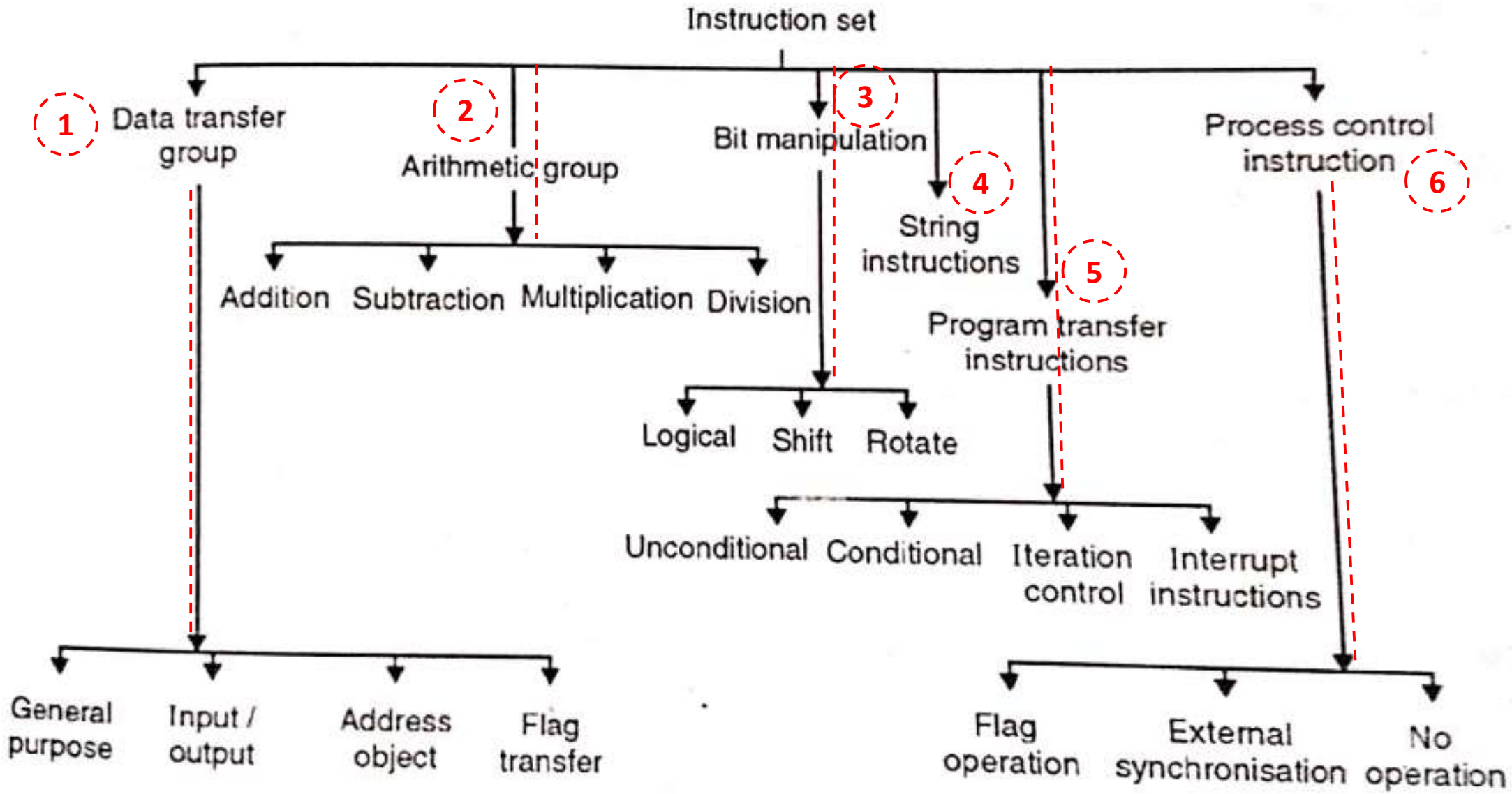
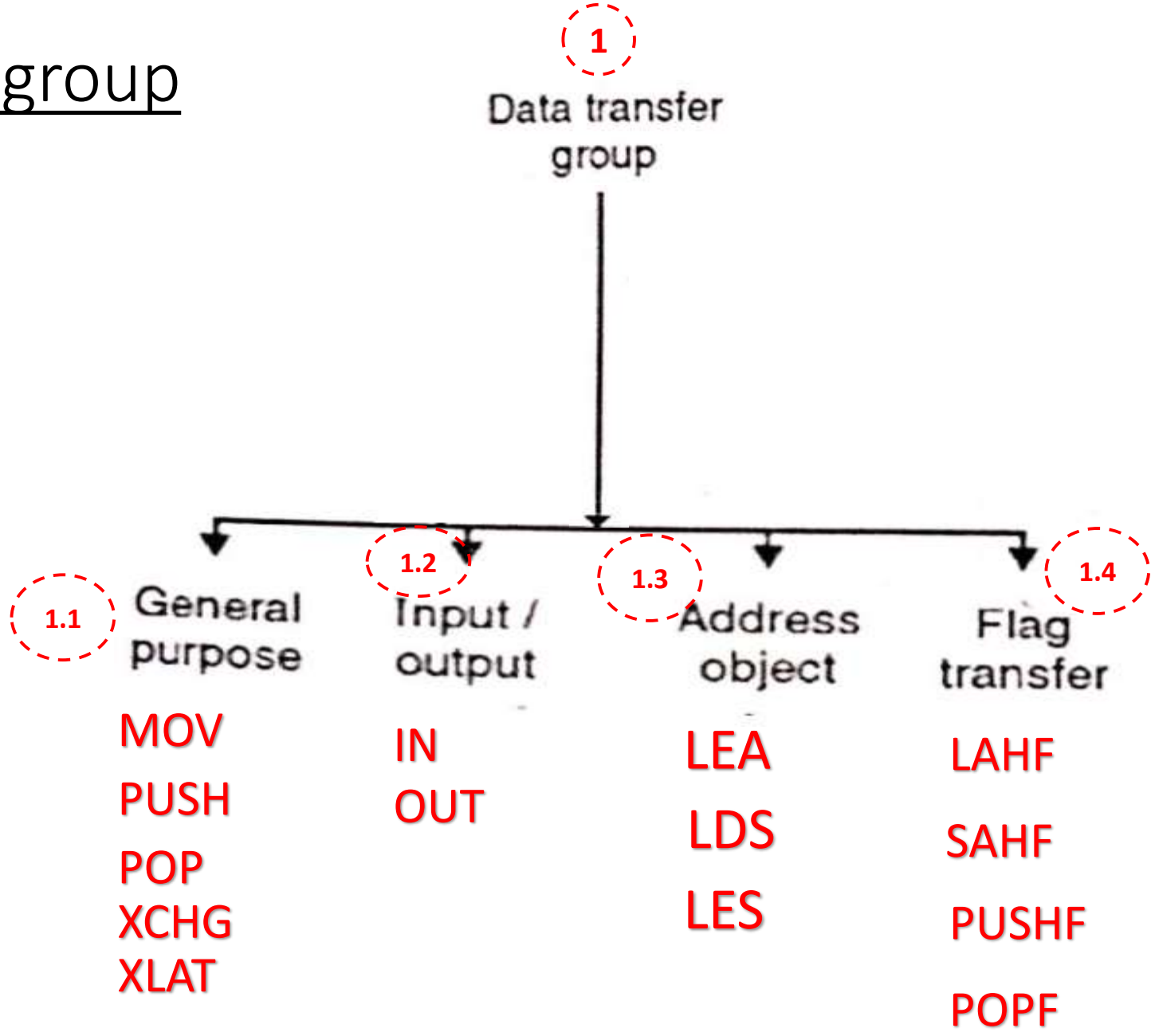
Instruction set of 8086

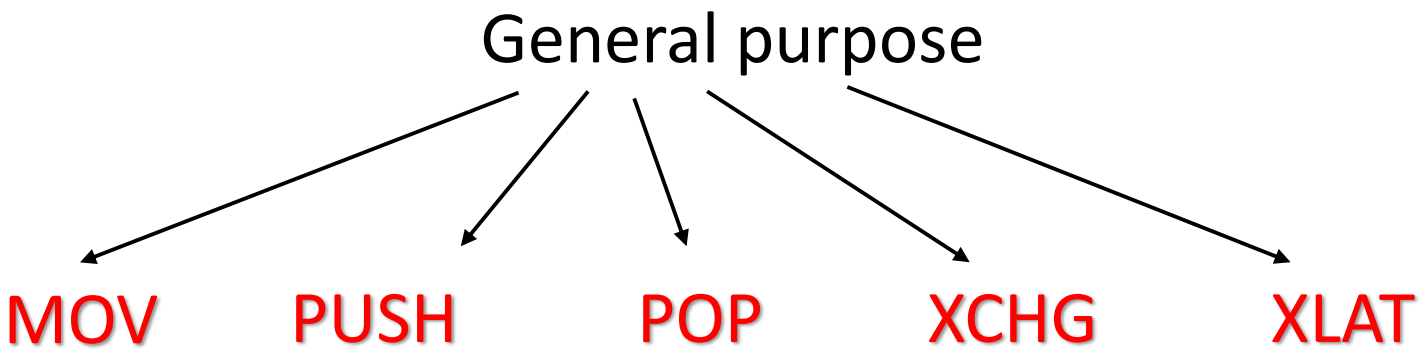
Instruction set of 8086



1. Data transfer group



1.1 General Purpose



MOV Instruction

The mov instruction copies a word or a byte of data from a fixed/specified source to a fixed/specified destination

Mnemonic: MOV destination , source

MOV Operand 1 , Operand 2

Operation :

Destination ⟵ Source

Operand 1 ⟵ Operand 2

Flags :

No Flags affected

Sr. No.	Destination	Source
1.	Memory	Accumulator
2.	Accumulator	Memory
3.	Register	Register
4.	Register	Memory
5.	Memory	Register
6.	Register	Immediate
7.	Memory	Immediate
8.	Seg-Reg	Reg – 16(GPR)
9.	Seg-Reg	Mem – 16
10.	Reg – 16(GPR)	Seg-Reg
11.	Memory	Seg-Reg

MOV memory , accumulator

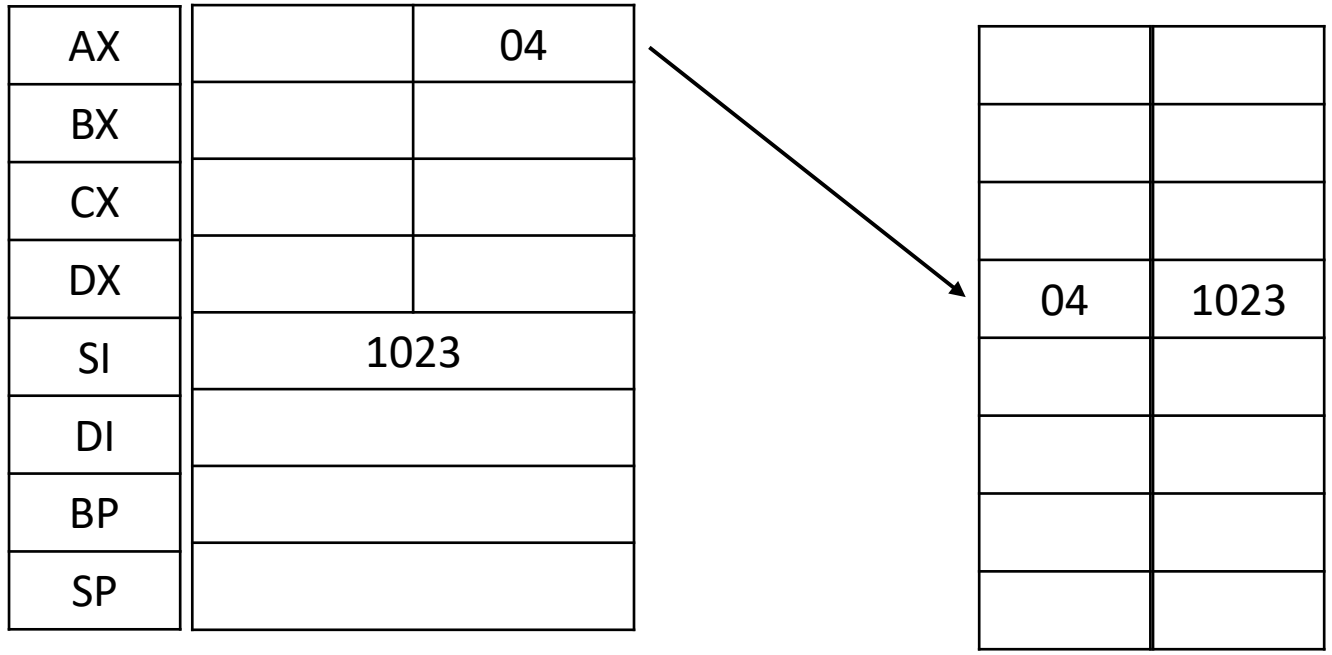
This instruction copies the contents of accumulator into memory location specified in the instruction

Mnemonic: MOV memory , accumulator

Operation :

memory ← accumulator

Example: MOV [SI] , AL

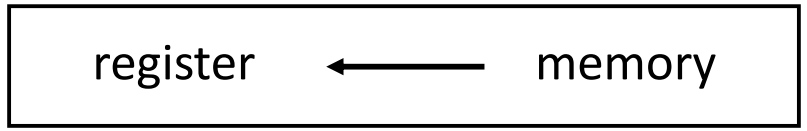


MOV register , memory

This instruction will copy the contents of the memory location specified in the instruction to the destination register

Mnemonic: MOV register , memory accumulator

Operation :



Example: MOV CX , COUNT[DI]

Addressing mode : register relative addressing mode

Example: MOV CX , COUNT[DI]

Addressing mode : register relative addressing mode

Example: MOV CX , [DI+1000]

EU

AX		
BX		
CX	AA	2B
DX		
SI		
DI	1100	
BP		
SP		

BIU

CS	2105
DS	2314
ES	
SS	
IP	187A

Starting address (CS + IP)

CS = 21050

+ 187A

228CA

For program :

DS = 23140

+1100

24240

+1000

25240

For Data :

228CA 228CB	Opcode of MOV instruction
228CC	00
228CD	10
228CE	
25240	12
25241	34

Here assuming COUNT = 1000 similar with relative addressing mode

Before Execution

Example: MOV CX , COUNT[DI] Addressing mode : register relative addressing mode Example: MOV CX , [DI+1000]

EU

AX		
BX		
CX	12	34
DX		
SI		
DI	1100	
BP		
SP		

BIU

CS	2105
DS	2314
ES	
SS	
IP	187E

CS = 21050

\+ 187E

228CE

228CA	Opcode of MOV instruction
228CB	
228CC	00
228CD	10
228CE	Opcode of next instruction
25240	34
25241	12

After Execution

PUSH Instruction

Register → Stack

This instruction is used to transfer the contents of the specified register are copied onto the stack in the following sequences:

1. The stack pointer is decremented by 1 and contents of higher order register copied on that location
2. The stack pointer is again decremented by 1 and contents of lower order register copied on that location

Mnemonic: PUSH source

Operation :

SP = SP – 2

SS : [SP] (top of the stack)=Operand

Flags :

No Flags affected

Addressing mode : register indirect addressing mode

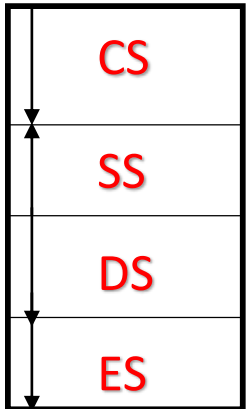
Example: PUSH AX

	AH	AL
AX	03	04
BX		
CX		
DX		
SI		
DI		
BP		
SP	2586	

SS

	0000	
	2584	
SP - 1	2585	03
SP	2586	
	FFFF	

The stack pointer is decremented by 1 and contents of higher order register copied on that location



	AH	AL
AX	05	04
BX		
CX		
DX		
SI		
DI		
BP		
SP	2584	

SS

	0000	
SP - 2	2584	04
SP - 1	2585	03
SP	2586	
	FFFF	

The stack pointer is again decremented by 1 and contents of lower order register copied on that location

SP – 2 =SP

POP Instruction

Register ⟵ Stack

This instruction is used to transfer the contents of the stack into register the following sequences:

1. The stack pointer is incremented by 1 and contents of the memory location are copied into higher order
2. The stack pointer is again incremented by 1 and contents of the memory location are copied into lower order register

Mnemonic: POP Destination

Operation :

SP = SP + 2

SS : [SP] (top of the stack)=Operand

Flags :

No Flags affected

Addressing mode : register indirect addressing mode

Example: POP AX

	AH	AL
AX		04
BX		
CX		
DX		
SI		
DI		
BP		
SP	2584	

SS

	0000	
SP	2584	04
SP + 1	2585	05
	2586	
	FFFF	

The copy the contents of lower order register copied on that location

	AH	AL
AX	05	04
BX		
CX		
DX		
SI		
DI		
BP		
SP	2586	

SS

	0000	
SP	2584	04
SP + 1	2585	05
SP + 2	2586	
	FFFF	

The stack pointer is incremented by 1 and contents of higher order register copied on that location

SP + 2 = SP

XCHG Instruction

Destination ⟷ Source

This instruction is used to exchange the contents of AX and BX

Mnemonic: XCHG

Operation :

Destination ⟷ Source

Flags :

No Flags affected

NOTE :

For AX and BX only

Addressing mode : Implied addressing mode

Example : XCHG AX,BX

	AH	AL
AX	3B	04
BX	01	02
CX		
DX		
SI		
DI		
BP		
SP		

Before Execution

	AH	AL
AX	01	02
BX	3B	04
CX		
DX		
SI		
DI		
BP		
SP		

After Execution

XLAT / XLTAB Instruction(Translate or Replace byte)

- This instruction replaces a byte in AL register with a byte from a look up table in the memory.
- Here the contents of AL before execution acts as index to the desired location in lookup table.
- Mostly this concept is used to convert BCD to ASCII or to seven segment.

Mnemonic: XLAT

Operation :

AL = DS :[BX + AL]

Flags :

No Flags affected

NOTE :

For AL and BX only

Addressing mode : Implied addressing mode

Before Execution

EU

AX		02
BX	20	00
CX		
DX		
SI		
DI		
BP		
SP		

BIU

CS	2105
DS	2314
ES	
SS	
IP	187A

CS = 21050

\+ 187A

228CA

For program :

For Data :

DS = 23140

+2000

25140

\+ 02

25142

228CA	Opcode of XLTA instruction
228CB	
228CC	
228CD	
228CE	
25140	12
25141	34
25142	35
25143	

After Execution

EU

AX		35
BX	20	00
CX		
DX		
SI		
DI		
BP		
SP		

BIU

CS	2105
DS	
ES	
SS	
IP	

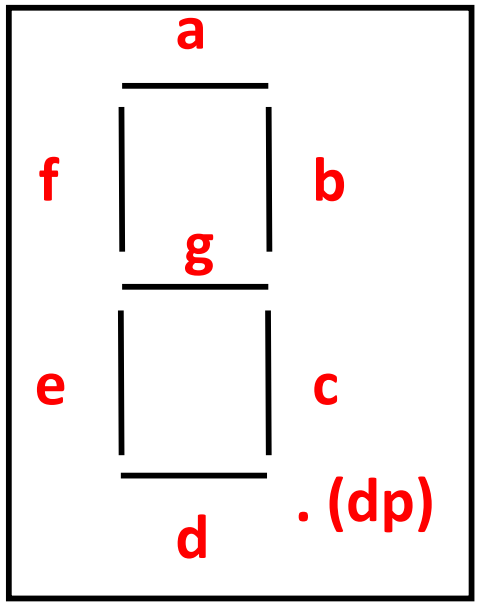
228CA	Opcode of XLTA instruction
228CB	
228CC	
228CD	
228CE	
25140	12
25141	34
25142	35
25143	

AL = DS :[BX + AL]

AX		01
BX	40	00

DS

FFFF	
4000	0’
4001	1’(06)
4002	2’
4009	9’



Seven Segment Display

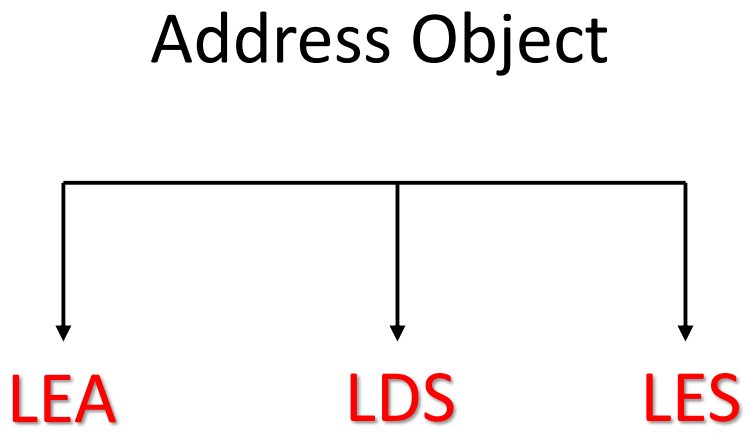
- Cathode = 0
- Anode = 1

dp	g	f	e	d	c	b	a			
0	0	0	0	0	1	1	0	=	06	1’

To get the code of 5

```
Mov BX, 4000
Mov AL , 01
XLAT
```

3.Address Object



LEA Load effective Address

- This instruction determines the offset of variables or memory location named as source and puts the offset in the indicated 16 bit register.
- This instruction is replaced by mov when assembly is possible.
- Normally the offset is loaded into index or base pointer register such as SI,DI,BX,BP

Mnemonic: LEA AX,COUNT

Operation :

Loads AX with the offset count

AX ← offset of COUNT

Flags :

No Flags affected

Addressing mode : register direct addressing mode

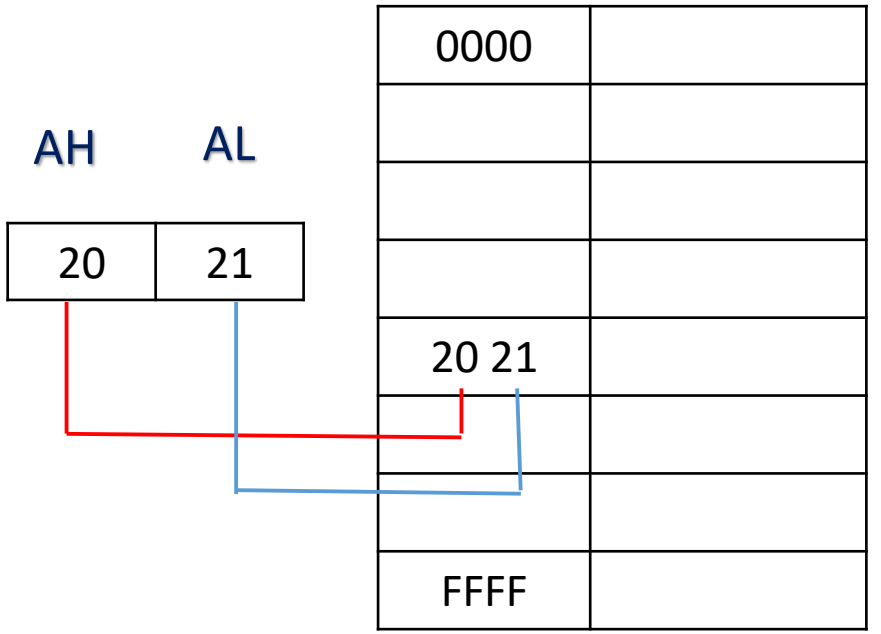
Before Execution

AH	AL
52	58

0000	
20 21	
FFFF	

After Execution

AH	AL
20	21



AX register is used to store offset value after execution of this instruction

LDS Load pointer with DS

Load register and DS with words from memory

- The source is always a memory location. DS is used as a segment register for memory.
- This instruction copies a word from two memory locations into register specified in instruction.
- It then copies a word from the next two memory locations into the DS register.

Mnemonic: LDS reg, source

Operation :

REG = first word,
DS = Second word

Flags :

No Flags affected

Addressing mode : register direct addressing mode

Example: LDS BX,count

Before Execution

Reg BX

02	08

CS	2105
DS	2314
ES	
SS	
IP	

DS

0000	
25185	01
25186	50
25187	23
25188	78
FFFF	

DS = 23140

+2045

25185

Assume count = 2045

Operation :

REG = first word,

DS = Second word

After Execution

Reg BX

50	01

CS	2105
DS	7823
ES	
SS	
IP	

DS

0000	
25185	01
25186	50
25187	23
25188	78
FFFF	

LES Load pointer using ES

Load register and ES with words from memory

- The source is always a memory location. DS is used as a segment register for memory.
- This instruction copies a word from two memory locations into register specified in instruction.
- It then copies a word from the next two memory locations into the DS register.

