towfiq.rakin@icloud.com

8086

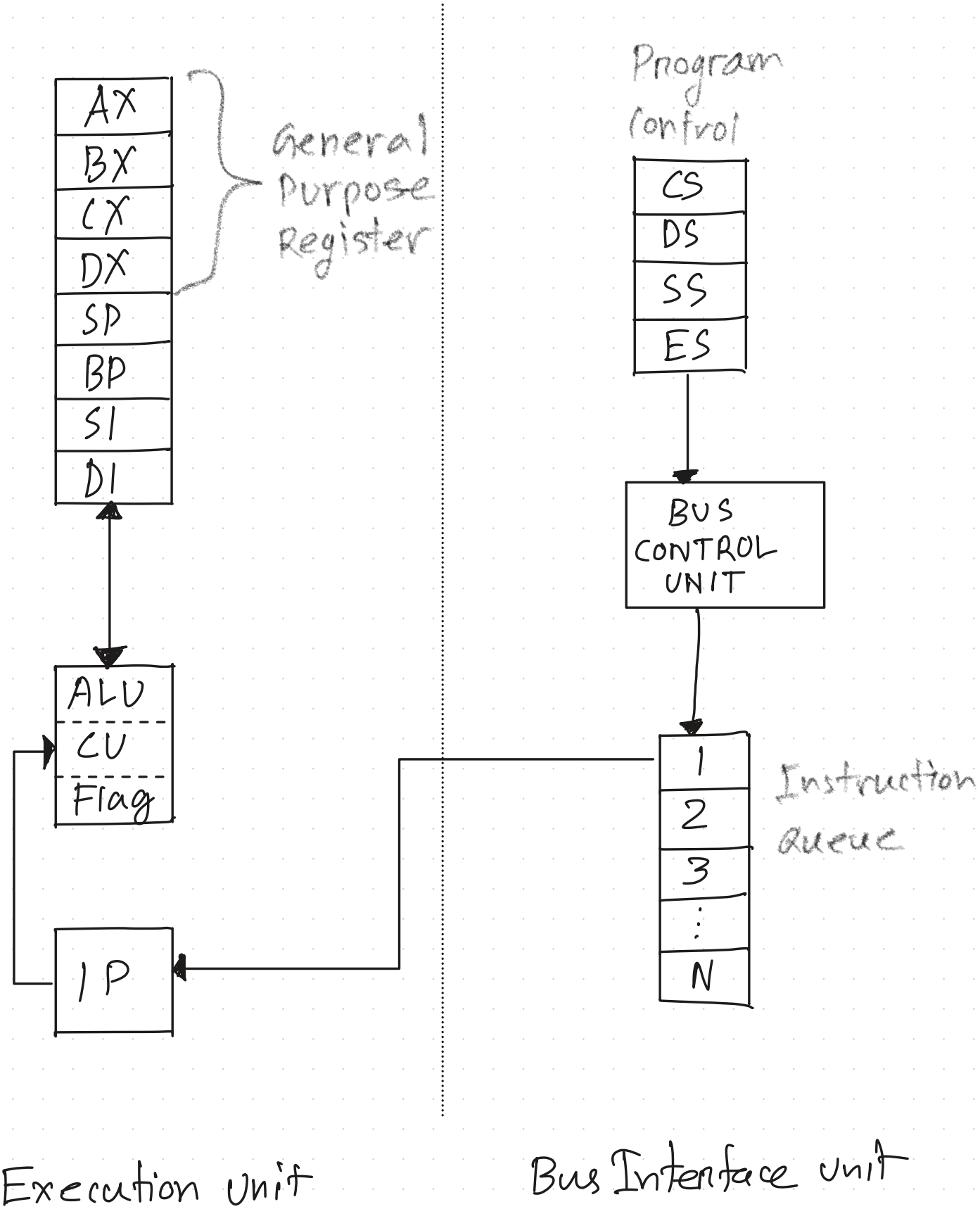
INTRODUCTION

8086 microprocessor is a enhanced version of 8085 that was designed by intel in 1976. It is a 16 bit microprocessor which has 16 bit data line and 20 bit address line. It provides up to 1MB storage.

FEATURES

i. It is a 16-bit processor.
ii. It has a 16-bit data bus.
iii. It has 20 bit address line and can access up to 2^{20} memory locations (1MB).
iv. It can support up to 64K I/O ports.
v. It provides 14 registers, 16-bit each.
vi. Prefetches up to 6 instruction bytes from memory and queues them in order to speed up the process.
vii. It uses two stages of pipelining i.e. fetch stage and execute stage.

