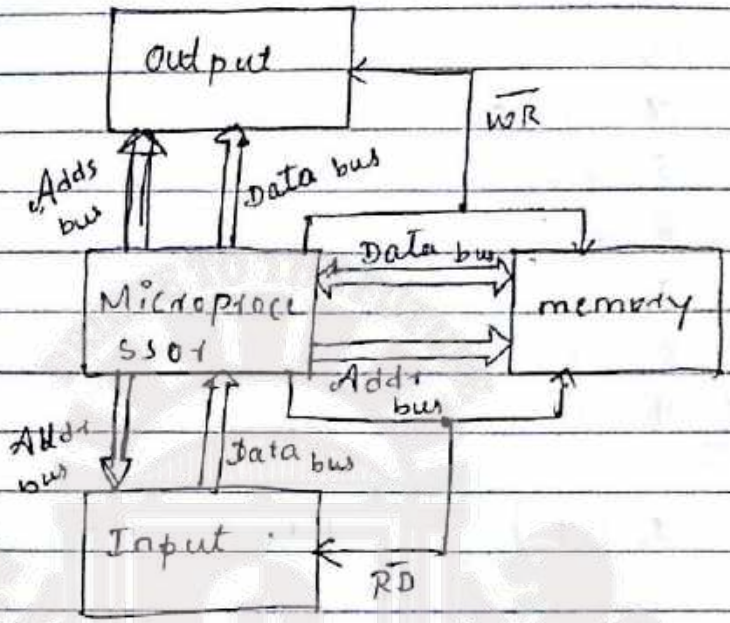
8/2/17

Microprocessor based computer system.

It has 5 components



$\overline{WR}$ → write signal
$\overline{Rd}$ → Read signal

control signal.

1) Input devices :- Keyboard, mouse, scanner, Joystick etc

Output devices :- monitor, Speaker, Printer, projector

Microprocessor :- 1) 8086 / 8080
2) 80186
3) 80286
4) 80386
5) 80486
6) Pentium
7) Pentium 4
8) Pentium 2
9) Pentium 3
10) Pentium 4
11) Core-i2
12) Core-i3
13) Core-i5
14) Core-i7

PAGE :
DATE : / /

Buses :- group of communication lines carries the same type of information in the form of bits, ie either data, or address or control information.

There are 3 types of buses.

1. Address bus :- It carries address information from microprocessor to memory device or i/o device. It is unidirectional.

Suppose, A processor has n-address lines or n-bits of memory address, The no of memory locations can be addressed into, that processor is 2^n

$n = 20$ bits

no of memory address $= 2^n = 2^{20} = 1M$

$1K = 1024 = 2^{10}$

$1M = 2^{20} = 1K \times 1K$

$1G = 2^{30} = 1K \times 1K \times 1K$

Address in Binary

$A_{19}A_{18}A_{17}A_{16}A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0$	Addr in Hex
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0H
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 0 0 0 1 H
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	0 0 0 0 2 H
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	0 0 0 0 3 H
⋮	⋮
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	F F F F F H

The above table shows processor has 20 address lines ie $2^{10} \times B$

$$2^{10} \times B = 2MB$$

size of each memory location is 8-bits or 1 byte. ∴, The no of bits can be stored in a processor which has 20 address bits is equal To :

$$2^{20} \times 8 \text{ bits} = 1M \times 8 \text{ bits} = 1MB$$

Data bus :- It carries data information from memory to processor or vice versa depending on the instruction. It also carries data information from processor to O/p device or I/p device to processor.

The width of data bus depends on the word length of the microprocessor.

Control bus :- It carries the control information from processor to i/p devices or memory device. $\overline{Rd}$ and $\overline{WR}$ are the control signals for Read and write operations respectively ..

Book No. 17

PAGE :
DATE : / /

1. Briefly explain the histry of intel family of microprocessors.

How many bit of Processor
data bus width
Add bus —11—
no of instruc-n
memory size etc

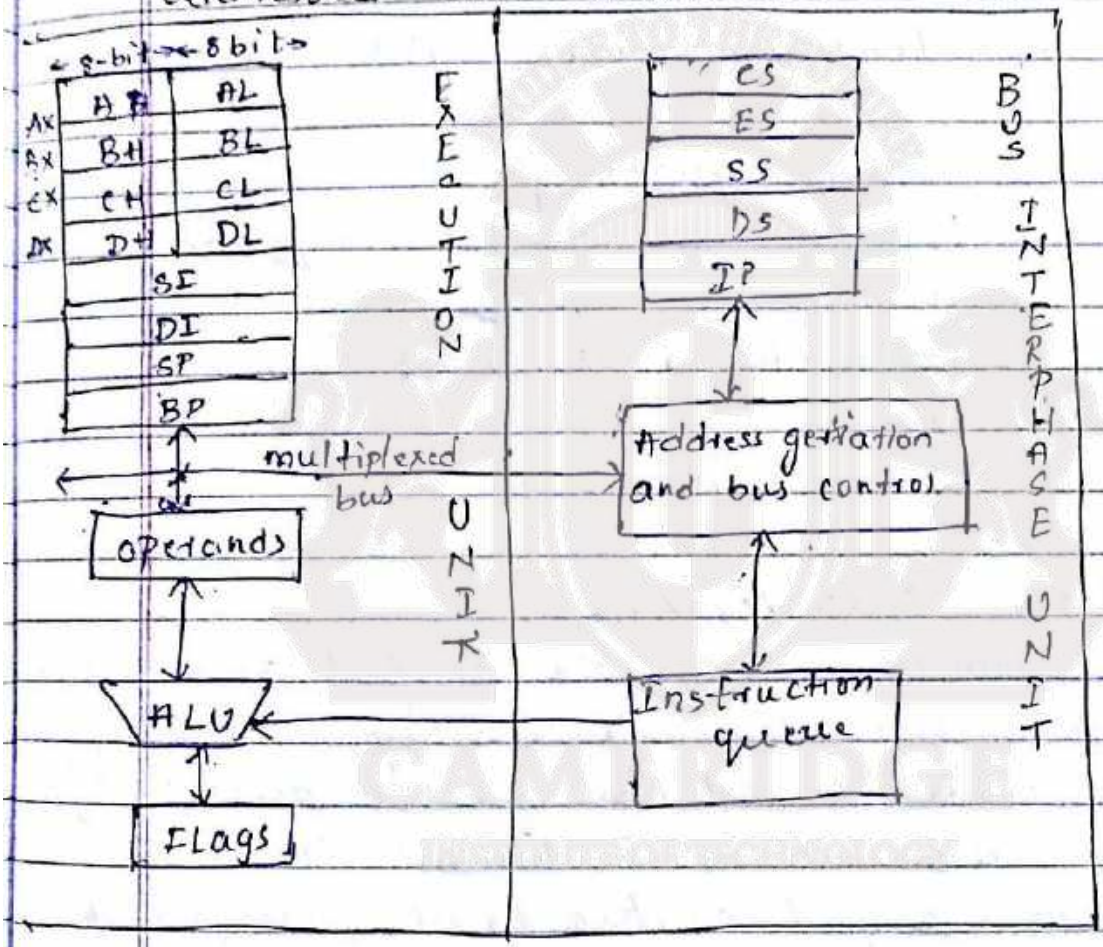
2.

1 GiB

PAGE :
DATE : / /

Internal block diagram of 8086/8088µP
[architecture of 8086/8088

1. 8086/88 is a 16 bit µP
2. width of data bus is 16-bit
3. width of Address bus 20 bits
4. no of memory location is that can be addressed into 8086/88 is 2^{20} = 1MB



General purpose Registers :-

→ AX [Accumulator register]

→ It is sometimes used to hold the result of Division and multiplication

→ It is used with I/O instructions.

→ It is also used as a 16-bit GP register

→ AL is an 8-bit accumulator.

→ It can also be used as two 8-bit independent registers AH and AL

BX :- Base register

→ It sometimes holds the address of a memory location

→ It is also used as 16-bit gp register.

→ It can also be used as two 8-bit independent registers BH and BL

CX :- Count register :-

→ It acts as a count reg for loop operation or instruction.

→ It is also used as a 16 bit GPR.

→ It is also used as two 8-bits independent register CH and CL.

DX :- Data register :-

→ It sometimes holds the partial result after the multiplication and division operations.

→ It holds the address of i/o port [2^{16}]

→ It is also used as a 16-bit GPR

→ It is also used as two 8-bit independent register DH and DL

Index Registers and pointers :-

SI [Source Index] :- It sometimes holds the address of source string in case of string instructions

→ It can also be used to store the address of a memory location in data segment.

DI [destination index] :-

→ It sometimes holds the address of destination string in case of string instructions.

→ It can also be used to hold the address of a memory location in data segment.

SP [stack pointer]

→ It holds the address of the top of the stack. ie it is used to access the data from the top of the stack, in the stack segment

BP [base pointer]

→ It is used to access data from any memory location in the stack segment

IP [Instruction pointer]

→ It points to the next instruction while executing current instruction.

→ ~~It hold~~ It holds the address of next ins" while executing current instruction

Segment Registers

CS [code segment register]

→ It holds the segment address of code segment.

ES [Extra segment]

It holds the segment addr of Extra segment

code	Extra	stack	Data

DS [Data Segment]
Holds the Segment addr of data Segment
SS [Stack segment]
Holds the Segment addr of stack segment.

1. List the 8-bit registers available in 8086

Ans AL, AH, BL, BH, CL, CH, DL, DH

2. List the memory pointer registers available in 8086

A: SI, DI, SP, BP, IP, BX

3. List 16-bit registers Available in 8086

A:- AX, BX, CX, DX, SI, DI, BP, SP, IP, CS, ES, SS, DS

4. List the segment reg available in 8086

A: CS, ES, SS, DS

Flag register :-

→ It is a group of flip-flops.
→ Each bit of bit of flip flop can be set or reset individually

B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
				0	D	I	T	S	Z		A		P		C

8086 flags are of 2 types.

PAGE :
DATE : / /

1 → status or conditional flags [O, S, Z, A, P, C]

→ These flags indicate the status or condition of microprocessor after the arithmetic or logical operations

2 → control flags [D, I, T]

→ These flags are used to control the operation the microprocessor.

11/2/19

1. carry flag :- [C]

→ carry flag will set if there is a carry generated from the MSB after the arithmetic operations.

→ carry flag is the borrow flag in subtraction operation.

2. Parity flag [P] :-

→ It indicates whether the result is odd parity or even parity after the arithmetic or logical operations.

→ If the the result is the even parity, P = 1 [set], else if P is odd parity P = 0 [reset]

3 A :- [Auxillary carry flag] :-

→ It indicates whether carry is generated from Bit B_3 to B_4. If the carry is generated from B_3 to B_4, A = 1 otherwise A = 0.

4. Zero flag [Z] :- It indicates whether the result is zero or not. If the result is zero, Z=1. else Z=0 [non-zero]

5. Sign flag [S] :-
→ It indicates whether the result is positive or negative.
→ If the result is negative, S=1 else S=0
→ If MSB = 1, -ve S=1 else +ve S=0

6. O [Overflow flag] :-
→ It will set if the result cannot fit into a specified register.

7. Interrupt flag [I] :- It controls the operation of an external interrupt pin of the µP. INTR.
→ It Enables or disables external interrupt.
→ If I = 1, Interrupt request is enabled else if I = 0 —— " —— disabled.

8 Direction flag :- [D] It selects auto Increment mode or auto decrement mode for string instructions
→ If D=0, Selects auto increment mode, if D=1 selects auto decrement mode.

9. TRap [T] :- It is used to Debug the application programmes. If T=1, processor enters into step by step executing mode.

PAGE :
DATE : / /

Overflow

Signed -2^{n-1} to $2^{n-1}-1$
Unsigned 0 to 2^n-1

if $n=4$.
-2^3 to 2^3-1
-8 to $+7$.

Unsigned 0 to 2^4-1
0 to 15.

Signed

$+7$, $+6$, $+13$ overflow

$n=4$

$+7$, -2, $+5$ (No, overflow)

Unsigned

$+13$, $+12$, $+25$ (overflow)

$+10$, $+3$, $+13$ (No overflow)

1. Show the states of status flags after the following operation.

8+15

1 — 38 H — 00111000
23 — 2F H — 00101111
16 — 67 — 01100111 — 67 H

C = 0 (Not set)
Z = 0 (Not set)
A = 1 [B_3 to B_4 carry]
S = 0 {MSB = 0}
P = 0 {odd parity}

12 + 6

16) 18 (1
16
2

96 H — 1.001 0110
8C H — 1000 1100
1 H — 0001 0010
22 H

= 22H C=1

C = 1
Z = 0
A = 1
S = 0
P = 1

(4/2/17

Memory Segmentation :- [Real mode memory addressing]

[Program segment, Physical and logical address

The memory segment has 4 segments
1. code segment
2. data segment
3. stack segment
4. extra segment

→ The memory size of 8086 is 2^{20} = 1MB. This is also called as physical memory of 8086

→ The physical address is a 20-bit address and it is divided into segment address and offset address.

→ The segment address defines the beginning of a memory segment.

ave

PAGE :
DATE : / /

→ The offset address selects any location within the memory segment. The maximum size of a memory segment is 64 KB.

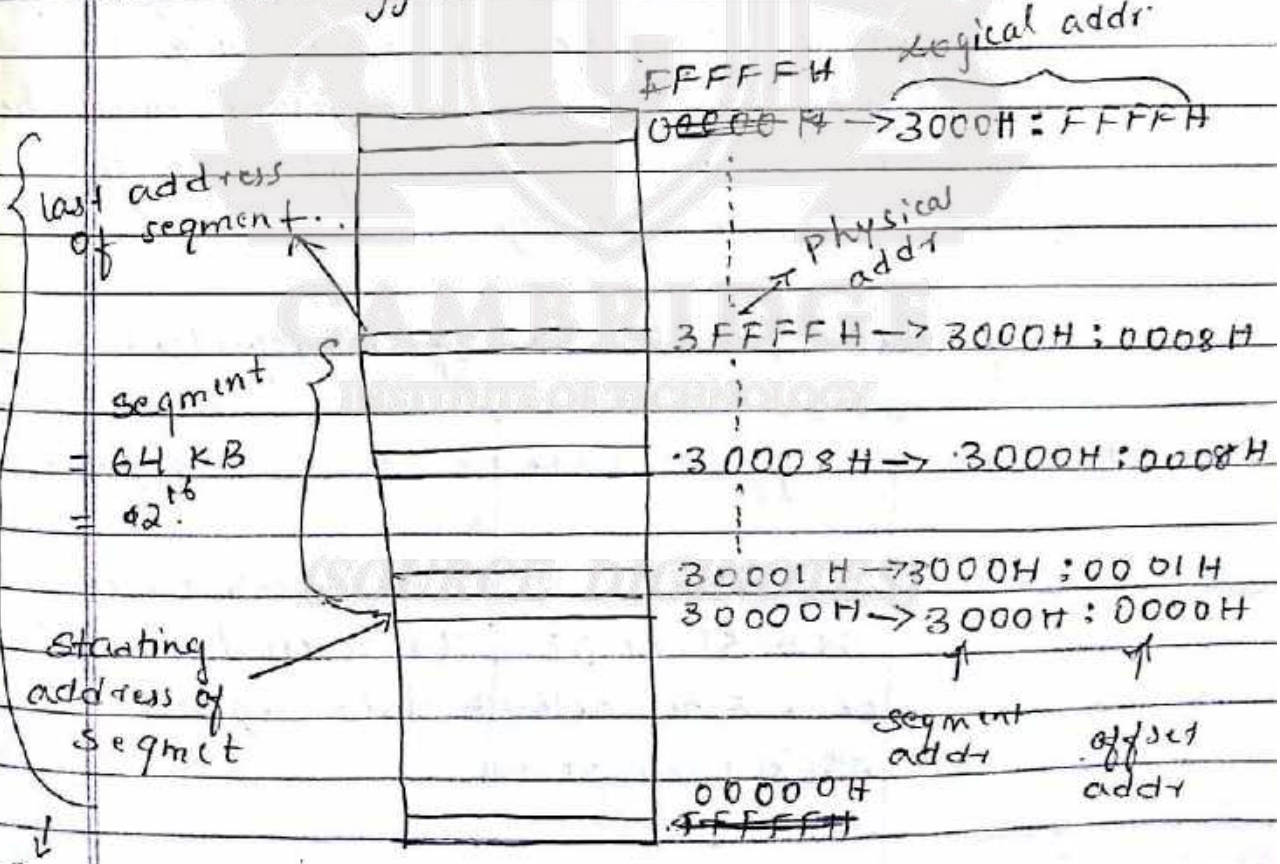
→ The segment address range from 0000H to FFFFH.

→ The physical address is a 20 bit address that is actually put on the address pin of 8086 microprocessor.

→ Physical address can range from 00000H to FFFFFH.

→ The ending address of a segment is found by adding FFFFH

→ The logical address consist of a segment value and an offset address



calculation of Physical addr.

Segment addr x 10H + Offset addr

Advantages :-

→ segment + offset addressing scheme allows programs to be relocated in the memory
→ A relocatable program is one that can be placed into any area of the memory and executed without change.
→ Because memory is addressed within a segment by an offset addr, the memory segment can be moved to any place in the memory system without changing any of the offset addresses and only the content of the ~~eff~~ segment register must be changed to execute the program in the new area of memory

Default Segment and offset registers

Segment addr	offset addr	Purpose
CS	IP	To fetch code
DS	BX or SI or DI or an 8-bit or 16-bit displacement present in the instruction	To access data in the data segment
SS	SP or BP	To access memory location in stack segment
ES	DI	To access extra segment in string instructions

1K

2

0008H

PAGE:
DATE: / /

ES 1

calculate the physical address of the memory locatio to accew code, data and stack

given DS = 1200H
CS = 2A10H
SS = 3940H
BX = 1900H
SP = 2ABCH
IP = 1939H

code = CS x 10 + IP.
= 24100H + 1939H.
= 25A39H

Data = DS x 10 + BX.
12000 + 1900H.
13900H.

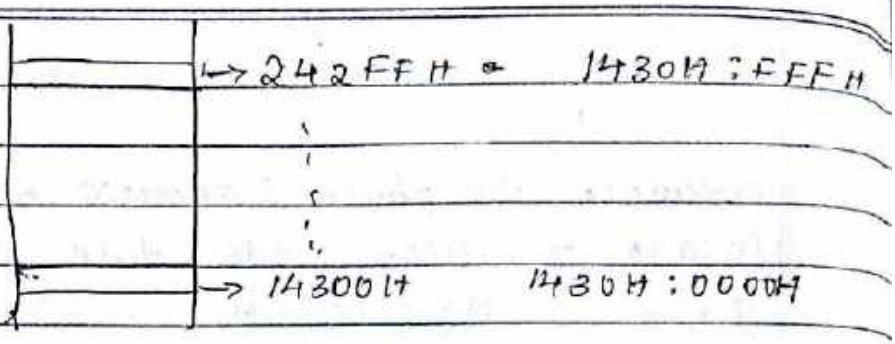
Stack SS x 10 + SP.
= 39400H + 2ABCH.
3BEBCH.

In the real mode, the starting and ending address of each segment located by the following segment register values.

given :- i) 1430H

starting addr = 1430H x 10H
= 14300H.