Mnemonic: LES reg, source

Operation :

REG = first word,
ES = Second word

Flags :

No Flags affected

Addressing mode : register direct addressing mode

Example: LES BX,count

Before Execution

Reg BX

02	08

CS	2105
DS	2314
ES	
SS	
IP	

DS

0000	
25185	01
25186	50
25187	23
25188	78
FFFF	

DS = 23140

+2045

25185

Assume count = 2045

Operation :

REG = first word,

ES = Second word

After Execution

Reg BX

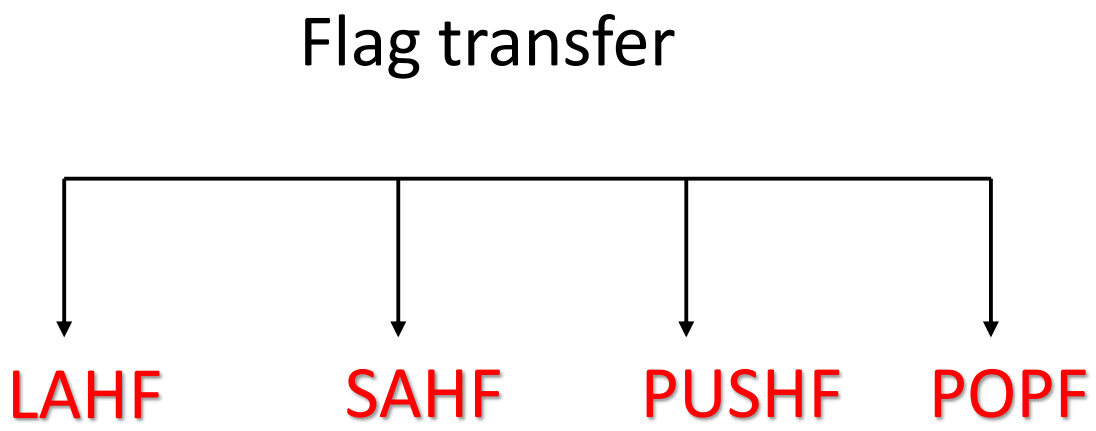
50	01

CS	2105
DS	
ES	7823
SS	
IP	

DS

0000	
25185	01
25186	50
25187	23
25188	78
FFFF	

4.Flag Instruction (Flag transfer)



LAHF Load AH reg from flags

Copy Lower byte of flag register to AH

- This instruction is used to transfer lower byte of flag register into AH register

Mnemonic: LAHF

Operation :

AH = lower byte of flag register

Flags :

No Flags affected

Addressing mode : Implied addressing mode

Example: LAHF

0909 = 0000 1001 0000 1001

FFFF = 1111 1111 1111 1111

1111 1111 1111 111

1 0000 1001 0000 1000

Before Execution

AH	AL
52	58

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF
0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	1

0C 11

After Execution

AH	AL
11	58

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF
0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	1

0C 11

SAHF Store AH reg in flags

Copy AH into Lower byte of flag register

- This instruction is used to transfer content AH register into lower byte of flag register

Mnemonic: SAHF

Operation :

lower byte of flag register = AH

Flags :

No Flags affected

Addressing mode : Implied addressing mode

Example: SAHF

0909 = 0000 1001 0000 1001

FFFF = 1111 1111 1111 1111

1111 1111 1111 111

1 0000 1001 0000 1000

Before Execution

AH	AL
03	58

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF
0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	1

0C 11

After Execution

AH	AL
07	58

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF
0	0	0	0	1	1	0	0	0	0	0	1	0	1	1	1

0C 07

PUSHF push flags onto stack

- This instruction is used to transfer flag register onto stack.
- The stack pointer is decremented by 1 and contents of higher order byte of flag register copied on that location
- The stack pointer is again decremented by 1 and contents of lower byte of flag order register copied on that location

Mnemonic: PUSHF

Operation :

SP = SP - 2

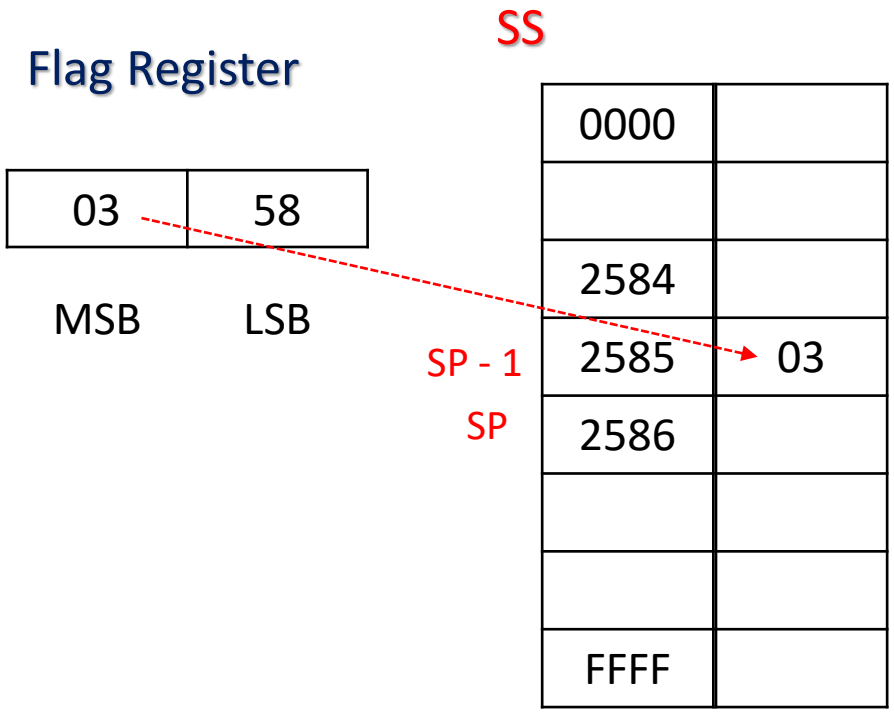
SS : [SP] (top of the stack)=Operand

Flags :

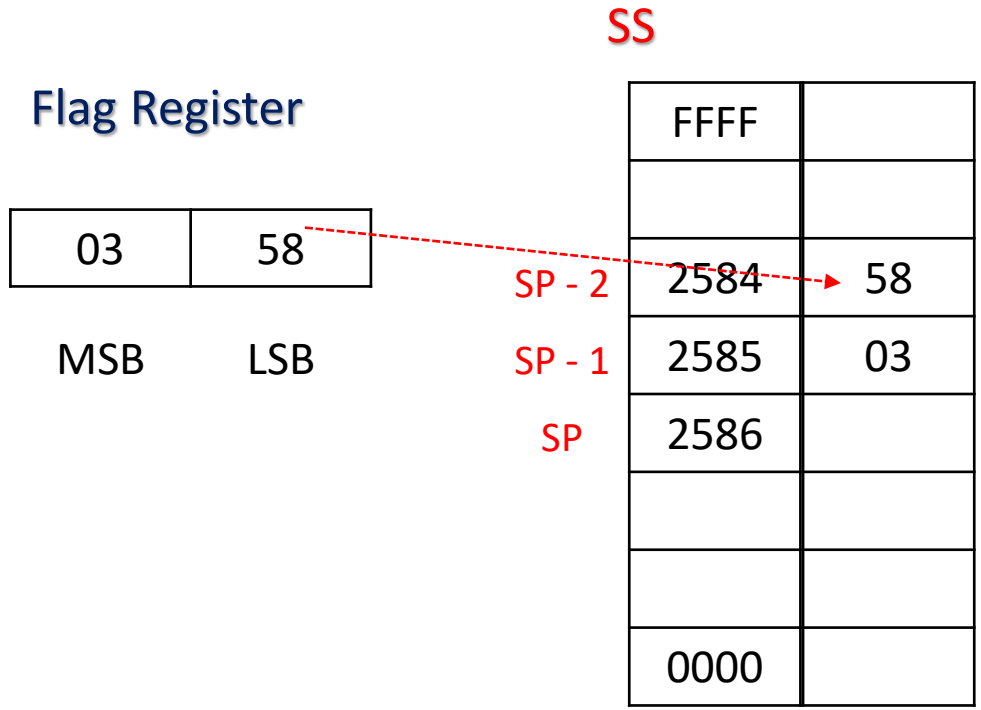
No Flags affected

Addressing mode : Register addressing mode

Example: PUSHF



The stack pointer is decremented by 1 and contents of higher order register copied on that location



The stack pointer is again decremented by 1 and contents of lower order register copied on that location

SP – 2 =SP

POPF Instruction

This instruction is used to transfer the contents of the stack into register the following sequences:

1. The stack pointer is incremented by 1 and contents of the memory location are copied into higher order byte of flag register
2. The stack pointer is again incremented by 1 and contents of the memory location are copied into lower order byte of flag register

Mnemonic: POPF

Operation :

SP = SP + 2

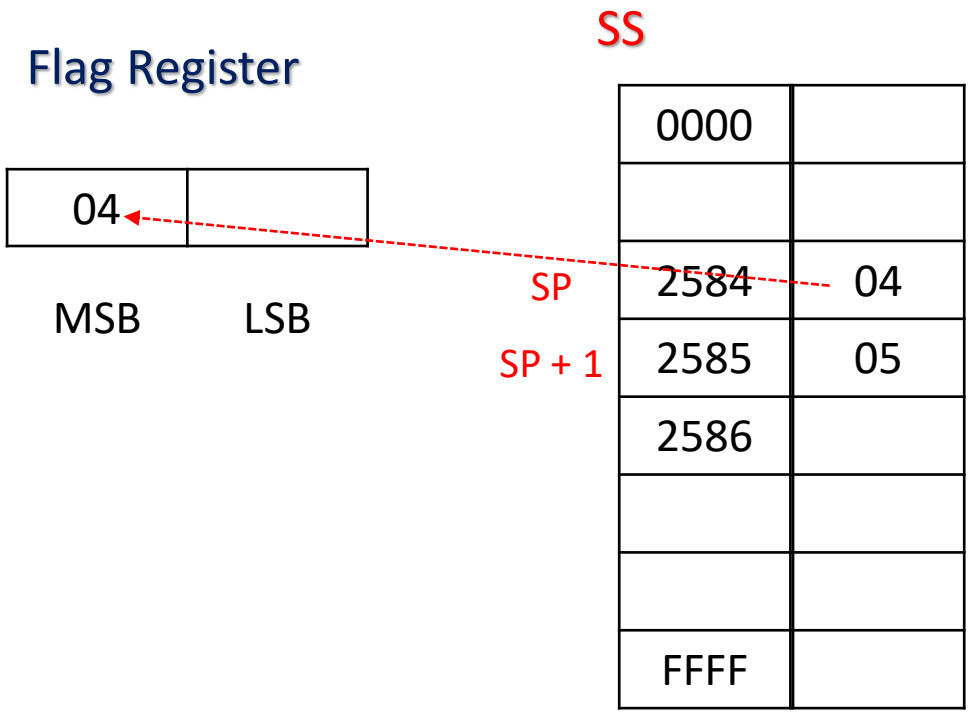
SS : [SP] (top of the stack)=Operand

Flags :

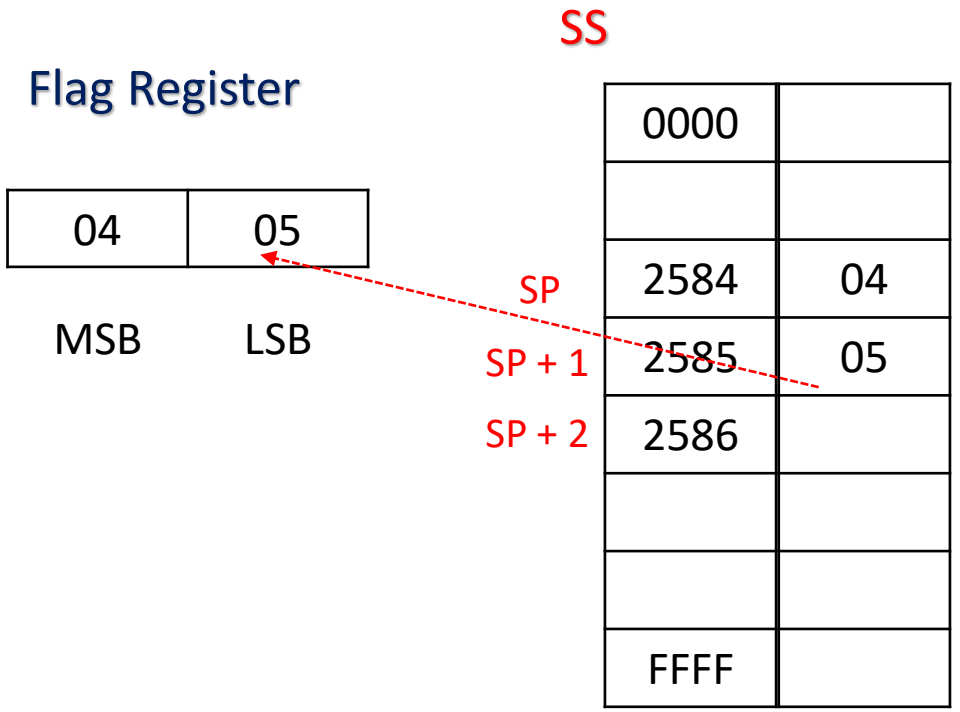
No Flags affected

Addressing mode : register addressing mode

Example: POP AX



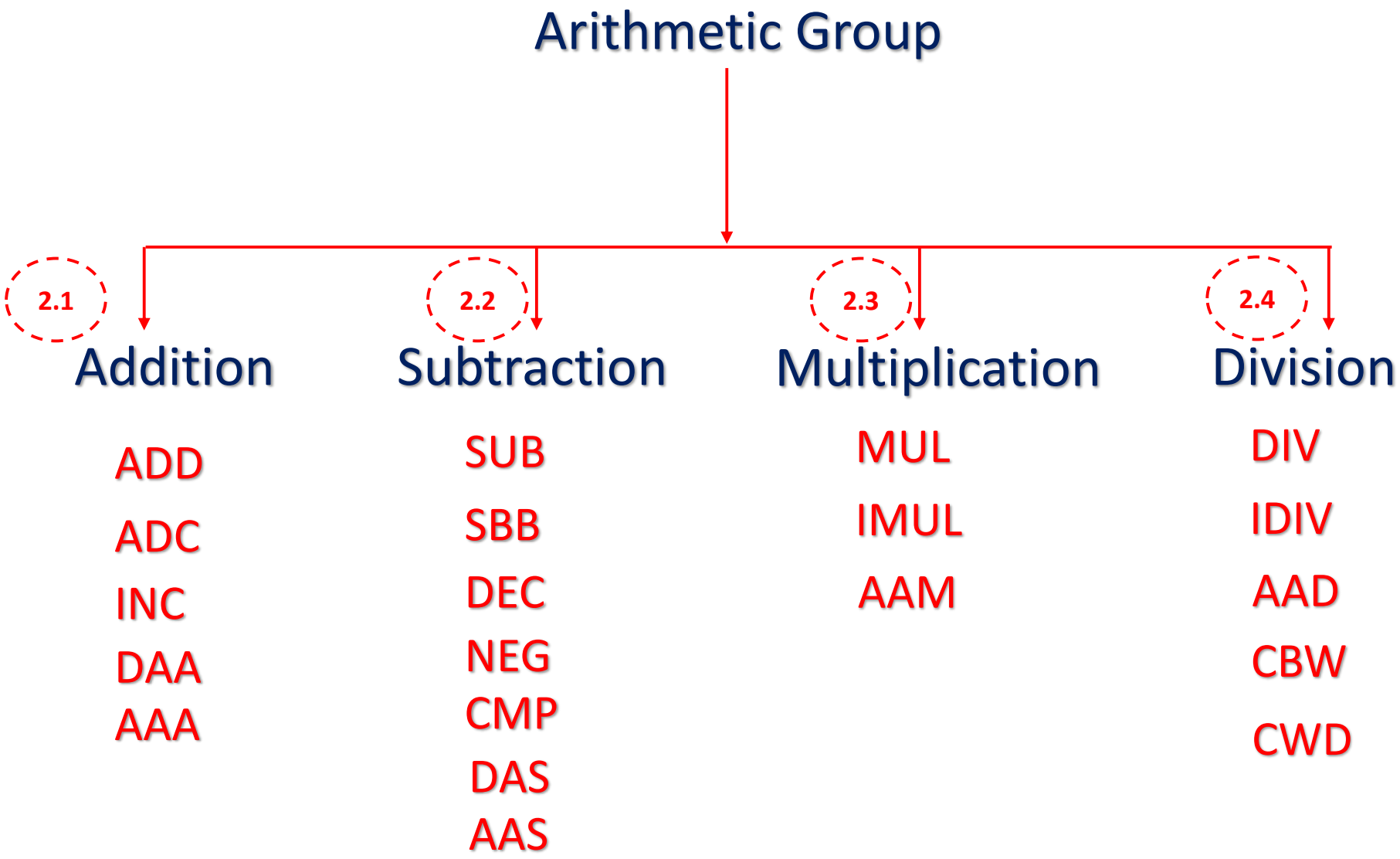
The stack pointer is again decremented by 1 and contents of lower order register copied on that location



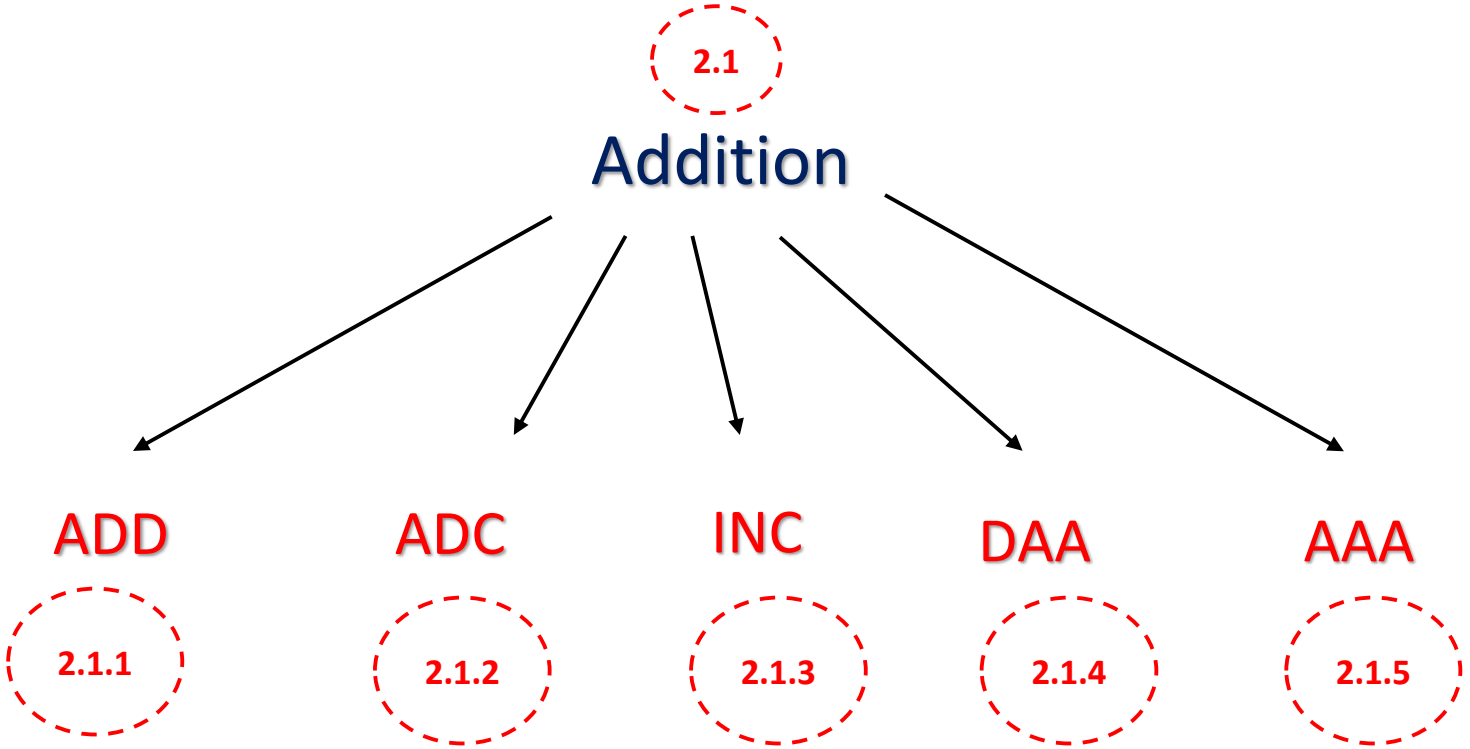
The stack pointer is decremented by 1 and contents of higher order register copied on that location

SP + 2 = SP

2. Arithmetic group



2.1 Addition Group



2.1.1 ADD – Add byte or word

This instruction adds a number from source to number from destination and puts the result to specified destination.

Mnemonic: ADD destination , source

ADD Operand 1 , Operand 2

Operation :

Destination ← Source + Destination

Operand 1 ← Operand 1 + Operand 2

Sr. No.	Destination	Source
1	Register	Register
2	Register	Memory
3	Memory	Register
4	Register	Immediate
5	Memory	Immediate
6	Accumulator	Immediate

Flags :

All Flags affected

ADD register , register

- This instruction adds the data in registers.
- The result is stored in register. It can be 8 bit/ 16 bit instruction.
- This instruction can be 8/16 bit.

Mnemonic: ADD register , register

Example: ADD BL, CL

BL ⟵ BL + CL

Operation :

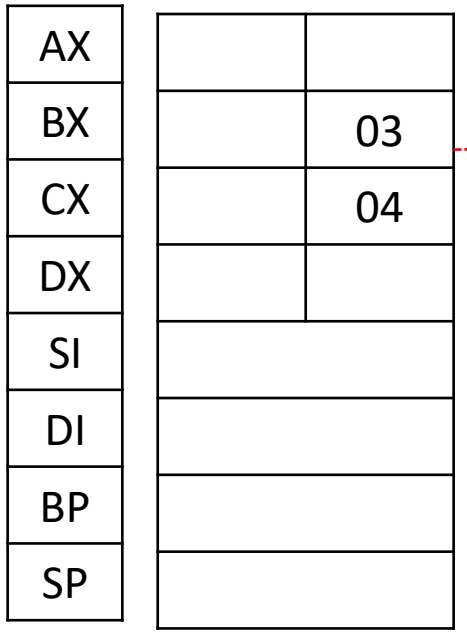
Register ⟵ Register + Register

Addressing mode : Register addressing mode

Flags :

All Flags affected

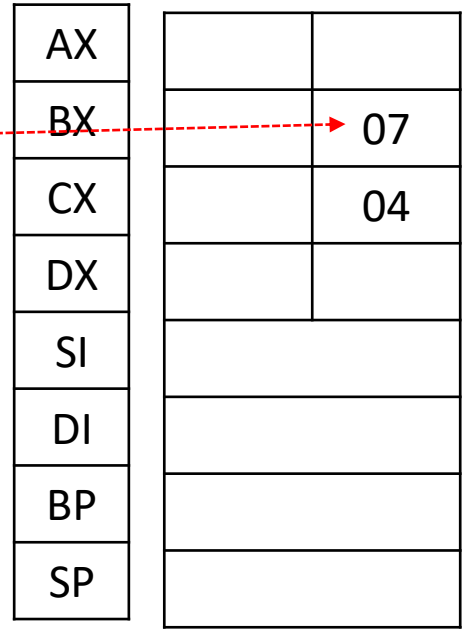
Before execution CL = 04 , BL = 03



Before Execution

ADD BL, CL

After execution CL = 04 , BL = 07



After Execution

ADD register , memory

- This instruction adds the data from memory location and required registers.
- The result is stored in register.
- This instruction can be 8/16 bit.

Mnemonic: ADD register , memory

Operation :

Register ⟵ Register + content of memory location

Example: ADD AX, [2000]

AX ⟵ AX + content of memory location

Addressing mode : Register addressing mode

Flags :

All Flags affected

Before Execution

EU

AX	02	03
BX		
CX		
DX		
SI	2000	
DI		
BP		
SP		

CS = 21050

\+ 187A

228CA

For program :

228CA	Opcode of ADD instruction
228CB	
228CC	
228CD	
228CE	
25140	04
25141	05
25142	35
25143	

BIU

CS	2105
DS	2314
ES	
SS	
IP	187A

For Data :

DS = 23140

\+2000

25140

After Execution

EU

AX	**06**	**08**
BX		
CX		
DX		
SI		
DI		
BP		
SP		

228CA	Opcode of ADD instruction
228CB	
228CC	
228CD	
228CE	
25140	04
25141	05
25142	35
25143	

BIU

CS	2105
DS	
ES	
SS	
IP	

2.1.2 ADC – Add with carry

- This instruction adds destination operand contents , source operand contents and Carry flag content.
- Result is stored back to destination operand.
- The source and destination can be 8/16 bit register or memory location. The source can also 8/16 bit register or memory location and immediate data.
- It is easy to perform multiple – precision arithmetic by using ADC instruction.

Mnemonic: ADD destination , source

Operation :

Destination ⟵ Destination + Source + CY

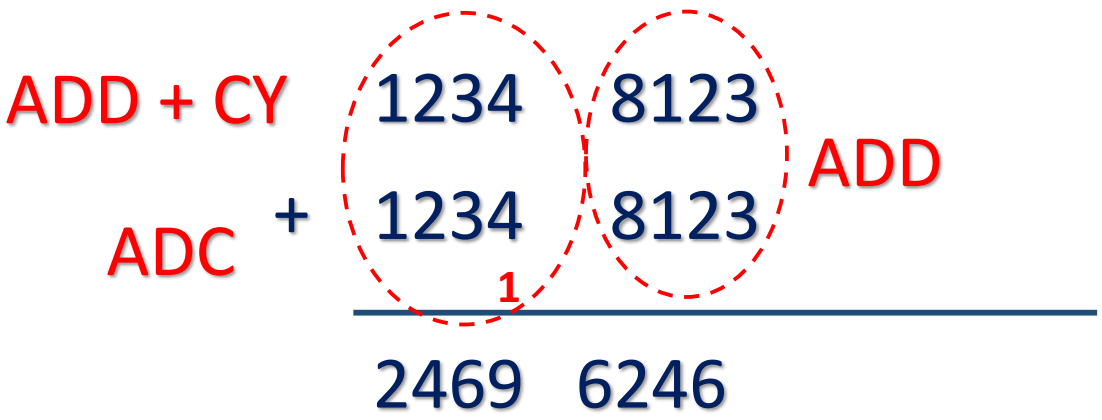
Flags :

All Flags affected

Two 16 bit number addition

ADD 1234
\+ 1234
2468

Two 32 bit number addition



2.1.3 INC – Increment byte or word by 1

- This instruction adds 1 to the destination operand.
- The operand can be a register or memory location.
- The operand may be a byte or word and it is treated as an unsigned binary number.

Mnemonic: INC destination

Operation :

Destination ← Destination + 1

Flags : All Flags affected except carry flag.(carry flag not changed)

Example: INC CX → IF CX = 1234 , after INC CX will be 1235

INC AL → IF AL = 08 , after INC AL will be 09

INC [2000] → This instruction increments the content of memory location 2000 by 1.

If 2000 = 04 , after INC it will be 05

2.1.4 DAA (Decimal Adjust After Addition)

This instruction is used to changed the contents of accumulator from a binary value to its equivalent BCD number.