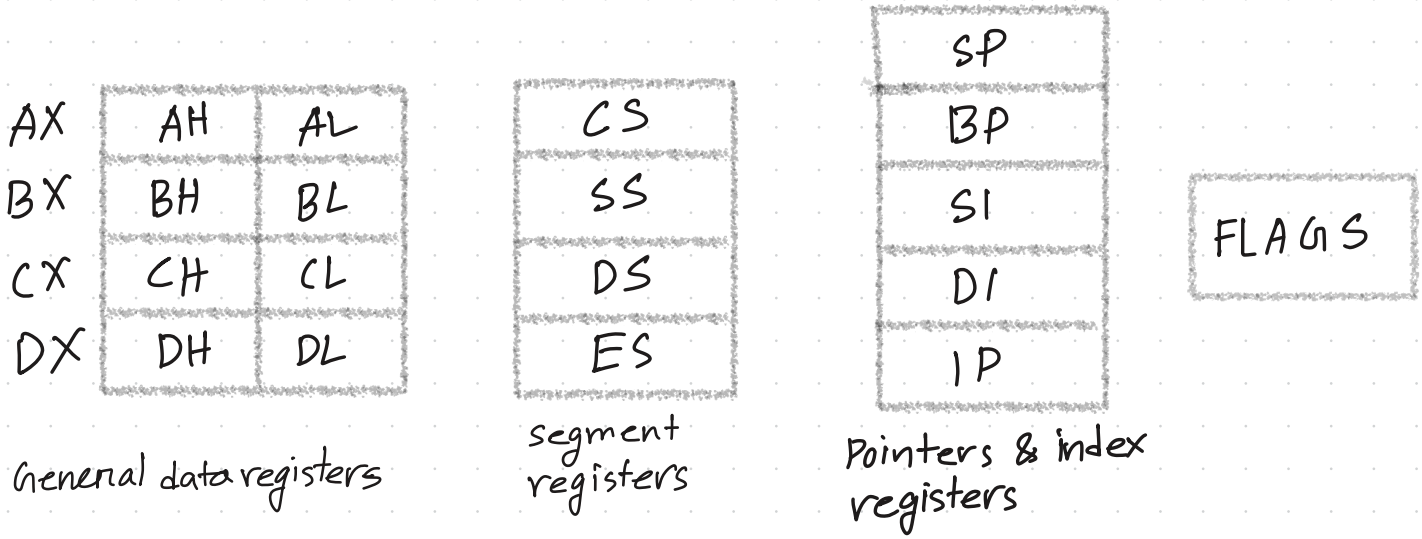
INTERNAL ARCHITECTURE



REGISTER ORGANISATION

The 8086 registers are classified into the following types:

- → General Purpose Registers
- → Segment Registers
- → Pointers and segment Registers
- → Flag Register



1. General Data Registers

i. The registers AX, BX, CX, DX are the general purpose 16-Bit registers.

ii. AX is used a 16-bit accumulator. The lower 8-bit is designed as AL & the higher 8-bit is designed AH.

iii. AL data registers can be used as either 16-bit or 8-bit. As example BX is a 16-bit register, but BL indicates the lower 8-bit while BH indicates the higher 8 bit.

iv. The register BX is used as offset storage for forming physical address.

v. The register CX is used as default counter in case of string and loop instructions.

vi. DX register may be used as an implicit operand or destination in case of few instructions.

2. Segment Registers

There are 4 segment registers:

→ Code segment (CS)
→ Data segment (DS)
→ stack segment (SS)
→ Extra segment (ES)

Code Segment Register (CS):

It is used for addressing memory location in the code segment of the memory, where executable program is stored.

Data Segment Register (DS):

It points the data segment of the memory where data is stored.

Extra segment Register (ES):

It refers to a segment in the memory which is another data segment in the memory.

Stack Segment Register (SS):

The stack segment is that segment of memory which is used to store stack data. while addressing any location in the memory bank, the physical address is calculated from two parts.

> Physical Address = Segment address X10H + offset address

The first part is segment address, the segment register contain 16-bit segment base addresses, related to different segment. The second part is the offset value in that segment.

