **not need a RET** instruction.	**Needs a RET** instruction to return back to main program.

Differentiate between

	PROCEDURE (FUNCTION)	**MACRO**
1	A procedure (Subroutine/ Function) is a set of instruction needed repeatedly by the program. It is **stored as a subroutine and invoked from several places by the main program.**	A Macro is similar to a procedure but is not invoked by the main program. Instead, the **Macro code is pasted into the main program wherever the macro name is written in the main program.**
2	A subroutine is **invoked by a CALL** instruction and control returns by a RET instruction.	A Macro is simply accessed by **writing its name**. The entire macro code is pasted at the location by the assembler.
3	**Reduces the size** of the program	**Increases the size** of the program
4	**Executes slower** as time is wasted to push and pop the return address in the stack.	**Executes faster** as return address is not needed to be stored into the stack, hence push and pop is not needed.
5	**Depends on the stack**	**Does not depend on the stack**

Type 1) **Iteration Control Instructions**

These instructions **cause** a series of **instructions to be executed repeatedly**.
The **number of iterations** is loaded **in CX** register.
CX is **decremented by 1**, after every iteration. Iterations occur **until CX = 0**.
The **maximum difference between** the **address** of the instruction and the address of the Jump **can be 127**.

1) LOOP Label
Jump to specified label if CX not equal to 0; and decrement CX.
Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
            .
            .
            .
        MOV BL, AL
        LOOP BACK       ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0.
```

2) LOOPE/LOOPZ Label (Loop on Equal / Loop on Zero)
Same as above except that looping occurs ONLY if Zero Flag is set (i.e. ZF = 1)
Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
            .
            .
            .
        MOV BL, AL
        LOOPZ BACK      ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0 and ZF = 1.
```

3) LOOPNE/LOOPNZ Label (Loop on NOT Equal / Loop on NO Zero)
Same as above except that looping occurs ONLY if Zero Flag is reset (i.e. ZF = 0)
Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
            .
            .
            .
        MOV BL, AL
        LOOPZ BACK      ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0 and ZF = 0.
```

Type 2) **Processor Control / Machine Control Instructions**

(these are instructions that directly operate on Flag Reg)

In the exam first explain the following instructions: **PUSHF, POPF, LAHF and SAHF**

For Carry Flag

1) STC

This instruction **sets** the **Carry Flag**. No Other Flags are affected.

2) CLC

This instruction **clears** the **Carry Flag**. No Other Flags are affected.

3) CMC

This instruction **complements** the **Carry Flag**. No Other Flags are affected.

For Direction Flag

4) STD

This instruction **sets** the **Direction Flag**. No Other Flags are affected.

5) CLD

This instruction **clears** the **Direction Flag**. No Other Flags are affected.

For Interrupt Enable Flag

6) STI

This instruction **sets** the **Interrupt Enable Flag**. No Other Flags are affected.

7) CLI

This instruction **clears** the **Interrupt Enable Flag**. No Other Flags are affected.

Note: There is no direct way to alter TF. It can be altered through program as follows:

To set TF:

```
PUSHF                  ; push contents of Flag register into the stack
POP BX                 ; pop contents of flag reg from the stack-top into BX
OR BH, 01H             ; set the bit corresponding to TF, in the BH register
PUSH BX                ; push the modified BX register into the stack
POPF                   ; pop the modified contents into flag register.
```

To reset TF:

```
PUSHF                  ; push contents of Flag register into the stack
POP BX                 ; pop contents of flag reg from the stack-top into BX
AND BH, FEH            ; reset the bit corresponding to TF, in the BH register
PUSH BX                ; push the modified BX register into the stack
POPF                   ; pop the modified contents into flag register.
```

Type 3)

External Hardware Synchronization Instructions

1) **ESC**

This is an 8086 **instruction-prefix** used to **indicate that the current instruction is for the 8087 NDP**.
We write a *homogeneous program* for the two processors 8086 and 8087.
Instructions are fetched by 8086 into its queue.
8087 duplicates the instruction queue of 8086 and monitors this queue.
When an instruction with **ESC prefix** (binary code 11011) is **encountered**, **8087 is activated**, and hence **it executes the instruction**.
8086 treats the instruction as NOP.
ESC has to be written before each 8087 instruction.

2) **WAIT**

This instruction is used to synchronize 8086 with the 8087 Co-Processor via the $\overline{\textbf{TEST}}$ input pin of 8086. Whenever 8087 is busy it puts a "1" on its BUSY o/p line connected to the $\overline{\textbf{TEST}}$ i/p of the μP.

The WAIT instruction makes the μP check the $\overline{\textbf{TEST}}$ pin.

If the μP checks the $\overline{\textbf{TEST}}$ pin and finds a "1" on it, 8086 understands that 8087 is busy and so it enters wait state. Here it does no processing.

It can **come out** of this idle state **in 2 ways**:

i. **$\overline{\textbf{TEST}}$ input is made low**
i.e. 8087 is no longer busy.
This takes 8086 completely out of the IDLE state.

ii. **Valid Interrupt on INTR or NMI**
In this case 8086 **exits wait state**, **executes the ISR** for the interrupt, and then **re-enters the WAIT state**. (*This is because the address of the WAIT instruction is what was pushed into the stack before executing the ISR.)*

Thus if **we write a WAIT instruction before every 8087 instruction**, we can ensure that 8087 is ready for executing its own instruction whenever it arrives.

WAIT can also be written before an 8086 instruction that requires the result of a previous 8087 operation.

3) **LOCK**
This is an 8086 **instruction prefix.**
It **prevents any external bus master from taking control of the system** bus during execution of the instruction, which has a **LOCK** prefix.
It causes 8086 to activate the $\overline{\textbf{LOCK}}$ signal so that no other bus master takes control of the system bus.

4) **NOP**
There is **no operation performed** while executing this instruction.
8086 requires 3 T-States for this instruction.
It is mainly used to insert time delays, and can also be used while debugging.

5) **HLT**
This instruction causes 8086 to **stop fetching any more instructions**.
8086 enters **Halt state**.
8086 can **come out** of this halt state only if there is a **valid hardware interrupt** (NMI or INTR) or **by reset**.

Type 4) **Interrupt Control Instructions**

1) **INT Type**
   This instruction causes an interrupt of the given type. The **'Type'** can be a number between **0 … 255**.
   The following action takes place:
   i. **PUSH Flag** Register onto the Stack. SP decremented by 2.
   ii. **IF** and **TF** are **cleared**. No other flags are affected.
   iii. **PUSH CS** onto the Stack. SP decremented by 2.
   iv. **PUSH IP** onto the Stack. SP decremented by 2. ∴ In all **SP decremented by 6**.
   v. **New value of IP** taken from location **type × 4**.
      Eg: INT 1 ; IP 🡨 {[00004] and [00005]} (as 1 × 4 = 00004H)
   vi. **New value of CS** taken from location **(type × 4) + 2**.
      Eg: INT 1 ; CS 🡨 {[00006] and [00007]}
      Execution of ISR begins from the address formed by new values of CS and IP.

2) **INTO (Interrupt on Overflow)**
   This instruction causes an interrupt of **type 4**, **ONLY if Overflow Flag (OF) is set**.
   The above sequence is followed and the control is transferred to the location pointed by 00010H.
   Eg: **INTO** ; *If OF = 1 then execute INT 4.*
   Please Note:- This is INTO (O for Overflow) and NOT INT 0 (i.e. Type 0 ==> Zero Divide Interrupt).

3) **IRET (Return from ISR)**
   This instruction causes the 8086 to return to the main program from an ISR.
   The following action takes place:
   i. **POP IP** from the Stack.
      SP incremented by 2.
   ii. **POP CS** from the Stack.
      SP incremented by 2.
   iii. **POP Flag Register** from the Stack.
      SP incremented by 2.
      ∴ In all **SP incremented by 6**.
   Execution of the **Main Program** continues from the address formed values of CS and IP restored from the stack.
   Please Note:- The original value of TF and IF are restored from the Stack. Also note that to come back from an ISR, the programmer must use the IRET instruction and not the normal RET instruction as the RET instruction will not POP back the Flag.

Type 5) String Instructions of 8086 *(Very Important ※ **10m**)*

A **String** is a **series of bytes** stored sequentially in the memory. String Instructions operate on such "Strings".
The **Source String is at a location pointed by SI in the Data Segment.**
The **Destination String is at a location pointed by DI in the Extra Segment.**
The Count for String operations is always given by CX.
Since CX is a 16-bit register we can transfer max 64 KB using a string instruction.
SI and/or DI are **incremented/decremented** after each operation depending upon the direction flag **"DF"** in the flag register.
If **DF = 0**, it is **auto increment**. This is done by **CLD instruction**.
If **DF = 1**, it is **auto decrement**. This is done by **STD instruction**.

1) MOVS: MOVSB/MOVSW (*Move String*)

It is used to **transfer** a word/byte **from data segment to extra segment**.
The offset of the source in data segment is in SI.
The offset of the destination in extra segment is in DI.
SI and DI are incremented / decremented depending upon the direction flag.

Eg:	**MOVSB**	; *ES:[DI] ← DS:[SI] … byte transfer*
		; *SI ← SI ± 1 … depending upon DF*
		; *DI ← DI ± 1 … depending upon DF*
	MOVSW	; *{ES:[DI], ES:[DI + 1]} ← {DS:[SI], DS:[SI + 1]}*
		; *SI ← SI ± 2*
		; *DI ← DI ± 2*

2) LODS: LODSB/LODSW (*Load String*)

It is used to **Load AL** (or AX) register with a byte (or word) **from data segment**.
The offset of the source in data segment is in SI.
SI is incremented / decremented depending upon the direction flag (DF).

Eg:	**LODSB**	; *AL ← DS:[SI] … byte transfer*
		; *SI ← SI ± 1 … depending upon DF*
	LODSW	; *AL ← DS:[SI]; AH ← DS:[SI + 1]*
		; *SI ← SI ± 2*

3)**STOS: STOSB/STOSW** (*Store String*)

It is used to **Store AL** (or AX) **into** a byte (or word) in the **extra segment**.
The offset of the source in extra segment is in DI.
DI is incremented / decremented depending upon the direction flag (DF).