Ending addr = 14300H x 10H + FFFFH
24 2FFFH



2. D100H

starting = D100H × 10H
D1000H.

ending = D100H × 10H + FFFFH.
D1000H + FFFFH.
E0FFFH //

3. If CS = 24F6H
IP = 634AH

Show the following.

i) Logical address

24F6H : 634AH

ii) The offset addr
634AH

iii) Physical addr
24F60H + 634AH.

2B2AAH.

PAGE:
DATE: / /

The lower addr of Code Segment
24F6H x 10H + 0000H
= 24F60H //

4. The upper range addr of code segment
24F6H x 10H + ~~6348~~ FFFFH
24F60 + FFFF.
34F5FH

4. If DS = 7FA2H and the offset is 438EH calculate
i) Physical addr
7FA2H x 10H + 438EH
= 7FA20H + 438EH = 83DAEH
83DAEH.

ii) lower range of data segment
~~7d~~ 7FA20H.

iii) Upper range addr
7FA20H + FFFFH = ~~8FA1F~~ 8FA1FH

iv) Logical address
7FA2H : 438EH.

5- If SS = 3500H and SP = FFFEH
i) physical Addr
3500H x 10H + FFFEH
= 35000
FFFF
~~4~~ 34FFEH

FF590
FFFF
0F58FH

ii) Lower range :-

:. 35000H .

iii) Upper Range .

35000H t FFFFFH .

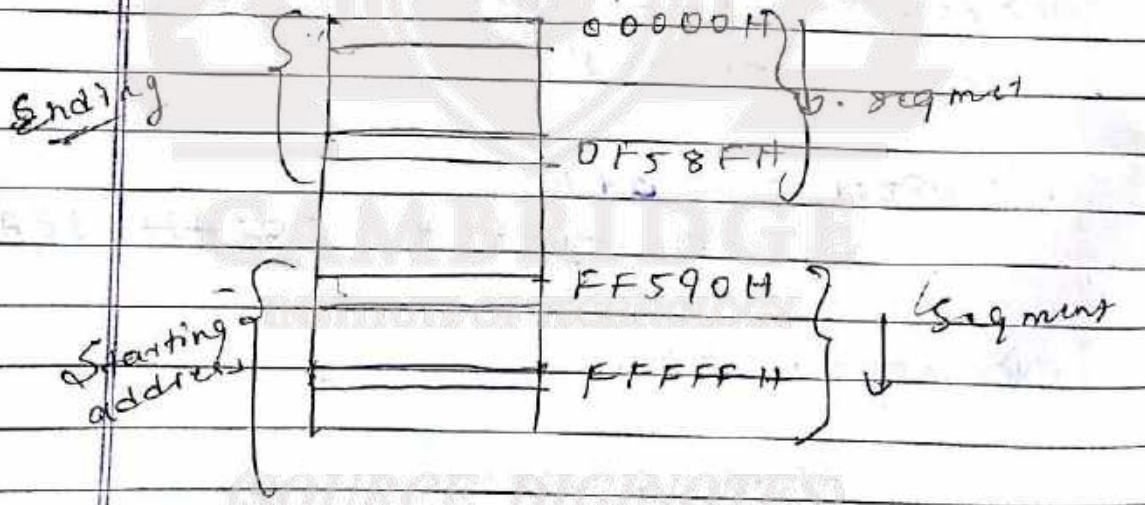
4AFFFH .

iv) Logical addrs .

3500H : FFFEH .

6. What is the range of Physical address in the code segment if CS = FF59H

FF590H to 0F58FH FF



PAGE :
DATE : / /

Addressing Modes :-

It is the method of Accessing data from the memory.

Types :-

1. Immediate addressing mode } 8086 µP
2. Register addressing mode
3. Direct A.M
4. Register Indirect A.M.
5. Register Relative A.M
6. Based - Indexed A.M.
7. Based - Indexed Relative A.M.
8.

1. Immediate A.M :-

In this addressing mode, the data is specified in the instruction itself.

Syntax

MOV Dest, Source
↓
Mnemonic

eg:- MOV AL, 38H ;
MOV AX, 8432H ;

2- Register AM :-

In this addressing mode, the data is specified using an 8-bit or 16-bit register and it is referred using a register.

intel i3, i5, i7 series
intel of.. 2 C
AMD - FX series.

Eg: MOV BL, AH; } valid
MOV AX, BX;

MOV CL, BX } invalid
MOV BX, CL

Because CL is 8 bit and BX is 16 bit
it operand mismatch.

3. Direct A.M

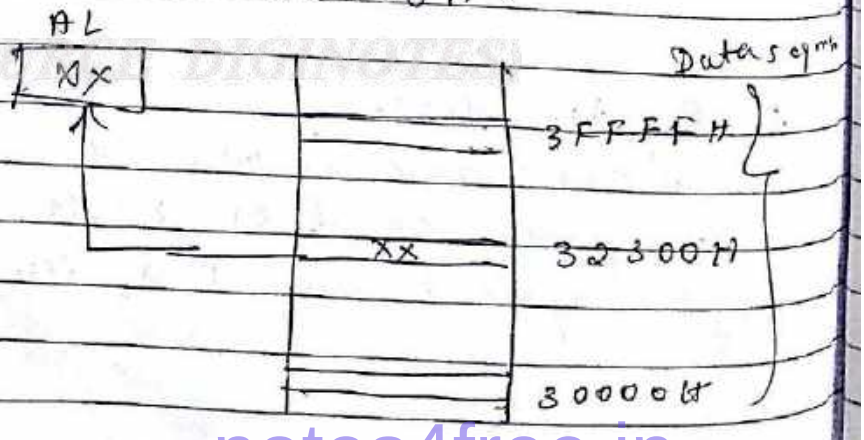
In this A.M, the offset address of a memory location in data segment is specified in the instruction itself.

Ex:
MOV AL, [2300H] ← offset address

Assume DS = 3000 H [segment address]

3000H : 2300H

Physical address of the memory location
= DS x 10H + offset address
= 3000H x 10H + 2300H.
= 32300H.



w.r.t X86 inst X84-

80386 386 endian 8

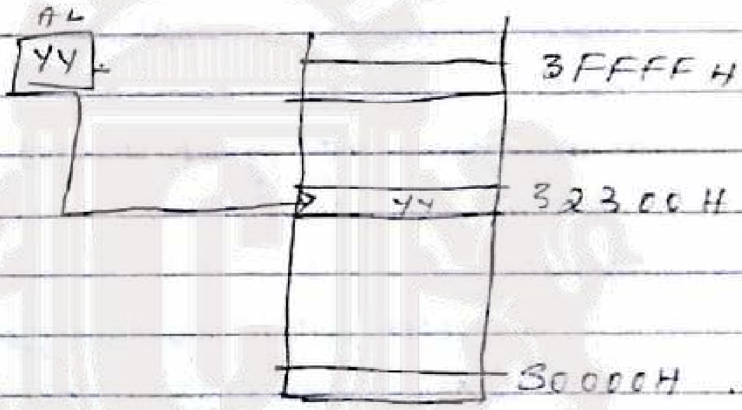
PAGE :
DATE :

After the execution of MOV AL, [2300H], the content of the memory location whose offset address is 2300H is copied into AL register.

ie the data XX is copied into AL register

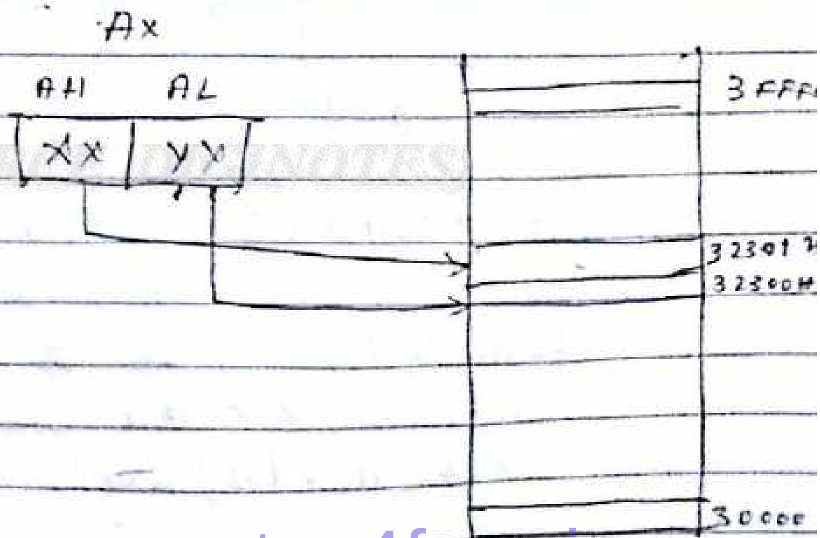
Ex 2:

· MOV [2300H], AL

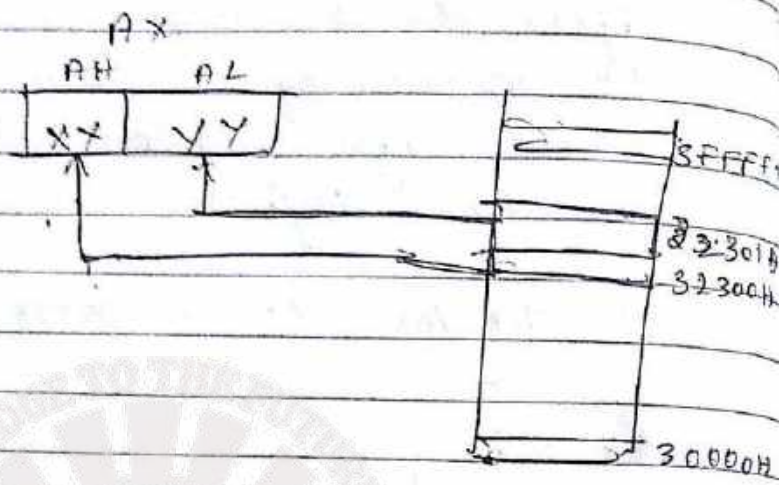


Example 3

Mov [2300H], AX



Ex 3 : MOV AX, [2300H]



4. Register Indirect AM :-

In this A.M, the offset address of the memory location is indirectly specified using a Register.

→ The registers used to specify the offset address are BX, BP, SI, DI, SP.

The physical addr of the memory location is calculated using expression.

DS X 10H + [BX or SI or DI]

(OR)

SS X 10H + [BP or SP]

Example :-

mov. SI, 2300H

MOV AL, [SI]

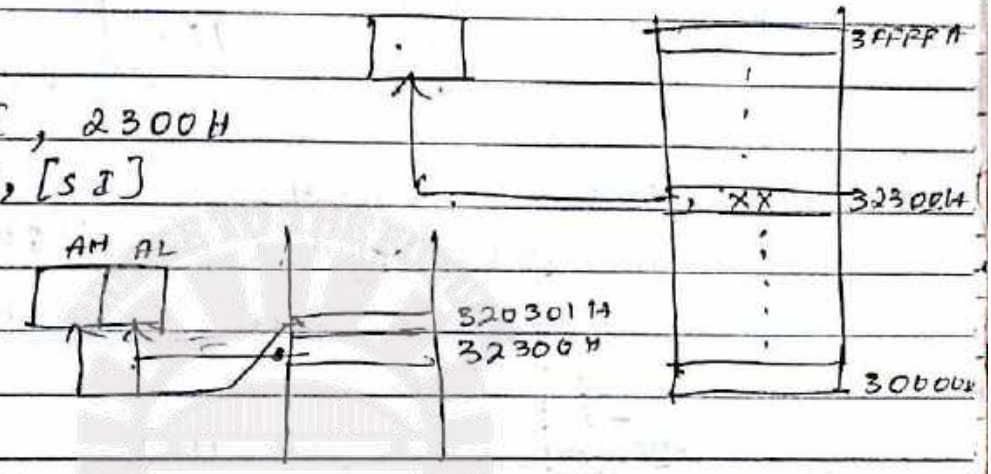
PAGE :
DATE : / /

Assume DS = 3000H

Physical addr = DS X 10H + SI
= 3000H X 10H + 2300H
= 32300H

2. Ex 1

mov SI, 2300H
mov AX, [SI]



Note :-

1. If the Base Registers [BX or BP] is used to specify the addr, the register indirect A.M is called as Based A.M

ex ① mov AL, [BX].
② mov AL, [BP]

2. If the index register [SI or DI] is used, the A.M is called as Indexed A.M

ex ① mov AL, [SI]
② mov AL, [DI]

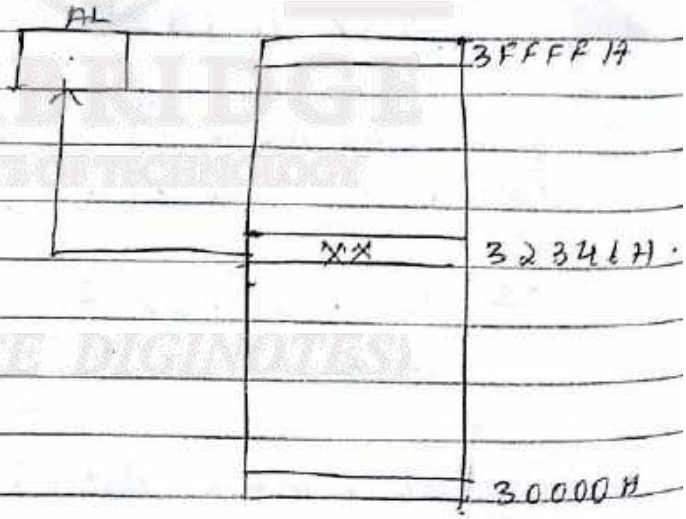
3. Register relative A.M :-
In this AM, the effective offset address of the memory location is calculated by adding the contents of the register [BX or BP or SI or DI or SP] with an 8-bit or 16-bit displacement specified in the instruction.

Physical addr: { DSX 10H + [BX / SI / DI] + Displacement
{ SSX10H + [BP / or / SP] + Displacement.

mov SI, 2300H
mov AL, 46H[SI] (or mov AL, [SI+46H])
Displacement

Assume DS = 3000H

P.A = DSX10H + SI + Displacement
= 3000H X 10H + 2300H + 46H
= 32346H



PAGE :
DATE : / /

Based Index A.M :-

In this A.M, the effective offset addr of the memory location is calculated by adding the contents of Base register [BX or BP] and the contents of Index Register [SI or DI]

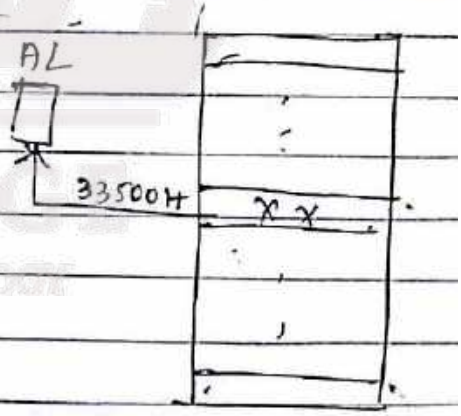
Physical addr = { DS X 10H + BX + (SI or DI)
SS X 10H + BP + (SI or DI) }

mov AL, [BX][SI].

mov BX, 1200 H
mov SI, 2300 H

Assume DS = 30000H

Physical addr = 30000H X 10H + BX + SI
30000H + 1200H + 2300H
= 33500H



Based Indexed Relative

In this AM, the effective offset addr of the memory location is calculated by : Adding the contents of Base register [BX or BP] and the contents of Index Register along with Displacement

Physical addr = { DS x 10H + BX + [SI or DI] + Displacement
: SS x 10H + BP + [SI or DI] + Displacement

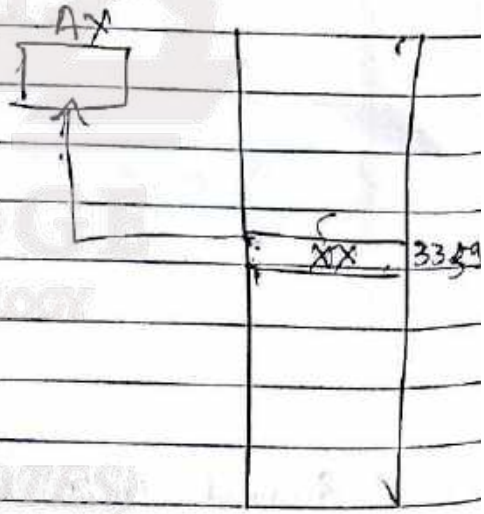
mov BX, 1200H
mov SI, 2300H

mov AL, 93H[BX][SI]

Assume DS = 3000H

30000 + 93H + 1200 + 2300

30000
1200
2300
93
33593H



PAGE:
DATE: / /

1. Identify the A.M of the following instructions

(i) mov AX BX
Register A.M

ii) mov CX, [BP]
Register Indirect ~~one~~ A.M and also belongs to Based A.M.

iii) ADD DX, [1836H]
Direct Addressing Mode.

iv) SUB BX, [SI] → Register indirect and Indexed A.M.

(v) mov DL, 2CH
Immediate A.M

(vi) ADD BX, [BP][DI]
Based Indexed A.M.

(vii) SUB CX, [BX + SI + 1938H]
Based index Relative A.M.

(viii) SUB CX, [SI + 383AH] → Register relative.

2. Identify the Addressing Mode of the following Instruction and also calculate the physical addr of the memory loc^{n} accessed by the microprocessor

Assume :- DS = 1800H
SS = 2610H
SI = 4936H
DI = 9320H
BP = 1930H
BX = C120H

1. MOV [BP], DX → Register Indirect

P.A. SS X 10H + BP.
2610H X 10H + 1930H
26100
1930
27A30H.

2. MOV CX, [SI] → Register Indirect

DS X 10H + SI
18000H + 4936H.
= 1C936H

18000
4936
1C936H

3. MOV [9300H], DX → Direct A.M.

DS X 10H + Offset
18000 + 9300H
= 21300H

18000
9300
21300H

4. MOV [BP+29H], AX Register relative

SS X 10H + BP + 29H
26100H + 1930H + 29H
= 27A59H

26100
1930
29
27A59

PAGE:
DATE: / /

5. mov [BX+DI], DL . Based Index

DS x 10H + [BX + DE]
18000H + [C120
9320
5440

= 23D440H

18000
C120
9320
BD440

6.

6. ADD CL, [BP+SI+2346H] .. Based index relative

SS x 10H + BP + [SI] + Displaceme

26100H x 10H + 1930H + H936H + 2346H .

26100
1930
4936
2346
2E6ACH

22
15

Pipelining and Instruction Queue:

In 8086, to speed up the execution of program, the instruction fetching and execution of instructions are overlapped each other. This technique is known as pipelining.