Addition of hexadecimal numbers :

Case 1 :

	30 h	24 h	25 h	28 h	28 h
	+ 20 h	+ 24 h	+ 25 h	+ 27 h	+ 28 h
Expected answer →	50 h	48 h	50 h	55 h	56h
Actual answer →	50 h	48 h	4A h	4F h	50 h

Case 2 :

	50 h	64 h	84 h
	+ 50 h	+ 54h	+ 94h
Expected answer →	100 h	118 h	168 h
Actual answer →	A0 h	B8 h	108 h

Case 3 :

	88 h
	+ 88 h
Expected answer →	176 h
Actual answer →	110 h

There is a gap of 6 numbers

Up to 9 numbers BCD and HEX numbers are same

From 10 to 15 in HEX , 6 Characters are used and after these 6 characters 10 onwards numbers are started.

Decimal	BCD	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	0001 0000	**A**
11	0001 0001	**B**
12	0001 0010	**C**
13	0001 0011	**D**
14	0001 0100	**E**
15	0001 0101	**F**
16	0001 0110	10
17	0001 0111	11

So there is a 6 characters gap between BCD and hex

2.1.4 DAA (Decimal Adjust After Addition)

This instruction is used to changed the contents of accumulator from a binary value to its equivalent BCD number.

Mnemonic: DAA

Addressing mode : Implied

Flags : AF,CF,PF,ZF and SF

0 0 0 0 **0 1 1 0**

higher nibble lower nibble

60/06

Working of DAA instruction :

AL

HN

If higher nibble is > 9 II Carry flag = 1 then

add **60**

LN

If lower nibble is > 9 II Auxiliary flag = 1 then

add **06**

Why 6 is added?

Why 6 is added?

Up to 9 numbers BCD and HEX numbers are same

From 10 to 15 in HEX , 6 Characters are used and after these 6 characters 10 onwards numbers are started.

Decimal	BCD	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	0001 0000	**A**
11	0001 0001	**B**
12	0001 0010	**C**
13	0001 0011	**D**
14	0001 0100	**E**
15	0001 0101	**F**
16	0001 0110	10
17	0001 0111	11

So there is a 6 characters gap between BCD and hex and to overcome this gap 6 is added

Addition of hexadecimal numbers :

Case 1 :

	30 h	24 h	25 h	28 h	28 h
	+ 20 h	+ 24 h	+ 25 h	+ 27 h	+ 28 h
Expected answer →	50 h	48 h	50 h	55 h	56h
Actual answer →	50 h	48 h	4A h	4F h	50 h

Case 2 :

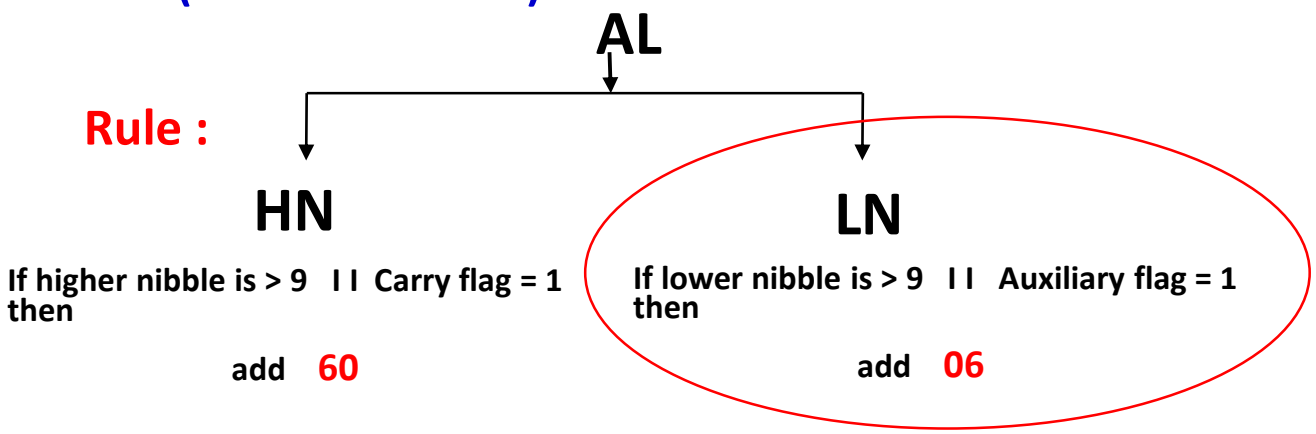
	50 h	64 h	84 h
	+ 50 h	+ 54h	+ 94h
Expected answer →	100 h	118 h	168 h
Actual answer →	A0 h	B8 h	108 h

Case 3 :

	88 h
	+ 88 h
Expected answer →	176 h
Actual answer →	110 h

There is a gap of 6 numbers

Case 1 : (for lower nibble)



a) Both conditions not true

```
  30 h
+ 20 h
------
  50 h
```

LN is not > 9 , Auxiliary flag = 0

Therefore , output same

b) one conditions true i.e. LN > 9

```
  25 h
+ 25 h
------
  4A h
```

LN > 9 , Auxiliary flag = 0

Therefore add 06 into LN

```
  0010 0101
+ 0010 0101
-----------
  0100 1010   4A h
+ 0000 0110   06 h
-----------
  0101 0000   50 h
```

c) one conditions are true i.e. Ln>! 9 and AF = 1

```
  28 h
+ 28 h
------
  50 h
```

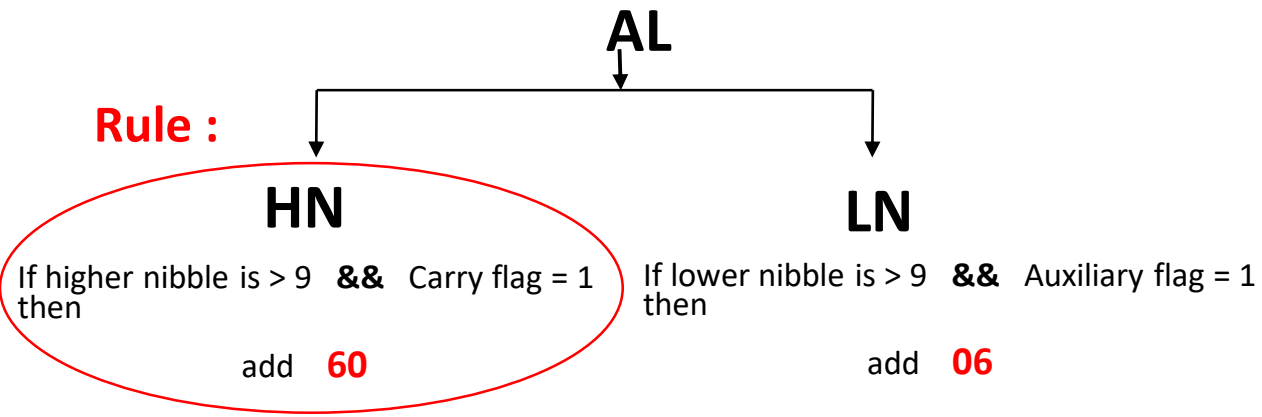
LN > 9 & Auxiliary flag = 1

Therefore add 06 into LN

```
  0010 1000
+ 0010 1000
      1
-----------
  0101 0000   50 h
+ 0000 0110   06 h
-----------
  0101 0110   56 h
```

Back

Case 2 : (for higher nibble)



a) Both conditions not true

```
  30 h
+ 20 h
  50 h
```

HN is not > 9 , Carry flag = 0

Therefore , output same

b) one conditions true i.e. LN > 9

```
  50 h
+ 50 h
  A0 h
```

HN > 9 , Carry flag = 0

Therefore add 06 into LN

```
  0101 0000
+ 0101 0000
-----------
  1010 0000   A0 h
+ 0110 0000   60 h
-----------
1 0000 0000   100 h
```

c) both conditions are true i.e. HN> 9 and CF = 1

```
  84 h
+ 94 h
  108 h
```

HN >9 & Carry flag = 1

Therefore add 06 into LN

```
  1000 0100
+ 1001 0100
-----------
1 0001 1000   108 h
+ 0110 0000   60 h
-----------
1 0111 1000   178 h
```

Prev carry

Back

Case 3 : (for both nibble)

AL

Rule :

HN

If higher nibble is > 9 || Carry flag = 1 then

add **60**

LN

If lower nibble is > 9 || Auxiliary flag = 1 then

add **06**

both conditions are true i.e. HN/LN > 9 and CF/AF = 1

```
   88 h
+  88 h
-------
  108 h
```

```
                1000  1000
          +     1000  1000
                  1
          --------------------
          1     0001  0000     100 h
       +        0110  0110     66 h
          --------------------
Prev      1     0111  0110     176 h
carry
```

AAA Instruction

Packed BCD

34

0011 0100

Unpacked BCD

34

03 04

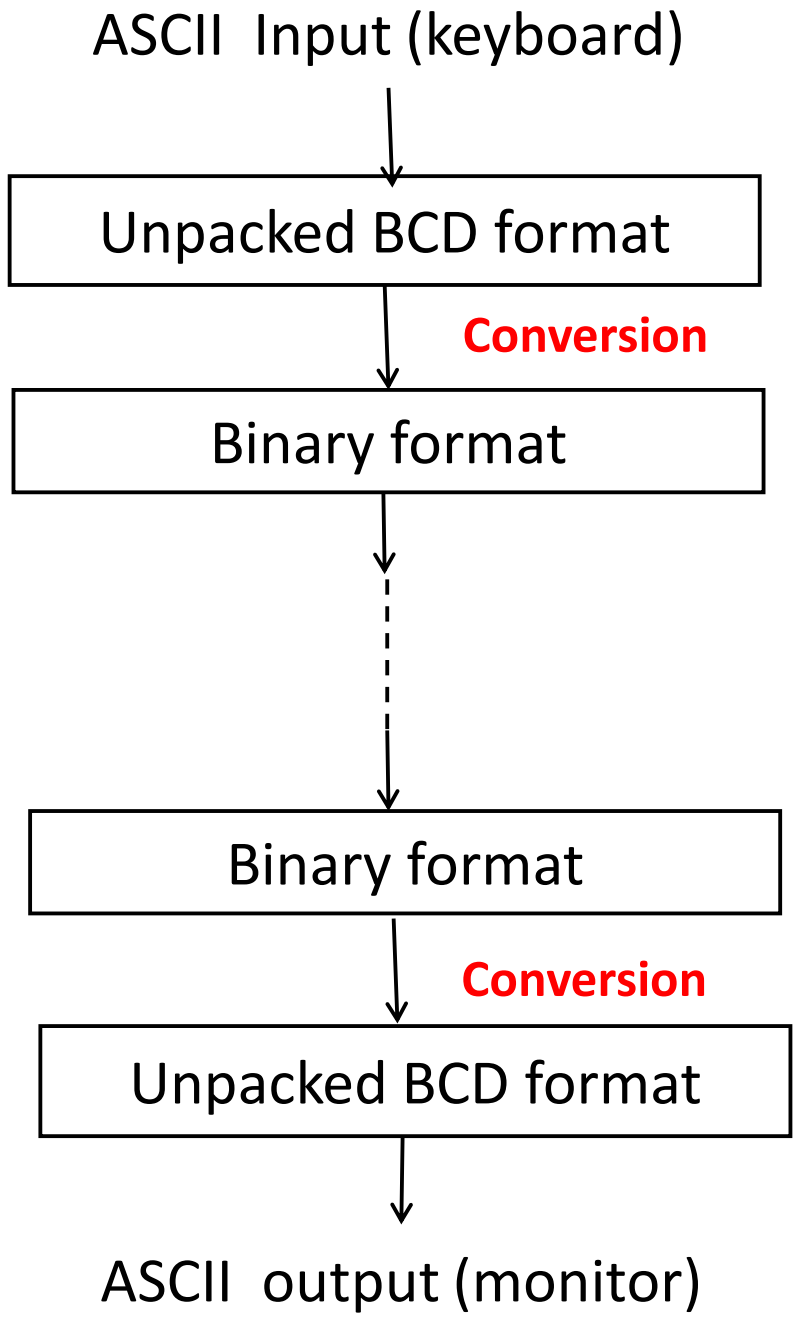
0000 0011 0000 0100

ASCII

34

0000 0100

- Computer follows ASCII input and output.
- We get from keyboard ASCII character and we have to send out ASCII character to monitor. Therefore we have to convert ASCII to binary



ASCII code for digit 0-9

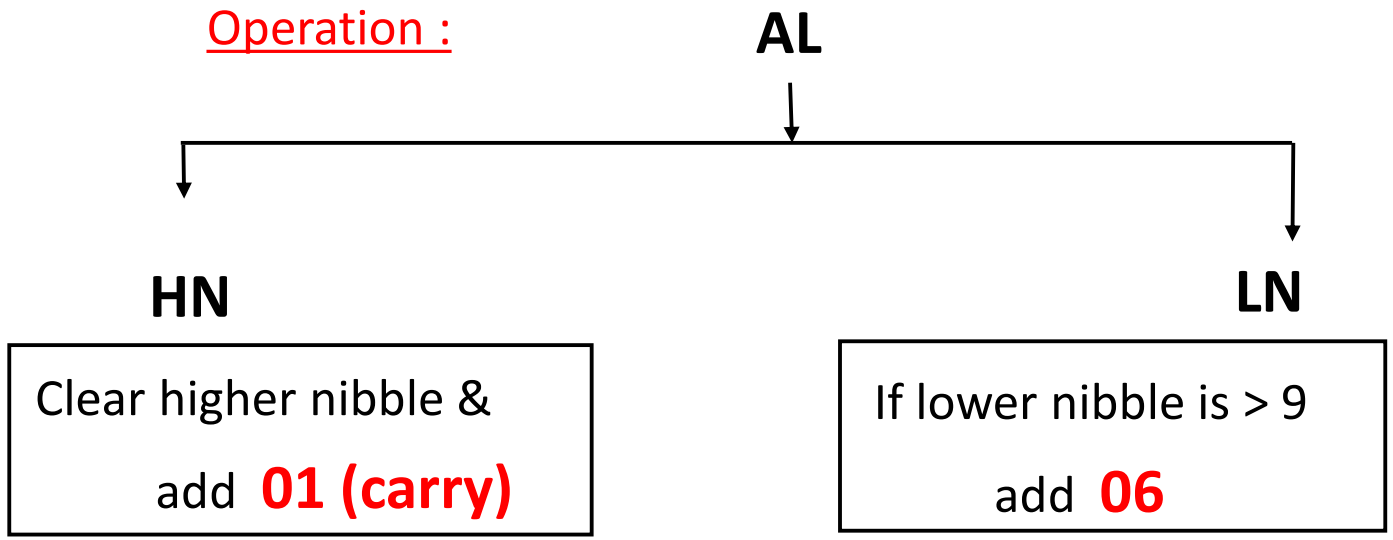
Key	ASCII (hex)	BCD (unpacked)
0	30	0011 0000
1	31	0011 0001
2	32	0011 0010
3	33	0011 0011
4	34	0011 0100
5	35	0011 0101
6	36	0011 0110
7	37	0011 0111
8	38	0011 1000
9	39	0011 1001

AAA (ASCII Adjust after addition)

- Numerical data coming into a computer from a terminal through keyboard is usually in ASCII code.
- The numbers 0 to 9 are represented by ASCII codes 30H to 39H .
- The 8086 allows to add the ASCII codes for two decimal digits without masking off “3” in the upper nibble for each.
- After addition , AAA instruction is used to make sure that the result is the correct unpacked BCD
- The AAA instruction works only AL register.

Mnemonic: AAA

Operation :



Flags :

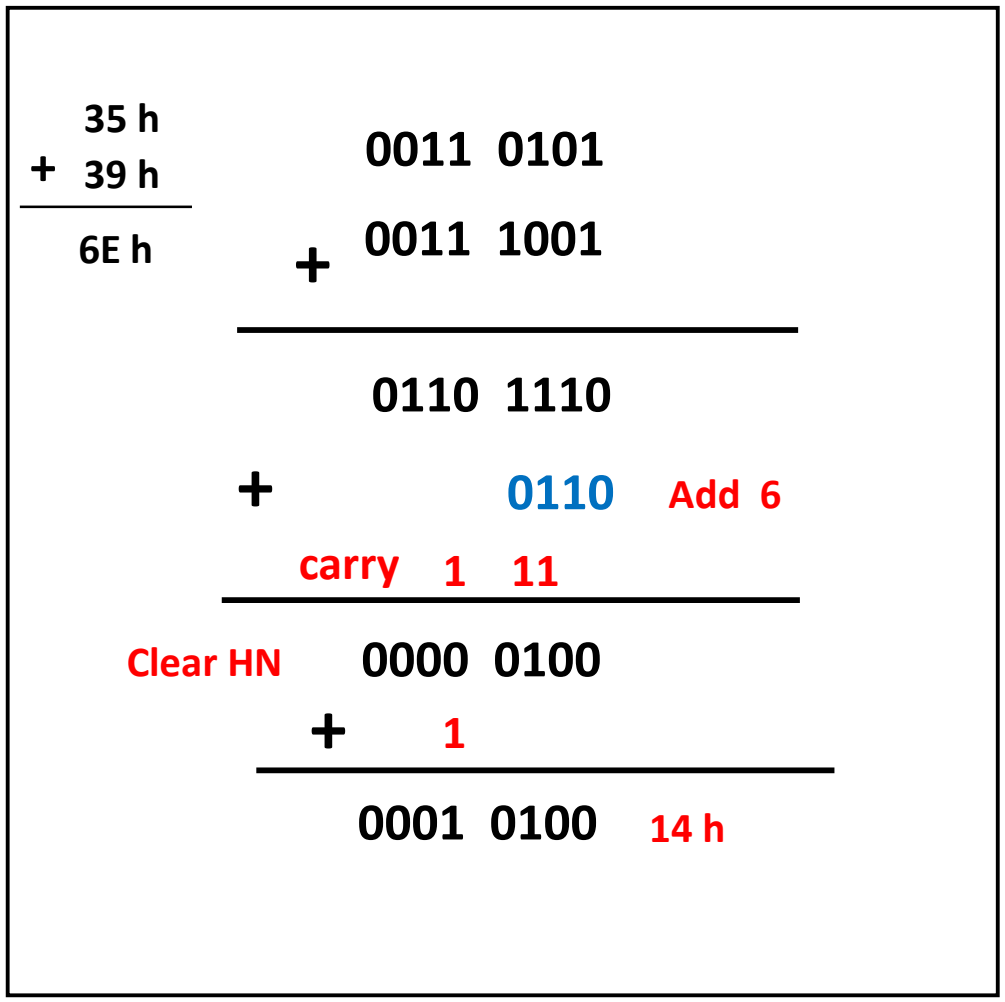
AF and CF flags are changed

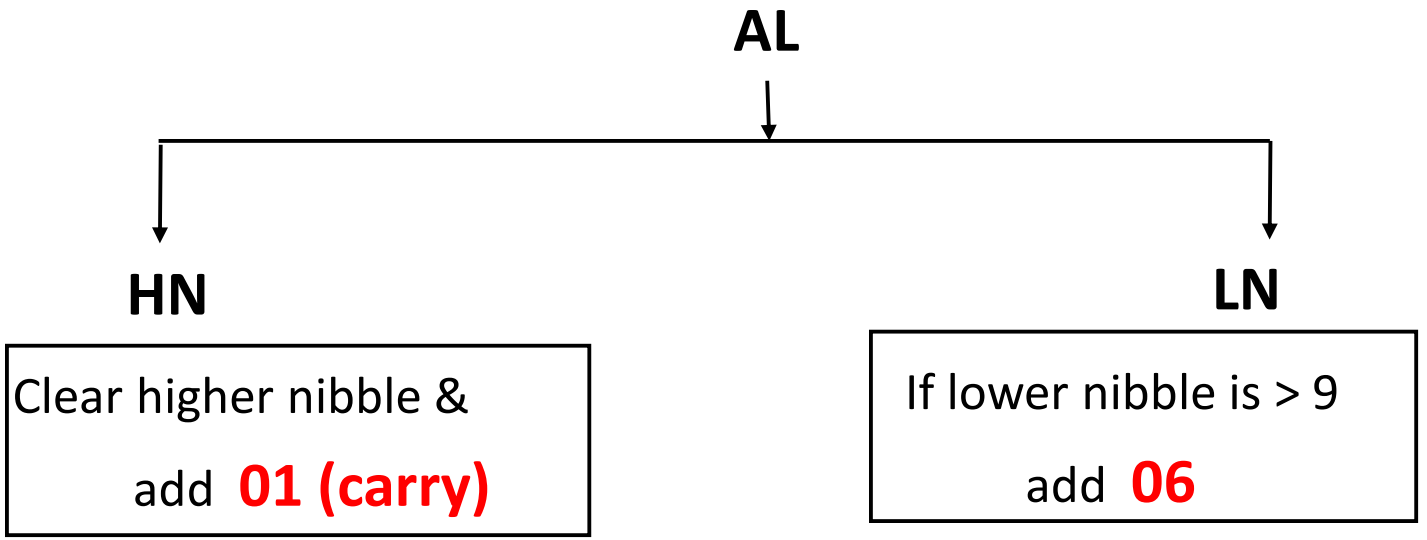
Addressing mode : Implied

AL = 0011 0101(ASCII 5)

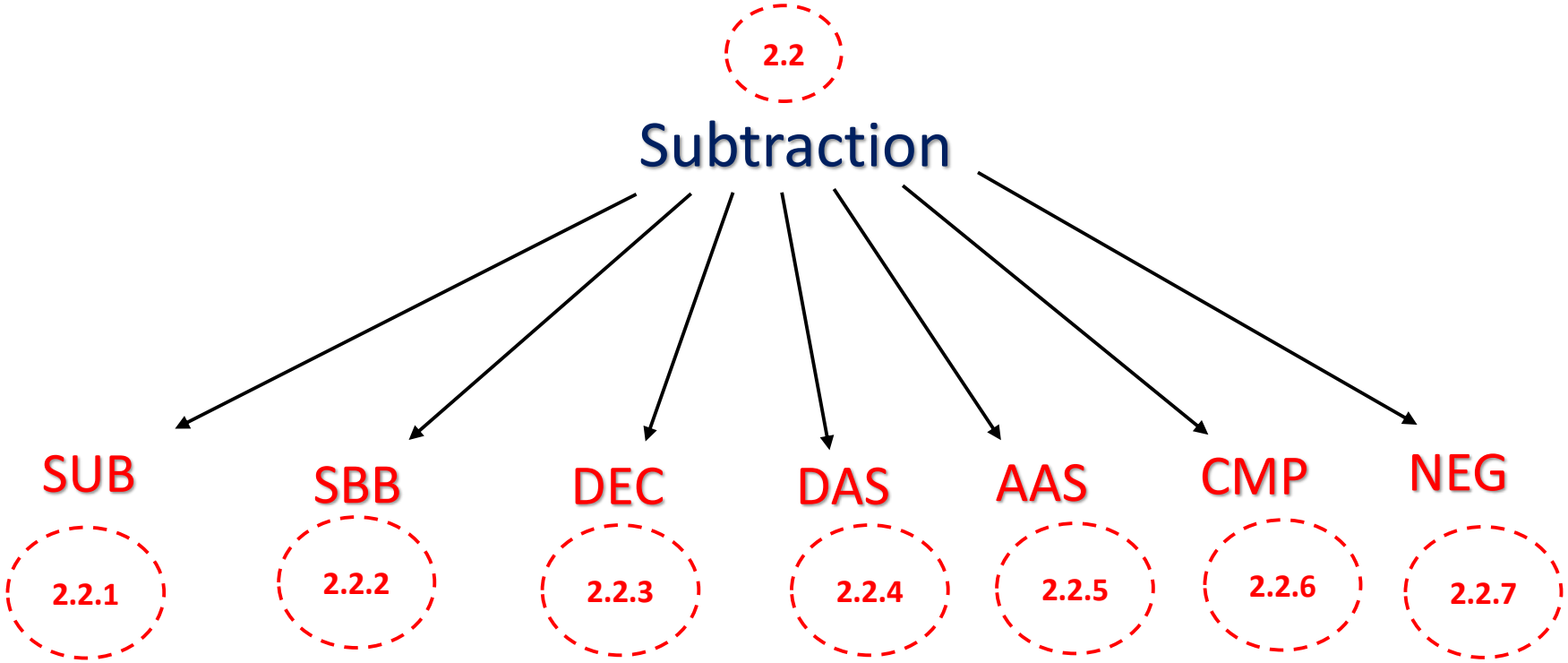
BL = 0011 1001(ASCII 9)

Rule :





2.2 Subtraction Group



2.1.1 SUB – Subtract byte or word

This instruction subtracts a number from source to number from destination and puts the result to specified destination.

Mnemonic: SUB destination , source

SUB Operand 1 , Operand 2

Operation :

Destination ← Destination - Source

Operand 1 ← Operand 1 - Operand 2

Sr. No.	Destination	Source
1	Register	Register
2	Register	Memory
3	Memory	Register
4	Register	Immediate
5	Memory	Immediate
6	Accumulator	Immediate

Flags :

All Flags affected

SUB register , register

- This instruction subtracts the data in registers.
- The result is stored in register.
- This instruction can be 8/16 bit.