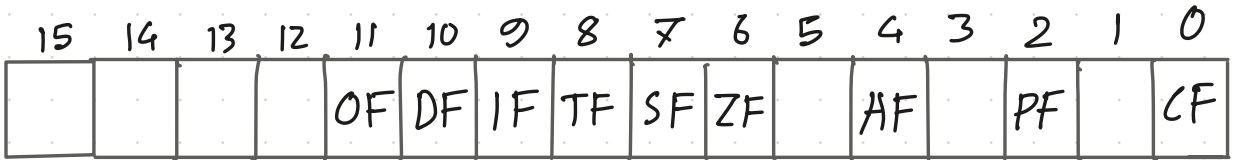
3. Pointer & Index Registers

The index and pointer registers are given below:

- IP: Instruction pointer - store memory location of next instruction to be executed. contains offset within the code segment.
- BP: Base Pointer, contains offset within data segment.
- SP: Stack Pointer, contains offset within stack segment.
- SI: Source index; used to store the offset of source data in data segment.
- DI: Destination index; used to store the offset of destination in data or extra segment.

4. Flag Register

The 8086 flag register contents indicate the result of computation in the ALU. It also contains some flag bits to control the CPU operation.

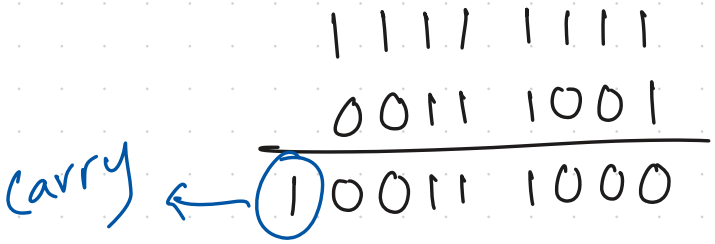


The description of each flag bit is as follows:

Status Flags

Carry flag (CF)

This flag is set 1. when there is a carry out of MSB.



Parity flag (PF)

This flag is set to 1. if the lower byte of the result contains even number of 1's.

Auxiliray carry flag (AF)

This flag is set 1, if there is a carry from the lowest nibble, or borrow from the lowest nibble.

```
  1011 1111
  0000 0100
-----------
  1100 0011
```

* Here, carry goes from 3rd bit to the 4th bit.

Zero flag (ZF)

This flag is set 1, if the result of previous instruction is zero.

Sign flag (SF)

This flag is set 1, if the result of any computation is negative.

if MSB = 1, SF = 1

MSB = 0, SF = 0

Overflow flag (OF)

This is flag set, if an overflow occurs. when the result is more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operation, and then the overflow will be set.

Trap flag (TF)

Interrupt flag (IF)

Direction flag (DF)

} Status Flags

EXAMPLE

* 7FFFH + 7FFFH → 0111 1111 1111 1111 + 0111 1111 1111 1111 = 1111 1111 1111 1110

CF=0 AF=1
PF=0 SF=1
ZF=0 OF=1 → Too positive numbers resulted negative number

* FFFFH + 0001H → 1111 1111 1111 1111 + 0000 0000 0000 0001 = 1 0000 0000 0000 0000

CF=1 AF=1
PF=1 SF=0
ZF=1 OF=0 → sum of negative (-) & positive (+) number is positive (+)

* FFFFH + FFFFH → 1111 1111 1111 1111 + 1111 1111 1111 1111 = 1 1111 1111 1111 1110

CF=1 AF=1
PF=0 SF=1
ZF=0 OF=0 → (-) + (-) = (-)

MEMORY ADDRESSING

In 8086 memory addressing, every memory location is referenced by a 16-bit segment and a 16-bit offset, and a 20-bit physical address placed on the bus is calculated as physical Address = segment X10H + offset. This scheme lets a 16-bit CPU address 1MB by combining a segment base with an offset within that 64KB segment.

Memory address schemes:

1. A physical Address such as 04A26H, is a 20-bit value that directly references a specific location.
2. A Logical Address: combines segment and offset address. Offset address is a location within 64K byte segment range.