In pipelining, when the n^{th} instruction is executed, $(n+1)^{th}$ instruction is fetched

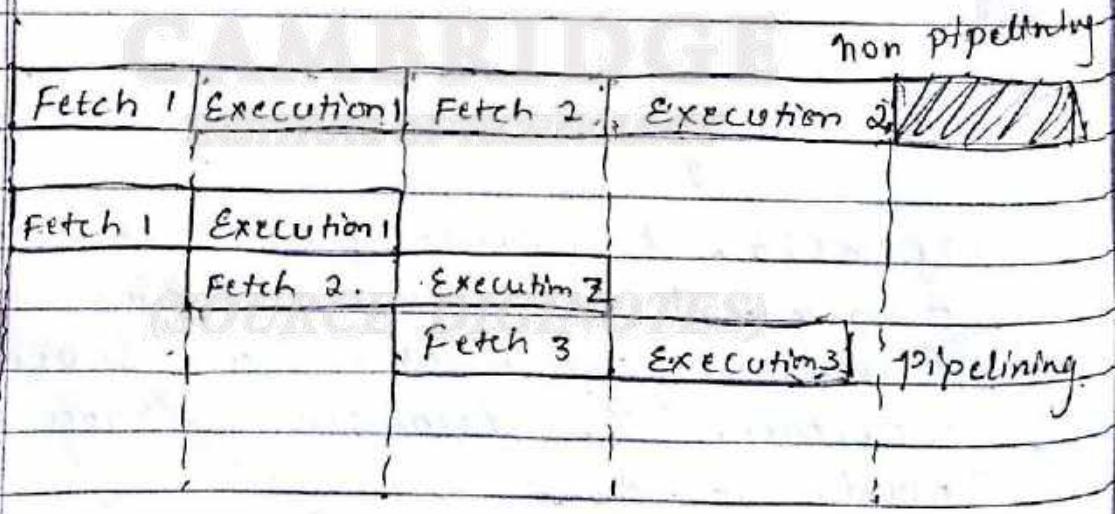
and thus the processing speed is increased

Intel implemented the concept of Pipelining by splitting the internal Architecture of the micro processor into two sections.
ie (i) execution unit [EU]
(ii) Bus interphace Unit [BIU]

These two sections works simultaneously, The BIU access the memory, while EU executes Instructions previously Fetched.

BIU has a buffer or instruction queue. The queue is 6-bytes long in 8086 and 4 bytes long in 8088.

queue is used to prefetch and store at maximum of 6-bytes of instruction code from the memory.



PAGE :
DATE : / /

:- Stack :-

Stack is a section of Read/write memory used by the CPU to store the information temporarily

→ The CPU needs this storage area since there are only limited no of registers.

The two registers used to access the stack are SS and SP

→ Every register inside the 8086 [except segment registers and SP] can be stored in the stack.

→ The storing the content of CPU registers into the stack is called push operation.

→ Loading the contents of the top of the stack into the CPU registers is called ~~Push~~ POP operation

→ PUSH and POP instruction are used to access the stack

PUSH instruction.

Syntax :- PUSH operand
↑
(Reg/memory)

ex: PUSH AL // push the content of Reg. AL onto the top of the stack
// SP is decremented by 1

ex :- push BX // push the content of Bx onto the stack, sp is decremented by 2

→ This instruction copies the register specified in the instruction into the stack.

→ The stack pointer [SP] is decremented by 1 or 2 depending on the size of the operand.

→

POP instruction :-

Syntax :- POP operand
(reg/memory)

→ This Instruction loads the specified register or memory location with the contents of the top of the stack.

→ SP is incremented by 1 or 2 depending on the size of the operand.

ex:- POP AL // loads the reg AL with content of TOS, SP is incremented by 1

Note :- ~~the register~~

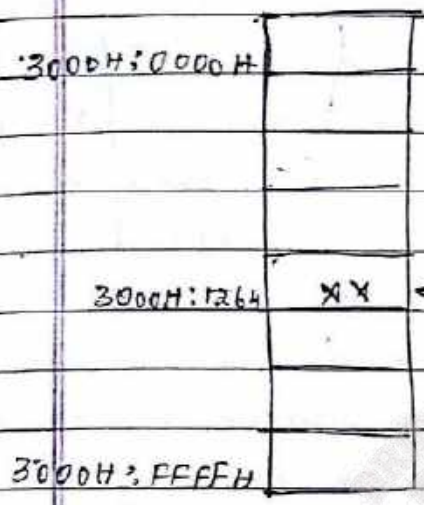
ex:- POP BX // sp incremented by 2 because the size of BX register is 2 - bytes.

Move :- MOV AX, 834CH
PUSH AX

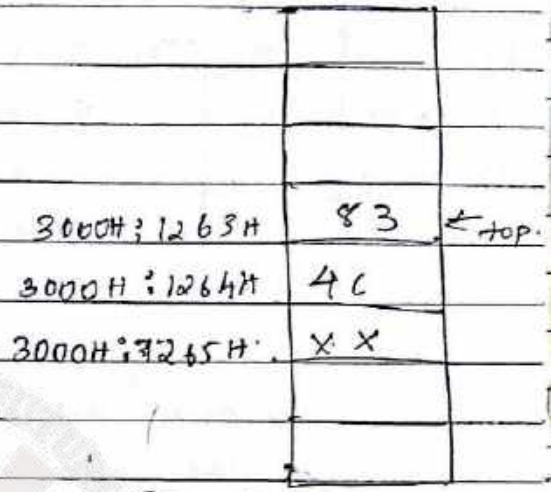
assume · SS = 3000H .
SP = 1265H

PAGE:
DATE: / /

Before execution



After execution



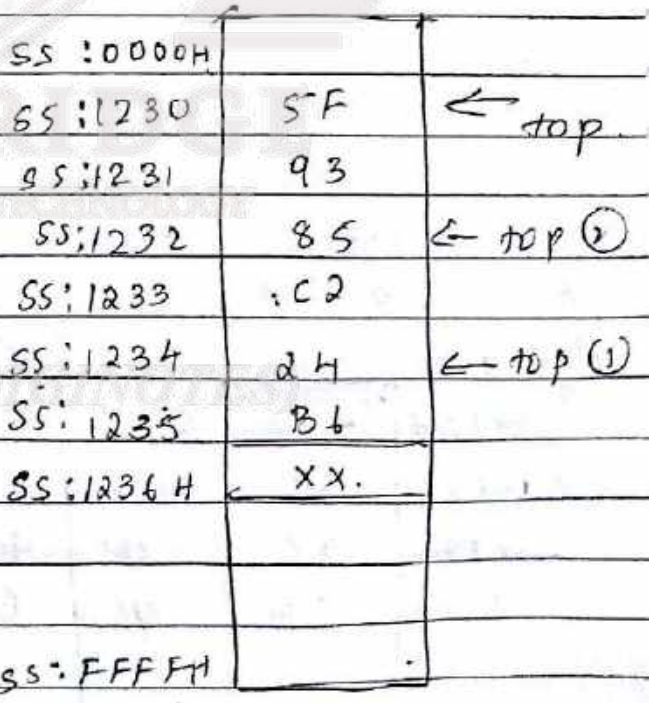
SP = 1263H

Example 2:-

Assuming that SP = 1236 H
AX = 24B6 H
DI = 85C2 H
DX = 5F93 H

Show the contents of the stack as each of the following instructions are executed sequence.

PUSH AX
PUSH DI
PUSH DX



SP = 1234H

After PUSH AX | After PUSH DI | PUSH DI

					SS:1230	5F	←
		SS:1232	55	←	SS:1231	93	
SS:1234	24	← SS:1233	C2		SS:1232	55	
SS:1235	B6	SS:1234	24		SS:1233	C2	
SS:1236	XX	SS:1235	B6		SS:1234	24	
		SS:1236	XX		SS:1235	B6	

1. Assuming that the stack is as shown below and SP = 18FB, show the contents of the stack and registers at each of the following instructions executed in sequence.

```
POP CX
POP DX
POP BX
```

Before

SS:18FBH	43	←
:18FCH	89	
:18FDH	C2	
:18FEH	3F	
18:FFH	9C	
1900H	6B	

SP = 18FBH

Ans

After POP CX: CX = 4389H

After POP DX: DX = C23FH

After POP BX: BX = 9C6BH

	After POP CX	After POP DX	After POP BX
18FDH	C2 ←		
18FEH	3F		
18FFH	9C	9C ←	1901H
1900H	6B	6B	

SP = 18FDH | SP = 18FFH | SP = 1901H

$ → End of the string

PAGE :
DATE : / /

Assembler directives

These are key words used in the assembly code. Assembler directives will direct the assembler to perform specific task during assembling process.

→ There is no equivalent machine code generated for an assembler directive.

1. Data type defination :-

DB [Data byte] :- It reserves a byte or bytes of memory location in the available memory.

N[0]	84 H
N[1]	93 H
N[2]	84 H

Ex :- ① N DB 84H

② N DB 84H, 93H, 84H

③ N DB ? [one byte of memory location is reserved but unallocated ie uninitialized]

④ N DB ?, ?, ? // 3 bytes of memory location

⑤

⑥ MSG DB "GOOD MORNING"
↑ ASCII

⑥ LIST DB 40 DUP(?)
40 * duplicate memory locations are reserved, but all are uninitialized

DUP → Assembler directive.

⑦ MSG DB 10, 13, " _____ "
↓ ↓
OD OA → Enter

2. DW [Data word]

It reserves a word or words of memory locations in the available memory.

```
SRI    LIST   DW   3896H, 4634H
```

List[0]	96
List[1]	38
List[2]	34
List[3]	46

3. DD [~~Double~~ Data double word]

It reserves Double word memory locations in the available memory

Double word = 4 bytes // 2 words.

N DD

N[0]	B6	} Double word
[1]	C2	
[2]	96	
[3]	43	

4. DQ [Data quad word]

Quad word = 8 bytes // 4 words

N DQ 4396C2B64804040402.

PAGE :
DATE : / /

N[0]	43		N[07]	02
[1]	96		.	04
[2]	C2			06
[3]	B6			48
[4]	48			B6
[5]	06			C2
[6]	04			96
[7]	02			43

5. Data Ten Bytes [DT].
. Reserves Ten bytes of memory locations.
N DT 43 96 C2 B6 48 06 04 02 11 41.

N[0]	41
N[1]	11
N[2]	02
N[3]	04
N[4]	06
N[5]	48
N[6]	B6
N[7]	C2
N[8]	96
N[9]	43

6. EQU :- This assembler directive is used to assign a label with a constant value.

LEN EQU 3
↑ ↑
Label. constant.

• model

It Specifies the no of data and code segment used in the program.

There are 4 models

• model	No. of code segments	No of Data Segments
SMALL	1	1
MEDIUM	>1	1
TINY/ COMPACT	1	>1
LARGE	>1	>1

LENGTH :- This directive is used to find the length of Data array or string

```
LIST DB 21H, 43H, CIH, 9AH.
MOV AX, LENGTH LIST.
```

MACRO and ENDM

(macro)

→ It is a group of instructions that perform one task

→ Macro is accessed during assembly with a name given to macro when defined.

→ Macro and Endm directives are used with macro sequence.

→ Macro assembler directive indicate the beginning of A macro sequence.

→ Endm directive indicate the end of the macro sequence.

PAGE :
DATE : / /

Example :

main Program

```
PRINT MACRO
   ---------
   ---------
   ---------
ENDM.
```

Name of the macro

```
   ---------
   ---------
PRINT
   ---------
   ---------
   ---------
PRINT.
```

Example 2 :- // with parameter.

```
PRINT MACRO .M.
. LEA DX, M
  MOV AH, 09H
  INT 21H
    ENDM.
.DATA
M&&I DB "- - - - -"
M2 DB "- - - - -"
```

```
   ---------
   ---------
PRINT M1   // Displays string M1
   ---------
   ---------
PRINT M2   // Displays string M2.
   ---------
   ---------
```

PROCEDURE [Subroutine]

→ It is equivalent to the function in C language.

→ It is a group of Instructions that perform one task.

→ It is a Reuseable Set of Instructions Stored in a memory once but used as often as necessary.

→ The CALL instruction links the procedure and RET instructions Returns from the procedure

→ STACK stores the Return address whenever a procedure is called during the Execution of a program.

→ The CALL Instruction push the Return addr onto the stack

→ RET Ins^n copies the return addr from stack into IP register.

→ The PROC Assembler directive Indicate the beginning of a procedure

→ ENDP Assembler directive Indicate the End of a procedure.

Example:

(Name of the procedure → PRINT)

```
——                  | PRINT PROC
——                  |   ——
——                  |   ——
CALL PRINT          |   ——
——                  |   ——
CALL PRINT          |   RET
                    | ENDP
```

PAGE :
DATE : / /

Difference b/w MACRO and PROCEDURE

MACRO	PROCEDURE
→ MACRO and ENDM assembler directives are used	→ PROC and ENDP assembler directives are used
→ Accessed during Assembly using the name given to macro.	→ Accessed using CALL and RET instructions During Execution of the program.
→ Machine code is generated for instructions each time a macro is called	→ machine code is generated only once and put in the memory.
→ with MACRO more code memory is required	→ with procedure, less code memory is required.
→ It doesn't use stack.	→ It uses stack inorder to save the return add.

OFFSET :-
When the assembler comes across the OFFSET Assembler directive along with a label, the offset addr of the Label is copied into the Register specified in the instruction.

Example NUM DB 23H
- - -
- - -

MOV BX, OFFSET NUM // LEA BX, NUM

Equivalent to LEA BX, NUM

EXTRN and PUBLIC :-

→ The Directive EXTRN informs the Assembler that the Names, labels and Procedure Declared after this directive have already been defined in some other assembly language modules

→ while in other modules where the names, Procedures and labels actually appear must be declared using PUBLIC Directive.

Example.

Module 1

```
Module 1
---
---
PUBLIC N1 DB 21H
PUBLIC N2 DB 28H
---
---
---
```

```
module 2 -
---
---
MOV AL, EXTRN N1
---
---
MOV CL, EXTRN N2
```

LOCAL :-

The labels, variables, constants, procedures declared using LOCAL Directive to be used only in that particular module

Ex

```
LOCAL N1 DB 28H.
LOCAL N2 DB ?
```

PAGE :
DATE : / /

GLOBAL :-
The labels, variables, constants, procedure declared using GLOBAL Directive can be used in any of the modules.

Ex:-
GLOBAL N1 DB 28H
GLOBAL N2 DB ?

~~EXTRN~~ :-
ORG :- This assembler directive directs the assembler to start a memory allotment for a particular segment from the declared address using ORG.
-> It gives the starting address of a module

Ex:-
ORG 1800H
↑
starting Address

7/3/17 SEGMENT OVERRIDE PREFIX :-

It allows the programmer to deviate from the default segment
MOV CX, [BX]
-> This instruction access data within the data segment by default.
-> Suppose that the data is available in the stack segment instead of data segment, the instn can be written as
MOV CS, SS:[BX]

This Ins^{n} addresses stack segment instead of data segment.

→ The prefix can be added to any instruct^{n} except jump inst^{n}.

Branch control :- [Program control transfer i

→ CALL :-

→ RET :-

JUMP Instructions

→ Unconditional jump.

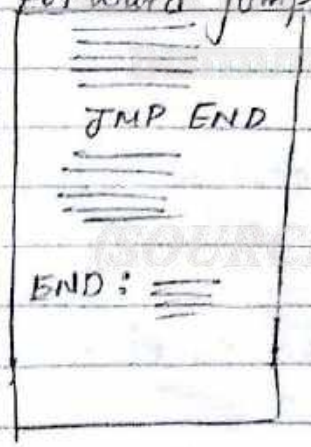
→ Conditional jump

UNCONDITIONAL JUMP :-

No condition is been tested before execution of the instruction.

Syntax:- JMP label.

Forward jump



backward jump

PAGE:
DATE: / /

JMP :

- short } Intrasegment jump.
- Near }
- Far. } Intersegment jump

Short jump :- This ins" allows jump to the memory locations within -128 to +127 from address following the jump instruction.

Example :- JMP Next.

JMP 08H ← displacement.

Instruction format : | opcode | Displacement |.

| EB | 08 |

1 byte 1 byte.

EB → the opcode of JMP.

Displacement is a signed number (8 bits).

$n = 8$.

-2^{n-1} to $+2^{n-1}-1$

-2^7 to 2^7-1.

-128 to +127

Near jump :-

Displacement is 2-bytes long. The displacement range b/w -32768 to +32767

Example :- JMP NEXT.

JMP 1234H.

Instruction format : | opcode | Displacement low | Displacement high |

ED	3H	12

1 byte 2 byte

$n \rightarrow 16$

-2^{n-1} to $+2^{n-1}-1$

-2^{15} to $2^{15}-1$

-32768 to 32767.

Far jump :- This is an inter segment jump.
→ The jump location is outside the segment. Therefore the segment and offset address of the jump location must be specified in the instruction itself.

Syntax :-

FAR JMP Label

segment address, offset addr

JMP 4000H : 1893H

Assembler directive

Jump location address

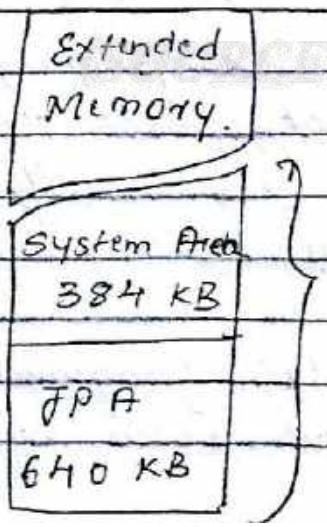
Conditional jump instructions

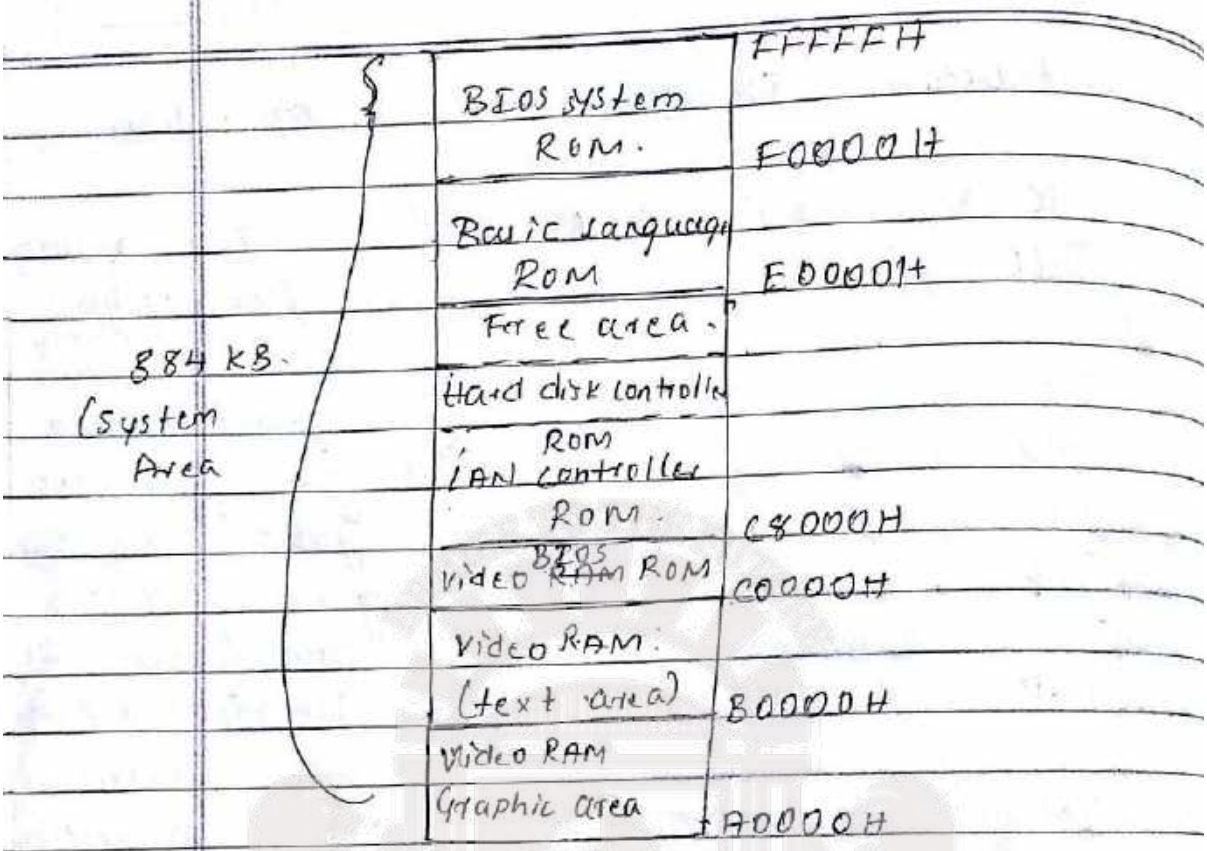
Some condition being tested before the execution of jump instruction. If cond is true the program control transfers to the label specified in the instruction. The condition of the status flags are checked.