Mnemonic: SUB register , register

Example: SUB BL, CL

BL ← BL - CL

Operation :

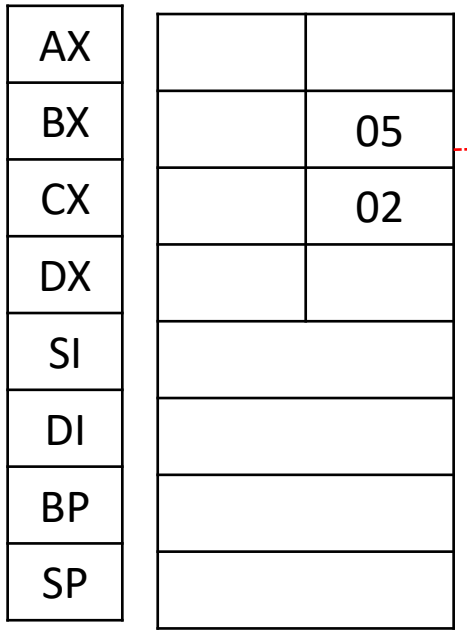
Register ← Register - Register

Addressing mode : Register addressing mode

Flags :

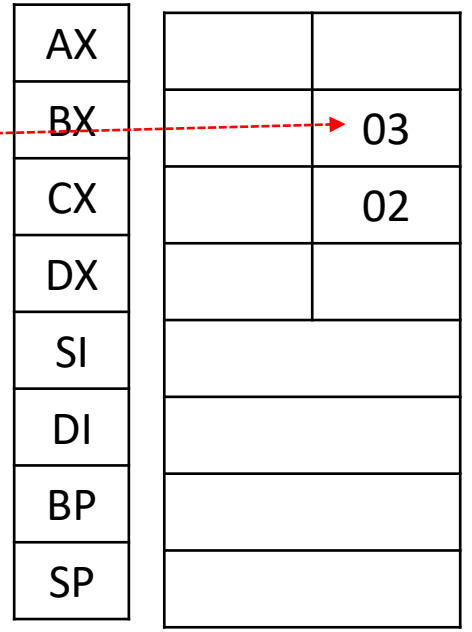
All Flags affected

Before execution CL = 02 , BL = 05



SUB BL, CL

After execution CL = 04 , BL = 07



Before Execution

After Execution

2.2.2 SBB – Subtract with borrow

- This instruction subtracts destination operand contents , source operand contents and Carry flag content.
- Result is stored back to destination operand.
- The source and destination can be 8/16 bit register or memory location. The source can also 8/16 bit register or memory location and immediate data.
- It is easy to perform multiple – precision arithmetic by using SBB instruction.

Mnemonic: SBB destination , source

Operation :

Destination ⟵ Destination - Source - CY

Example: SBB BL, CL

Flags :

All Flags affected

2.2.3 DEC – Decrement byte or word by 1

- This instruction subtracts 1 from the destination operand.
- The operand can be a register or memory location.
- The operand may be a byte or word and it is treated as an unsigned binary number.

Mnemonic: DEC destination

Operation :

Destination ← Destination - 1

Flags : All Flags affected except carry flag.(carry flag not changed)

Example: DEC CX → IF CX = 1234 , after DEC CX will be 1233

DEC AL → IF AL = 08 , after DEC AL will be 07

DEC [2000] → This instruction decrements the content of memory location 2000 by 1.

If 2000 = 04 , after DEC it will be 04

2.2.6 CMP –Compare byte or word

- This instruction compares a word/byte from source with byte/word from destination.
- The comparison is done by subtracting the source byte or word from the destination byte or word.
- The result is not stored in either of the destination or source.
- The destination and source remain unchanged, only flags are updated.

Mnemonic: CMP Destination , Source

Operation :

Destination - source

Flags : AF, OF, SF, ZF, PF are updated

Compare	CF	ZF	SF	
Source > Destination	1	0	1	Subtraction required borrow, CF =1
Source < Destination	0	0	0	No borrow required, CF =0
Source = Destination	0	1	0	Result of subtraction is zero.

******* **After each operation in this instruction carry flag changed……**

if 0 then it will become 1

if 1 then it will become 0

Example1: CMP BL, CL

Where, BL = 05 , CL = 04

Where, BL = 08 , BL = 03

05 = 0 0 0 0 0 1 0 1 : BL

04 = 0 0 0 0 0 1 0 0 : CL

+ 05 h

- 04 h

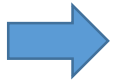
---

04 = 0 0 0 0 0 1 0 0

- 04 = 1 1 1 1 1 0 1 1

+ 1

0



2's Complement **CY = 1** **Carry changed**

---

1 1 1 1 1 1 0 0

05 = 0 0 0 0 0 1 0 1

+

- 04 = 1 1 1 1 1 1 0 0

1 1 1 1 1 1

---

0 0 0 0 0 0 0 1

CY = 0 **Carry changed**

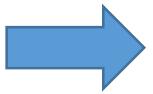
Compare	CF	ZF	SF	
Source > Destination	1	0	1	Subtraction required borrow, CF =1
Source < Destination	0	0	0	No borrow required, CF =0
Source = Destination	0	1	0	Result of subtraction is zero.

Example: CMP BL, CL

Where, BL = 04 , CL = 05

04 = 0 0 0 0 0 1 0 0 : BL

05 = 0 0 0 0 0 1 0 1 : CL



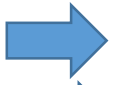
+ 04 h

- 05 h

Where, BL = 04 , BL = 06

05 = 0 0 0 0 0 1 0 1

- 05 = 1 1 1 1 1 0 1 0



1's Complement

+ 1

2's Complement

0

---

1 1 1 1 1 0 1 1

CY = 1 **Carry changed**

04 = 0 0 0 0 0 1 0 0

+

- 05 = 1 1 1 1 1 0 1 1

0

---

1 1 1 1 1 1 1 1

CY = 1 **Carry changed**

Compare	CF	ZF	SF	
Source > Destination	1	0	1	Subtraction required borrow, CF =1
Source < Destination	0	0	0	No borrow required, CF =0
Source = Destination	0	1	0	Result of subtraction is zero.

Example: CMP BL, CL

Where, BL = 05 , CL = 05

05 = 0 0 0 0 0 1 0 1 : BL

05 = 0 0 0 0 0 1 0 1 : CL

+ 05 h

+ - 05 h

Where, BL = 07 , CL = 07

05 = 0 0 0 0 0 1 0 1

- 05 = 1 1 1 1 1 0 1 0 → **1's Complement**

+ 1 → **2's Complement**

0

1 1 1 1 1 0 1 1

CY = 1 **Carry changed**

05 = 0 0 0 0 0 1 0 1

+ - 05 = 1 1 1 1 1 0 1 1

1

0 0 0 0 0 0 0 0

CY = 0 **Carry changed**

Compare	CF	ZF	SF	
Source > Destination	1	0	1	Subtraction required borrow, CF =1
Source < Destination	0	0	0	No borrow required, CF =0
Source = Destination	0	1	0	Result of subtraction is zero.

2.2.7 NEG –Negate byte or word

- This instruction replace the number is a destination with 2's complement of that number.
- The destination can be register or memory.

Example: NEG BL

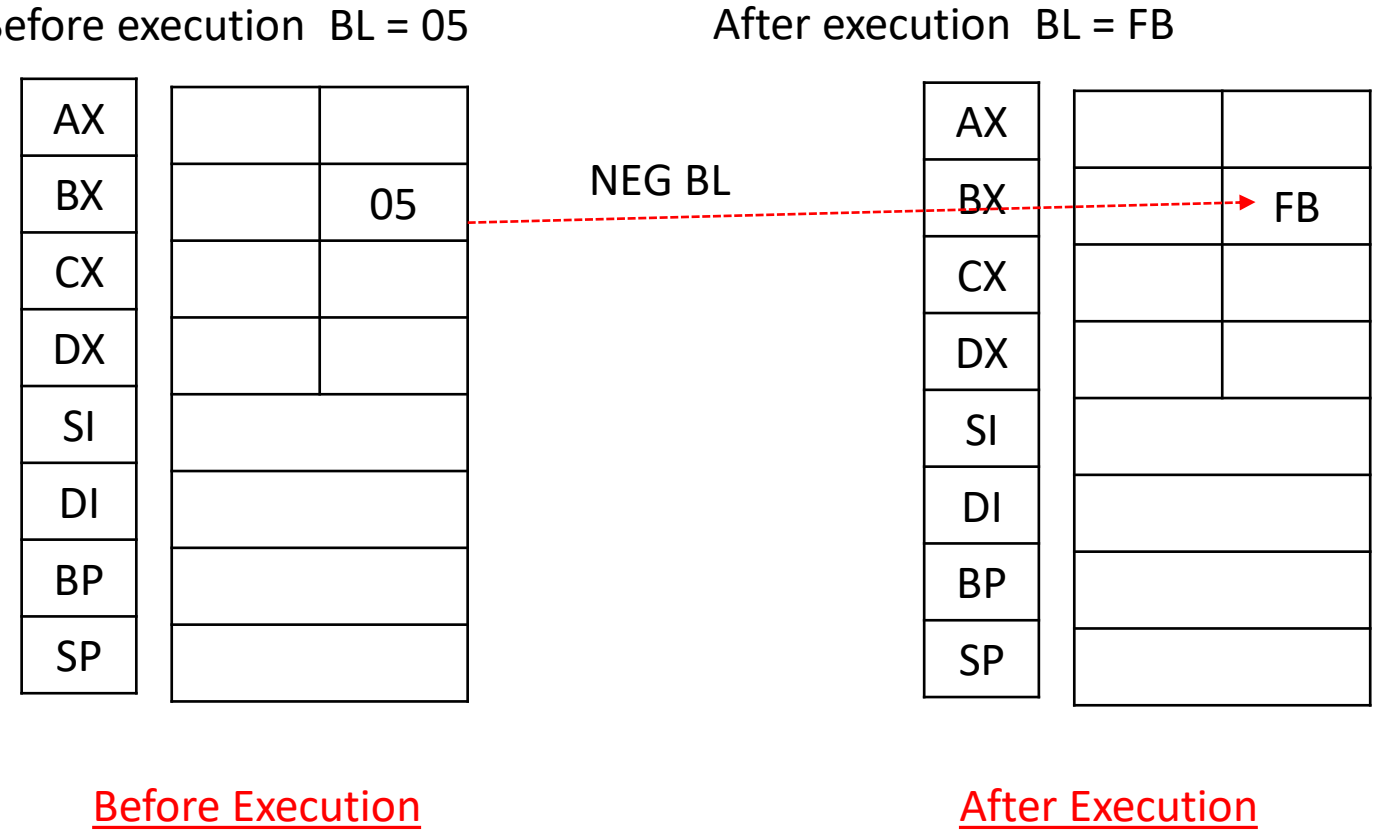
BL ⟵ 2's complement of BL

Mnemonic: NEG Destination

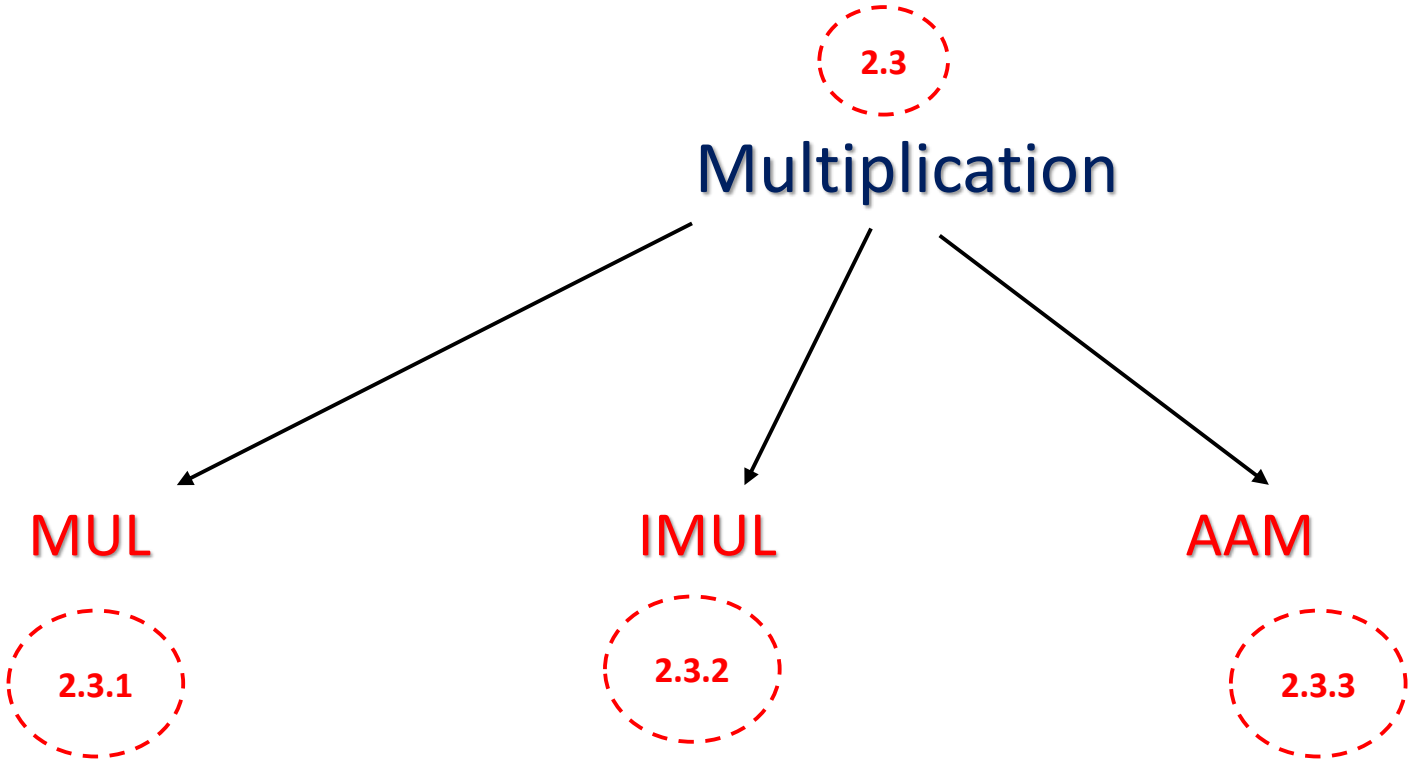
Operation :

Destination ⟵ 2's complement of Destination

Flags : All flags are updated



2.3 Multiplication Group



2.3.1 MUL – Multiply byte or word unsigned

- This instruction multiplies an unsigned byte from source with an byte in the AL register or an unsigned word from source with an unsigned word in AX.
- When a byte is multiplied by contents of AL, the result is stored in AX.
- The MSB of result is stored in AH register and the LSB of result is stored in the AL register.
- When a word is multiplied by contents of AX, then MSB of result is stored in DX register and the LSB of result is stored in the DX register.

Example: MUL BL

Where AL = 09 H,
BL = 02 H

Operation :

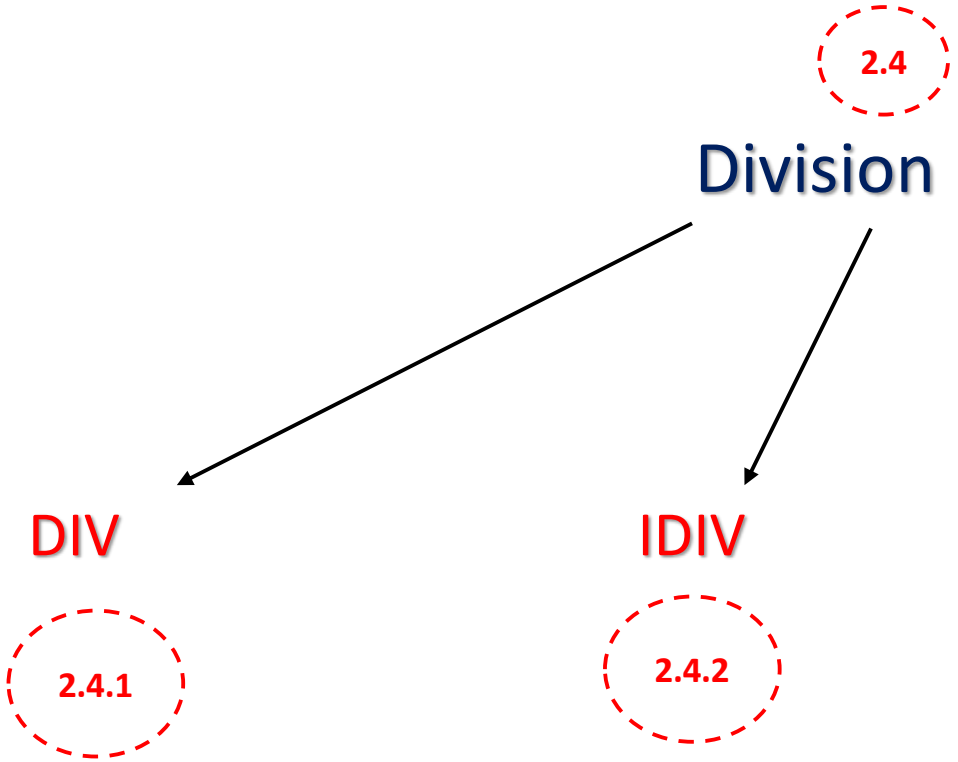
Mnemonic: MUL multiplier

AH = 00 H
AL = 12 H

X **09 h**
02 h
0012 h

Flags : AF,PF,SF,ZF undefined
CF and OF will both be 0

2.4 Division Group



2.4.1 DIV – Multiply byte or word unsigned

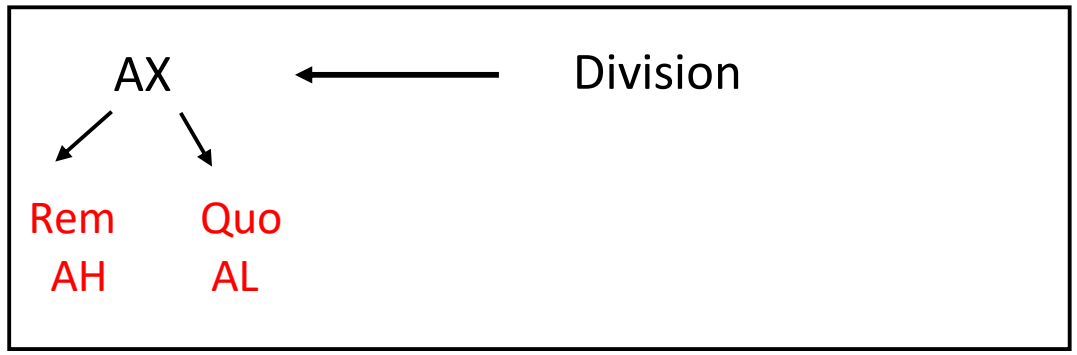
- This instruction divides an unsigned byte from source with an byte in the AL register or an unsigned word from source with an unsigned word in AX.
- When a byte is divided , the result is stored in AX.
- The remainder of result is stored in AH register and quotient of result is stored in the AL register.
- For 16 bit operation DX register is used.

Example: DIV BL

Where AL = 08 H,
BL = 02 H

Operation :

Mnemonic: DIV Divider



08 h
/ **02 h**
---
04 h → Q

AH = remainder
AL = quotient

Addressing modes of 8086

It is a manner in which an operand is given in instruction

1. Immediate addressing mode (Data in Instruction)

2. Register addressing mode (Data in register)

3. Direct addressing mode (Address in Instruction)

4. Indirect addressing mode (Address in Register)

5. Implied addressing mode (Nothing is given in instruction)

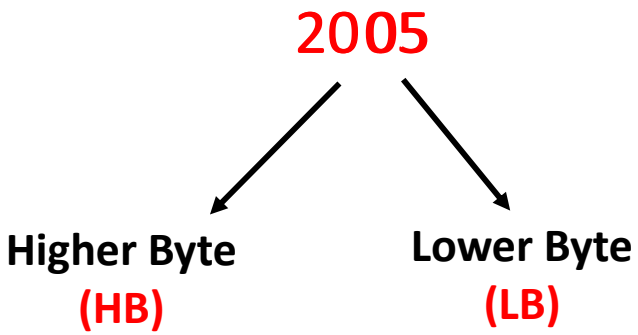
1. Immediate addressing mode (Data in Instruction)

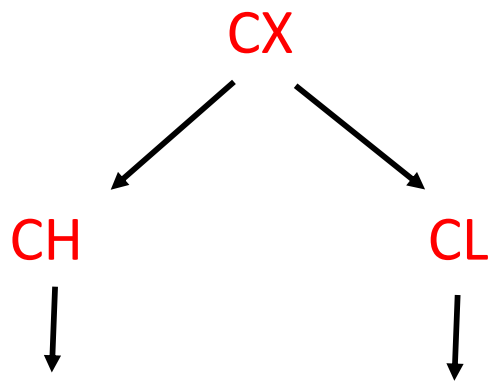
In immediate addressing mode the data to be used is immediately given in the instruction.

Example :

MOV CL, 02 H ⟶ 02 (8 bit data) is transfer into reg. CL

MOV CX, 2005 H ⟶ 2005 (16 bit data) is transfer into reg. CX in following manner :





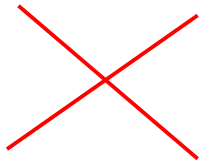
2. Register addressing mode (Data in register)

In register addressing mode data to be operand is in general purpose register

Example :

MOV CL, BL → Content (8 bit data) of reg. BL is transferred into reg. CL

MOV CX, BX → Content (16 bit data) of reg. BX is transferred into reg. CX

MOV CL, BX  It is not possible to perform 16 bit operation with 8 bit reg or vice versa.

3. Direct addressing mode (Address in Instruction)

In direct addressing mode operand is given by a direct address where the data is present.

Anything in [ ] refers address

Example 1 :

MOV CL, [2000] → The Content which present at memory location 2000 transfer into reg. CL

CL ← DS : [2000]

DS

0000	1A
0001	08
.	
.	
2000	04
2001	05
2002	06
.	
FFFF	

→ CL = 04

Note :

- Data is always refer from data segment (DS)
- DS has starting address
- From 0000 to FFFF these are the offset
- BIU section of 8086 generate 20 bit physical address using following formula :

PA = Seg address * 10 h + offset

Direct addressing mode (Address in Instruction)

In direct addressing mode operand is given by a direct address where the data is present

Example 2 :

MOV [2000], CL ⟶ The Content of reg. CL transfer into memory location 2000

CL ⟶ DS : [2000]. assume CL =07

DS

0000	1A
0001	08
.	
.	
2000	~~04~~ 07
2001	05
2002	06
.	
FFFF	

Note :

- Data is always refer from data segment (DS)
- DS has starting address
- From 0000 to FFFF these are the offset
- BIU section of 8086 generate 20 bit physical address using following formula :

PA = Seg address * 10 h + offset

Direct addressing mode (Address in Instruction)

In direct addressing mode operand is given by a direct address where the data is present

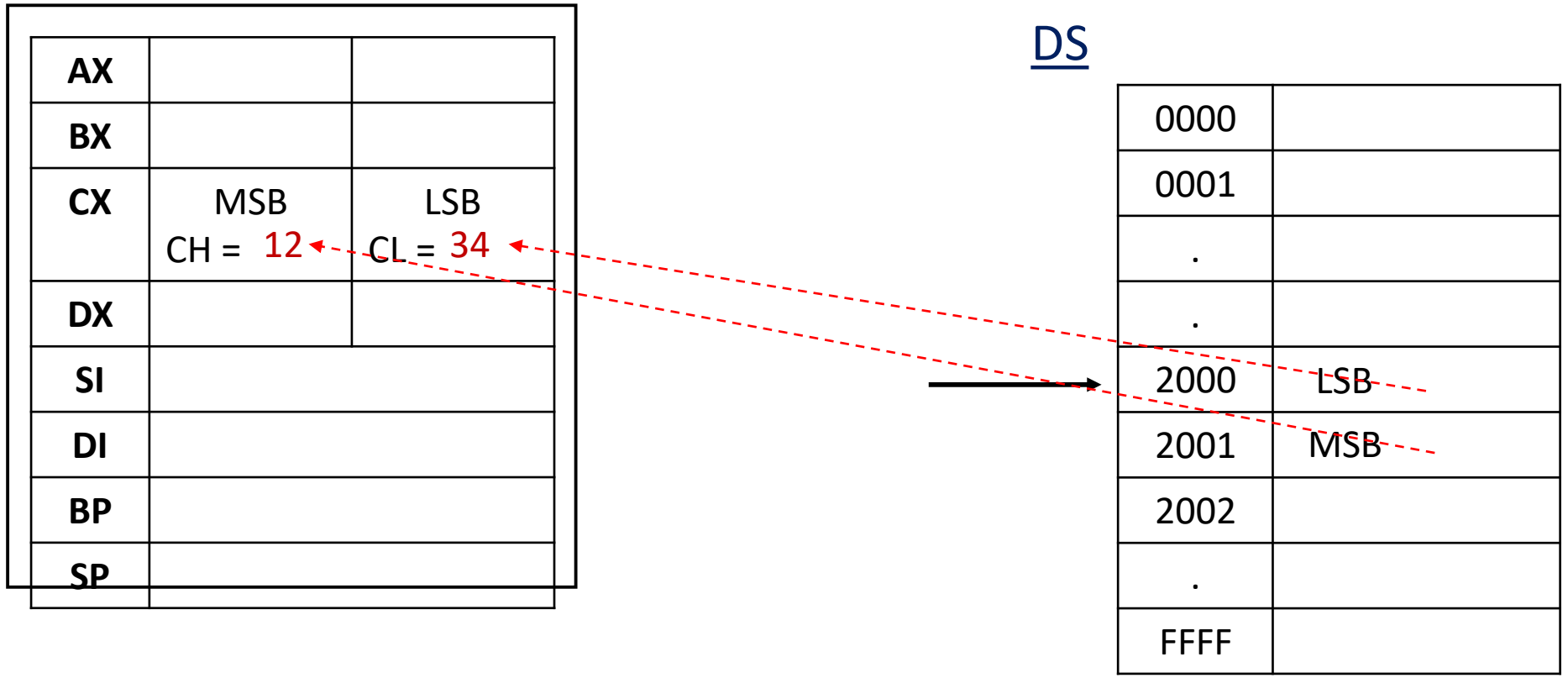
Example 3 : MOV CX , [2000] ⟶ The Content of reg. CX transfer into memory location 2000

CX ⟵ DS : [2000]

16 bit data 1234 is stored into memory locations in following manner :