Specifying Address

To represent a segment address and its relative offset we use notation:

Segment : Offset

020A : 1BCD

↙ segment ↘ offset

Common Segmen - Offset pairs:

Segment	Offset	Special Purpose
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

How to calculate

Physical Address = Segment X 10H + Offset

* 1005H : 5555H → Logical Address

1005H X 10H → 10050 H

10050H → segment
+ 5555H → offset
155A5H → Physical Address

* 020AH : 1BCDH

020AH X 10H → 020A0H

020A0 H
1BCD H
03C6D H → Physical Address

* 1356H : 2E20H

1356H X 10H → 13560 H

13560 H
2E20 H
16380 H → Physical Address

CT 1 END

BASIC INSTRUCTION

MOV AX, 1234H ⟶ Data to Register

MOV AX, BX ⟶ Register to Register

MOV AX, [1234H] ⟶ Memory to register

MOV [2345H], [1234H] ⟶ INVALID X

XCHG AX, BX ⟶ Exchange the value of AX & BX

ADC ⟶ Add with carry (Previous OP)

MULTIPLY

MUL source (8/16 Bit)

MUL BL ⟶ In case of 8 Bit data, 8086 will multiply the source will AL register and store the 16 Bit into AX.

So, AX = AL * BL

MUL BX ⟶ In case of 16 Bit data, source will be multiplied with AX. Higher 16 bit will be stored in DX and lower in AX.

So, DX (16bit) AX (16bit) = AX * BX [32 bit Result]

IMUL ⟶ Signed Multiplication

DIVISION

DIV BL ⟶ When the operand is 8 Bit,

AL = AX / BL

AH = Reminder

AH	AL
R	Q

DIV BX ⟶ When the operand is 16 bit.

AX = (DX AX) / BX

DX = Reminder

DX	AX
R	Q

IDIV ⟶ Signed Division

ADDRESSING MODES

The way of specifying different source by an instruction is known as **addressing modes.**

There are 8 types of addressing mode of 8086 microprocessor.

1. Immediate Addressing Mode

The data operand is the part of the instruction.

```
MOV AX, 1234H
```

→ Data Operand

2. Register Mode

Both operands are register.

```
MOV AX, BX
```

3. Direct Mode

The effective address is directly given into the instruction.

```
MOV AX, [1234H]
```

→ Offset Address

4. Register Indirect Mode

The address of memory location stored in a register

```
MOV AX, [BX]
```

→ Here BX holds the Offset address.

5. Register Relative

The data is available at an effective address formed by adding 8/16 bit displacement with the content of any one of the register BX, BP, SI, DI in the default (DS or ES) segment.

MOV AX, 50H[BX] → DS X 10H + (BX + 50H)
Offset

FLOW CONTROL INSTRUCTION

The jump and loop instructions transfer control to another part of the program. This transfer can be unconditional or can depend on a particular combination of status flag settings.

JUMP

JMP – This instruction unconditionally transfer the control o execution to the specified address.

Example:

```
MOV  AL, 2030H
MOV  BL, 3465H
MUL  AL, BL
JMP  A
INC  AL
INC  BL
A: XCNG AL, BL
```

Here the instruction JMP allows the program to bypass following two instruction and to execute specified instruction unconditionally.

Conditional JUMP

To implement a conditional jump, the cpu looks at the Flags registers. If the condition for the jump are true, the cpu adjust the IP to point to the destination label so that the instruction at this label will be done next. If the condition is false, then the IP is not altered; this means the next in the line will be done.

Example:

```
MOV AL, 2
MOV BL, -5
CMP AL, BL
JGE label
   Print "AL<-5"
label:
   Print "AL>-5"
```

→ Jump if AL is greater or equal to BL (as set by the CMP inst) As a result, it will bypass the following line and execute → print "AL>-5"

CMP

The cmp instruction subtracts second operand from the first operand updates the value of flag registern. But it doesnit store the result of that subtraction anywhere.

CMP AX, BX ⟶ Does AX - BX and updates the flags (CF, OF, SF, ZF, AF, PF)

Table 6.1 Conditional Jumps

Signed Jumps

Symbol	*Description*	*Condition for Jumps*
JG/JNLE	jump if greater than jump if not less than or equal to	ZF = 0 and SF = OF
JGE/JNL	jump if greater than or equal to jump if not less than or equal to	SF = OF
JL/JNGE	jump if less than jump if not greater than or equal	SF <> OF
JLE/JNG	jump if less than or equal jump if not greater than	ZF = 1 or SF <> OF

Unsigned Conditional Jumps

Symbol	*Description*	*Condition for Jumps*
JA/JNBE	jump if above jump if not below or equal	CF = 0 and ZF = 0
JAE/JNB	jump if above or equal jump if not below	CF = 0
JB/JNAE	jump if below jump if not above or equal	CF = 1
JBE/JNA	jump if equal jump if not above	CF = 1 or ZF = 1