PAGE:
DATE: / /

Instruction	Condition tested	Operation
JC	C=1	Jump if carry
JNC	C=0	Jump if no carry
JZ	Z=1	Jump if zero
JNZ	Z=0	Jump if no zero
JS	S=1	Jump if Negative
JNS	S=0	Jump if +ve.
JP	P=1	Jump if even parity
JNP	P=0	Jump if odd parity
JO	O=1	Jump if overflow
JNO	O=0	Jump if no overflow
JL/ JB	C=1	Jump if below
JG/ JA	C=0	Jump if above.
JE / JZ	Z=1	Jump if equal
JNE / JNZ	Z=0	Jump if not equal
JGE/ JAE	C=0, Z=1	Jump if Above or Equal.
~~JLE/ JBE~~		
JLE/JBE	C=1, Z=1	Jump if below or Eq

Memory Map of IBM PC.





The main memory is divided into 3-parts

1. Transient program Area [TPA]
2. System Area
3. Extended memory system

→ The first 1 MB memory is called the real mode memory because Intel-μPs are designed to function in this area using real mode operation.

→

: The TPA holds the disk operating system and other programs that control the computer system. It also stores any currently active or inactive DOS Applications.

The System area contains programs on ROM's or flash memory and areas of read/write

PAGE :
DATE : / /

[RAM] memory for data storage like graphics area, text area, Video BIOS ROM, LAN controller ROM, Hard disk controller, Bios system ROM etc.

9/3/17

MODULE – 2

ARITHMETIC INSTRUCTION

Unsigned Addition

ADD, ADC, INC.

(1) ADD :-

Syntax :- ADD Dest, source.

operation :- [Dest] + [source]

Dest ← [Dest] + [source]

→ All A.M are valid, All status flags [C, P, Z, S, A, O] are modified after execution.

→ Both destination and source both cannot be memory locations.

Example: ADD AL, CH // AL ← [AL] + [CH]

ADD AX, BX // AX ← [AX] + [BX]

ADD AX, [2800H] // Direct A.M

ADD AL, 89H // Immediate A.M

ADD AX, [SI] // Register Indirect

ADD AX, 93H[SI] // Register relative

ADD AX, [BP][SI] // Based Indexed

ADD AX, 98H[BP][SI] // Based indexed Rel

MOV [BX], [SI] // Invalid bcoz both source and destination are memory locations

PAGE :
DATE : / /

(2) ADC :- [Addition with Carry]

syntax:- ADC Dest, Source.

operation :- Dest ← (Dest) + (Source) + (C)

↑ carry flag.

Ex ADC AL, CH ; AL ← [AL] + [CH] + [C]

(3) INC :- [Increment]

syntax:- INC SOURCE.

operation :- Source ← [Source] + 1

All A.M are valid except Immediate A.M.

Ex:- INC AL // Register A.M

INC AX // Register A.M

INC 83H // Immediate AM.

INC BYTEPTR [BX] // 1 byte.

↓ Assembler directive to know the size.

INC WORDPTR [BX] // 2 byte.

→ BYTEPTR and WORDPTR are Assembler directives, Indicate the size of the Operand.

→ BYTEPTR Indicate that the memory location specified in the instruction is a byte location.

→ WORDPTR Indicate that the memory loc-n specified in the ins-n is a word [2-byte] location.

(1) Show the status of C, P, S, B, Z, flags after the execution of the following instructions.

P.T.O.

MOV AL, C9H
ADD AL, 9EH

	C	9
	9	E
1	16	7 H

1100 1001
1001 1110
1 0110 0111

C = 1, S = 0, A = 1
P = 0, Z = 0

2. MOV AL, D9H
ADD AL, 62H
ADD AL, 28H

	D9	
1	62	
	3B	
	3B	
	28	
	1	
	64 H	

D9H = 1101 1001
62H 0110 0010
3BH 1 0011 1011

3BH 0011 1011
28H 0010 1000
(C) 0110 0100 = 64H

C = 0
P = 0
S = 0
A = 1
Z = 0

CLC :- clear carry

4/3/17

1. Write a program to calculate the total sum of 5 bytes of data

```
.MODEL SMALL
.DATA
   LIST DB 02H, FFH, BCH, 93H, 91H
   COUNT DB 05
   SUM DB ?
.code
   MOV AX, @DATA ; Immediate addressing mode, in any of the segment reg is not possible
   MOV DS, AX
   MOV CX, COUNT ; Loop count in CX
   MOV SI, OFFSET LIST ; copy the offset addr of list to SI
   MOV AL, 00H ; clear AL
NEXT: ADD AL, [SI]
   INC SI ; Increment pointer
   DEC CX ; Decrement count
   JNZ NEXT ; if count ≠ 0, continue Addition
   MOV SUM, AL ; Store the result in sum.
   MOV AH, 4CH
   INT 21H
   END
```

2. Write a program to calculate the sum of 5 words

```
.MODEL SMALL
.DATA
   LIST DW 1111H, 2345H, 6732H, 6318H, 1231H
   COUNT DW 05
   SUM DW ?
```

```
.CODE
    MOV AX, @DATA
    MOV DS, AX
    MOV CX, COUNT
    MOV SI, OFFSET LIST
    MOV AX, 0000H
    CLC
NEXT: ADC AX, [SI]
    INC SI
    INC SI
    DEC CX
    JNZ NEXT
    MOV SUM, AX
    MOV AH, 4CH
    INT 21H
    END
```

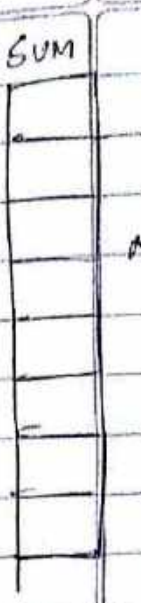
3. W.A.P That adds the following two multi-WORD Numbers and Save the result in a memory loc".

NUM1 = 1111 1111 1111 1111 H

NUM2 = FFFF FFFF FFFF FFFF H

```
.MODEL SMALL
.DATA
    NUM1 DQ 1214 3111 1111 1111 H
    NUM2 DQ FFFF FFFF FFFF FFFF H
    SUM DQ ?
.CODE
    MOV AX, @DATA
    MOV DS, AX
    CLC ; clear carry flag
    MOV CX, 04H loop count
```

copy the offset address of NUM1, NUM2, SUM to SI, DI and BX respectively




```
MOV SI, OFFSET NUM1
MOV DI, OFFSET NUM2
MOV BX, OFFSET SUM
NEXT: MOV AX,[SI]
ADC AX,[DI]
MOV [BX], AX
INC SI
INC SI
INC DI
INC DI
INC BX
INC BX
DEC CX
JNZ NEXT
MOV AH, 4CH
INT 21H
END
```

→ add the first word of NUM1 to first word of NUM2 and continue addition. store the result in SUM.

→ Increment the pointers

→ continue addition. if loop count is not zero.

20/3/19

Unsigned Subtraction :-

SUB

Syntax : SUB Dest, Source

Operation :- Dest ← (Dest) − (Source)

Ex: SUB AL, BH ; AL ← (AL) − (BH)
SUB AX, BX ; AX ← (AX) − (BX)
SUB AX [ASI]

→ All A.Ms are valid; All status flags are modified after the execution.

→ Internally the CPU performs the following operations

step1 :- Take the 2's complement of Substrand (source)

step2 :- Add it to the Minuend (destination)

step3 :- Invert the carry flag and auxillary carry flag.

Ex: MOV AL, 3FH
MOV BL, 23H
SUB BL, AL ; BL ← (BL) − (AL)

BL ← 23H − 3FH

23H = 0010 0011

3FH = 0011 1111

23H = 0010 0011
1100 0000
1110 0100

carry = 0
Invert carry = 1
Auxillary carry = 0
Invert Aux carry = 1

C = 1
A = 1
Z = 0
S = 1
P = 1

1010
0101
←0111

PAGE:
DATE: / /

Ex2:- MOV AL, ACH
SUB AL, 39H ; AL ← (AL) − (39H).
A ← (AC) − (39H)

AC → 1010 1100
39 → 0011 1001
AC → 1010 1100
2's(39) → 1100 0111
carry = 1 0111 0011

C = 0
A = 0
S = 0
P = 0
Z = 0

Invert carry = 0.
AUX carry = 1
Invert Aux carry = 0.

SBB :- [Subtract with Borrow]
Here content of carry flag is also subtracted.

Dest ← (Dest) − (source) − (C) ← carry flag.

Ex:-
SBB AL, BH ; AL ← [AL] − (BH) − (C)
SBB AX, BX ; AX ← (AX) − (BX) − (C)
SBB AX, [SI]

⇒ All A.M's are valid.
⇒ All status flags are modified after execution.

Dec :-
Syntax: DEC source.
operation :- source ← (source) − 1

All A.M's are valid except Immediate Mode

Ex:- DEC AL ; DEC 49H // Invalid
DEC AX ;

DEC 49H // Invalid Immediate A.M.
DEC Byte ptr[SI] // valid
DEC wordptr[SI] // valid, using Assembler directives.

CMP (compare)

Syntax :- CMP dest, source

Operation :- (Dest) - (source)

CMP internally performs subtraction operation but after the subtraction, Result is not stored in destination, and all status flags are modified after the execution.

→ All A.M's are valid

→ In general, conditional jump instructions are executed after CMP instruction.

Unsigned Multiplication :-

MUL

Syntax :- MUL source.

Operation :-

(i) 8 bit x 8 bit multiplication

16 bit ← 8 bit x 8 bit.

(product) (multiplicand) (multiplier)

AX ← (AL) x (source)

(ii) 16 bit x 16 bit multiplication.

32 bit ← 16 bit x 16 bit

(product) (multiplicand) (multiplier)

(DX AX) ← (AX) x (source)

PAGE :
DATE : / /

Source maybe a Register or a memory location, but it cannot be an immediate data.

MUL BL ; AX ← (AL) × (BL).
MUL BX (DXAX) ← (AX) × (BX)
MUL 38H // Invalid bcoz Immediate Addr

MUL BYTEPTR[SI] // 8bit x 8bit, operand = memory location

MUL WORDPTR[SI] // 16 bit x 16 bit

21/8/17

Ex:- MOV AL, 09H
MOV CL, 04H
MUL CL

AX ← (AL) × (CL).
AX ← 09H × 04H
AX ← (36)$_{10}$.
AX ← 24H

Unsigned Division :-

DIV. syntax DIV SOURCE.

operation :

1. 8-bit Quotient, 8-bit Remainder ← { 16-bit (Dividend) / 8-bit (Divisor).

(AL) Quotient, (AH) Remainder ← { (AX) / (Source)

16 bit Quotient, 16 bit Remainder ← 32-bit (dividend) / 16-bit (divisor)

AX (Quotient), DX (Remainder) ← DX AX / Source

All A.M's are valid except Immediate A.M.

Ex: ① DIV CL ; AL (Q), AH (R) ← (AX) / (CL)

② DIV CX ; AX (Q), DX (R) ← (DX, AX) / CX

③ MOV AX, 0009H
MOV CL, 02H
DIV CL.

AL (Q), AH (R) ← (AX) / (CL)

AL ← 04H, AH ← 01H ← 0009H / 02H

16 1000/

PAGE:
DATE: / /

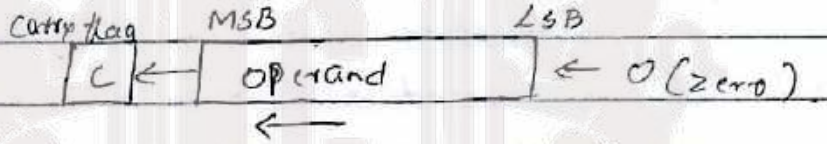
unsigned signed.

Shift Instructions SHL, SHR, SAL, SAR.

SHL :- Syntax: SHL operand, count

→ The operand may be a Register or a memory location.
→ count Specifies the no. of bit positions to be shifted
→ If the count is 'one', it can be Specified in the instruction itself
→ If the count is more than one, it must be Specified using 'CL' Register

Operation :-



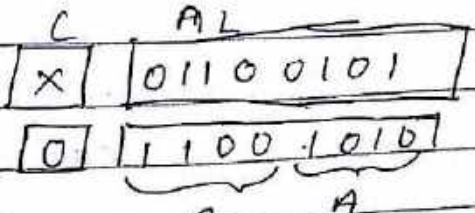
Ex1 SHL BL, 1

In the above example, the content of BL is shifted Left by one bit position.

Ex2 MOV CL, 4
SHL BL, CL
operand
count

The content of BL is shifted four bit positions Left

Ex3 MOV AL, 65H
SHL AL, 1



AL ← CAH Carry = 0

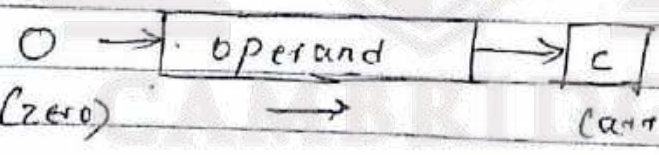
Ex 4 :- MOV CL, 3
MOV AL, 7AH [7]← 0 1 1 1 1 0 1 0
SHL AL, CL [1] ← 1 1 0 1 0 0 0 0
CL = 3 D0H
AL ← D0H Carry = 1

SHR :-

Syntax :- SHR operand, count.

→ The operand may be Register or Memory loc^n
→ Count specifies the no of bit position to be shifted
→ If the count is 'one' it can be specified in the instruction itself
→ If the count is more than one it must be specified in CL register

Operation :-



Ex1 :- SHR BL, 1

Ex2 :- MOV CL, 4
SHL BL, CL

Ex3 :- MOV AL, 65H 0 1 1 0 0 1 0 1 C 1
SHL AL, 1 AL ← 0 0 1 1 0 0 1 0
AL ← 32H 32H

PAGE:
DATE: / /

Exa: MOV CL, 3 . 0111 1010 . 0
MOV AL, 7AH . 01111010
SHL AL, CL . 0F
or AL ← 0FH

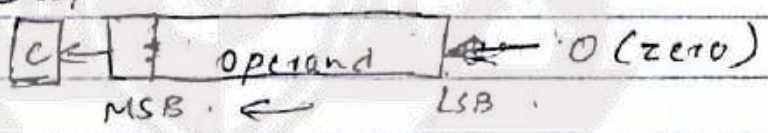
21/2/17

SAL [shift arithmetic Left] :-

Syntax SAL operand, count .

-> The operand may be Register or memory location.
-> count specifies the no of bit position to be shifted
-> If the count is 'one' it can be specified in the instruction itself.
-> If the count is more than one. It must be specified in CL register.

Operation



Ex: SAL BL, 1 .
& In the above example, the content of BL is shifted Left by one bit position

Ex2 MOV CL, 4
SHL BL, CL
operand count

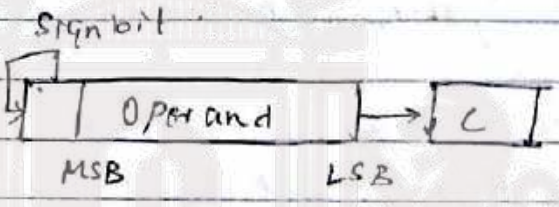
The content of BL is shifted four bit operand position Left.

SAR [shift arithmetic Right]

Syntax:- SAR operand, count

→ The operand may be register or memory location
→ Count specifies the no. of bit position to be shifted
→ If the count is one it can be specified in the instruction itself.
→ If the count is more than one it must specify in CL register.

Operation:-



Ex:1 MOV AL, 9AH
SAR AL, 1

1001 1010 CY
1100 1101
AL ← CDH carry flag = 0

Ex:2 MOV CL, 4
MOV AL, 85H
SAR AL, CL

1000 0101
1111 1000 carry = 0
F 8 H
AL ← F8H

PAGE:
DATE: / /

Ex 3 MOV CL, 3
MOV AL, 74H.
SAR AL, CL.

7 3
0111 0100 Carry = 1.

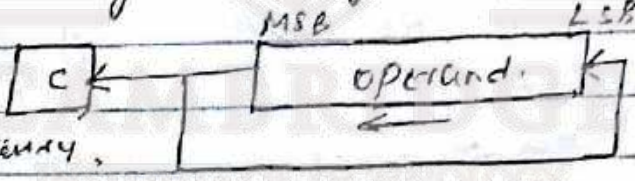
0 0 0 0 1 1 1 0
AL ← 0EH.

Rotate Instruction ROL, ROR, RCL, RCR.

ROL (Rotate Left).

Syntax: ROL operand, count.

→ The operand may be a Register or a memory location.
→ Count specifies no of bits positions to be Rotated left.
→ If the count is one, it can be seperated specified in the instruction itself.
→ If the count is more than one, it must be specified using CL Register.



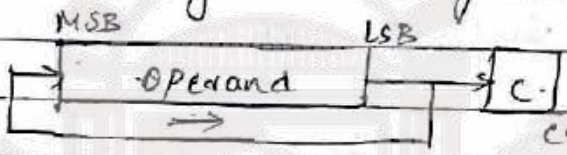
Ex:1 MOV AL, A3H 1010 0011
ROL AL, 1 0100 0111
47H
AL ← 47H. C=1

Ex:2 MOV CL 3 10011000
MOV AL, 9BH C 1101 1100
ROL, AL, CL DCH.

ROR [Rotate Right]

Syntax :- ROR operand count.

→ The operand may be register or memory location.
→ count specifies the no of bit positions to be rotated Right.
→ If the count is one, it can be specified in the instruction itself.
→ If the count is more than one, it must be specified using a CL register.



ex1 :- MOV AL, A3H — 1010 0011
ROL AL, 1 — 1101 0001
D1H
AL ← D1H — c = 1

ex2 :- MOV CL, 3 — 1001 1011
MOV AL, 9BH — 0111 0011
ROL AL, CL. — 73H
AL ← 73H

22/8/17

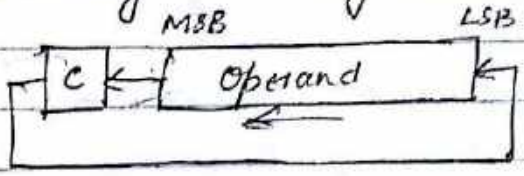
RCL :- (Rotate left with carry)

Syntax :- RCL operand count

→ The operand may be registed or memory location
→ count specifies the no of bit position to be rotated left with carry.
→ If the count is one, it can be specified in the instruction itself.