12 34

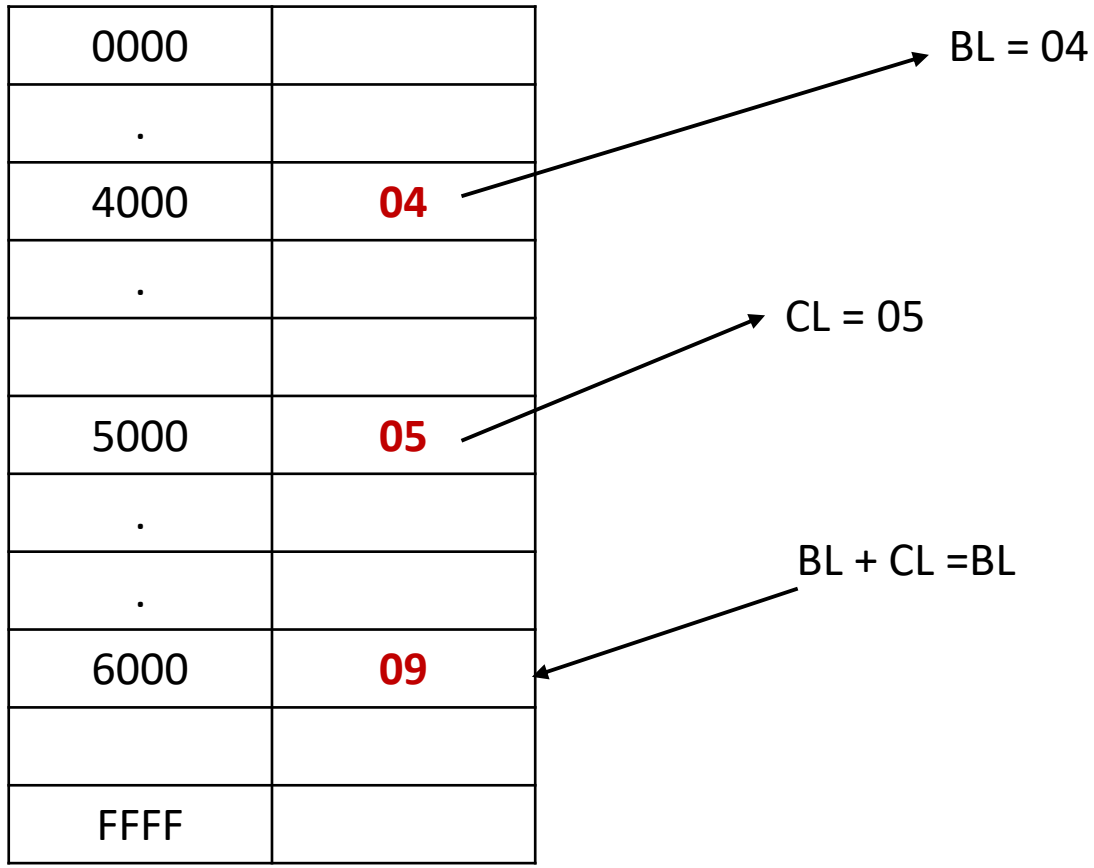
MSB LSB



Direct addressing mode (Address in Instruction)

1. Move content of 4000 th location into BL register
2. Move content of 5000 th location into CL register
3. Add contents of BL and CL
4. Store the result on 6000 th memory location

DS



1. Mov BL, [4000]
2. Mov CL, [5000]
3. Add BL, CL
4. Mov [6000], BL

4. Indirect addressing mode (Address in register)

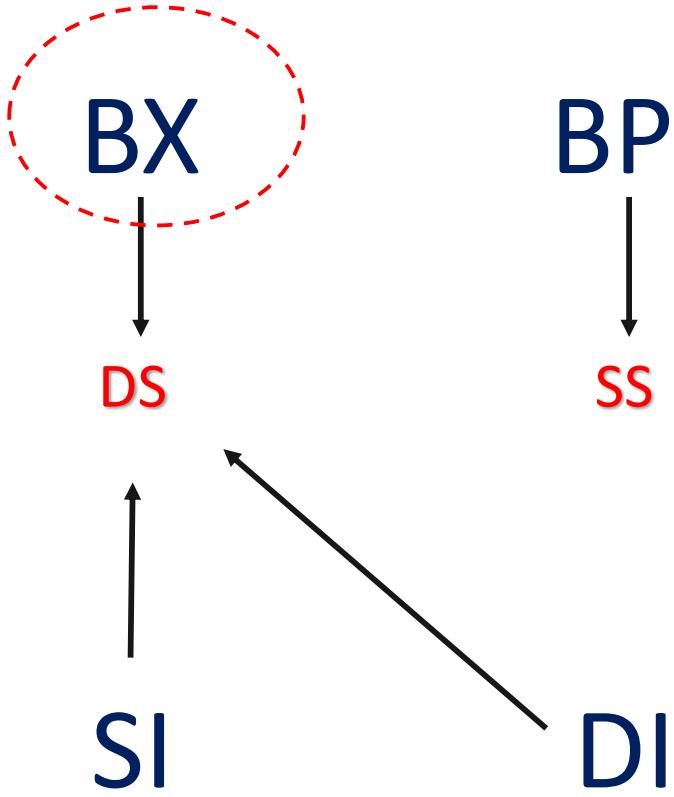
In indirect addressing mode the instruction dose not have the address of the data to be operated on. But the instruction points where the address is stored i.e. it is indirectly specifying the address of memory location where the data is stored or is to be stored.

There are four types of indirect addressing mode :

1. Register Indirect (address simply given by register)
2. Register relative (address in reg + relative)
3. Based indexed (address in base reg + index reg)
4. Based relative indexed (address in base reg + index reg + relative)

Rules related with register

Only BX register is used



1.Register Indirect addressing mode (Address in reg)

In direct addressing mode operand is given by a direct address where the data is present

Example : MOV CL,[BX] ⟶ The Content of memory location 4000 transfer into CL

CL ⟵ DS : [4000].

Note :

- Only BX register is used

DS

0000	1A
0001	08
.	
.	
4000	04
4001	05
4002	06
.	
FFFF	

CL = 04

MOV BX, 4000H Load 4000 H immediate data into reg BX

MOV CL,[BX] Now 4000 H will be treated as a memory location and content of this location will be transfer into reg. CL

Difference between direct and indirect addressing :

Direct addressing :

Mov BX, [4000]

CL ⟵ DS : [4000]

DS

0000	1A
0001	08
.	
.	
4000	07
4001	05
4002	06
.	
FFFF	

Indirect addressing :

Mov BX, 4000H ⟶ Initialisation of BX with 4000

Mov CL, [BX]

CL ⟵ DS : [4000]

Advantage of Indirect addressing:

1. Suppose we want to transfer 100 location from 4000 then using direct addressing mode we have to write 100 times above instruction for e.g. MOV CL,[4001] , MOV CL,[4002] and so on.

2. By using indirect addressing we can implement following code

```
Mov BX, 4000H
Mov CL, [BX]
INC BX
```

In C prog:

```
Arr[0]      If x = a[5], x will get data at arr[5]
....        If x = a[9], x will get data at arr[9]
Arr[9]      Direct addressing :
```

```
i = 5
x = a[i]    Indirect addressing :
```

2.Register relative addressing mode (Address in reg + relative)

In this mode, the operand address is calculated using one of the base registers and an 8 bit or a 16 bit displacement.

Example : MOV CL,[BX + displacement] → The Content of memory location 4002 transfer into CL

MOV CL,[BX + 02 h]

CL ← DS : [4000 + 02].

MOV CL,[BX + displacement] Increment memory locations by displacement

MOV CL,[BX - displacement] Decrement memory locations by displacement

Note : • Only BX register is used

DS

	0000	1A
	0001	08
	.	
	.	
BX →	4000	04
	4001	05
BX + 02h →	4002	06 → CL = 06
	.	
	FFFF	

Example : MOV CL, [BX - 03 h]

DS

	0000	1A
	0001	08
	.	
	.	
BX - 03h →	4000	04 → CL = 04
	4001	05
	4002	06
BX →	4003.	08
	FFFF	

3.Based Index addressing mode (Address in base reg + Index)

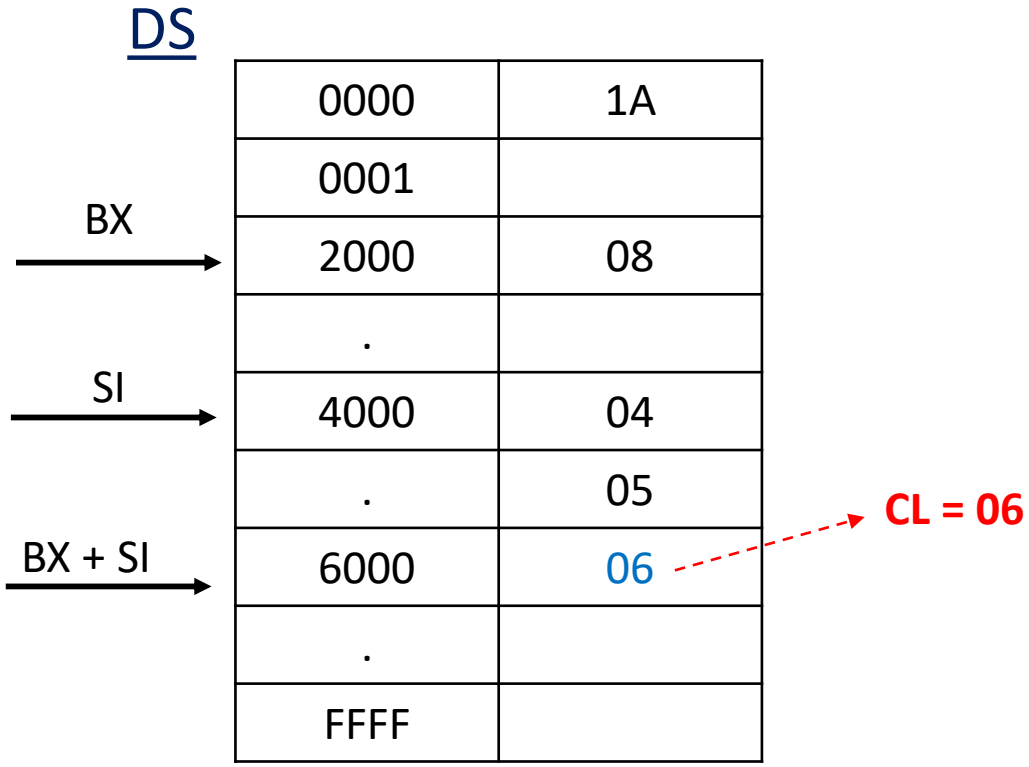
In this mode, operand address is calculated as base register plus an index register and an 8 bit or a 16 bit displacement.

Example : MOV CL,[BX + SI] ⟶ This instruction moves a byte from the address pointed by BX + SI in data segment to CL.

CL ← DS : [4000 + 2000].

Note :

- Only BX register is used



4.Based relative Index addressing mode (Address in base reg + Index + relative)

In this mode, operand address is calculated as base register plus an index register and 8 or 16 bit displacement.

Example :

MOV CL,[BX + SI + displacement] ⟶ This instruction moves a byte from the address pointed by BX + SI + 02 in data segment to CL.

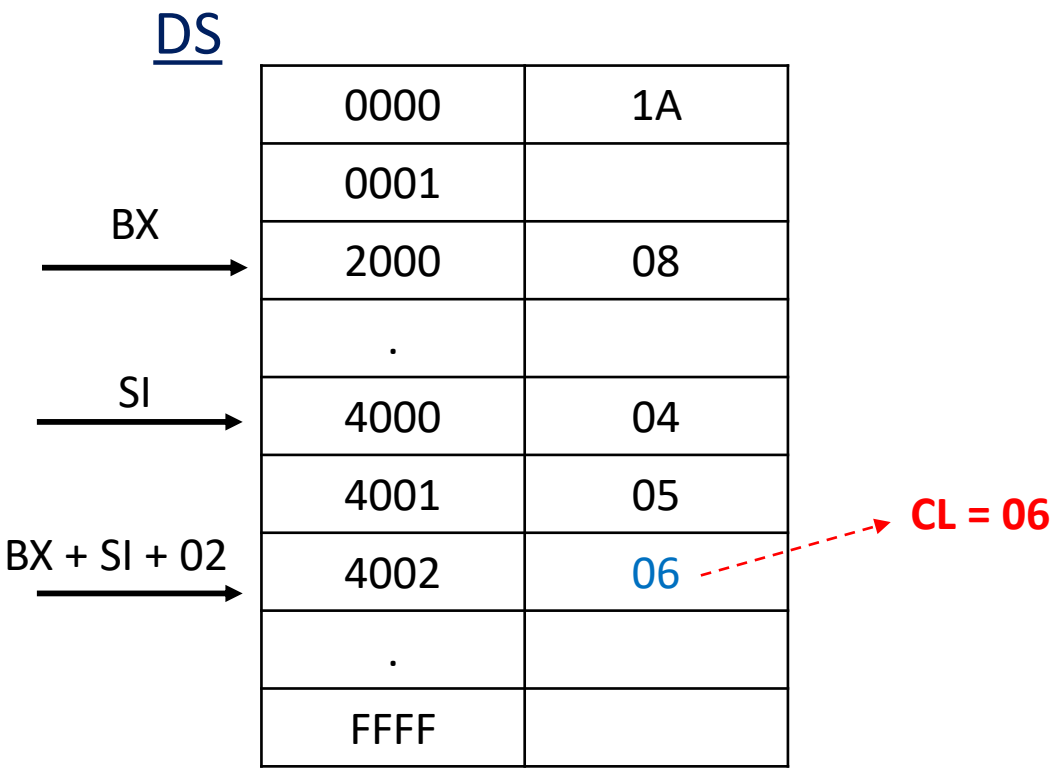
MOV CL,[BX + SI + 02h]

CL ⟵ DS : [2000 + SI + 02].

Note : • Only BX register is used

MOV CL,[BX +SI + displacement] Increment memory locations by displacement

MOV CL,[BX + SI - displacement] Decrement memory locations by displacement



5. Implied addressing mode (Nothing is given in instruction)

In this mode, the operands are implied and are hence not specified in the instruction.

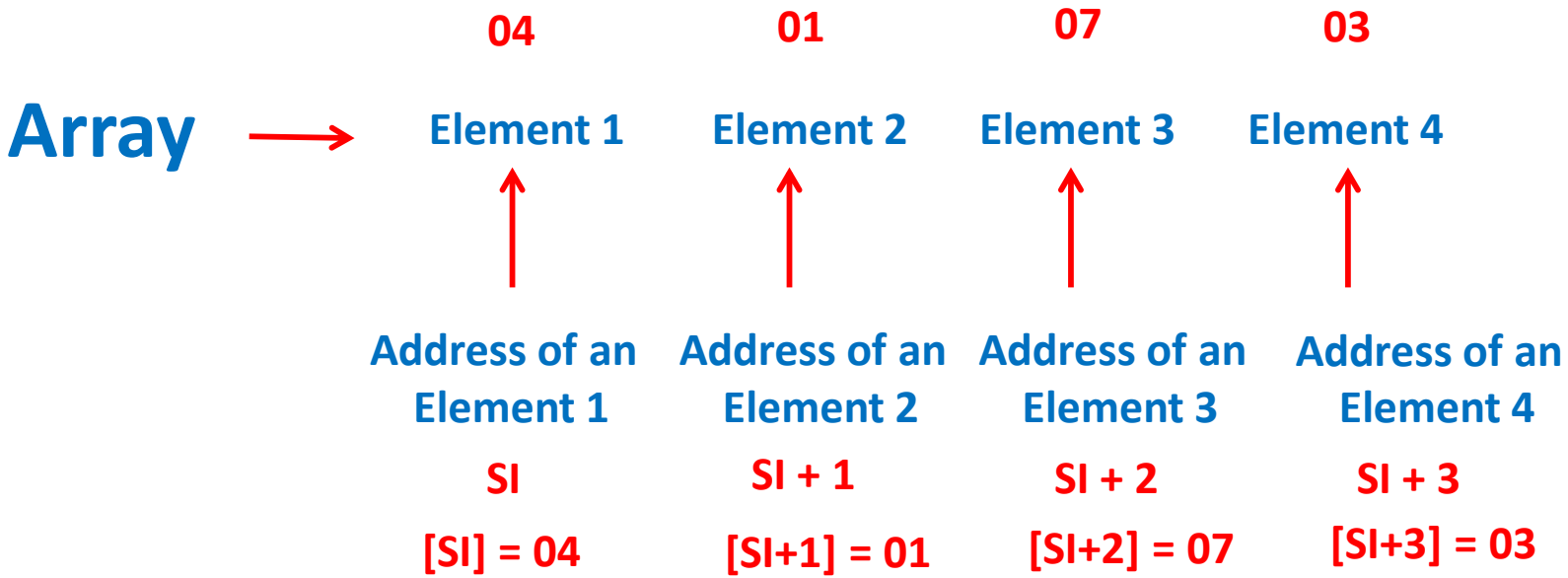
Example :

Operations are related with specific register

STC ⟶ Set carry flag

DAA ⟶ Decimal adjust after addition

Create an array using TASM



Array is stored into the data segment, hence each element having address.

SI is the pointer for data segment.

Assign offset address of an array to SI pointer.

Steps to create an array :

Create an array variable and put elements into it : Arr db 04,01,07,03H

Assign offset address with SI pointer: Mov si , offset Arr

OR

LEA si , Arr

Transfer first element of an array into reg : Mov al , [si]

For next element : Mov al , [si + 1] OR Inc si
Mov al , [si]

DS

SI →	2001	04
SI+1 →	2002	01
SI+2 →	2003	07
SI+3 →	2004	03

SI = 2001

[SI] = [2001] = 04

WAP to create an array and print all elements of an array

```
.model small
.data
arr db 05,05,03,04,05h
.code
   mov ax,@data
   mov ds,ax
   mov si , offset  arr or  lea si, arr
   mov dh,05h
back: mov al,[si]
   inc si
   mov ch,02h
   mov cl,04h
   mov bh,al
i2: rol bh,cl
   mov dl,bh
   and dl,0fh
   cmp dl,09
   jbe i4
   add dl,07h
i4: add dl,30h
   mov ah,02h
   int 21h
   dec ch
   jnz i2
   mov dl,' '
   mov ah,02h
   int 21h
   dec dh
   jnz back
   mov ah,4ch
   int 21h
end
```

1

Initialization of an array

2

Assign offset address

Counter for an elements 3

Mov first element into al 4

Increment SI pointer by 1 to point next element

Print first element using 30-37 logic 5

Assign blank space between two element 5

Decrement counter of an element, if zero then terminate program else print second element 6

Dh = 05 Dh = 04 Dh = 03 Dh = 02 Dh = 01 Dh = 00

Al = 05

Al = 05

Al = 03

Al = 04

Al = 05

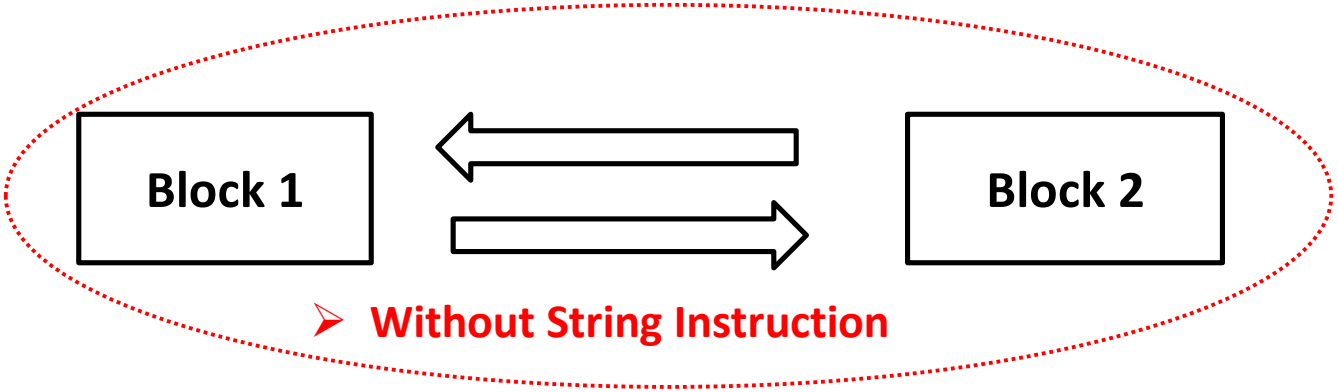
05 05 03 04 05

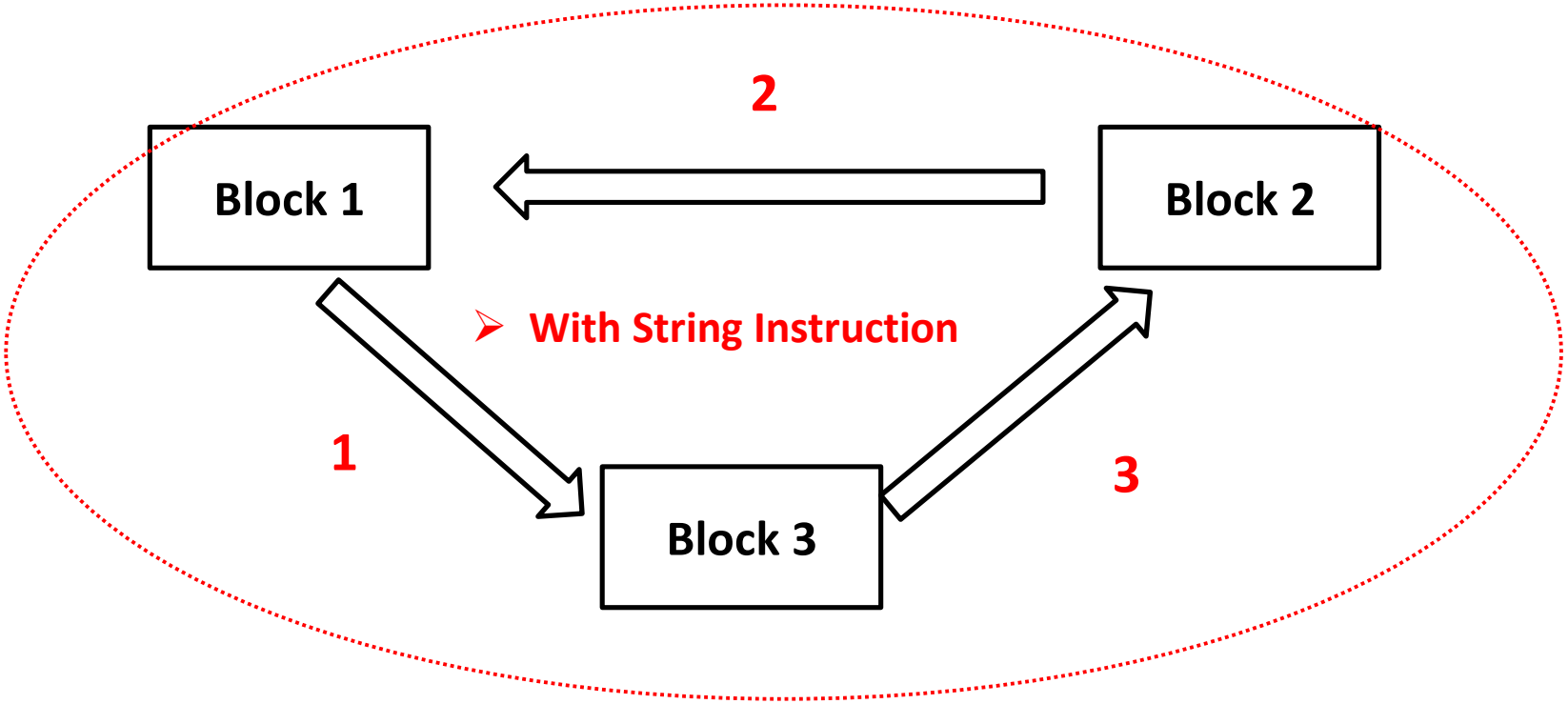
Terminate program

DS

SI →	2001	05
SI+1 →	2002	05
SI+2 →	2003	03
SI+3 →	2004	04
SI+4 →	2005	05

Block Exchange





➢ Without String Instruction

➢ With String Instruction

Block Exchange without string instruction

WAP to exchange 05 data bytes present at two memory blocks 1000 and 2000 onwards respectively without using string instructions