Single-Flag Jumps

Symbol	*Description*	*Condition for Jumps*
JE/JZ	jump if equal jump if equal to zero	ZF = 1
JNE/JNZ	jump if not equal jump if not zero	ZF = 0
JC	jump if carry	CF = 1
JNC	jump if no carry	CF = 0
JO	jump if overflow	OF = 1
JNO	jump if no overflow	OF = 0
JS	jump if sign negative	SF = 1
JNS	jump if nonnegative sign	SF = 0
JP/JPE	jump if parity even	PF = 1
JNP/JPO	jump if parity odd	PF = 0

LOOP

The loop instruction can be used to implement a for loop. It has the form.

```
LOOP  destination-label
```

The count for the loop is the register CX which is initialised to loop count. Execution of loop ints causes to decrement CX automatically, and if CX is not 0, the loop continues. If CX=0, the next instruction after loop is done.

Example: Write a count controlled loop to display a row of 10 stars.

Solution:

```
MOV CX, 10      ----------> Number of stars i.e. count
MOV AH, 2       ----------> function to print char
MOV DL, '*'     ----------> store char '*'
TOP:
    INT 21H     ----------> Execute function
LOOP TOP        ----------> Repeat until CX is 0
```

LOGICAL INSTRUCTION & MASKING

Logical instruction of 8086

→ AND des, src

→ OR des, src

→ XOR des, src

→ NOT des, src

[des → Destination
src → Source]

MASKING

AND

0 → Clear — b AND 0 = 0

1 → Preserve — b AND 1 = b

OR

0 → Preserve — b OR 0 = b

1 → Set 1 — b OR 1 = 1

XOR

1 → Compliment — b XOR 1 = $\overline{b}$

0 → Preserve — b XOR 0 = b

EXAMPLE 01

AND

Let's assume AL → 10010010. If we want to clear the MSB of AL, what will be the masking bitstream?

ANS:

Using AND operation, 0 → clear

Logical Expression,

AND AL, 01111111

AL →	10010010
MASK →	01111111
AND	00010010

EXAMPLE 02

set the value of AL = 10110000 in 0 position bit.

OR

ANS:

Logical Expression,

OR AL, 00000001

AL →	10110000
MASK →	00000001
OR	10110001

EXAMPLE 03

set the even position of AL 000 10100

7 6 5 4 3 2 1 0

ANS:

Logical Expression,

OR AL, 01010101

AL	00010100
MASK	01010101
OR	01010101

PROBLEM

```
MOV AL, 23H
MOV BL, 32H
INC AL
ADD BL, AL
```

Compliment the 3rd and 5th bit. what will be the masking bit after the execution of Last line? Show the change is number in each line.

Solution:

MOV AL, 23H ⟶

AL

0	0	1	0	0	0	1	1

MOV BL, 32H ⟶

BL

0	0	1	1	0	0	1	0

INC AL ⟶

AL

0	0	1	0	0	1	0	0

(24H)

ADD BL, AL ⟶

```
BL  00110010
+AL 00100100
-------------
BL  01010110
```

(56H)

Now, compliment of 3rd and 5th bit,

```
XOR BL, 00101000
```

Masking bit

```
BL   01010110
MASK 00101000   (+)
-------------
BL   01111110
```

SHIFT & ROTATE

SHL (shift left) [Multiplication of 2]

Syntax:
- SHL destination, 1
- SHL destination, CL

[CL contains N. N = total number of shifts]

A 0 is shifted into the rightmost position.

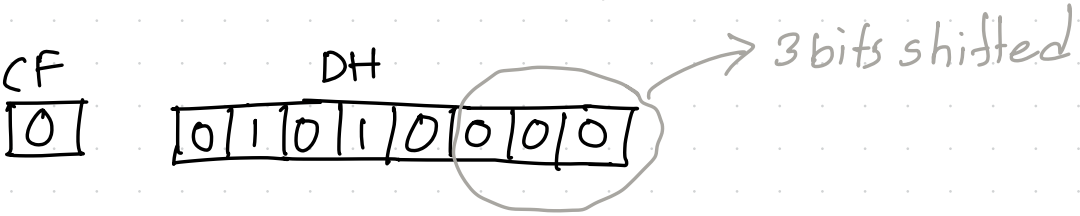
Effects on flags:

CF = last bit shifted out

OF = 1 if result changes sign on last shift.

Example Suppose DH contains 8AH and CL contains 3. what are the values of DH and of CF after the execution of SHL DH, CL?

Solution: Binary equivalent of DH = 10001010. After 3 left shifts, CF will contain 0. The new content of DH can be obtained by erasing leftmost three bits and adding three zero bits to the right end.