PAGE:
DATE: / /

→ If the count is more than one, it must be specified using CL register.

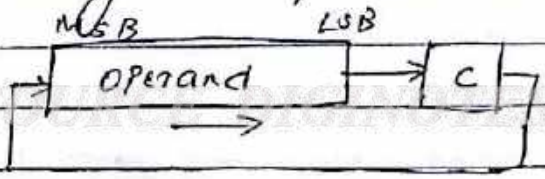


Ex1 MOV AL, C3H.
RCL AL, 1

1100 0011
1000 0111
AL ← 87H . C = 1.

RCR : [Rotate Right with carry]
Syntax: RCR operand count.

→ The operand may be register or memory location
→ The count specifies the no of bit position to be rotated Right with carry.
→ If the count is one, it can be specified in the instruction itself.
→ If the count is more than one, it must be specified using CL register



Ex: MOV AL, C3H.
RCL AL, 1.

1100 0011 [X] C

1100 0110

AX ← E8FH

1. Write an Assembly level program to count the no of ones in a given 8-bit data

Input : 8-bit data (46H).
~~Output~~ : 0100 0110
output : 3.

```
.Model Small.
.data.
    A dB 46H
    Out dB ?
.code
    MOV Ax,@ Data
    MOV DS, Ax
    Mov AL A  MOV CL 08
    MOV CX 0;
B   ROL AL, 1
```

```
.model small .
. data
    N DB ?
.code
  MOV CL, 08H // loop count .
  MOV BL, 0  // counter .
  MOV AL, NUM //
NEXT: ROL AL, 1  // Rotate left and check if
      JNC SKIP     carry flag is set .
      INC BL  // if carry flag is set, increment
SKIP: DEC CL .      the count .
      JNZ NEXT // Check if 8 rotations
      MOV N, BL   completed. if not repeat .
```

2. write an ALP to count the no of 1's in a given 16 bit data .

```
.Model small .
. data :
    Num dw ABCDH .
    N  dw b ?
.code .
  A DB 46H .
  MOV AX, @ Data .
  MOV DS, AX .
  MOV CX 16
  MOV BL, 0 .
  MOV AX, NUM .
NEXT: ROL AX .
      JNC SKIP .
      INC BL .
SKIP: Loop NEXT .
      MOV N, BL
```

0 0 0 1 0 0 0 0

5. Write an ALP to count the no of posith and negative numbers in an array of 8 bit no's

Rotate the given no left by 1-bit position and check the carry flag. If carry flag is set [ie MSB of the given no is 1 and id is negative]. The given no is negative, otherwise it is positive.

```
.Model small
.Data
    List db 30H, 20H, 50, 40H, 3AH.
    Len DB ($-LIST).
    POS DB ?  // positive count
    NEG DB ?  // negative count.

.CODE
    MOV AX, @DATA
    MOV DS, AX;
    MOV CX, LEN // Loop count.
    LEA .SI, .List.
    MOV BL, 0.  //Neg
    MOV BH, 0 // pos
NEXT  MOV AL, [SI]
      ROL AC, 1 → JMP EXIT
      JC SKIP
      INC BH  // Inc pos count
SKIP: INC BL  // Inc Neg count
EXIT: INC SI
      Loop NEST
```

If LSB is 1, the no is odd
else it is even (LSB)

2 10 S

PAGE:
DATE: / /

```
MOV POS, BH .
MOV Neg, BL
END .
```

4. Write an ALP to count the no of Even and Odd no's in a given array of ~~no~~ 8 bit no's

```
.MODEL SMALL
.
```

30/5/14

Logical Instructions

→ OR
→ AND
→ XOR
→ NOT
→ TEST

1. **OR :-**

Syntax : OR Dest, source

Operation : Dest ← (Dest) Bitwise OR (source)

→ All the A.M's are valid

→ It performs Bitwise OR operation b/w Destination & source operands and result is stored in destination.

→ Both source and destination cannot be memory locations. All status flags are modified except 'C' and 'A'

Ex:

OR AL, BL ; AL ← (AL) Bitwise OR (BL)
OR AX, CX
OR AX, [SI]
OR AL, 3AH.

2. **AND**

Syntax :- AND Dest, source

operation :- Dest ← (Dest) & Bitwise AND (source)

→ All A.M's are valid.

→ It performs Bitwise AND operation b/w Destination and source operands and result is

stored in destination.

→ Both source and Destination cannot be memory locations. All status flags are modified except 'c' and 'A'

Ex AND AL, BL ; AL ← (AL) Bitwise AND (BL).

5. XOR.

Truth table:

A	B	Y.
0	0	0
0	1	1
1	0	1
1	1	0.

Syntax:- XOR Dest, Source.

Operation:- Dest ← (Dest) Bitwise XOR (source)

→ All A.M's are valid.

→ It performs Bitwise XOR operation blw destination and source operands and result is stored in destination.

→ Both source and Destination cannot be memory locations, All status flags are modified except 'c' and 'A'

Ex:

XOR AL, BL ; AL ← (AL) Bitwise XOR (BL).

XOR AL, 8AH

4. TEST :-

Syntax :- TEST Dest, Source.

Operation :- (Dest) $\overset{\text{Bitwise AND}}{\wedge}$ (Source)

TEST AL, BL ; (AL) $\overset{\text{Bitwise AND}}{\wedge}$ (BL)

TEST AX, CX

TEST AL, 8AH

→ All addressing modes are valid.

→ Test Internally performs Bitwise AND Operation But the result is not stored in the destination. After the execution all the status flags are modified except 'C' and 'A'

5. NOT

Syntax :- NOT Operand.

Operation :- operand ← $\overline{(\text{operand})}$ ← Bitwise complement.

→ All AM's are valid except Immediate A.M.

→ Operand can be a Register or memory location but cannot be an Immediate Data

Ex:- NOT AL

NOT AX

NOT 38H // Invalid

NOT BYTEPTR [SI]

NOT WORDPTR [SI]

1. Show the status of 'P', Z, S flags after the execution of following set of instructions in following sequence:

1. MOV BL, 8AH
OR BL, 9CH ; BL ← (BL) OR 9CH.
BL ← 8AH OR 9CH.
BL ← 9EH.

8AH : 1000 1010.
9CH 1001 1100
1001 1010
9 E H.

P = 0, Z = 0, S = 1

2. MOV BL, 8AH.
AND BL, 9CH.
BL ← 88H.

8AH : 1000 1010
9CH : 1001 1100
1000 1000
88H.

P = 1
Z = 0
S = 1

3. MOV BL, 8AH
XOR BL, 9CH
BL ← 16H
P = 0
Z = 0.
S = 0.

8AH : 1000 1010
9CH 1001 1100
0001 0110
16H.

4. MOV BL, 8AH.
XOR BL, BL
Z = 1
S = 0
P = 1

1000 1010
1000 1010
0000 0000
00H.

XOR Instruction can be used to clear the content of a register.

5. MOV AL, 8AN
NOT AL ; AL ← $(\bar{AL})$

$\bar{AL}$ = 10001010
$\bar{AL}$ = 01110101
AL ← 75H

2. Write ~~Write~~ An ALP to count no of even and odd no's in a given array of 8-bits

1) MOV AL, 04H → EVEN.
TEST AL, 01H

AL = 04H = 0000 0100
AND 0000 0001
0000 0000 Z = 1.

If Z = 1, the given no is EVEN, if Z = 0 the given no is odd.

```
.MODEL SMALL
.DATA
    LIST DB 03H, 05H, 07H, 02H
    LEN DB ($-LIST)
    ODDCOUNT DB ?
    EVENCOUNT DB ?
.CODE
    MOV AX, @DATA.
    MOV DS, AX
    MOV CX, LEN  //loop count
    MOV BH, 0  //odd count
```

```
        MOV BL,0    // Event cont.
        LEA SI, LIST // copy offset addr of LIST to SI
NEXT:   MOV AL, [SI]
        TEST AL, 01H
        JZ EVEN     // If Z=1, no is even
        INC BH      // odd count (Increment)
        JMP EXIT.
EVEN:   INC BL      // Increment even count.
EXIT:   INC SI.
        LOOP NEXT.
        MOV EVENCOUNT BL.
        MOV ODDCOUNT BH.
        MOV AH, 4CH.
        INT 21H.
        END.
```

31/3/17

BCD

BCB Arithmetic Instruction.	
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Decimal — Binary coded decimal.

66

88
+10
68

58
176

case i) Lower nibble > 9.

Ex:- 25 H
48 H
6D H
06 H
73 H

25
48
73

ie D > 9

case ii)

48
49
91 H
06 H
97 H

48
49
97

Auxillary carry flag
A=1

case iii)

84 H
42 H
C6 H
60 H
126 H

84
42
126

higher nibble > 9
ie C > 9

C=1
AL=26

case iv) Higher nibble < 9 but carry flag = 1

94 H
92 H
26 H
60 H
186 H

94
92
186

C=1 AL = 86

DAA [Decimal Adjust After addition]

· Syntax :- DAA

This instruction is used to convert the result of the addition of two packed

[C] → 1 88
88
10
66
176

25 → packed
02 05 → unpacked.

PAGE :
DATE : / /

BCD numbers to a valid BCD number

→ After the addition the result must be present in 'AL' Register in order to adjust the result

→ DAA Instⁿ works as follows.

case i

After the Addition if lower nibble is greater than 9 or if auxillary carry flag A=1, 06 H is added to AL

case ii After the addition if the higher nibble is greater than 9 or if carry flag sets ie if C=1, 60 H is added to AL

ex:- MOV AL, 25H ; AL ← 25H.
ADD AL, 48H ; AL ← (AL) + 48H
DAA AL ← 6DH.
AL ← 73H.

eg 2:- MOV AL 84H ; AL ← 84H.
ADD 42AL 42H. AL ← (AL) + 84H.
DAA. AL ← 126H

ex:- MOV AL 48H ; AL ← 48H
ADD AL 49H ; AL ← (AL) + 49H
DAA. AL ← 97H

Ex:- MOV AL 92H. AL ← 92H.
ADD AL 94H. AL ← (AL) + 94H.
DAA AL ← 186H.

77

DAS [Decimal adjust After Subtraction]

Syntax :- DAS.

→ This is similar to DAA.

→ This instruction is used to convert the result of the Subtraction of two packed BCD no. to valid BCD no.'s

→ After the subtraction the result must be present in 'AL' Register in order to adjust the result

→ DAS instruction works as follows

Case i :-

After the subtraction if lower nibble is greater than 9 or if Auxiliary carry flag A=1, 06 is subtracted from AL.

Case ii) :- After the subtraction if higher nibble is greater than 9 or if carry flag (=1, then 60H is subtracted from AL.

Ex
$$\begin{array}{r} 58\,H \\ -19\,H \\ \hline 3F\,H. \\ -06\,H \\ \hline 39\,H. \end{array}$$

$$\begin{array}{r} 58 \\ -19 \\ \hline 39 \end{array}$$

$$\begin{array}{r} 16 \\ 9 \\ 04 \quad 10 \\ \hline 08 \end{array}$$

MOV AL, 58H — AL ← 58H.

SUB AL, 19H. — AL ← (AL) − 19H.

AL ← 58 − 19H

AL ← 3FH.

DAS — AL ← 39H.

Ex 2

MOV AL 41H
SUB AL, 19H.

41 H
- 19 H
28 H
- 06 H
22 H

∴ A = 1 · DAS.

Ex 3

C = 1

31 H 19 H.
41 H
0 D8 H.

C = 1

31/3/17 Ascii Arithmetic Instructions

Digit	ASCII code
0	30H
1	31 H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H

AAM (Ascii adjust after multiplication)
syntax :- AAM

→ This instruction, After execution converts the product available in AX into unpacked BCD form.
→ This Instruction is executed after MUL Instruction.

Ex:1 MOV AX,0048 ; AX ← 0048
AAM ; AX ← 0408

Ex:2 09 x 04
0036
↓
0306

MOV CL 04 ; CL ← 04
MOV AL 09 ; AL ← 09
MUL CL ; AX ← (AL) x (CL)
AX ← 09 x 04
AX ← 0036
AAM ; AX ← 0306

V.2 AAD [ASCII Adjust before division]

→ This instruction, after execution converts two unpacked BCD Digits in AH and AL to the equivalent Hexadecimal number in AX.

→ AAD Ins^n must be executed before the execution of div instruction

Ex:

58 (16) = 3AH.

0508 → 0058 → 003A H
4 4 04

MOV CL, 04
MOV AL, 08 } AX ← 0508
MOV AH, 05
AAD ; AX ← 0058
AX ← 003AH.
DIV CL ; (AX) / CL = 003AH / 04

Machine control Instructions :-

1. HLT :- This Instruction stops the execution of Instructions and halt the processor.

2. WAIT :- This Instruction to monitors the Test Input pin. If Test bar = 0, when wait Instruction is executed the µP waits until Test bar goes to logic 1.

3. LOCK :- when an Instruction with LOCK prefix executes the logic level of lock bar output pin goes to logic zero

4. NOP [No Operation] When microprocessor encounters NOP Instruction, it does nothing but consumes 3 clock cycle to execute.

4/4/17

Interrupts of 8086/88

An Interrupt is an external event that informs the microprocessor that a device needs its service.

→ Intel processors include two hardware pins INTR and NMI, that receives external Interrupts.

→ The processor also has software Interrupts using instructions.

→ 8086/88 has total of 256 interrupt from INT 00
INT 01
⋮
INT 255 (FFH)

Interrupt Service Routine [ISR]

→ With every Interrupt there must be a program associated with it.

→ When Interrupt is invoked, it is asked to run a program to perform certain task. This subroutine is called ISR.

→ The address of ISR is fetched from Interrupt vector table IVT.

→ For each Interrupt type, 4 memory locations are reserved in IVT

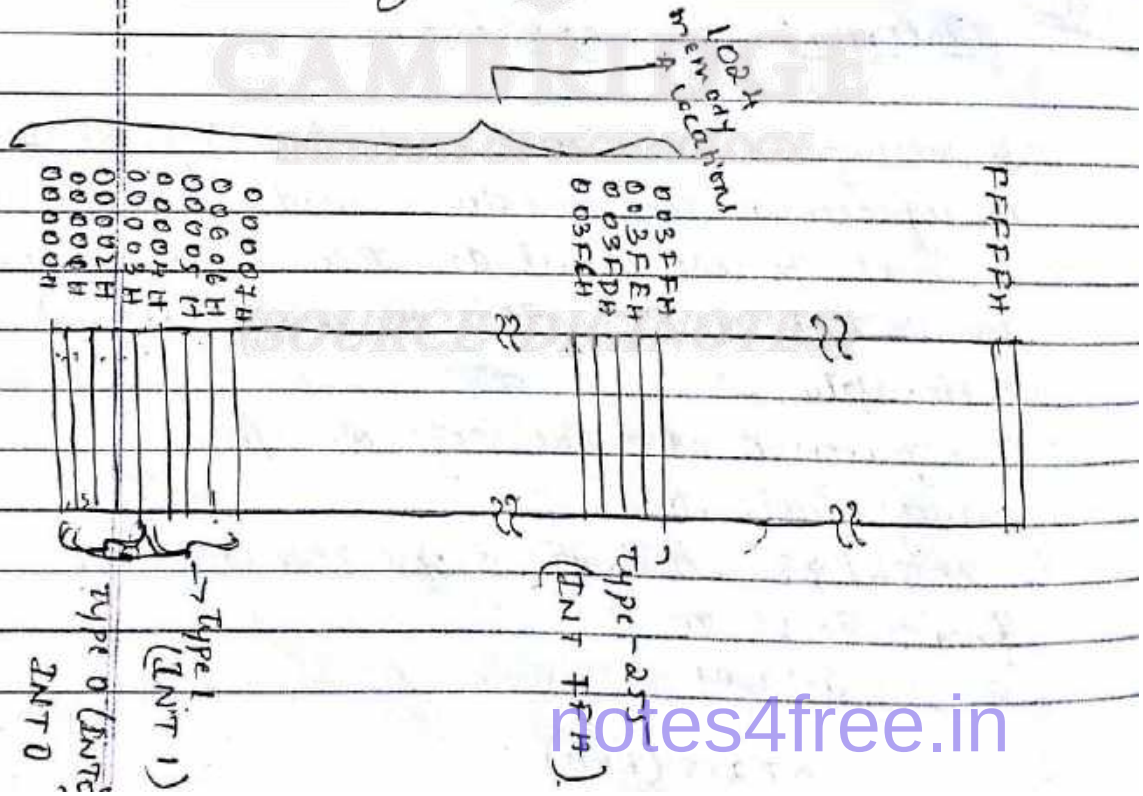
→ 2-bytes for offset addr and 2-bytes for Segment addr of Interrupt SR.

→ IVT is located in the first 1024 memory locations.

→ IVT has 256 Interrupt vectors

→ The Intel reserved the first 32 interrupt vectors.

→ The last 224 interrupt vectors are available for users.



Segment high
segment low
offset high
offset low

} Type n. (Each vector)

5. [4|17]

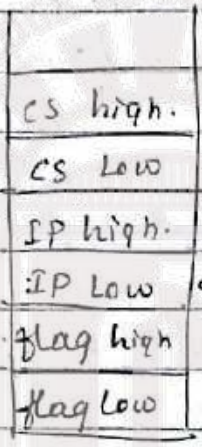
Steps taken by the processor when an Interrupt occurs [processing steps of interrupt]

→ When an interrupt occurs, The processor performs the following steps :-

1. flag register content is pushed onto the stack [saving flag register] and SP is decrement ~~and~~ -ed by 2.
2. Interrupt flag 'I' and Trap flag 'T' are disabled
3. The current content of IP is pushed on to the stack, and SP is decremented by 2.
4. Current content of CS is pushed onto the stack and SP is decremented by 2. [step 3 and 4 are saving the return addr.]
5. The INT no is multiplied by 4 to get the physical address of IVT where offset address and segment addr of ISR is available.
6. offset addr and Segment addr from IVT are copied into IP and CS registers respectively ~~from~~

7. From new CS and IP, the processor starts fetching the instructions and executes them. These instructions belongs to ISR.

8. The last instruction of ISR must be IRET to get IP, CS and Flag register back from the stack and make the CPU Run the code from where it left off



Hardware Interrupts

→ Two i/p interrupt pins of Intel UP used for hardware interrupts are INTR and NMI.

→ INTR is an i/p interrupt to microprocessor which can be masked or unmasked using 'I' flag

→ NMI is an i/p to CPU which cannot be masked or disabled

→ These two interrupts are external interrupts activated by logic 1 when an interrupt is received through NMI type 2 or INT 02 Interrupt occurs

Software Interrupts :-

→ If an ISR is called upon as a result of the execution of an instruction, such as

INT n where n is b/w 0 to 255
$0 \leq n \leq 255$, it is referred as software interrupts.

ex:- (1) INT 21H (DOS call)
(2) INT 10H (BIOS call)

8086 dedicated interrupts [Predefined interrupts]

1. INT 0 (Type 0) → Divide by Zero error
Type 0 interrupt occurs whenever an attempt is made to devide a no. by zero anywhere in the programs.