Block 1

SI →	1000	01
SI+1 →	1001	02
	1003	03
	1004	04
	1005	05

Block 2

DI →	2000	05
DI+1 →	2001	04
	2003	03
	2004	02
	2005	01

[SI] = 01 & [DI] = 05

Mov content of SI in AL & content of DI in AH

AL = 01 & AH = 05

Exchange the content of AL and AH

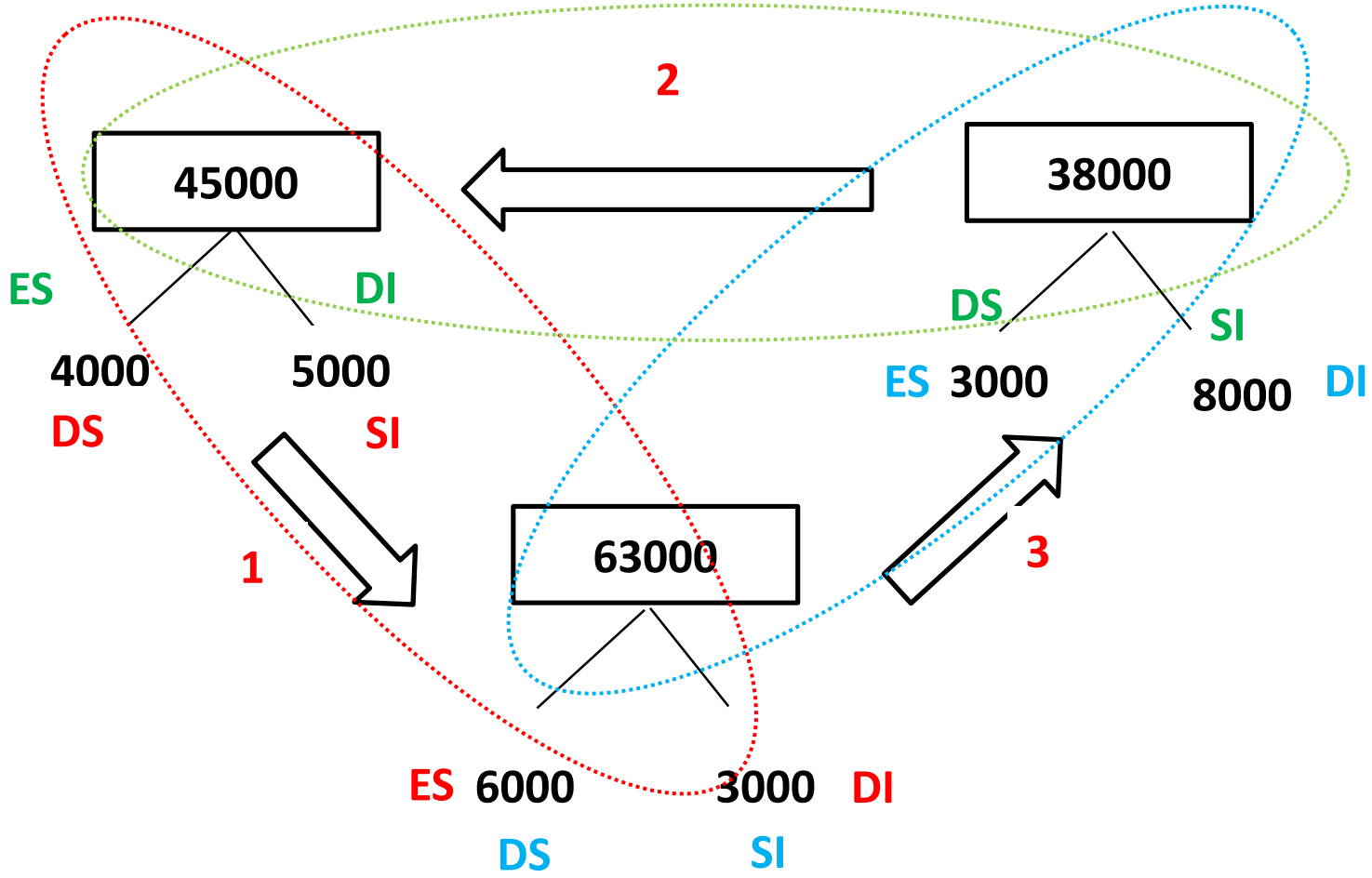
AL = 05 & AH = 01

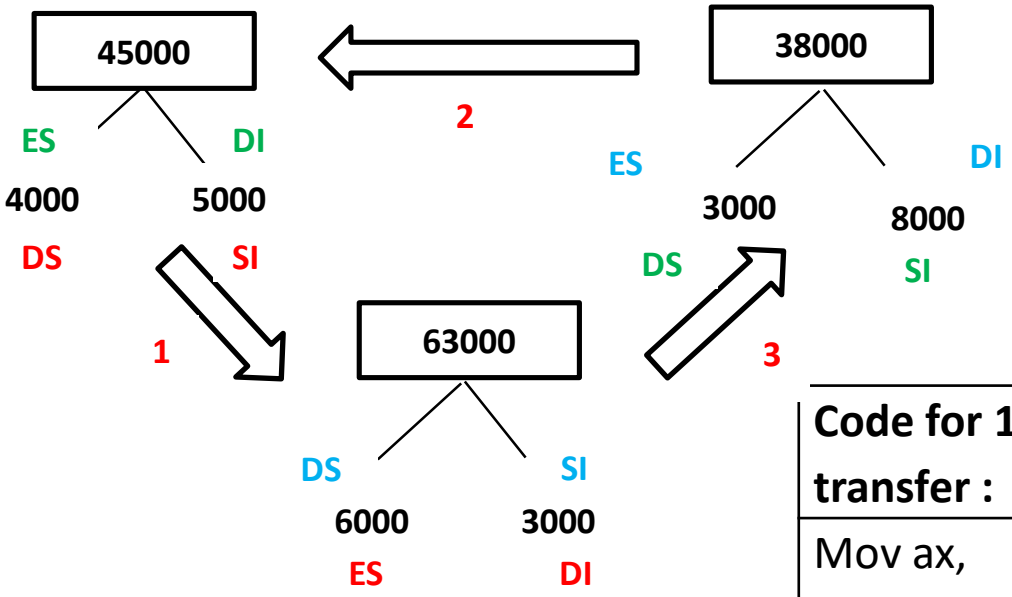
Store the exchanged value into the blocks

Increment SI and and DI by 1 to point next memory location

Block Exchange string instruction

WAP to exchange 150 data bytes present at two memory blocks 45000 and 38000 onwards respectively using string instructions



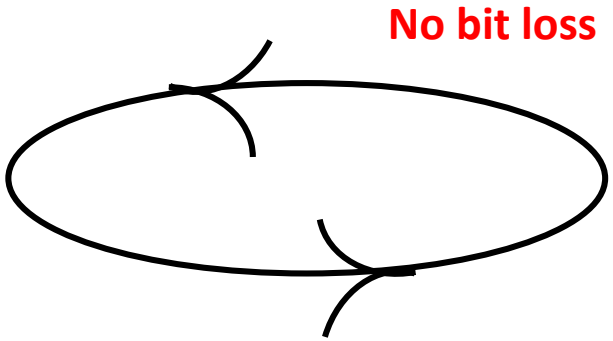


Code for 1st transfer :	Code for 2nd transfer :	Code for 3rd transfer :
Mov ax, 4000	Mov ax, 3000	Mov ax, 6000
Mov ds, ax	Mov ds, ax	Mov ds, ax
Mov ax, 6000	Mov ax, 4000	Mov ax, 3000
Mov es, ax	Mov es, ax	Mov es, ax
Mov si, 5000	Mov si, 8000	Mov si, 3000
Mov di, 3000	Mov di, 5000	Mov di, 8000
Mov cx, 150	Mov cx, 150	Mov cx, 150
Cld	cld	Cld
REP Movsb	REP Movsb	REP Movsb

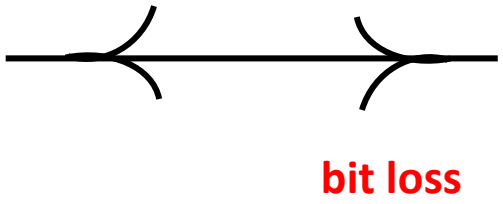
Rotate and Shift Instructions

Difference between rotate and shift

Rotate: It is a circular movement



Shift: It is a horizontal movement



Rotate Instructions:

1. **ROL** : Rotate bits to left
2. **ROR** : Rotate bits to right
3. **RCL** : Rotate bits to left along with carry
4. **RCR** : Rotate bits to right along with carry

NOTE :

1. Always CL reg is used to store the count value for rotation
2. If count is 01 then directly used along with instruction i.e. ROL BL, 01h
3. If count is other than 01 then CL reg is used to store the count value

example :
```
MOV CL, 04h
ROL  BL, CL
```

example :
```
MOV CL, 04h
ROL  BX, CL
```

CL is sufficient for both 8 bit as well as 16 bit data because maximum 8 bit rotation is 8 times and for 16 bit 16 times

1. ROL : Rotate bits to left

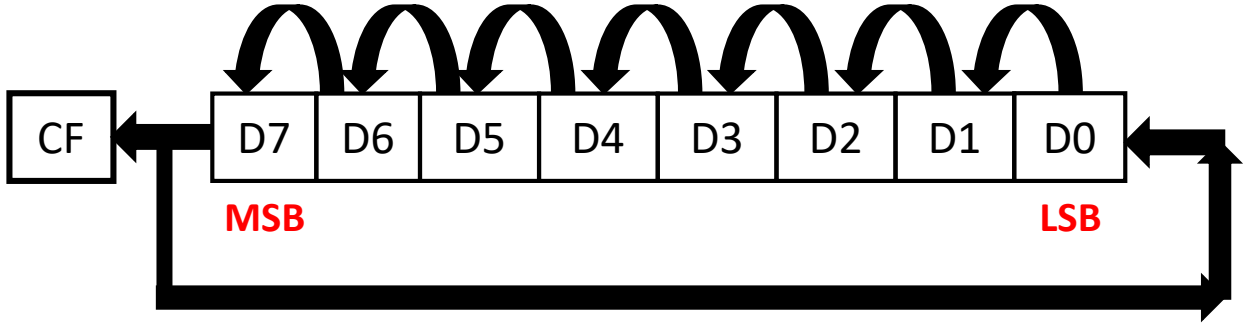
Example:

```
MOV CL, 01h
ROL  BL, CL
```

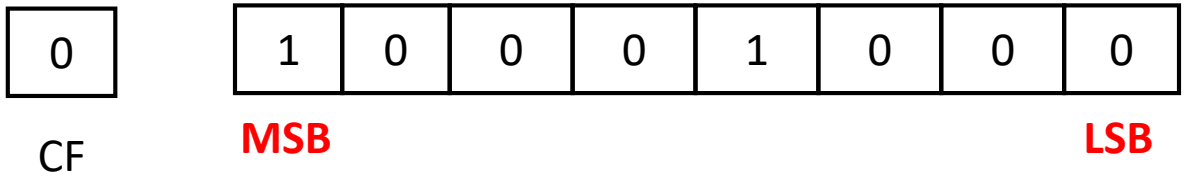
1. This instruction is used to rotate 8 bits and 16 bits to the left.
2. MSB moves into the LSB
3. MSB also copies into the carry flag

Mnemonic: **ROL destination , Count**

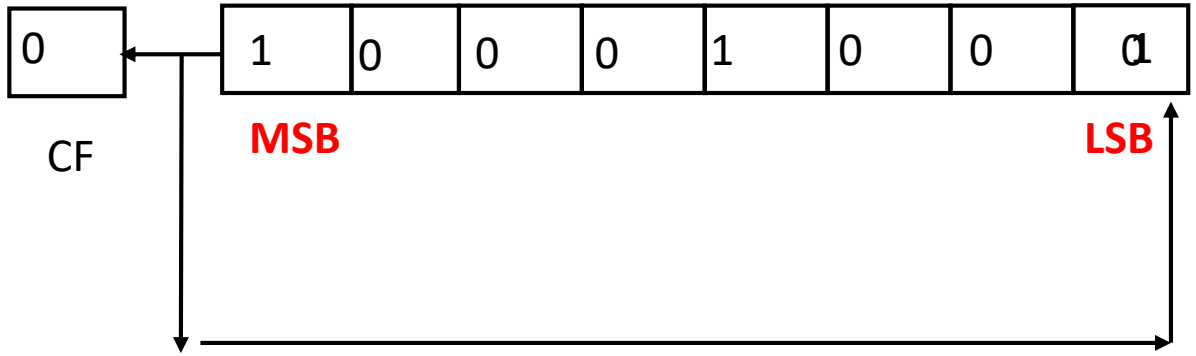
Operation :



Before Execution **BL= 88 h**



After Execution **BL= 11 h**



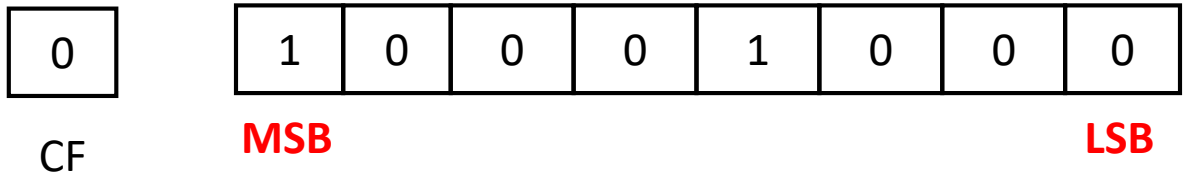
2. ROR : Rotate bits to right

Example:

```
MOV CL, 01h
ROR  BL, CL
```

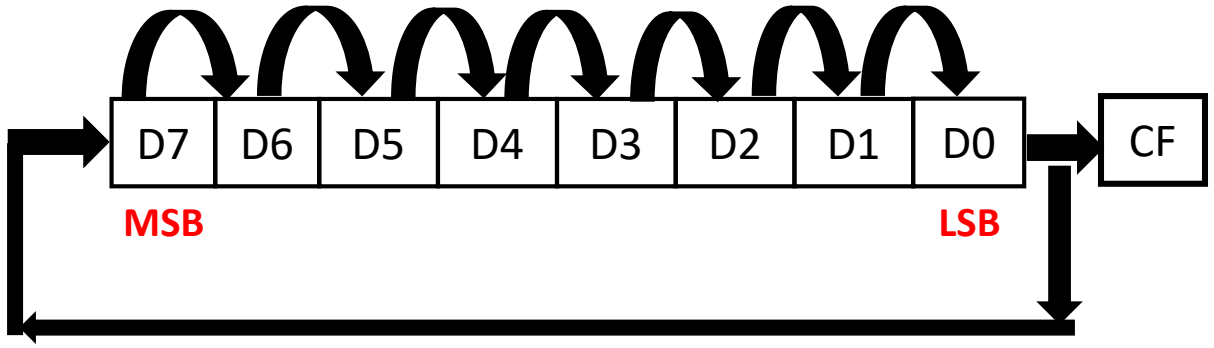
1. This instruction is used to rotate 8 bits and 16 bits to the right.
2. LSB moves into the MSB
3. LSB also copies into the carry flag

Before Execution **BL= 88 h**

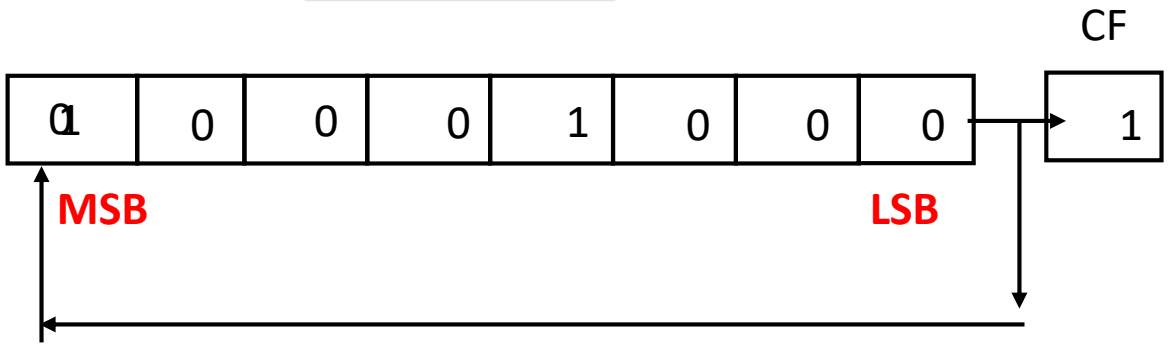


Mnemonic: **ROR destination , Count**

Operation :



After Execution **BL= 44 h**



3. RCL : Rotate bits to left with carry

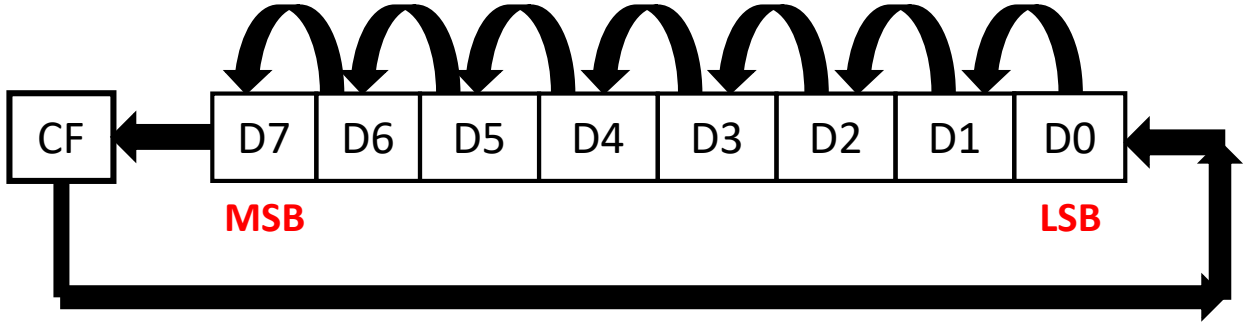
Example:

```
MOVE CL, 01h
RCL  BL, CL
```

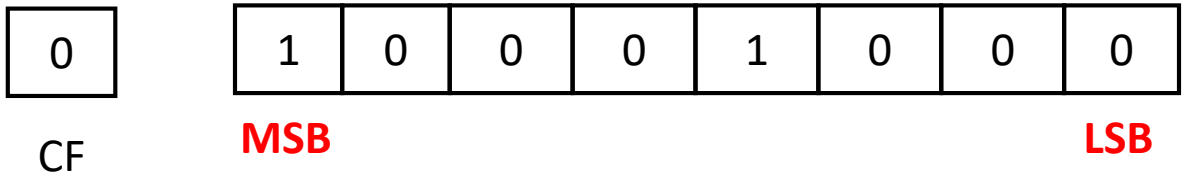
1. This instruction is used to rotate 8 bits and 16 bits to the left along with carry
2. LSB moves into the carry flag
3. Previous carry flag moves into the LSB

Mnemonic: **RCL destination , Count**

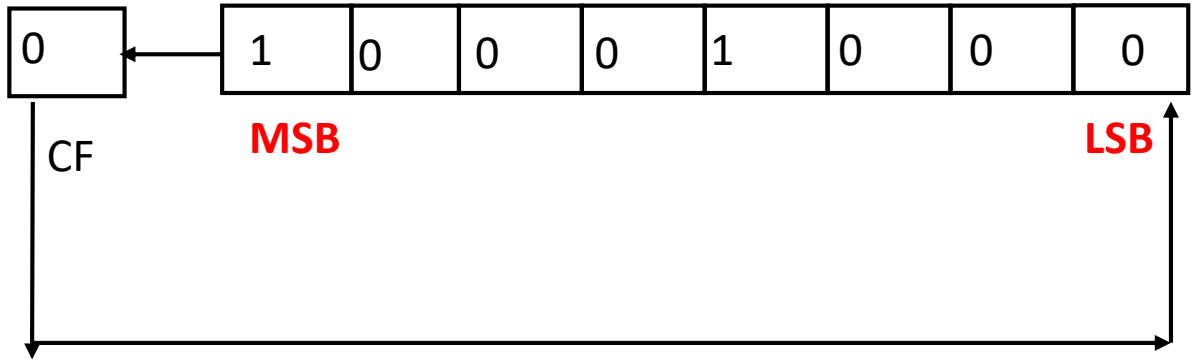
Operation :



Before Execution **BL= 88 h**



After Execution **BL= 10 h**



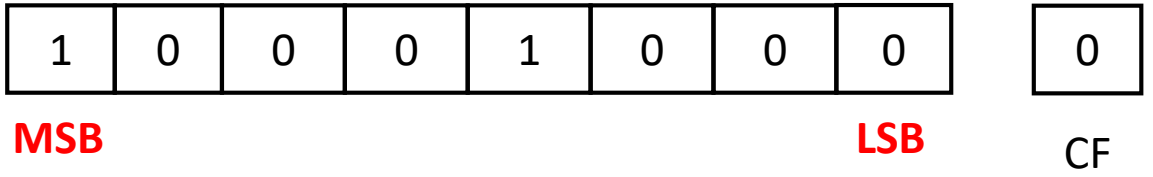
4. RCR : Rotate bits to right with carry

Example:

```
MOVE CL, 01h
ROR  BL, CL
```

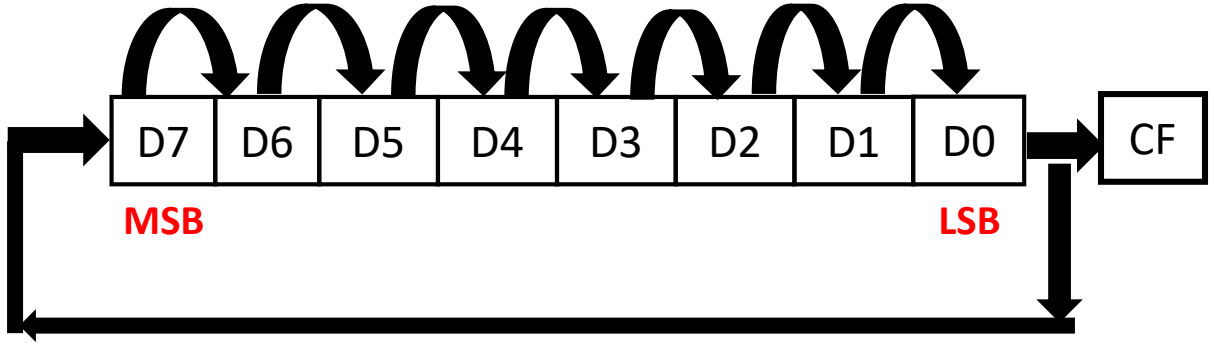
1. This instruction is used to rotate 8 bits and 16 bits to the right.
2. LSB moves into the MSB
3. LSB also copies into the carry flag

Before Execution **BL= 88 h**

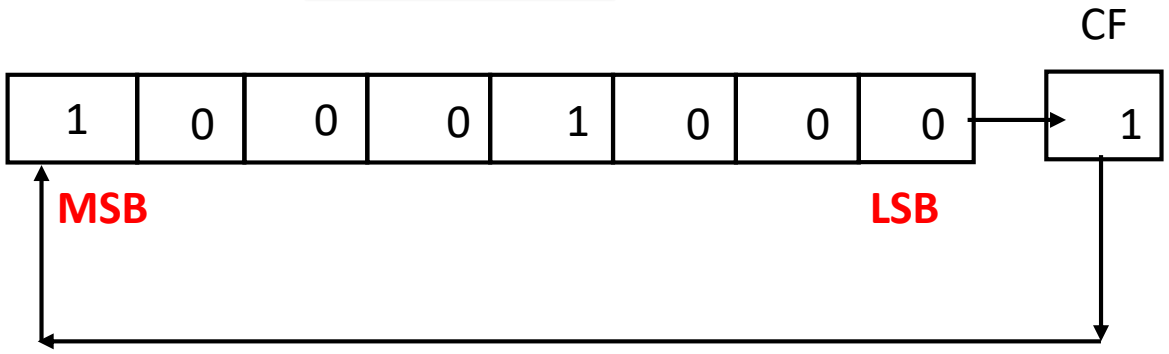


Mnemonic: RCR destination , Count

Operation :



After Execution **BL= C4 h**



Shift Instructions:

1. **SHL/SAL** : Shift bits to left
2. **SHR** : Shift bits to right
3. **SAR** : Shift arithmetic right

NOTE :

1. Always CL reg is used to store the count value for shifting
2. If count is 01 then directly used along with instruction i.e. SHL BL, 01h
3. If count is other than 01 then CL reg is used to store the count value

example :
```
MOV CL, 04h
SHL  BL, CL
```

example :
```
MOV CL, 04h
SHL  BX, CL
```

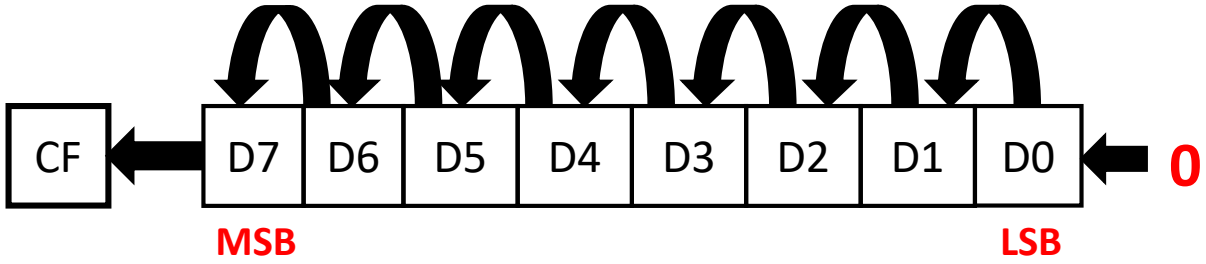
CL is sufficient for both 8 bit as well as 16 bit data because maximum 8 bit rotation is 8 times and for 16 bit 16 times

1. SHL/SAL : Shift bits to left

1. This instruction is used to shift 8 bits and 16 bits to the left.
2. MSB shifted into the carry.
3. LSB gets a 0.

Mnemonic: SHL/SAL destination , Count

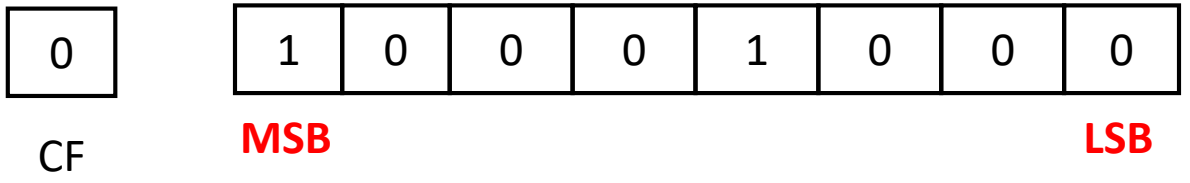
Operation :



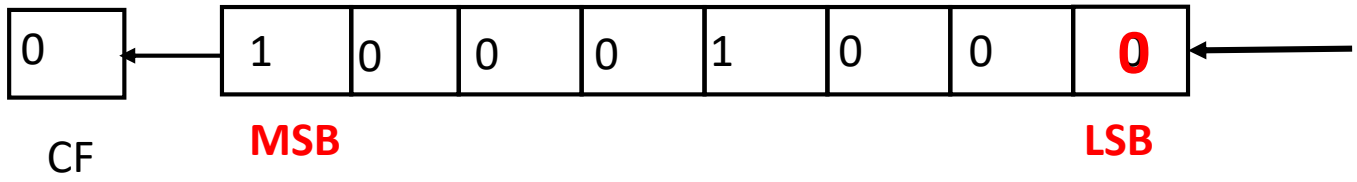
Example:

```
MOV BL, 88 h
MOV CL,01h
SHL  BL, CL
```

Before Execution **BL= 88 h**



After Execution **BL= 10 h**



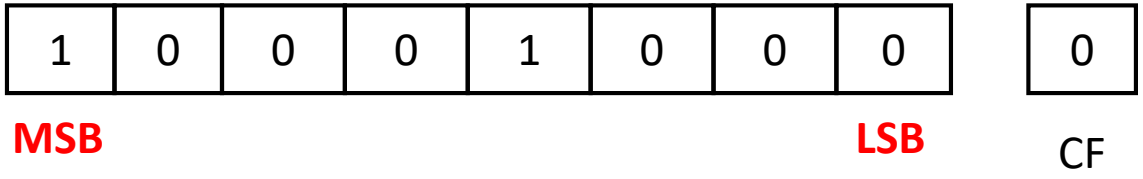
2. SHR : Shift bits to right

Example:

```
MOVE CL, 01h
SHR  BL, CL
```

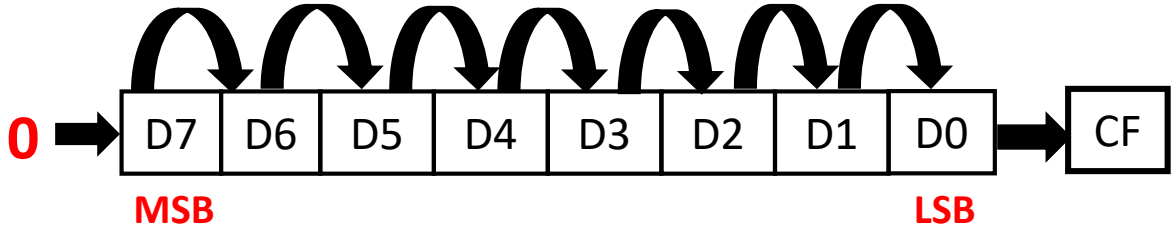
1. This instruction is used to shift 8 bits and 16 bits to the right.
2. LSB shifted into carry flag.
3. MSB gets 0.