DH = 50H

Characteristics:

MOV AL, 5 ⟶ 00000101 = 05D

SHL AL, 1 ⟶ 00001010 = 10D

SHL AL, 1 ⟶ 00010100 = 20D

Thus we can multiply by 2 by left shift.

SHR (Shift Right)

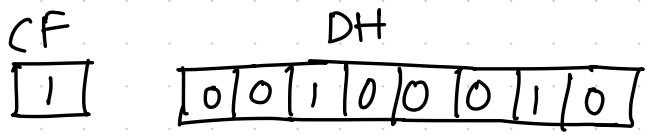
The instruction SHR (shift Right) performs right shifts on the destination operand. The format for single shift,

SHR destination, 1
SHR destination, CL ⟶ For multiple shifts.

The effects on flag is same as SHL.

Example Suppose DH contains 8AH and CL contains 2. what are the values of DH and of CF after the execution of SHR DH, CL?

solution: Binary equivalent of DH = 10001010. After 2 right shifts, CF will contain 1. The new content of DH can be obtained by erasing rightmost two bits and adding two zero bits to the left end.



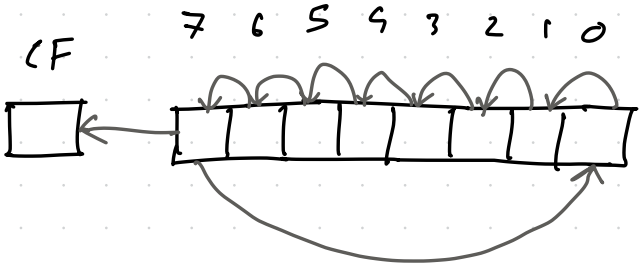
DH = 22H

The **SAR** instruction retains the original MSB value, effective for signed division.

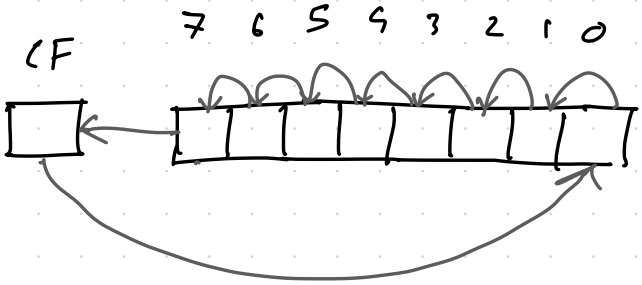
Rotate Left

ROL (Rotate left) shifts bits to the left. The MSB is shifted into the rightmost bit. The CF also gets the bit shifted out from the MSB.

ROL destination, 1
ROL destination, CL



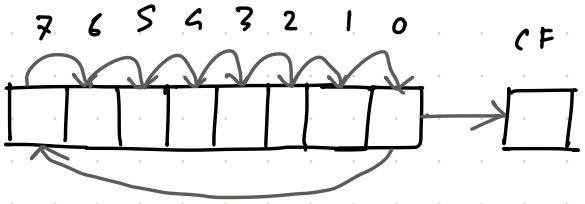
RCL (Rotatcle through carry left) does the same thing but with CF included.



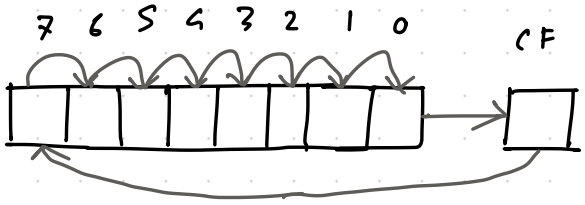
Rotate Right

same as ROL but in opposite direction.

ROR →



RCR →



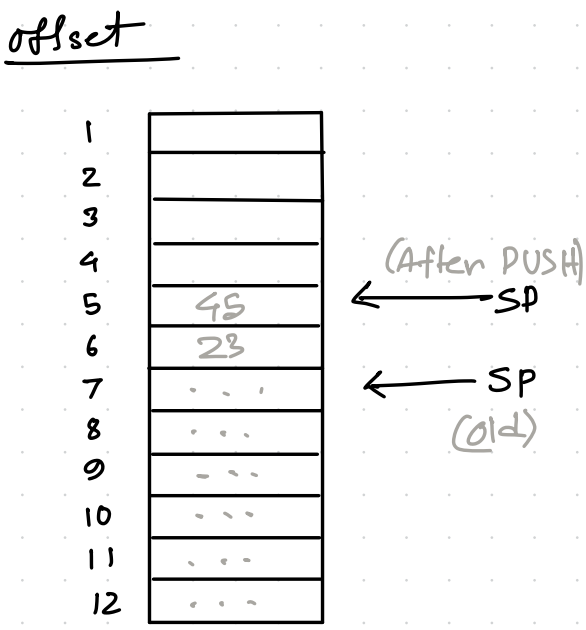
STACK

Segment : Offset

SS : SP

SS → Stack segment

SP → Stack pointer (top)



Stack follows LIFO structure.