2. INT 1 (Type 1) → Single step Execution.
If T=1, The processor enters into step by step execution mode, ie Type 1 interrupt occurs when T=1

3. INT 2 :- (Type 2) → Non-maskable interrupt.
When a logic 1 is placed on NMI pin, Type-2 interrupt occurs.

4. INT 3 :- (Type 3)
This is a special instruction, when it is executed, Type 3 interrupt occurs.

5. INT 4 :- (Type 4) → Overflow.
This is a special vector used with INTO Instruction. The INTO Instruction interrupts the processor, when if there is

An overflow in signed Arithmetic ie Type-4 interrupt occurs if overflow flag is set

DOS Interrupt INT 21H.

INT 21H is provided by DOS when OS is loaded in CPU. There are many functions associated with INT 21H, depending on the no. present in AH register

i) INT 21H option 09H

Function :- To display a string on the monitor

AH ← 09H

DX ← offset address of string

Ex:

```
MSG1 DB "Abdul $"
     ===
     LEA DX, MSG1
     MOV AH, 09H
     INT 21H
```

6/4/17

ii) INT 21H option 02H

To display a single character on the monitor

AH ← 02H

DL ← ASCII code of the character to be displayed.

Ex: MOV ~~AH~~ DL, '$' } character $ is displayed
MOV AH, 02H
INT 21H

(2) · MOV DL, 34H } → ASCII CODE of 4.
MOV AH, 02H
INT 21H } 4 is displayed.

3. INT 21H ~~option~~, Function 01H
Function: To Read a character with echo

AH ← 01H (Function no)
after execution · AL ← ASCII code of Read char.

Ex:- MOV AH, 01H } wait until the key is pressed.
INT 21H } When the key is pressed, the ASCII code of the character is copied into AL register.

4. INT 21H Function 0AH
Function: To Read a string and store it in a buffer.

AH ← 0AH (Function number).
DX ← offset address of the buffer

Ex:- Buffer DB 40 DUP(?)
LEA DX, Buffer
MOV AH, 0AH
INT 21H

Buffer

40	09	C	A	M	B	R	I	D	G	E

↓ Size of string ↘ Actual size.

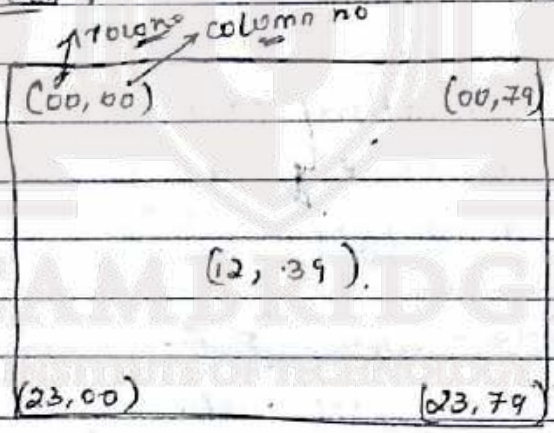
BIOS Interrupt using INT 10H

→ INT 10H subroutines are burnt into ROM BIOS.
→ INT 10H routine are used to communicate with the computers screen video.
→ There are many functions associated with INT 10H
→ some of the functions are.

1. Changing the color of the text.
2. changing background colour of text.
3. clearing the screen
4. changing the location of the cursor
5. to get the location of the cursor

Monitor screen in text mode is divided into 80 columns and 24 rows

Text screen :-



1. clearing the Screen using INT 10H, Function 06H

To use INT 10H to clear the screen, following registers must be loaded with certain values

AH ← 06 (Function number)
AL ← 00 (Entire page) // page number
BH ← 07 (Normal Attribute) // special attribute

(00) CH ← Row value of the ~~sole~~ Starting point
(00) CL ← column " "
(24) DH ← Row value of the ending point.
(79) DL ← column " "

Write an ALP to clear the entire screen.

```
//.MODEL SMALL
//. CODE
    MOV AX, @DATA.
    MOV DS, AX.
    MOV AH, 06
    MOV AL, 00
    MOV BH, 07
    MOV CH, 00.
    MOV CL, 00.
    MOV DH, 23
    MOV DL, 79
    INT 10H.
```

Setting the cursor to a specified location using INT 10H, Function 02H

AH ← 02 (Function no).
BH ← 00 (Page no, page 00)
DH ← Row number
DL ← column number } of the cursor location.

WAP to move the cursor to ~~move~~ the position (15, 28)

```
MOV AX, @DATA
MOV DS, AX
MOV AH, 02H
MOV BH, 00
MOV DH, 15
MOV DL, 28
INT 10H
```

WAP to clear entire screen and move the cursor to location (21, 56)

```
MOV AX, @Data
MOV DS, AX!
MOV AH, 06
MOV AL, 00
MOV BH, 07
MOV CH, 00
MOV CL, 00
MOV DH, 23
MOV DL, 79
INT 10H
MOV AH, 02
MOV BH, 00
MOV DH, 21
MOV DL, 56
INT 10H
```

7/11/17

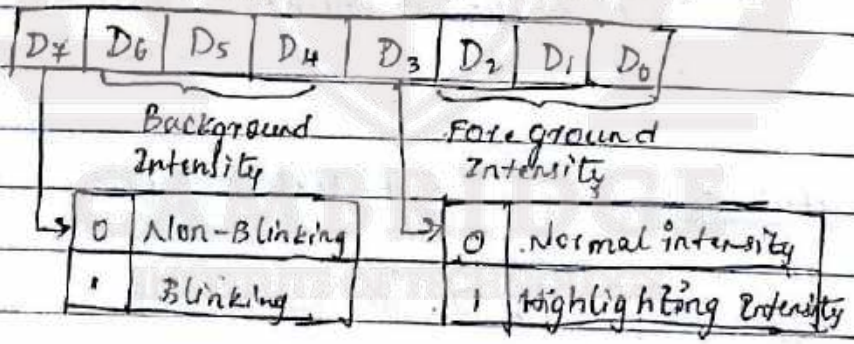
→ Changin the video mode using INT 10H, function 00H :

AH ← 00H (fun. no)
AL ← 07H (monochrome text mode)
03H (CGA Text mode)

W.A.P using INT10H to change the video mode to CGA text mode :-

```
MOV AH, 00 (Fun no)
MOV AL, 03 (CGA text mode)
INT 10H
```

Attribute byte in Monochrome monitors :-
There is an attribute associated with each characters on the Screen.
Attribute provides information to the video circuitary such as colour, intensity of the characters, Background colour etc.



Binary	Hex	Result
$D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0$		
0 0 0 0 0 0 0 0	00H	White on black.
0 0 0 0 0 1 1 1	07H	White on black Normal In
0 0 0 0 1 1 1 1	0FH	White on black High Intens
1 1 1 1 0 0 0 0	F0H	Black on white blinking

Registers used

AH ← 09H (Fun^n no)

BH ← Page no.

AL ← ASCII code of character

CX ← Count for the repetation.

BL ← Attribute byte

WAP using INT 10H to display the letters 'D' in 200 locations with the attribute black on white blinking. D are black and Screen background is white

```
MOV AH,00H ; set monochrome text mode.
MOV AL,07  ;
INT 10H
MOV AH,09H (fn^n no)
MOV BH,00H (page)
MOV AL, 44H (ASCII code of 'D')
MOV CX,200 (Repeat in 200 Locations)
MOV BL,0F0H (Attribute byte for black on
INT 10H                white blinking)
```

Attribute byte in CGA text mode :-

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
B	R	G	B	I	R	G	B
Background					Foreground		

B → Blinking: 0 – non-blinking, 1 – blinking

I → Intensity: 0 – high intensity, 1 – low intensity

I	R	G	B	colour
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	white
1	0	0	0	Grey
1	0	0	1	Lightblue
1	0	1	0	light green
1	0	1	1	light cyan
1	1	0	0	light red.
1	1	0	1	light magenta
1	1	1	0	light brown
1	1	1	1	light white (high by intensity white)

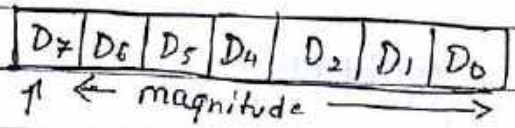
WAP that puts white space (ASCII - 20H) on entire screen. use high intensity white on a blue background attribute.

```
MOV AH, 00H
MOV AL, 03H (CGA text mode)
INT 10H
MOV AH, 09H (fn^n no)
MOV BH, 00H (page no)
MOV AL, 20H (ASCII of white space).
MOV CX, 800H (entire screen 24x80=()10 = 800H)
MOV BL, 1FH (Attribute byte) 1FH = 0001, 1111
                                   Blue  high intensity white
INT 10H
End
```

MODULE - 3

SIGNED ARITHMETIC OPERATIONS :-

Signed byte operand (8-bit)



$n = 8$

Range of numbers $= -2^{n-1}$ to $+2^{n-1} - 1$

$= -2^7$ to $+2^7 - 1$

$= -128$ to $+127$

To represent negative numbers.

1. Write the magnitude of the number using 8-bits without sign bit.
2. Take the ones complement by inverting each bit.
3. Take the 2'S complement by adding one to the 1'S complement

To represent 14

14 = 0000 1110

= 1111 0001 (1'S complement)

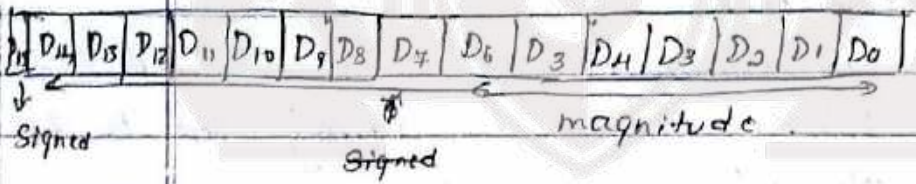
1

-14. 1111 0010 2'S complement = F2H

sign bit — magnitude.

Range of no's	Binary	Hex
0	0000 0000	00H
+1	0000 0001	01H
+2	0000 0010	02H
⋮	⋮	⋮
+127	0111 1111	7FH
-128	1000 0000	80H
-127	1000 0001	81H
-126	1000 0010	82H
⋮	⋮	⋮
-1	1111 1111	FFH

Signed word size operand (16-bit)



$n = 16$ -32768 to $+32767$

Range of numbers $= -2^{n-1}$ to $+2^{n-1}-1$

$= -2^{16}$ to $+2^{16}-1$

Range	Binary	Hex
0	0000 0000 0000 0000	0000H
+1	0000 0000 0000 0001	0001H
+2	0000 0000 0000 0010	0010H
⋮	⋮	⋮
+32767	0111 1111 1111 1111	7FFFH
-32768	1000 0000 0000 0000	8000H
⋮	⋮	⋮
-1	1111 1111 1111 1111	FFFFH

Overflow condition :- In 8-bit operations The overflow flag will set if any one of the following ~~errors~~ condition occurs.

1. There is a carry from bit D_6 to D_7 but no carry out from D_7.
2. There is a carry out from D_7 but there is no carry from bit D_6 to D_7

~~Check the Status~~.

ex 1. MOV AL, -128
MOV CL, -2
ADD AL, CL

Check the status of overflow, carry and sign flag after the execution.

n = 8

-128 = 1 000 0000
-2 = 1111 1110.
-126 = 0111 1110

incorrect result.

OF = 1 → There is ~~no~~ a carry out from D_7
CF = 1 but no carry from D_6 to D_7
SF = 0

Ex 2 MOV AL, -5
MOV CL, -2.
ADD AL, CL

OF = 0
CF = 1
SF = 1

-5 = 1111 1011
-2 = 1111 1110
-7 = 1 1111 1001

0010
1101
1
0111
0000

iii) MOV AL, +11
MOV CL, +19
ADD AL, CL.

OF = 0
CF = 0
SF = 0

```
+11   0000 1011
+19   0001 0011
      0001 1100   = 1EH.
```

iv) MOV AX, +21230
MOV CX, +3523
ADD AX, CX.

```
+21230 = 0101 0010 1110 1110
+3523  = 0000 1101 1100 0011
         0110 0000 1011 0001
```

OF = 0
CF = 0
SF = 0.

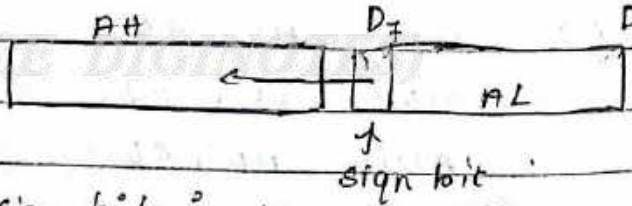
12/4/17

Instructions for conversion from 8-bit to 16 and 16-bit to 32-bit

i) CBW (convert byte to word)

syntax :- CBW

operation :-



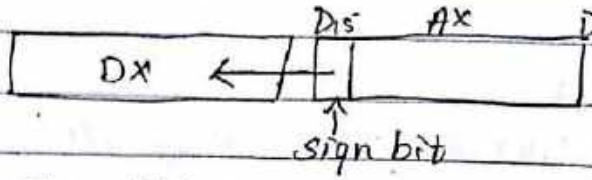
Copy sign bit in AL to AH

Ex:- MOV AL, 9AH ; AL = 9AH = 1001 1010
CBW ; AH = 1111 1111 AL = 1001 1010
↑ sign bit.

i) CWD (convert word to double word)

Syntax:- CWD

operation.



Copy Sign bit of AX to DX

ex:- MOV Ax, 4CA1H;

CWD;

Ax = 4CA1H = 0100 1100 1010 0001

↑ sign bit

Dx = 0000 0000 0000 0000

Ax = 0100 1100 1010 0001

STRING instructions.

→ Each string instruction allows data transfer that are either a single character [Byte] or double characters [word].

→ The index registers SI and DI are used along with string instructions to point to source string and destination string respectively. SI and DI

→ After the execution of a string byte instructions, are automatically incremented by 1, if the Direction flag D=0.

→ SI and DI are automatically Decremented by 1 if D=1.

→ After the execution of string word instructions, SI & DI are automatically incremented by 2 if D=0

→ SI and DI are automatically decremented by 2 if D = 1

MOVS
- MOVSB (move string Byte)
- MOVSW (move string word)

MOVSB

syntax :- MOVSB

operation :- (ES):(DI) ← Byte (DS):(SI)

SI ← SI+1 } if D=0
DI ← DI+1 } auto increment mode.

SI ← SI −1 } autoDecrement mode.
DI ← DI − 1 }

```
MOVSB = { CLD // clear D  ie D=0 autoinc
          MOV AL, [SI]
          MOV [DI], AL
          Inc SI
          Inc DI
```

```
MOVSB = { STD // Set D=1  autoDecrement
          MOV AL, [SI]
          MOV [DI], AL
          Dec SI
          Dec DI
```

MOVSW

Syntax : MOV SW

operation : (ES):(DI) ← word (DS):(SI)

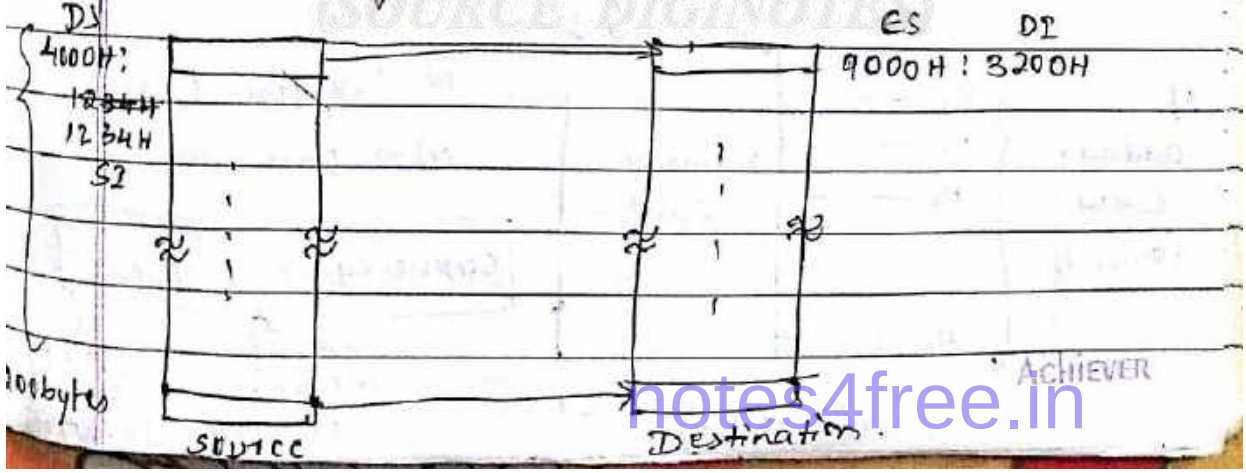
SI ← SI + 2 } if D = 0.
DI ← DI + 2 } auto increment.

SI ← SI - 2 } if D = 1
DI ← DI - 2 } autodecrement

MOVSW = { CLD // clear D auto increment.
MOV AX, [SI]
MOV [DI], AX.
Add SI, 2
Add DI, 2.

MOVSB = { STD // set D = 1 autodecrement.
MOV AX, [SI]
MOV [DI], AX
Sub SI, 2
Sub, DI, 2.

1. Write a sequence of instructions to move 200 bytes of Data from a block of memory loc^n starting with 4000H : 1234H to a block of memory loc^n starting with 9000H : 3200H.



```
MOV DS, 4000H
MOV SI, 1234H
MOV ES, 9000H
MOV DI, 3200H
MOV CX, 200  // Loop count
NEXT: MOVSB
      LOOP NEXT.
```

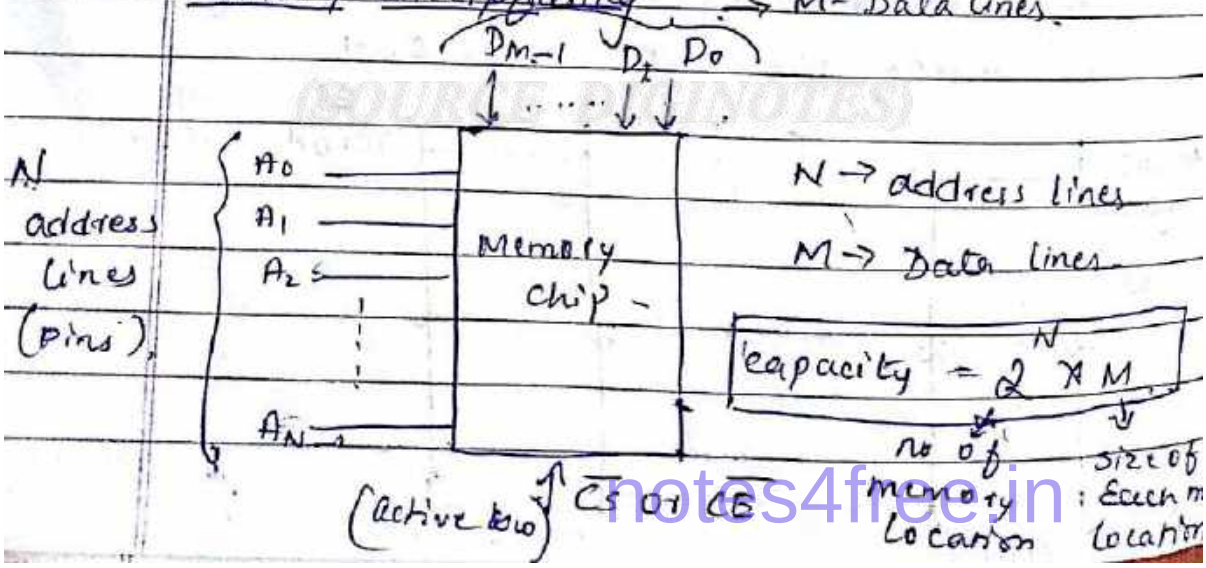
(or)

```
CLD // clear D  autoincrement.
MOV DS, H000H
MOV SI, 1234H
MOV ES, 9000H
MOV DI, 3200H
MOV CX, 200
NEXT: MOV AL, [SI]
      MOV [DI], AL
      INC SI
      INC DI
      LOOP NEXT.
```

18/12/17

Interfacing

1. Memory Interfacing



CMPS

- CMPSB [compare string Byte]
- CMPSW [compare string Word].