Before Execution **BL= 88 h**

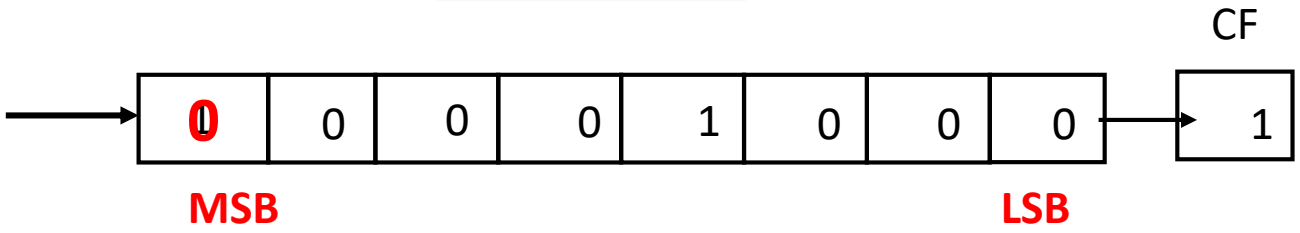


Mnemonic: SHR destination , Count

Operation :



After Execution **BL= 44 h**



Hence this instruction numbers are assumed to be unsigned

By putting 0 in MSB, going to change the sign of number i.e. makes the no. +ve

If we don't want to change the sign of number which means if +ve then +ve and if –ve then –ve (signed numbers) then use next shift instruction.

3. SAR : Shift Arithmetic right

Example:

```
MOVE CL, 01h
SHR  BL, CL
```

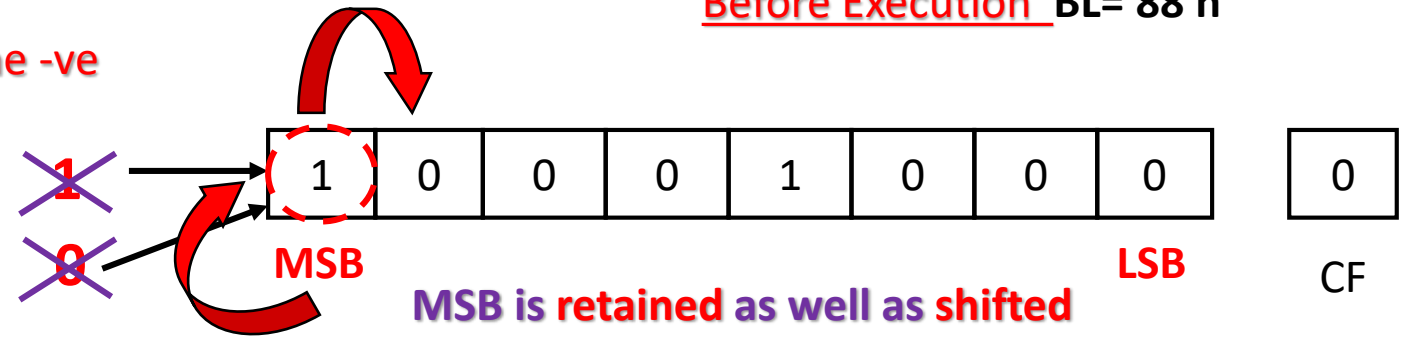
1. This instruction is used to shift 8 bits and 16 bits to the right.
2. LSB shifted into carry flag.
3. **Retain** and **shift** the MSB

Mnemonic: SAR destination , Count

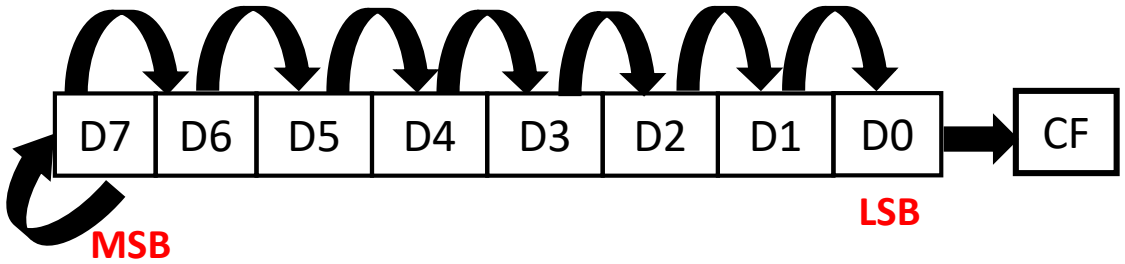
Number will become -ve

Number will become +ve

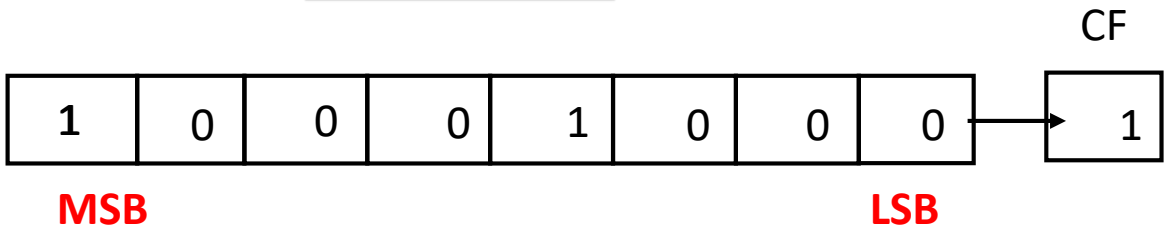
Before Execution **BL= 88 h**



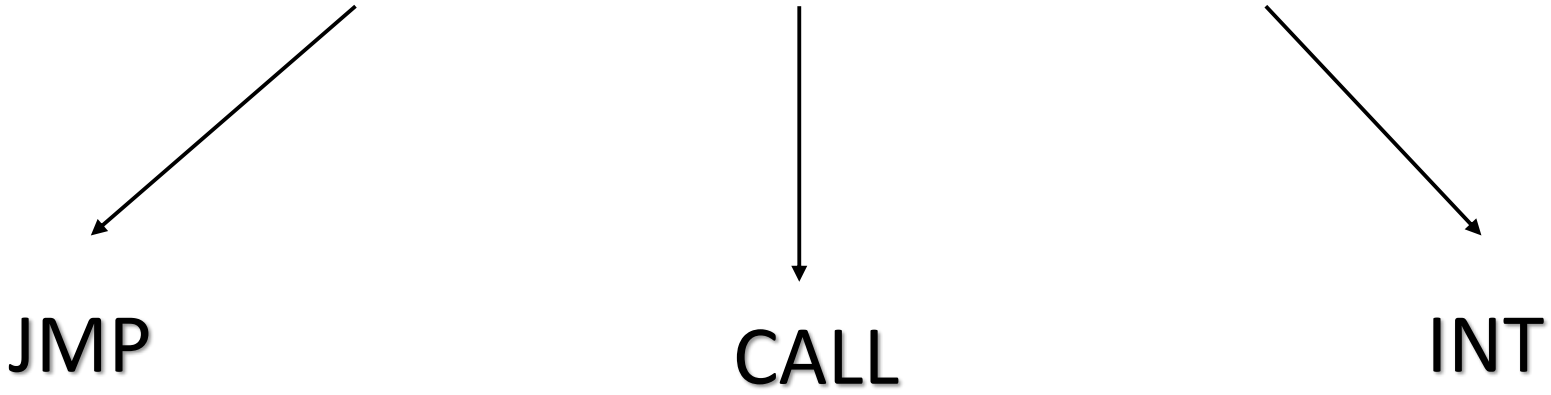
Operation :



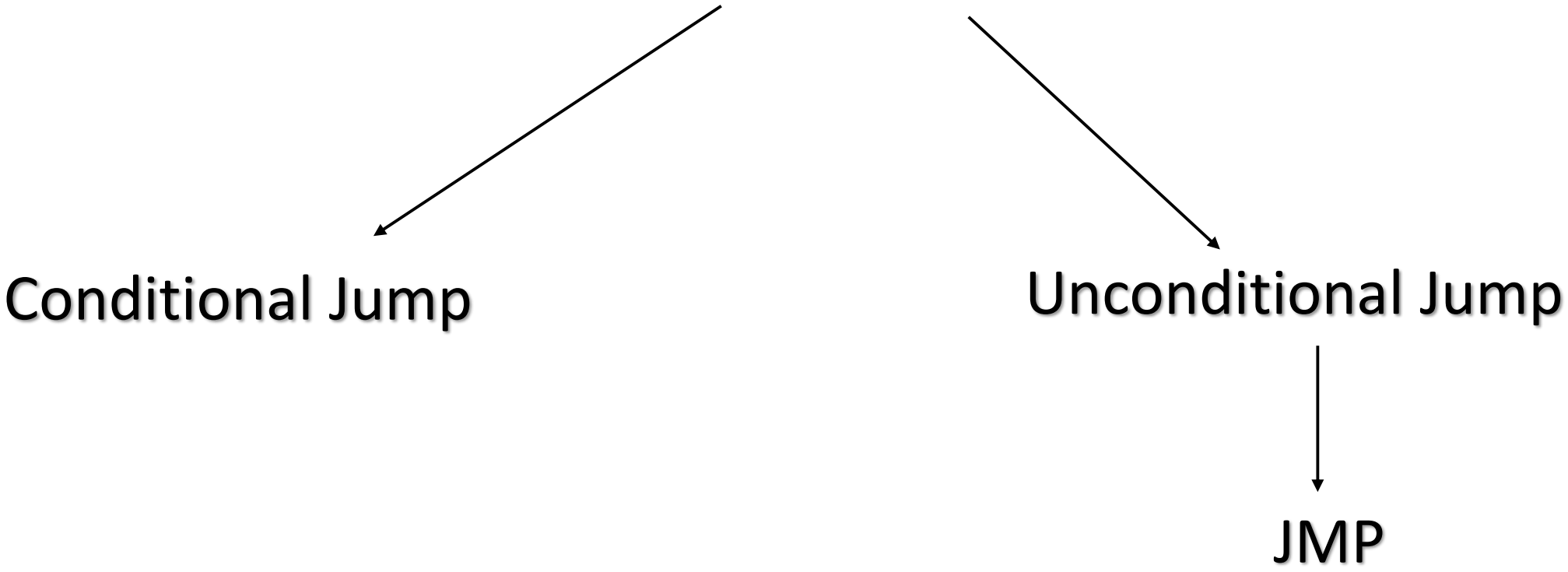
After Execution **BL= 44 h**



Program Transfer Group

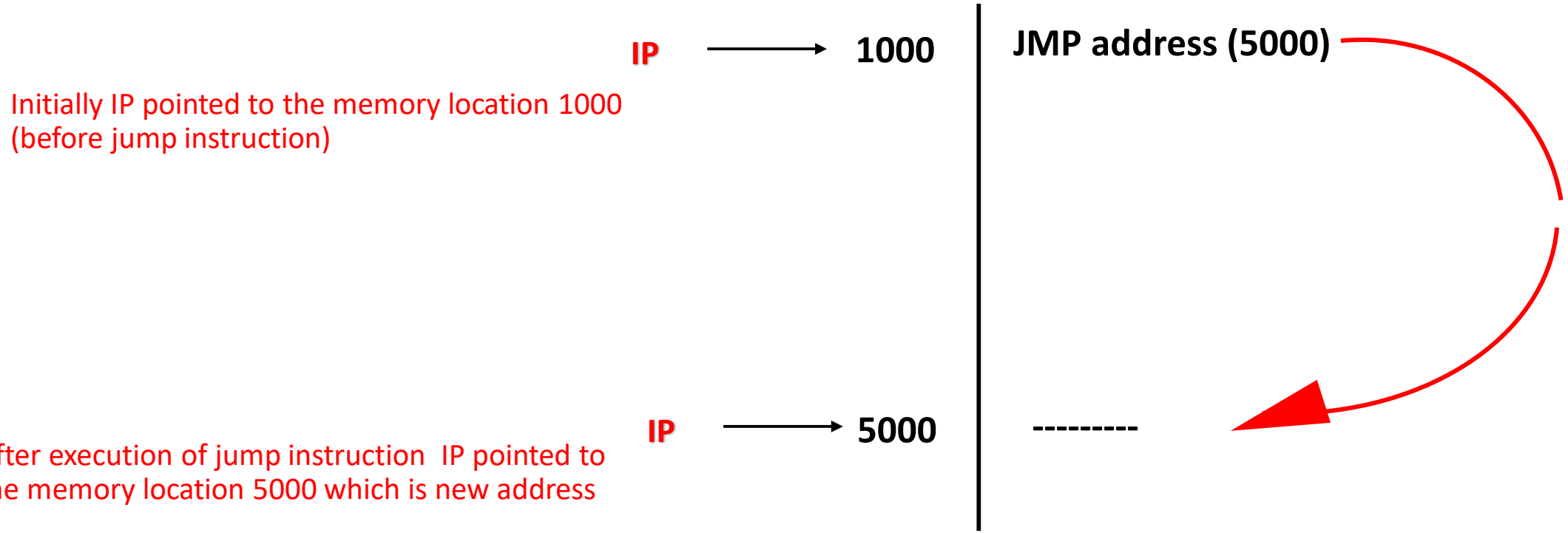


Jump Instruction

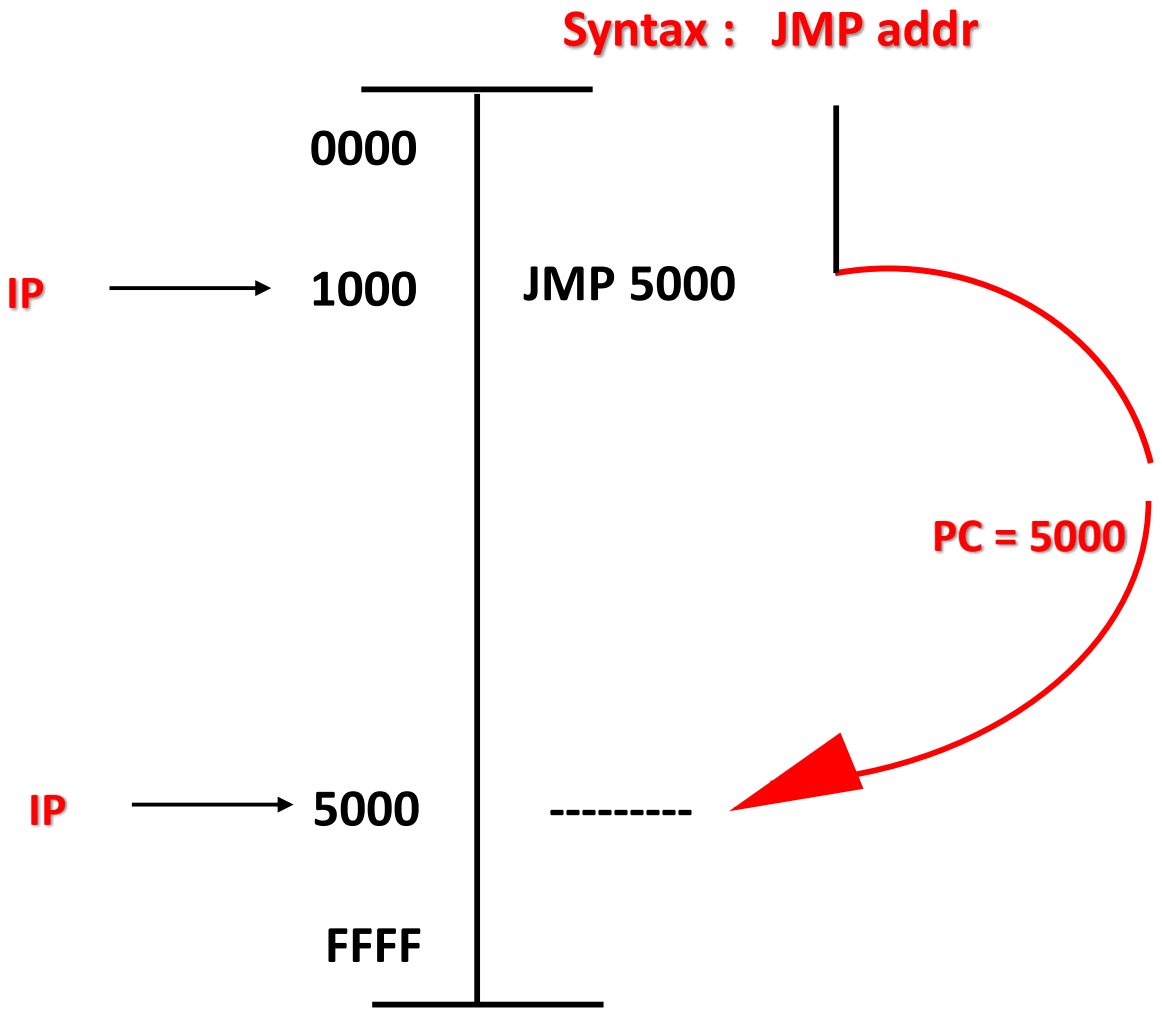


Unconditional Jump

JMP instruction of 8086 :



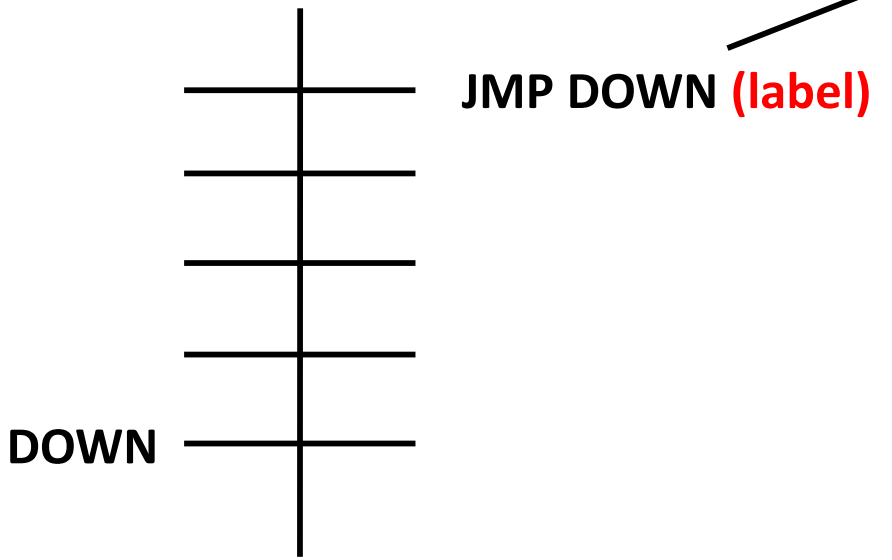
JMP



This is all about Theory part

Practically :

Assembler will substitute this address.



Note:

As a programmer we don’t know the physical address, because we are not writing program in memory, so assembler will responsible to convert symbols or labels into a physical address.

JMP– Jump to specified address:

- This instruction will cause the 8086 to fetch its next instruction from the location specified in the instruction rather than from next location after JMP instruction.
- There are two basic types of JMP, Near and Far

Mnemonic: JMP address

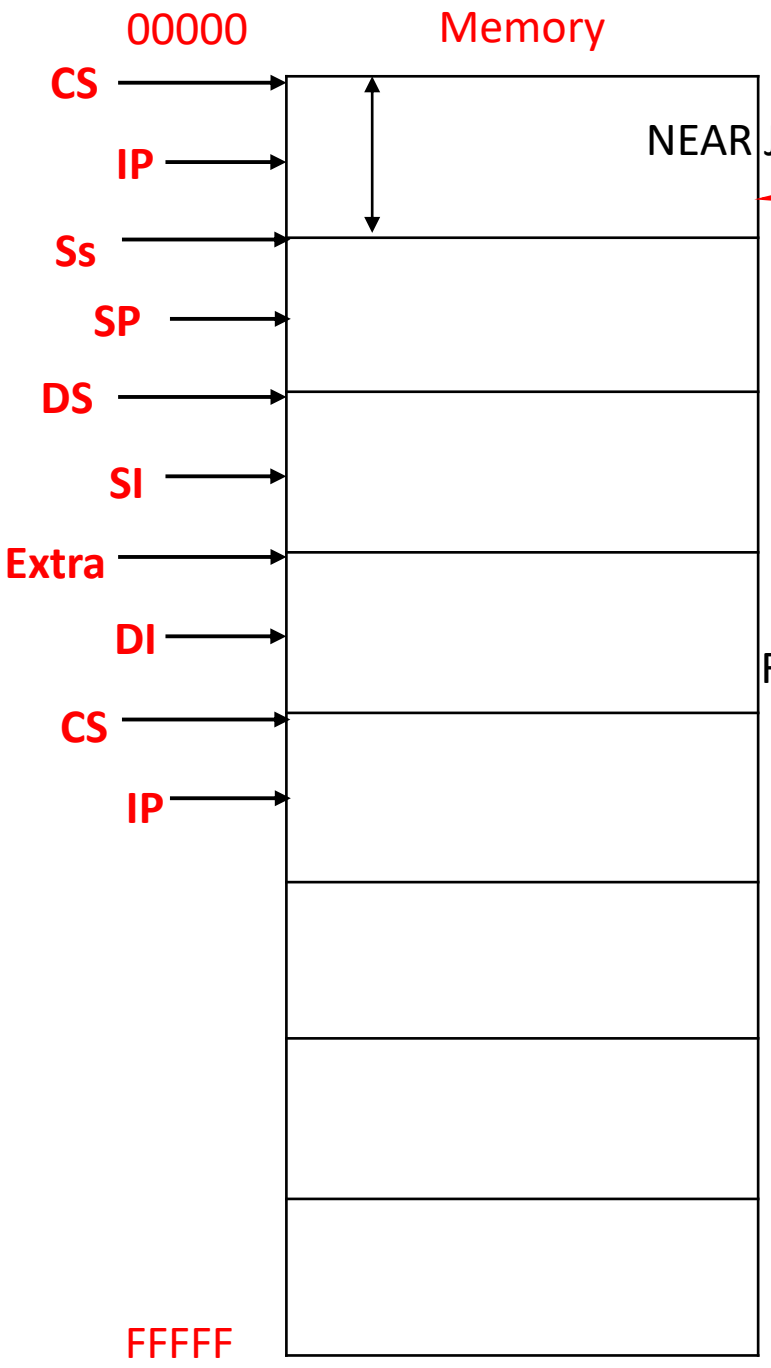
Operation :

Jump to specified address

Flags :

No Flags are affected

Example: JMP



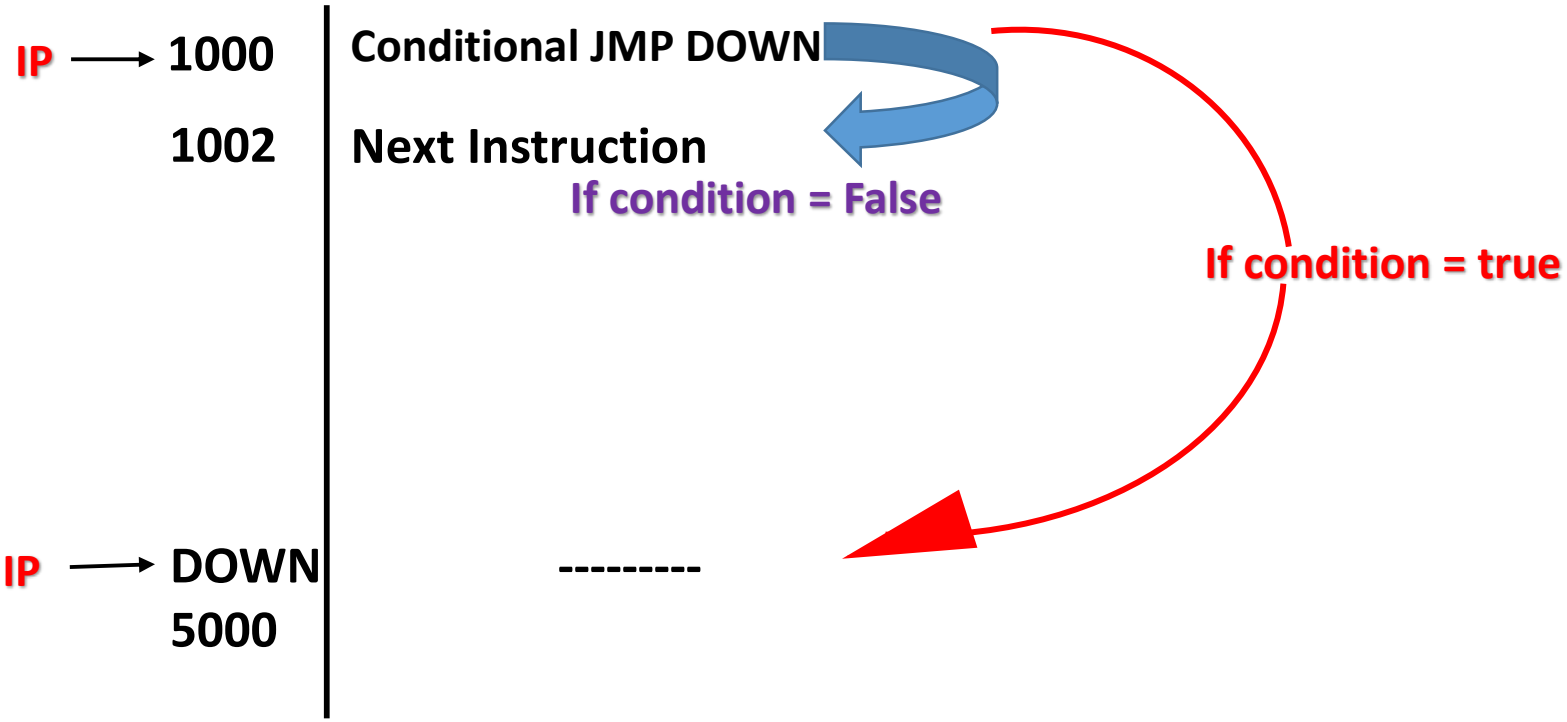
Near JMP and Far JMP further described as direct and indirect. If the destination address is specified within instruction it is called as direct and if the destination address is in register or memory location

NEAR JMP	FAR JMP
Destination location is in the same segment.	Destination location is in the different segment.
When NEAR JMP only IP changed and CS will remains same.	When Far JMP then both IP and CS changed.
Less stack memory is required	More stack memory is required
It is also called as an INTRA segment JMP	It is also called as an INTER segment JMP

Conditional Jump :

- Conditional transfer instructions are also called as conditional jumps instructions.
- There are total 18 conditional jump instruction as per given in the table.
- For conditional jump instructions, if **condition** is **true** then only control is transferred to the target specified in the instruction.
- if **condition** is **false** then control is transferred to the instruction that follows the conditional JMP.