PUSH

To add a new data to the stack, we use PUSH. The syntax is:

PUSH source ⟶ (can't be 8 bit (AH, AL, BL ...)

Let's assume, we want to push AX into the stack.

PUSH AX ⟶ 2345H

If the sp points 7, the higher order 8bit of AX, 23H will moved to 6, and lower 8-bit, 45H will move to offset 5. Thus after PUSH operation, SP will decrease by 2.

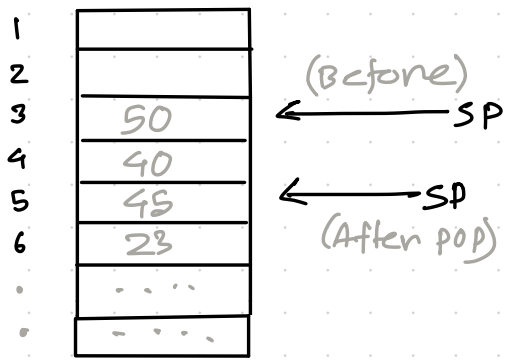
POP

After pop operation, top of the stack will move to destination and SP is increased by 2.

MOV BX, 4050H

PUSH BX

POP AX ⟶ 4050H



INTERRUPT

An interrupt is a condition that halts the microprocessor temporarily to work on a different task then return to its original task.

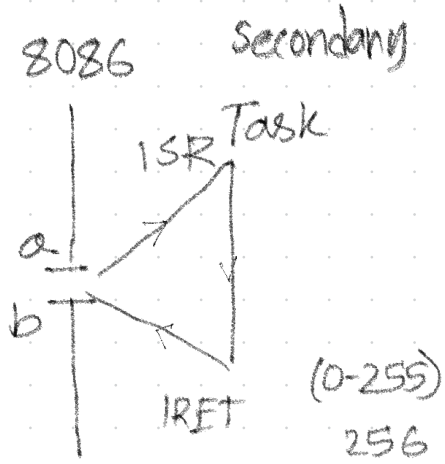


Fig: Interrupt

*ISR → Interrupt Service Routine

*IRET → Interrupt Return

ISR Says what type 0 interrupt is (or other types).

IRET returns back to main program (point b in fig).

(**)