CMPSB

Syntax:- CMPSB

-> This instruction compares byte in data segment pointed by SI with a byte in extra segment pointed by DI

-> SI and DI automatically incremented by 1 if D=0, else SI and DI are automatically decremented by 1 if D=1

-> [DS]:[SI] $\xleftrightarrow[\text{compare}]{\text{Byte}}$ [ES]:[DI].

-> SI ← SI+1, DI ← DI+1 if D=0.

SI ← SI-1, DI ← DI-1 if D=1

CMPSWI

Syntax:- CMPSWI.

-> This instruction compares ~~byte~~ word in data segment pointed by SI with a ~~byte~~ word in extra segment pointed by DI

-> SI and DI automatically incremented by 2. if D=0 else SI and DI are decremented by 2.

[DS]:[SI] $\xleftrightarrow[\text{compare}]{\text{Byte}}$ [ES]:[DI]

SI ← SI+2, DI ← DI+1 if D=0.

SI ← SI-1, DI ← DI-1 if D=1

Advanced Reboot option
c-Cleaner

Date :
Page No :

SCAS

- SCASB [scan string Byte]
- SCASW [scan string word]

SCASB

syntax :- SCASB

This instruction compares a ~~by~~ byte in AL with a byte in an extra segment memory location pointed by DI.

DI is incremented by 1 if D=1

(or)

DI is dec by 1 if D=1

Operation

$$(AL) \xleftarrow[\text{compare}]{\text{byte}} [ES]:[DI]$$

→ DI ← DI+1 , if D=0

→ DI ← DI-1 , if D=1

SCASW :-

Syntax :- SCASW

Operation :-

$$(AX) \xleftarrow[\text{Compare}]{\text{word}} [ES]:[DI]$$

→ DI ← DI+2 , if D=0

→ DI ← DI-2 , if D=1

REP prefix

REP instruction is a prefix to another instruction. REP prefix allows a string instruction to perform

the operation repetedly.

→ REP assumes the CX no of times the instruction to be repeated.

→ The CX content decremented automatically after every execution of REP

REP ⇒ Repeat the following instruction until CX = 0

or

Repeat if CX ≠ 0

And Exit if CX = 0

Ex:- MOV CX, 8 } Execution of CMPSB
REP CMPSB } is Repeated 8 times

REPE [Repeat if Equal]

{ Repeat until CX = 0 (and)
Repeat if operands are equal

{ Exit if CX = 0
and
Exit if operands are not equal.

REPNE [Repeat if not equal]

{ Repeat until CX = 0 and
Repeat if operands are not equal.

{ Exit if CX = 0 and.
Exit if operands are equal.

REPE = REPZ

REPNE = REPNZ.

1. Assuming that there is a spelling of "Europe" in an electronic dictionary and a user types in "Euorope". Write a program that compares these two strings and displays the following messages depending on the result.
(i) if they are equal, display the string "Spelling is correct" else display spelling is not correct.

```
.MODEL SMALL
.DATA
    DICT DB "Europe$"
    TYPED DB "Euorope$"
    MSG1 DB "Spelling is correct$"
    MSG2 DB "Spelling is incorrect$"
.CODE
    MOVE AX, @DATA
    MOV DS, AX
    MOV ES, AX
    CLD    // set auto increment mode for SI and DI
    LEA SI, DICT.
    LEA DI, TYPED
    MOV CX, 06
REPE CMPSB
    CMP CX, 0
    JNE incorrect.
    LEA DX, MSG1
    MOV AH, 09
    INT 21H.
    JMP EXIT
incorrect: LEA DX, MSG2
    MOV AH, 09
    INT 21H
EXIT : MOV AH, 4CH
    INT 21H
    END.
```

$\overline{CS}$ → chip select

Organization = $2^N \times M$

Note

$2^{10} = 1024 = 1K$.
$2^{11} = 2^*K$.
$2^{12} = 4K$.
$2^{13} = 6K\ 8K$.
$2^{14} = 16K$.
$2^{15} = 32K$.
$2^{16} = 64K$
$2^{17} = 128K$
$2^{18} = 256K$
$2^{19} = 512K$
$2^{20} = 1M$
$2^{21} = 2M$
$2^{22} = 4M$
$2^{23} = 8M$
$2^{24} = 16M$
$2^{25} = 32M$
$2^{26} = 64M$
$2^{27} = 128M$
$2^{28} = 256M$
$2^{29} = 512M$
$2^{30} = 1G$

1. Determine the capacity and organization of a memory chip which has 12 address lines and 8-data lines

Capacity = $2^{12} \times 8 = 4K \times 8 = 32$ Kbits

EEPROM → Electrically Erasable Programmable ROM

Organization :- $2^n \times M$.

$= 4K \times 8$.

2. A 512k chip has 8-data lines, Determine
8. the no of address lines

Ans $8 \times 2^n = 512k$.

$\frac{512k}{8} = 2^n$

$64k = 2^n$ $\quad 2^{16} = 64k$

$n = 16$

Organization $= 2^{16} \times 8$

3. Find the organization and capacity of the following
i) EEPROM with $A_0 - - A_{14}$ & $D_0 - - D_7$

$A_0 - - A_{14}$: Address line; $D_0 - - D_7$: Data line

$n = 15$
$M = 8$.

Organization $2^{15} \times 8$.

capacity $= 2^{15} \times 8 = 32K \times 8 = 256$ kb
or 32KB

4. (ii) SRAM with $A_0 - - A_{12}$ $D_0 - - D_3$

$n = 13$
$M = 4$.

Organization $= 2^{13} \times 4$.

capacity $= 8K \times 4 =$ 32K bits $= 4KB$

4. Find the capacity, Address and Data pins in the following.

i) 16K x 8 ROM.

$= 2^{14} \times 2^{10} \times 8$

$= 2^{14} \times 8$

Address lines = 14

Data lines = 8.

ii) 256K x 4 DRAM.

Address line = 18.

Data line = 4.

capacity = 128 KB.

iii) 4M x 8.

$2^{22} \times 8$, n = 22, M = 8.

iv) 8M x 8 = n = 23, M = 8

$2^{23} \times 8$

(v) 32M x 8

$2^{25} \times 8$ n = 25, m = 8.

(vi) 32G x 8

$2^{35} \times 8$ = n = 35, M = 8

(vii) 128 G x 8

$2^{37} \times 8$ = n = 37, M = 8.

20/4/17

Incomplete 6-7 pages

8-bit memory interface

Inorder to attach a memory device to the microprocessor. It is necessary to decode the addr sent from the microprocessor.

For example, if a 2K×8 memory chip has to be interfaced to the microprocessor, the 11 addr lines of the microprocessor to be connected to the chip and the remaining 11 address line of the µP to be connected to a decoding logic.

The o/p of decoding logic must be connected to the enabled i/p ($\overline{CS}$ or $C\overline{E}$) of the Decoder.

8088 µP has 20-bit address bus ($A_0 - A_{19}$) and 8-bit Data bus ($D_0 - D_7$)

→ Design a NAND gate decoder that selects EPROM of 2K×8 for the memory location range. FF800H - FFFFFH of 8088 µP

Soln memory chip
- 2K×8 ⇒ Memory chip as 11 address line ($A_0 - A_{10}$) 8-data lines.

$2 \times 2^{10} \times 8$

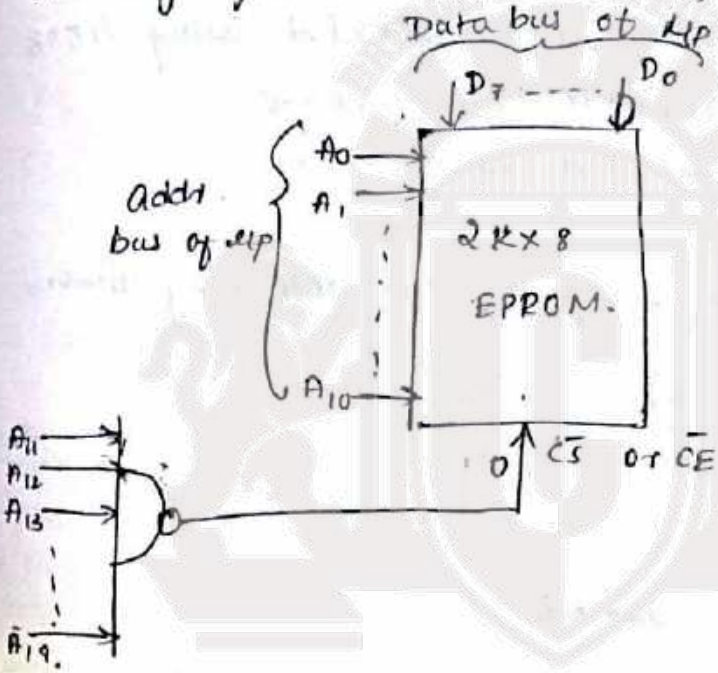
$= 2^{11} \times 8$

Address in Binary.

A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Hex
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	FF800H
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	FF801H
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	FF802H
⋮									⋮											
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH

To be connected to decoding logic.

To be connected to memory chip.



2. Address Range 40000H to 41FFFH using 8K×8 chip

8K×8

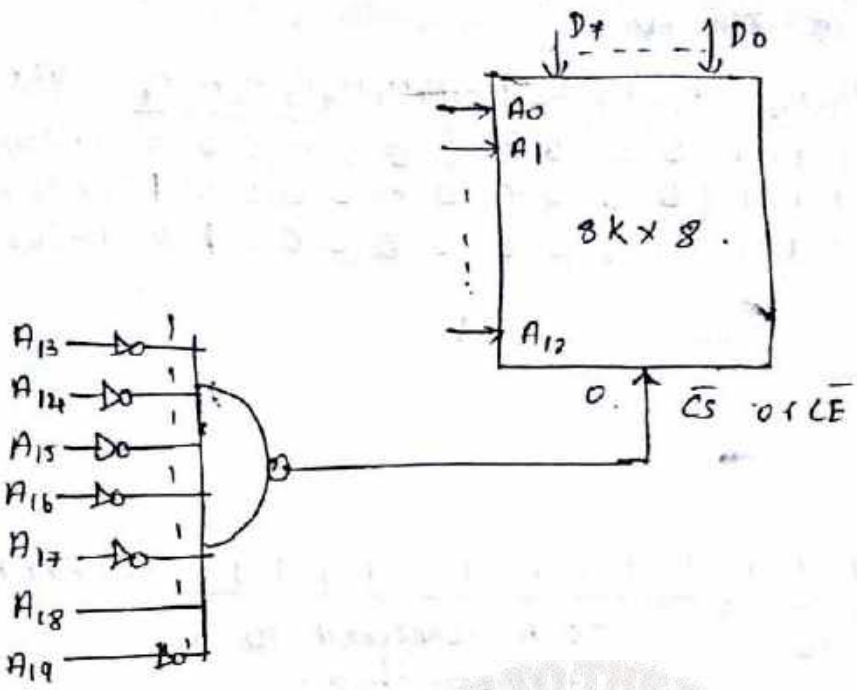
$2^3 \times 2^{10} \times 8$

$2^{13} \times 8$

13 → address lines.

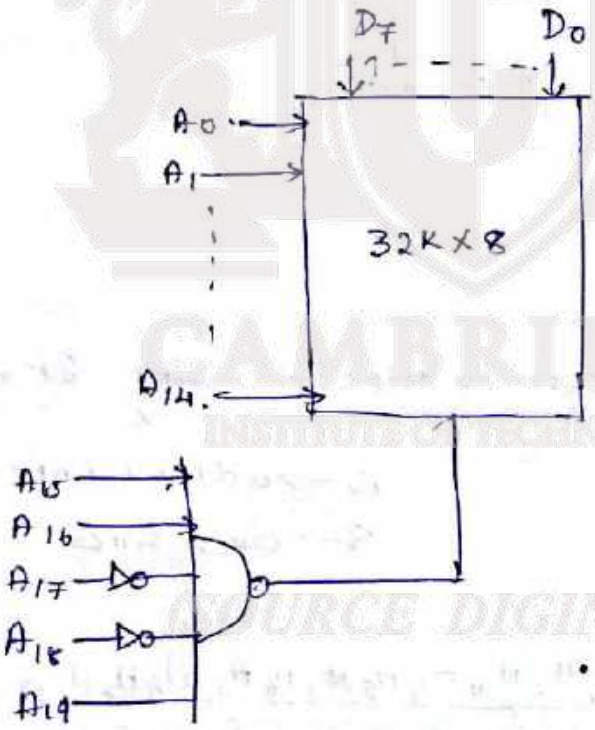
8 → data lines

A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	40000
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	40001
⋮									⋮											
0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	41FFF



1. Address range DF000H to DFFFFH using 4Kx8
2. DF800H to DFFFFH using 2Kx8
3. C0000H to C0FFFH using 2Kx8

3. Find the address range of the following memory design.

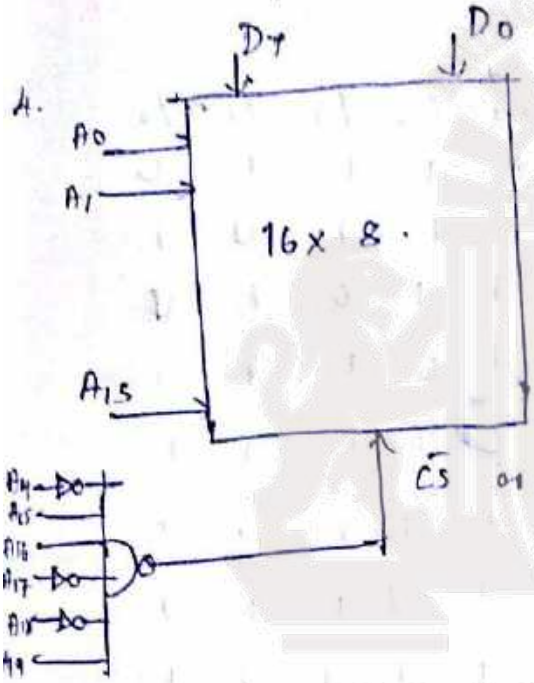


A_{19} A_{18} A_{17} A_{16} A_{15}	A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0	Hex
1 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	98000H
1 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	98001H
1 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	98002H
1 0 0 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	9FFFFH

Address rang : 98000H to 9FFFFH.

group of (1111) ⇒ enable.

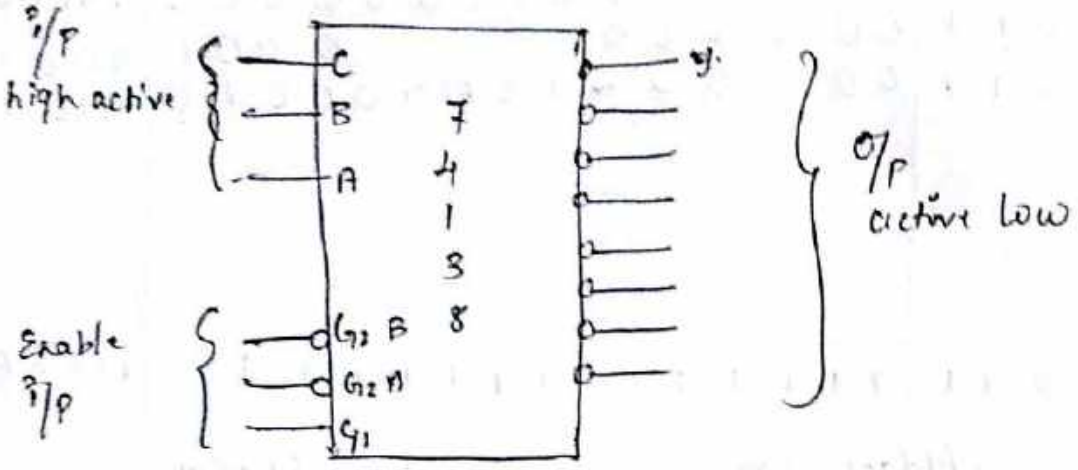
A.



A_{19} A_{18} A_{17} A_{16} A_{15} A_{14}	A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0	Hex
1 0 0 1 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	98000H
1 0 0 1 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1	
1 0 0 1 1 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1	9BFFFH

Address range 98000H to 9BFFFH

3-8 Decoder (IC 74138)



G_{2B}, G_{2A} } Active low G_1 → Active high.

C	B	A	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

(1) → Design memory system using 64 kx 8 memory dvp using 74138 decoder to select the memory addr range C0000H to CFFFFH

No of addr line = 16

64k × 8

$= 2^6 \times 2^{10} \times 8 = 2^{16} \times 8$

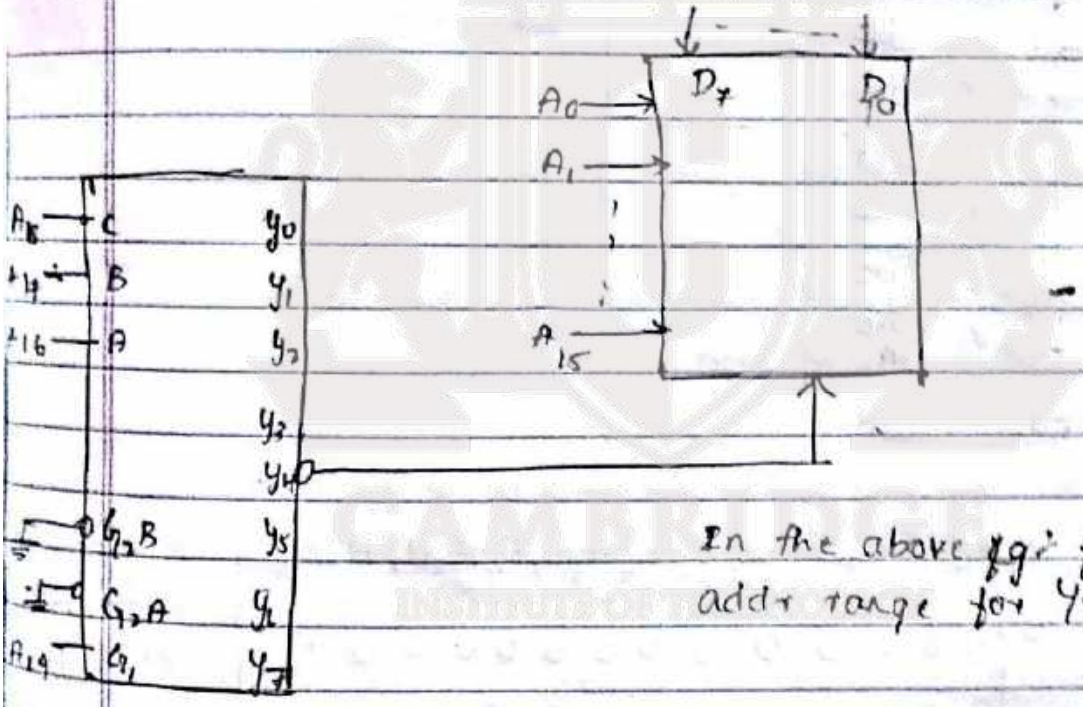
data lines = 8

A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Hex
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C0000H
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	C0001H
1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	CFFFFH

CBA

74138.