```
Eg:     STOSB           ; ES:[DI] ← AL … byte transfer
                        ; DI ← DI ± 1 … depending upon DF

        STOSW           ; ES:[DI] ← AL; ES:[DI+1] ← AH … word transfer
                        ; DI ← DI ± 2 … depending upon DF
```

4)**CMPS: CPMSB/CMPSW** (*Compare String*)

It is used to **compare** a **byte** (or word) **in** the **data segment with** a **byte** (or word) **in** the **extra segment**.
The offset of the byte (or word) in data segment is in SI. The offset of the byte (or word) in extra segment is in DI.
SI and DI are incremented / decremented depending upon the direction flag.
Comparison is done by subtracting the byte (or word) from extra segment from the byte (or word) from Data segment.
The Flag bits are affected, but the result is not stored anywhere.

```
Eg :CMPSB               ; Compare DS:[SI] with ES:[DI] … byte operation
                        ; SI ← SI ± 1 … depending upon DF
                        ; DI ← DI ± 1 … depending upon DF

    CMPSW               ; Compare {DS:[SI], DS:[SI+1]}
                        ; with {ES:[DI], ES:[DI+1]}
                        ; SI ← SI ± 2 … depending upon DF
                        ; DI ← DI ± 2 … depending upon DF
```

5)**SCAS: SCASB/SCASW** (*Scan String*)

It is used to **compare** the contents of **AL** (or AX) **with** a **byte** (or word) **in** the **extra segment**.
The offset of the byte (or word) in extra segment is in DI.
DI is incremented / decremented depending upon the direction flag (DF). Comparison is done by subtracting a byte (or word) from extra segment from AL (or AX). The Flag bits are affected, but the result is not stored anywhere.

```
Eg:SCASB                ; Compare AL with ES:[DI] … byte operation
                        ; DI ← DI ± 1 … depending upon DF

   SCASW                ; Compare {AX} with {ES:[DI], ES:[DI+1]}
                        ; DI ← DI ± 1 … depending upon DF
```

REP (*Repeat prefix used for string instructions*)

This is an **instruction prefix**, which can be used in string instructions.
It can be **used with string instructions only**.
It **causes** the **instruction** to be **repeated CX number** of times.
After each execution, the **SI** and **DI** registers are **incremented/decremented** based on the **DF** (Direction Flag) in the Flag register **and CX is decremented.**
i.e. **DF = 1; SI, DI decrements.**

Thus, it is important that before we use the REP instruction prefix the following steps must be carried out:
CX must be initialized to the Count value. If **auto decrementing** is required, **DF** must be **set using STD** instruction **else cleared** using **CLD** instruction.

```
EG:     MOV CX, 0023H
        CLD
   REP  MOVSB
```

The above section of a program will cause the following string operation

ES:[DI] ← DS:[SI], SI ← SI + 1, DI ← DI + 1, CX ← CX – 1

to be executed 23H times (as CX = 23H) in auto incrementing mode (as DF is cleared).

6) **REPZ/REPE** (*Repeat on Zero/Equal*)
It is a conditional repeat instruction prefix.It behaves the same as a REP instruction **provided** the Zero Flag is set (i.e. **ZF = 1**).It is used with CMPS instruction.

7) **REPNZ/REPNE** (*Repeat on No Zero/Not Equal*)
It is a conditional repeat instruction prefix.It behaves the same as a REP instruction **provided** the Zero Flag is reset (i.e. **ZF = 0**).It is used with SCAS instruction.

Please Note: 8086 instruction set has only 3 instruction prefixes :

1) **ESC** *(to identify 8087 instructions)*
2) **LOCK** *(to lock the system bus during an instruction)*
3) **REP** *(to repeatedly execute string instructions)*
   For a question on instruction prefixes (asked repeatedly), explain the above in detail.