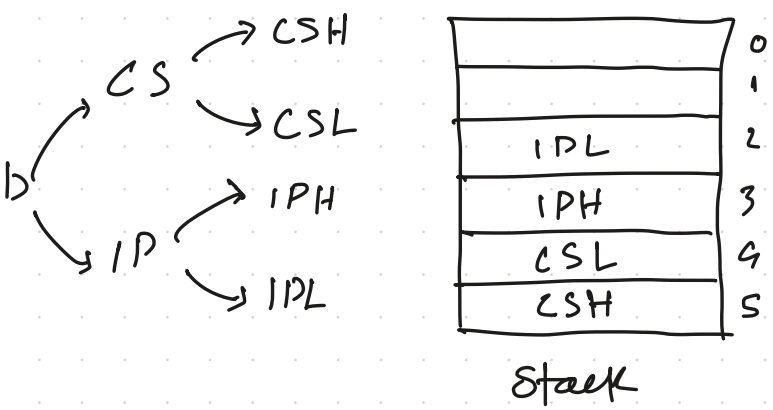
CS : IP → CS holds the segment address

→ IP holds offset address

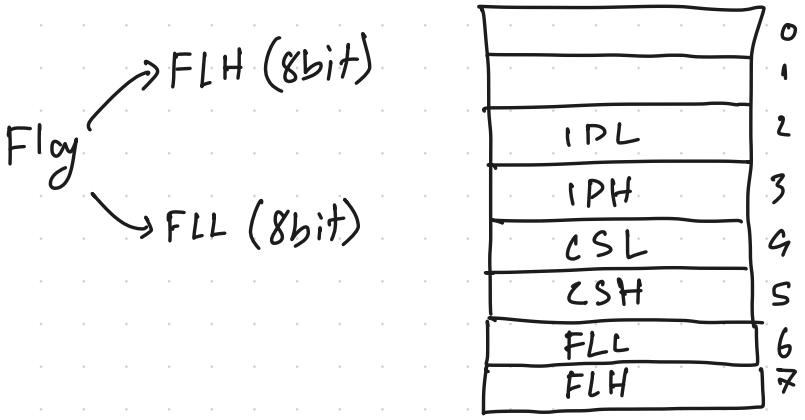
CS → Code Segment

IP → Instruction Pointer

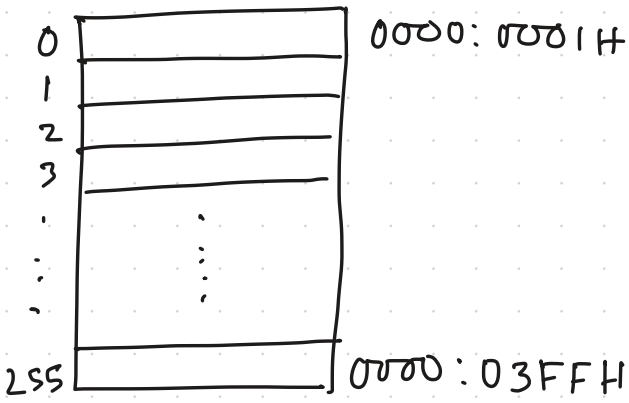
So, the address of b needs to be stored temporarily so that IRET knows from where the program should continue. Thus the address of b [CS:IP] stores in stack.



Interrupt and main program may have different flag values. As a result, values of flag register also needs to be stored is the stack in order to continue the main program.



INTERRUPT VECTOR TABLE



→ Total 256 type

→ Each type = 4 byte

→ Total = 256 X 4 = 1KB

IVT is a structured

QUESTION

** Determine the physical address of the ISR for the given IVT if type '0' interrupt is executed.

	CS : IP	P. A.	memory	
IPL	0000 : 0000	00000H	05H	✓
IPH	0000 : 0001	00001H	34H	
CSL	0000 : 0002	00002H	00H	✓
CSH	0000 : 0003	00003H	20H	
	2000 : 3405			
	2000 : 3406			

Type 0, N = 0

IP = 4 X N = 4 X 0

= 00000H

CS = 4N + 2 = 4X0 + 2

= 00002H

∴ IP = 3405H & CS = 2000H

∴ PA = CS X 10 + IP = 2000 X 10 + 3405

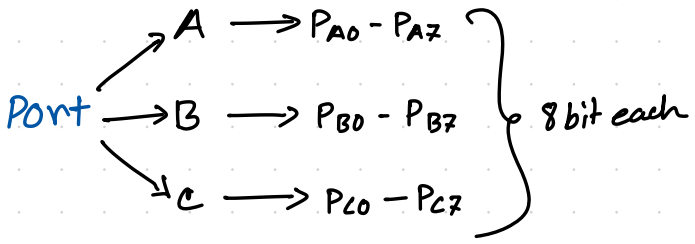
= 20000 + 3405 = 23405H

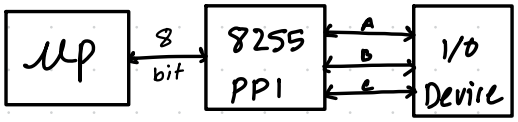
MID ENDS HERE

8255 PPI

PPI = Programmable Peripheral Interface

8255 PPI is a general purpose programable I/O device designed to interface the CPU with peripherals.





* μp can not be directly connected with I/O devices. μp runs much faster than I/O devices which can cause data loss and lagging.

* 8255 is used to Synchronise speed.

Port C → C upper ($P_{C4} - P_{C7}$)

Port C → C lower ($P_{C0} - P_{C3}$)

Group A → Port A + Port C_U [Main Task]

Group B → Port B + Port C_L [Interrupt]

$\overline{CS}$ (Chip Select) → 0 → Operate

$\overline{CS}$ (Chip Select) → 1 → Ignores CPU

$\overline{RD}$ → CPU read from 8255

$\overline{WR}$ → CPU write to 8255

A_0, A_1 } Port select

A_0	A_1	
0	0	A
0	1	B
1	0	C_U
1	1	C_L

Data Bus Buffer: Connects the μp 8 bit data bus with 8255 ports and control register.