Initially IP pointed to the memory location 1000 (before jump instruction)

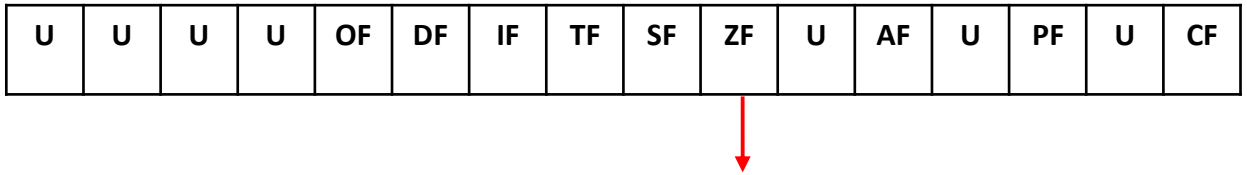


After execution of jump instruction IP pointed to the memory location 5000 which is new address

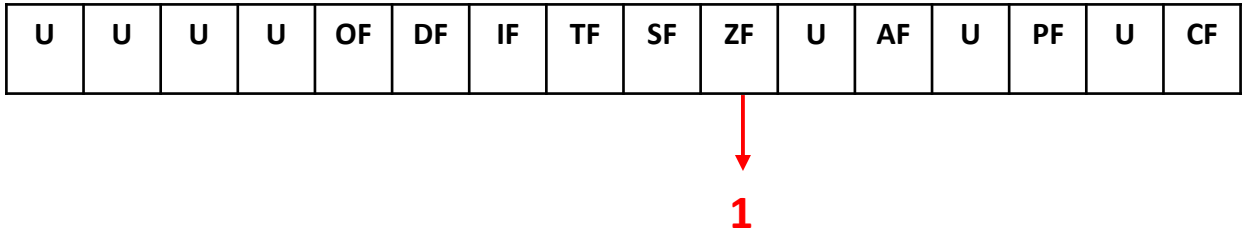
Common Operations		
JC	JUMP if carry	CF = 1
JNC	JUMP if not carry	CF = 0
JZ/JE	JUMP if zero (or equal)	ZF = 1
JNZ/JNE	JUMP if not zero (or not equal)	ZF = 1
JP/JPE	JUMP if parity (or even parity)	PF = 1
JNP/JPO	JUMP if not parity (or odd parity)	PF = 0
JCXZ	JUMP if CX is zero	CX = 0000H
Signed Operations		
JO	JUMP if overflow	OF = 1
JNO	JUMP if not overflow	OF = 0
JS	JUMP if sign (– ve)	S = 1
JNS	JUMP if not sign (+ ve)	S = 0
JL/JNGE	JUMP if less (i.e. either greater nor equal)	SF ⊕ OF = 1
JNL/JGE	JUMP if not less (i.e. either greater or equal)	SF ⊕ OF = 0

JLE/JNG	JUMP if less or equal (i.e. not greater)	(SF ⊕ OF) + ZF = 1
JNLE/JG	JUMP if neither less nor equal (i.e. greater)	(SF ⊕ OF) + ZF = 0
Unsigned Operations		
JB/JNAE	JUMP if below (i.e. neither above nor equal)	CF = 1
JNB/JAE	JUMP if not below (i.e. either above or equal)	CF = 0
JBE/JNA	JUMP if below or equal (i.e. not above)	CF ⊕ ZF = 1
JNBE/JA	JUMP if neither below nor equal (i.e. above)	CF ⊕ ZF = 0

	Mov al , 00h	AL= 00
	Mov bl, 01h	BL = 01
	Mov cl, 05h	CL = 05
BACK :	Add al , bl	AL= 00+01 = 01
	Dec CL	CL=05-01=04
	Jnz BACK	ZF=0,non zero Condition is true then jump to Back
	End	

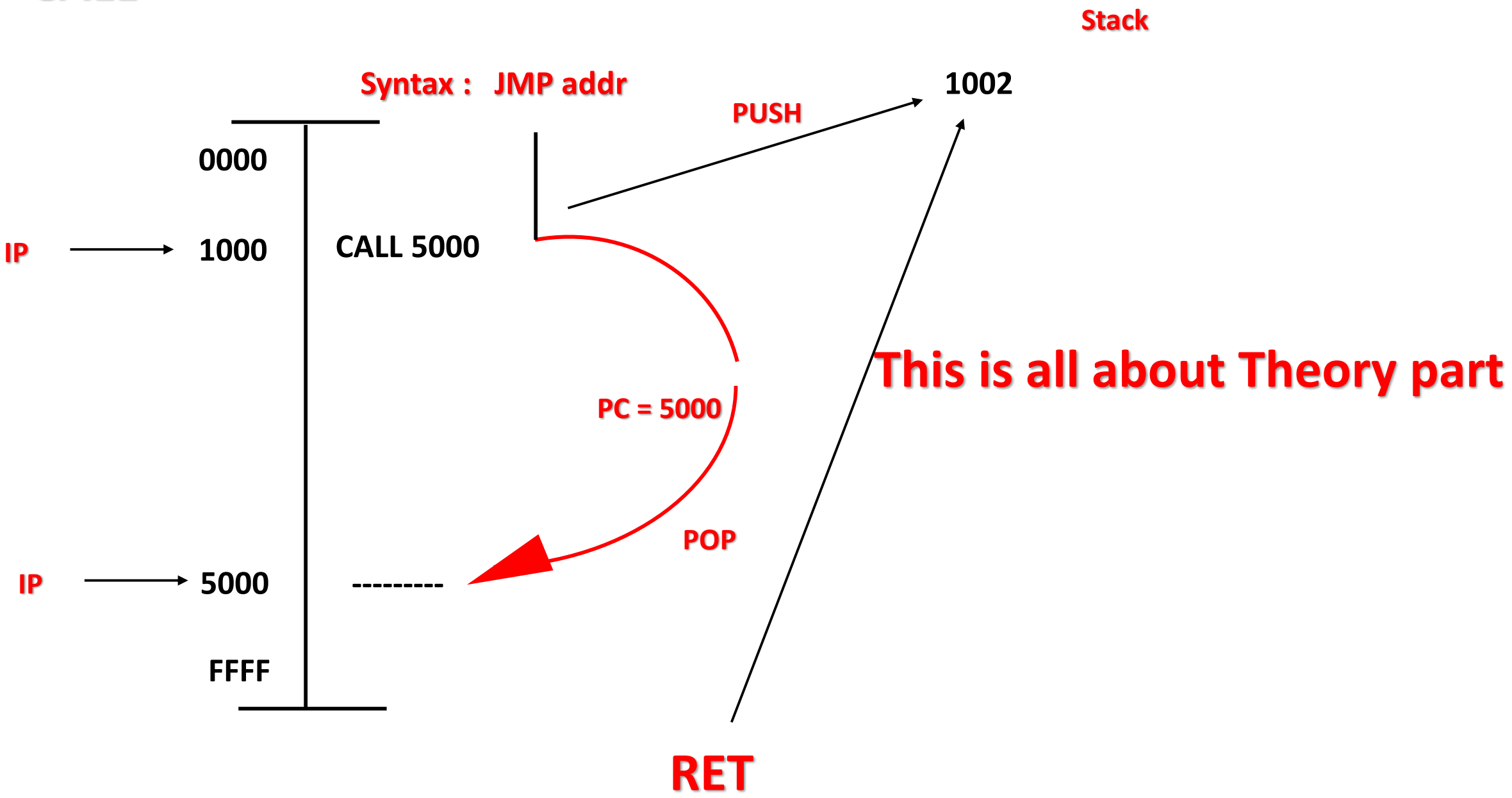


jump to Back



Execute next instruction

CALL



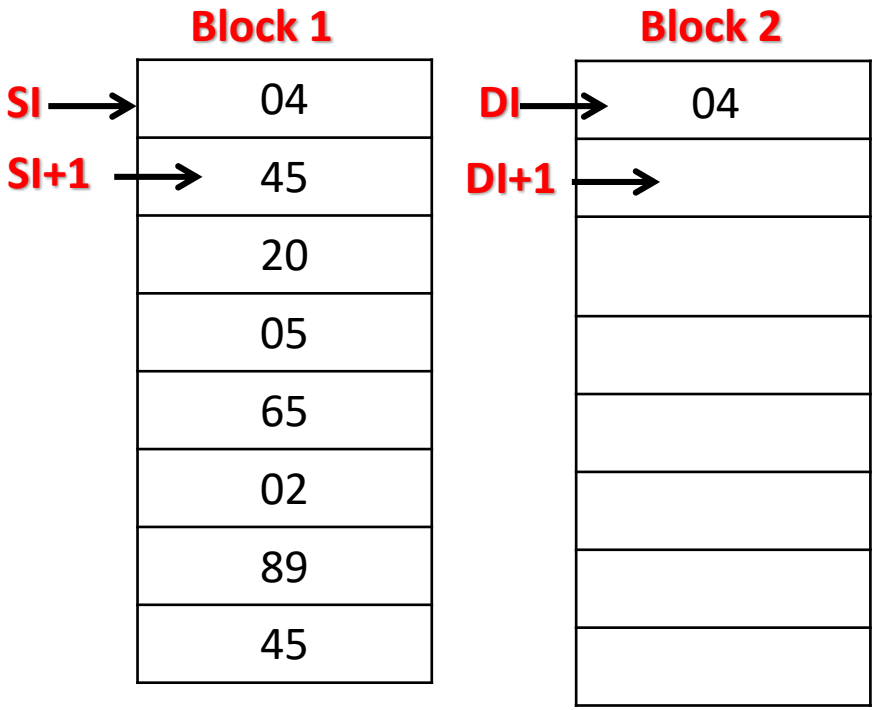
This is all about Theory part

String Instructions

There are two methods

Why string instructions are used????

When we want to perform operations on block of data with speed then string instructions are used.



Without String instruction

Use SI pointer to point Block 1 (source data)

Move first element into any reg.
MOV BL , [SI]

Use DI pointer to point Block 2 (destination data)

Transfer first element from reg to destination block
MOV [DI] , BL

Increment both pointer by 1 to point next location
INC SI
INC DI

Repeat above steps till data transfer overs, so need to execute loop.

Using String instruction

Use specific single string instruction to perform given transfer

- Less number of coding required
- Suitable for large number of data
- Data operation speed is fast as compare to manual data transfer

Example : there are two blocks , block 1 and block2. Block 1 having elements and block 2 is empty. We want to copy block 1 data into block 2

String Instructions

MOVS

LODS

SCANS

CMPS

STOS

Key Points for string instructions:

1. SI - Index offset for source string
2. DI - Index offset for destination string
3. CX - Default counter register
4. AL/AX – Register which required to stored data for string instructions.
5. CLD - clear direction flag (DF = 0)
6. SLD – Set direction flag. (DF= 1)

DF = 0 : Auto increment SI,DI pointers
DF = 1 : Auto decrement SI,DI pointers

Prefix used with string instructions:

1. REP – Repeat
2. REPE – Repeat if equal
3. REPNE – Repeat if not equal

MOVS B/W/D :

This instruction is used to transfer the contents of source to destination.

Operation : ES: [DI] ← DS: [SI]

Types :

1. MOVSB : Moves 8 bit content of source to destination : SI = SI +1 and DI = DI + 1 if DF =0
SI = SI -1 and DI = DI - 1 if DF =1

2. MOVSW : Moves 16 bit content of source to destination : SI = SI +2 and DI = DI + 2 if DF =0
SI = SI - 2 and DI = DI - 2 if DF =1

3. MOVSD : Moves 32 bit content of source to destination : SI = SI +4 and DI = DI + 4 if DF =0
SI = SI - 4 and DI = DI - 4 if DF =1

LODS B/W/D :

This instruction is used to load string byte into AL and string word into AX register. This instruction copies a byte or word from a string location pointed by SI into the AL/AX register.

Operation : AL ← DS: [SI]

Types :

1. LODSB : Load 8 bit content of AL from source : AL ← DS: [SI]
   SI = SI +1 if DF =0
   SI = SI -1 if DF =1

2. LODSW : Load 16 bit content of AX from source : AX ← DS: [SI]
   SI = SI +2 if DF =0
   SI = SI - 2 if DF =1

3. LODSD : Load 32 bit content of EAX from source : EAX ← DS: [SI] (80386)
   SI = SI +4 if DF =0
   SI = SI - 4 if DF =1

STOS B/W/D :

This instruction is used to load string byte from AL and string word from AX register. This instruction copies a byte or word from a AL/AX reg into string location pointed by DI in extra segment.

Operation : AL → ES: [DI]

Types :

1. STOSB : Store 8 bit content of AL into destination : DI = DI + 1 if DF =0
DI = DI - 1 if DF =1

AL → ES: [DI]

2. STOSW : Store 16 bit content of AX into destination : DI = DI + 2 if DF =0
DI = DI - 2 if DF =1

AX → ES: [DI]

3. STOSD : Store 32 bit content of EAX into destination : DI = DI + 4 if DF =0
DI = DI - 4 if DF =1

EAX → DS: [DI] (80386)

SCAS B/W/D :

This instruction is used to compare a byte/word into AL/AX with a byte pointed by DI in ES.

Operation : Compare AL/AX with ES: [DI]

Types :

1. SCASB : Compare 8 bit content of AL with destination : DI = DI + 1 if DF =0
DI = DI - 1 if DF =1

2. SCASW : Compare 16 bit content of AX with destination : DI = DI + 2 if DF =0
DI = DI - 2 if DF =1

3. SCASD : Compare 32 bit content of EAX with destination : DI = DI + 4 if DF =0
DI = DI - 4 if DF =1

WAP to find the displacement at which the data 0DH is present from an array of data stored from location 'Arr', the no. of bytes of data in the array is 80.

```
.model small
.data
arr db 80 dup(?)

.code

    lea di,arr
    mov al, ODh
   mov CX, 80
   CLD
Repne scasb
```

For this there are two ways to exit :

1. **If match is found**
2. **If match not found and CX=0**

CMPS B/W/D :

This instruction is used to compare a byte/word into source string of DS pointed by SI with a byte/word pointed by DI in ES.

Operation : Compare DS:[SI] with ES: [DI]

Types :

1. CMPSB : Compare 8 bit content of source with destination : SI = SI +1 and DI = DI + 1 if DF =0
SI = SI -1 and DI = DI - 1 if DF =1

2. CMPSW : Compare 16 bit content of source with destination : SI = SI +1 and DI = DI + 1 if DF =0
SI = SI -1 and DI = DI - 1 if DF =1

3. CMPSD : Compare 32 bit content of source with destination : SI = SI +1 and DI = DI + 1 if DF =0
SI = SI -1 and DI = DI - 1 if DF =1

```
.model small
.data
Src db 06 dup(?)
Des db 06 dup(?)
.code
    lea Si,Src
    lea Di,Des
    mov CX, 06
    CLD
    Repe cmpsb
```

DS:[SI]	Source	ES:[DI]	Destination
1000	67	2000	67
1001	56	2001	56
1002	0E	2002	0E
1003	89	2003	89
1004	A0	2004	A0
1005	12	2005	41---12

Transfer one array into another empty array without using string instruction

```
.model small
.data
a db 04,02,03,05,06h
e db 05 dup(?)

.code
   mov ax,@data
   mov ds,ax
   lea si,a
   mov es,ax
   lea di,e

   mov ch, 05h

Back: mov al,[si]
      inc si
      mov [di],al
      inc di
      dec ch
      jnz back

      lea di,e

      mov dh,05h

back1: mov al,[di]

       mov ch,02h
       mov cl,04h
       mov bh, al
i2:    rol bh,cl
       mov dl,bh
       and dl,0fh
       cmp dl,09h
       jbe i4
       add dl,07h
i4:    add dl,30h
       mov ah,02h
       int 21h
       dec ch
       jnz i2

       mov dl,' '
       mov ah,02h
       int 21h
       inc di
       dec dh
       jnz back1
       mov ah,4ch
       int 21h
   end
```

array with elements

Blank array to store 5 elements

Initialization of data segment as a source and assign SI for array a.

Initialization of data segment as a destination and assign DI for array e.

Count value to transfer elements from array a to array e

This loop is used to perform transfer from array a to array e

Point first element of array e to print

Count value to print elements

Take first element of an array e

To print first number from array

To assign space between two digits

Transfer one array into another empty array using string instruction

```
.model small
.data
a db 04,02,03,05,06h
e db 05 dup(?)
.code
   mov ax,@data
   mov ds,ax
   lea si,a
   mov es,ax
   lea di,e
   mov ch, 05h
   CLD
Rep movsb
   lea si,e
   mov dh, 05h
back: mov al,[si]
      mov ch,02h
      mov cl,04h
      mov bh, al
i2:   rol bh,cl
      mov dl,bh
      and dl,0fh
      cmp dl,09h
      jbe i4
      add dl,07h
i4:   add dl,30h
      mov ah,02h
      int 21h
      dec ch
      jnz i2
      mov dl,' '
      mov ah,02h
      int 21h
      inc si
      dec dh
      jnz back
      mov ah,4ch
      int 21h
   end
```

array with elements

Blank array to store 5 elements

Initialization of data segment as a source and assign SI for array a.

Initialization of data segment as a destination and assign DI for array e.

Count value to transfer elements from array a to array e

Clear DF for auto increment of SI & DI

Movsb is used to transfer 8bit data of AL reg from source to destination. Repeat this until CH =00 h

Point first element of array e to print

Count value to print elements

Take first element of an array e

To print first number from array

To assign space between two digits

Processor control Instructions of 8086

Processor Control Instructions

Flag Operations

- STC
- CLC
- CMC
- STD
- CLD
- STI
- CLI

No Operation

- NOP

External synchronization

- HLT
- WAIT
- ESC
- LOCK

Flag Operations

STC – Set Carry Flag

Example: STC

This instruction is used to set the carry flag.

Mnemonic: **STC**

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF
															1

Operation : **CF = 1**

Addressing Mode : Implied addressing mode

Flags : Except carry flag no other flags are affected.

CLC – Clear Carry Flag

Example: CLC

This instruction is used to clear the carry flag.

Mnemonic: **CLC**

0

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

Operation : **CF = 0**

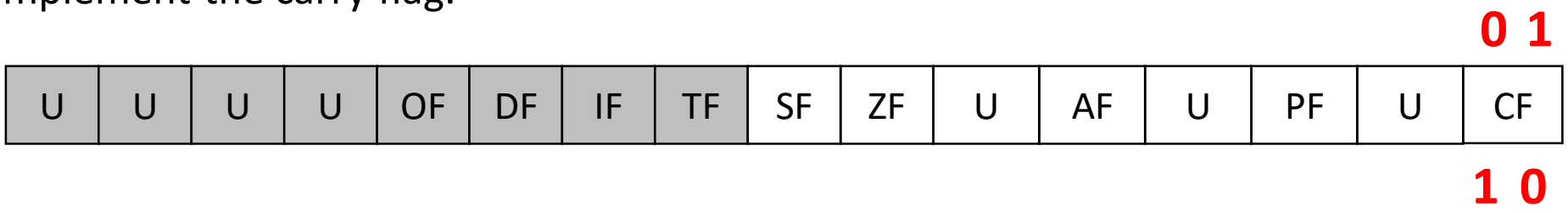
Addressing Mode : Implied addressing mode

Flags : Except carry flag no other flags are affected.

CMC – Complement Carry Flag

Example: CMC

This instruction is used to complement the carry flag.



Mnemonic: CMC

Operation : **CF = $\overline{CF}$**

If CF = 0 then after execution of CMC instruction CF = 1

If CF = 1 then after execution of CMC instruction CF = 0

Addressing Mode : Implied addressing mode

Flags : Except carry flag no other flags are affected.

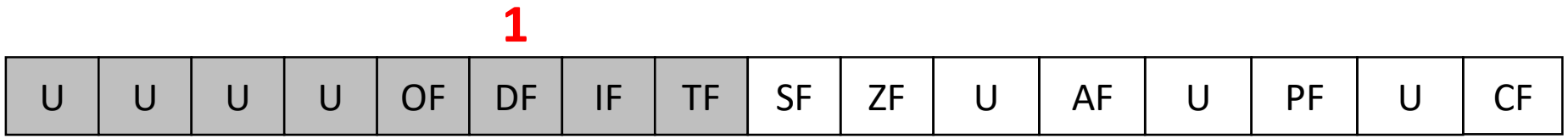
STD – Set Direction Flag

- This instruction is used to set the direction flag.
- DF flag is used in string instruction.
- If DF= 1 then in case of string instructions SI and DI automatically decremented.

Example: STD

Mnemonic: **STD**

Operation : **DF = 1**



Addressing Mode : Implied addressing mode

Flags : Except Direction flag no other flags are affected.

CTD – Clear Direction Flag

- This instruction is used to clear the direction flag.
- DF flag is used in string instruction.
- If DF= 0 then in case of string instructions SI and DI automatically incremented.

Example: CTD

Mnemonic: CTD

Operation : **DF = 0**

0

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

Addressing Mode : Implied addressing mode

Flags : Except Direction flag no other flags are affected.

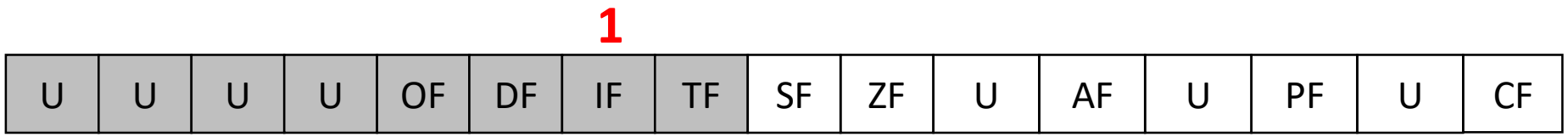
STI – Set Interrupt Enable Flag

- This instruction sets the interrupt flag to 1.
- This enables INTR interrupt of the 8086.

Example: STI

Mnemonic: **STI**

Operation : **IF = 1**



Addressing Mode : Implied addressing mode

Flags : Except Interrupt flag no other flags are affected.

CLI – Clear Interrupt Enable Flag

- This instruction resets the interrupt flag to zero.
- If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input.

Example: STI

Mnemonic: CTI

Operation : **IF = 0**

0

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

Addressing Mode : Implied addressing mode

Flags : Except Interrupt flag no other flags are affected.

No Operations

NOP – No Operation

Example: NOP

- The execution of this instruction causes the CPU to do nothing.
- This instruction uses three clock cycles and increments the instruction pointer to point to the next instruction.
- It can be used to increase the delay of delay loop.

Mnemonic: **NOP**

Operation : **Do nothing**

Addressing Mode : Implied addressing mode

Flags : Dose not affect any flag.

External synchronization

HALT – Halt until interrupt or reset

- The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 enters into a halt state. To come out of the halt state, there are 3 ways given below.
  (i) Interrupt signal on INTR pin
  (ii) Interrupt signal on NMI pin
  (iii) Reset signal on reset pin.
- It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

Mnemonic: HLT

Addressing Mode : Implied addressing mode

Flags : Dose not affect any flag.

WAIT – Wait for test pin active

When this instruction executes, the 8086 enters on idle condition in which it is doing no processing.

- The 8086 will stay in this idle state until 8086 $\overline{\text{TEST}}$ input pin is made low or on interrupt signal is received on the INTR or NMI interrupt pins.
- If a valid interrupt occurs while the 8086 is in the idle state, the 8086 will return to idle state after the interrupt service procedure executes.
- It is used to synchronize the 8086 with external hardware. Such as 8087 math processor.

Mnemonic: **WAIT**

Addressing Mode : Implied addressing mode

Flags : Dose not affect any flag.

3. ESC – Escape to external processor

Mnemonic	ESC external – opcode, source.
Operation	This instruction is used to pass instruction to a coprocessor, such as 8087 math co-processor which shares the address and data bus with on 8086. • The instruction for the Coprocessor are represented by a 6 bit code embedded in the escape instruction. • When the 8086 fetches on ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6 bit code specified in the instruction. • In most cases 8086 treats the ESC instruction as a NOP in some cases 8086 will access a data item in memory for co-processor.

4. LOCK – Lock bus during next instruction

Mnemonic	LOCK
Operation	Many multiprocessor systems contain several microprocessors. Each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a shared system bus so that each can access system resources such as disk drives or memory • Each microprocessor takes control of the system bus. Only when it needs to access some resource.
	• Lock prefix allows a microprocessor to make sure that another processor does not take control of the system bus. • While it is in the middle of a critical instruction which uses the system bus when an instruction with lock prefix executes the 8086 will assert its bus lock signal output. This signal is connected to an external bus controller, which then prevents any other processor from taking over the system bus.
Example	LOCK XCHG SEMAPHORE, AL : The XCHG instruction requires two bus accesses. The lock prefix prevents another processor from taking control of system bus between two accesses.