TO memory chip

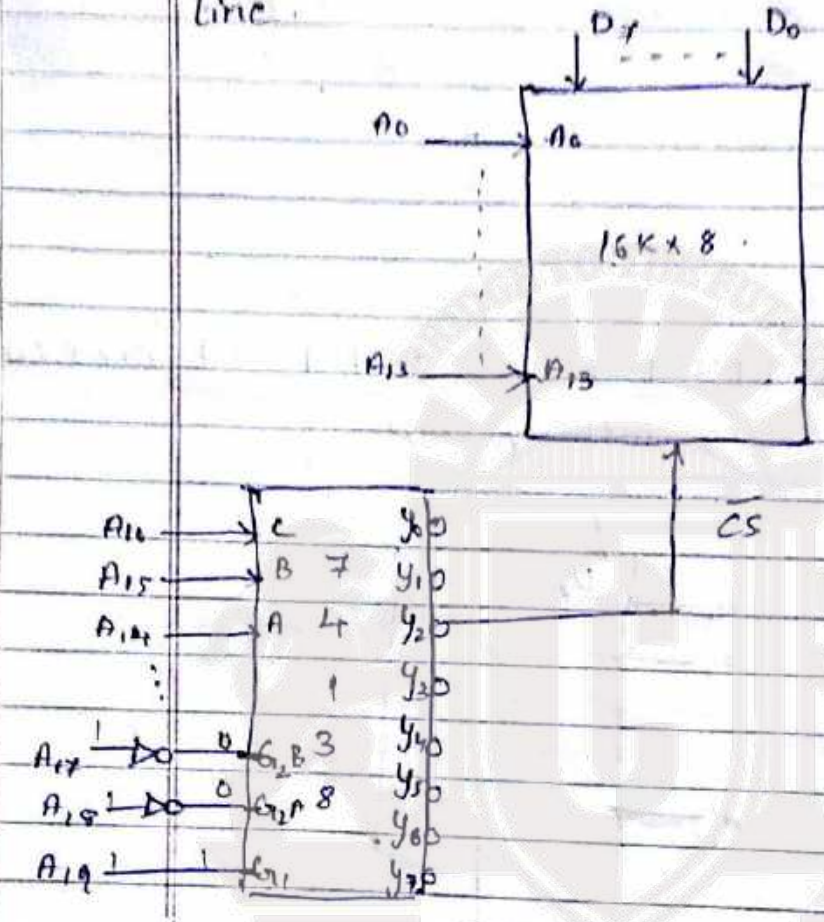


In the above fig: find the addr range for Y_2 & Y_7.

A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	H
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A0000H
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	A0001H
1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	AFFFFH

Y_2

1. Looking at the design in figure 1, find the address range for y_1, y_2, y_7. and also determine the block size controlled by each line.



A_{19} A_{18} A_{17}	A_{16} A_{15} A_{14}	A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0	Hex
1 1 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	
1 1 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	E8000H
1 1 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	E8001H
1 1 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	E8002H
	y_2		
1 1 1	0 1 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	EBFFFH

∴ Address range for y_2 is E8 000H to EBFFFH

ii)

A_{19} A_{18} A_{17}	A_{16} A_{15} A_{14}	A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0	Hex.
1 1 1	1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	F0000H
1 1 1	1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	
1 1 1	1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	
	.	.	
	y_4	:	
		:	
		:	
1 1 1	1 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	F3FFFH

C B A

∴ Address range is F0000H to F3FFFH.

iii)

A_{19} A_{18} A_{17}	A_{16} A_{15}	A_{14}	A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
1 1 1	1 1	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1	1 1	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1	1 1	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
			.
	y_7		:
			:
			.
1 1 1	1 1	1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

C B A

Address range is FC000H to FFFFFH.

Data Integrity in RAM and ROM.

→ To ensure Integrity of contains of ROM, Every PC must perform a check-sum calculation.

→ The process of check-sum will detect any corruption the contents of the data in ROM.

→ Checksum method uses a checksum byte for error check.

→ This checksum byte is an extra byte that is appended at the end of a series of bytes of data.

→ To calculate the checksum byte, the following steps are performed.

Step1 Add the byte together and drop the carry at the end.

Step2 take the 2's complement of the total sum and is the checksum byte.

→ At the receiving end, the error checking is performed by adding all bytes, including the checksum byte.

→ The result must be zero. If the result is not zero, one or more bytes of data have been corrupted.

Ex: Assume that we've 4-bytes hexadecimal data 25H, 62H, 3FH and 52H.

→ Find the check-sum byte

→ Perform the check-sum operation to ensure data integrity

16) 24 (1
16
8

5.

PAGE :
DATE : / /

→ If the second byte 62H had been changed to 22H show how checksum detects error.

i) 25H
62H
3FH
52H
---
1 08H ⇒ 0001 1000
8
1111 0111

1

1111 1000
E8H.

Carry Ignore.

Checksum Byte = E8H.

ii) 25H
62H
3FH
52H
E8H
---
1 00H ⇒ No Error.

iii) 25H
22H
3FH
52H
~~E8H~~
E8H.

E0H ⇒ Non-zero Sum ⇒ Error detected.

2. Assuming that the last byte of the following data is the checksum-byte, show whether the data has been corrupted or not.

Data = 28H, C4H, 9EH, BEH, 87H, 65H
83H, 50H, A7H, 51H

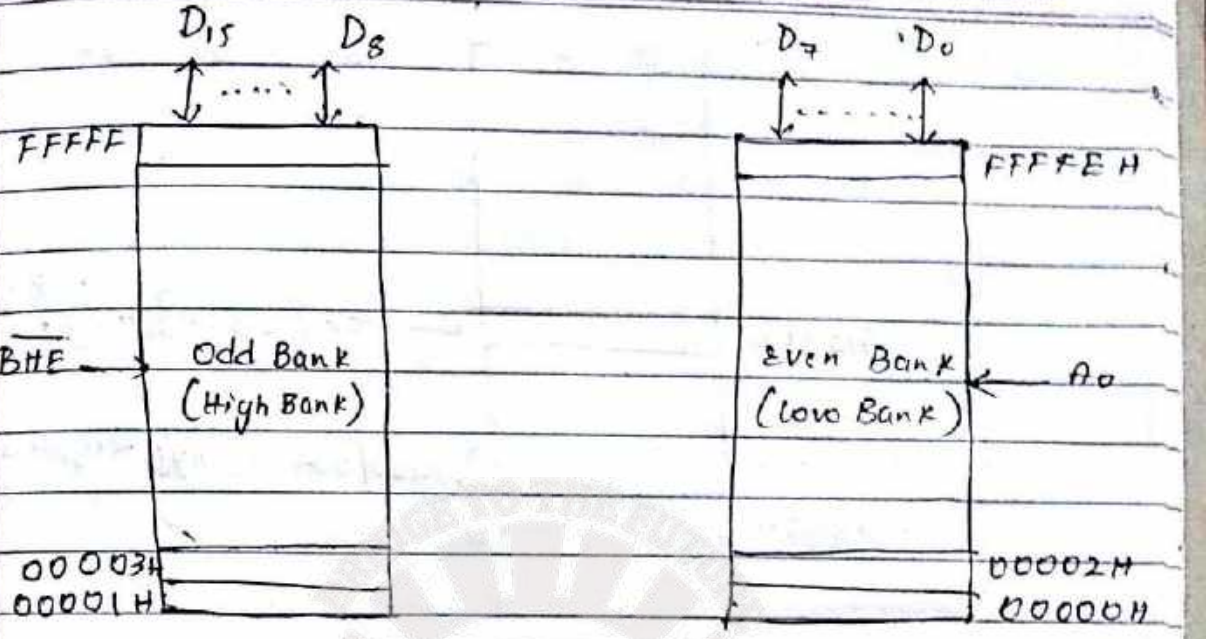
```
   4
   28H
1  C4H
   9EH  1
1  BEH  1
   87H  1
1  65H
   83H
   50H
   A7H  1
   51H
  ----
   00H  ⇒ no error in the data.
```

∴ 16-bit memory interphasing :- [8086]

The address space of 16-bit microprocessor is divided into two memory banks namely High bank (odd Bank), Low Bank [Even Bank]

→ Low Bank is connected to the lower half of 16-bit data bus [ie $D_0 - D_7$]

→ High bank is connected to the upper half of the data bus [ie $D_8 - D_{15}$]



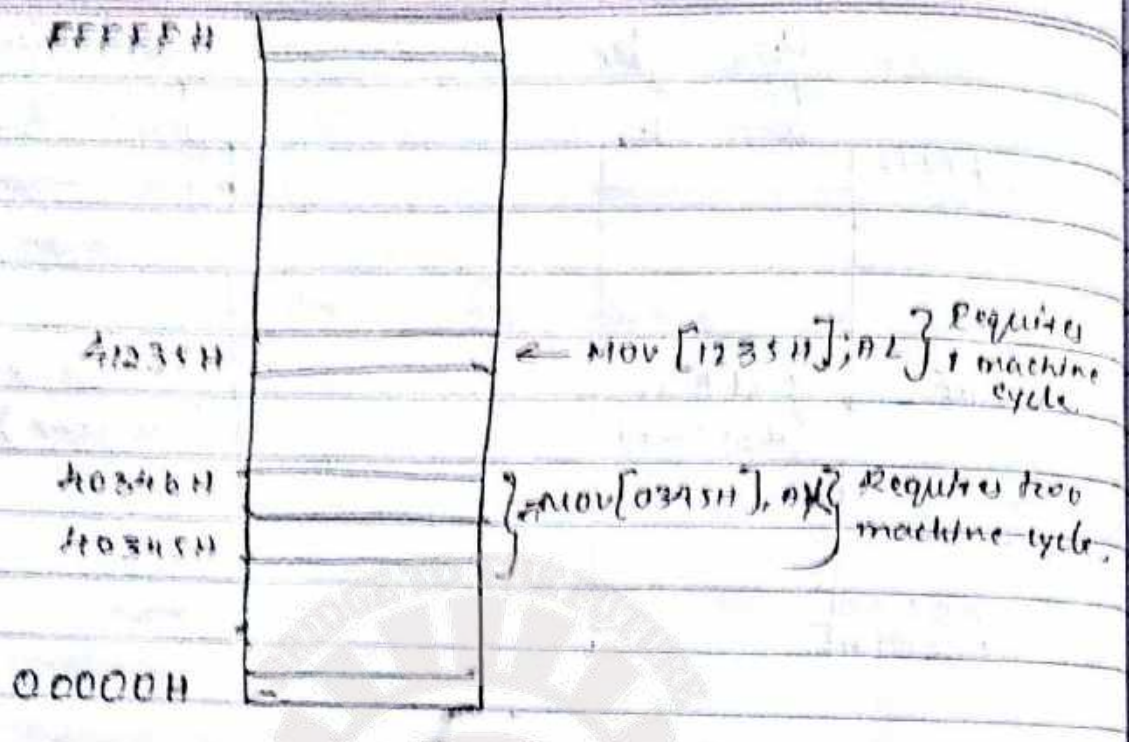
To distinguish b/w odd and even banks, the microprocessor provide a signal called $\overline{BHE}$ [Bus high enable] signal along with the Address line A_0. The following table shows the selection of banks.

$\overline{BHE}$	A_0	Enabled Bank
0	0	Both banks are enabled for 16-bit data transfer
0	1	High bank is enabled for 8-bit data transfer (D_8-D_{15})
1	0	Low bank is enabled for 8-bit (D_0-D_7).
1	1	Both banks are disabled.

Accessing Data in Odd and Even Bank.

Accessing data in 8-bit µP (8088)
(Only one memory bank)

P.T.O



Assume DS = 4000H.

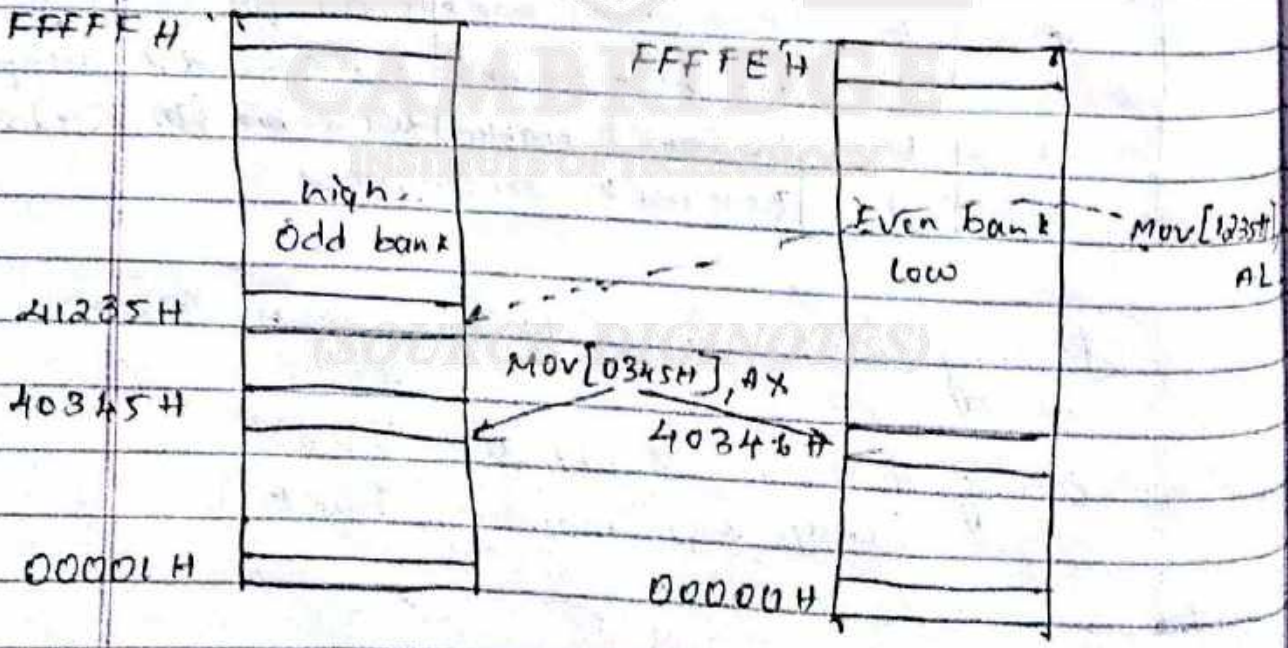
MOV [1235H], AL

Physical addr = DS×10H + Offset

= 40000 + 1235H

= 41235H

Accessing data in 16-bit µP (8086)



Machine cycle = 4 clock signal.

→ Accessing odd bank and requires one machine cycle.

For

MOV [0345H], AX → accessing odd and even bank in one machine cycle.

23/4/16

: I/O Interfacing :-

I/O Instruction :-

IN :- This instruction will receive 8-bit or 16-bit data from an I/O port.

→ The received data is available in AL or AX depending on the size of the data

Syntax :- ~~IN Source Destinat~~

IN Dest, source
(AL or AX) (Port)

Format 1 :

IN AL, Port address (Port addr is 8-bit)
IN AX, port address

Format 2 : (i) MOV DX, Port address } Port addr is
IN AL, DX } 16-bit

(ii) MOV DX, Port addr.
IN AX, DX

→ In Format 1 port address is 8-bit and we can have 256 port address (0 to from 00H to FFH

$2^{16} = 65536$

→ In Format 2, port addr is 16-bit and we can have 65536 port address

OUT

Syntax : OUT Dest, source
(Portaddr) (AL or AX)

Format 1

(i) OUT Portaddr, AL

(ii) OUT portaddr, AX

Format 2 (i) MOV DX, Portaddr
Out DX, AL

(ii) Mov DX, Portaddr
Out DX, AX

Ex:-

1. In a given 8088 based system, port addr 22H is an i/p port for monitoring the temperature, write instructions to monitor the temperature in the port continously. If Temperature reaches 100° the ~~the~~ reg BH should contain 'Y'

port addr.

NEXT: IN AL, 22H } Reading temp from port 22H
CMP AL, 100 } check the temp, if temp reaches
JNZ NEXT } 100, exit and store 'Y' in BH
MOV BH, "Y"

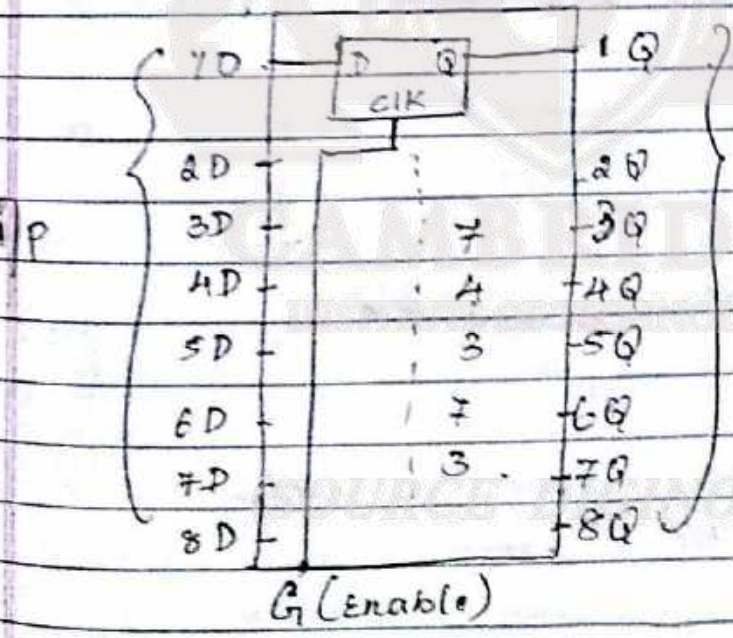
2. Write assembly language instruction to I/P data from port 300H and send it out to Port 304H.

Port addr is 16-bit

```
MOV DX, 300H   } Read data given 300H
IN AL, DX.     }

MOV DX, 304H   } send data to port 304H
Out DX, AL     }
```

I/O Address decoding

We use an IC for decoding ie 74373 → Latch



Truth table.

CLK	D	Q
1	1	1
1	0	0
0	Previous State	

Eg Show the design of an O/p port with an I/O address 99H using AND Gate, ~~and~~ inverters, and 74373 Latch.

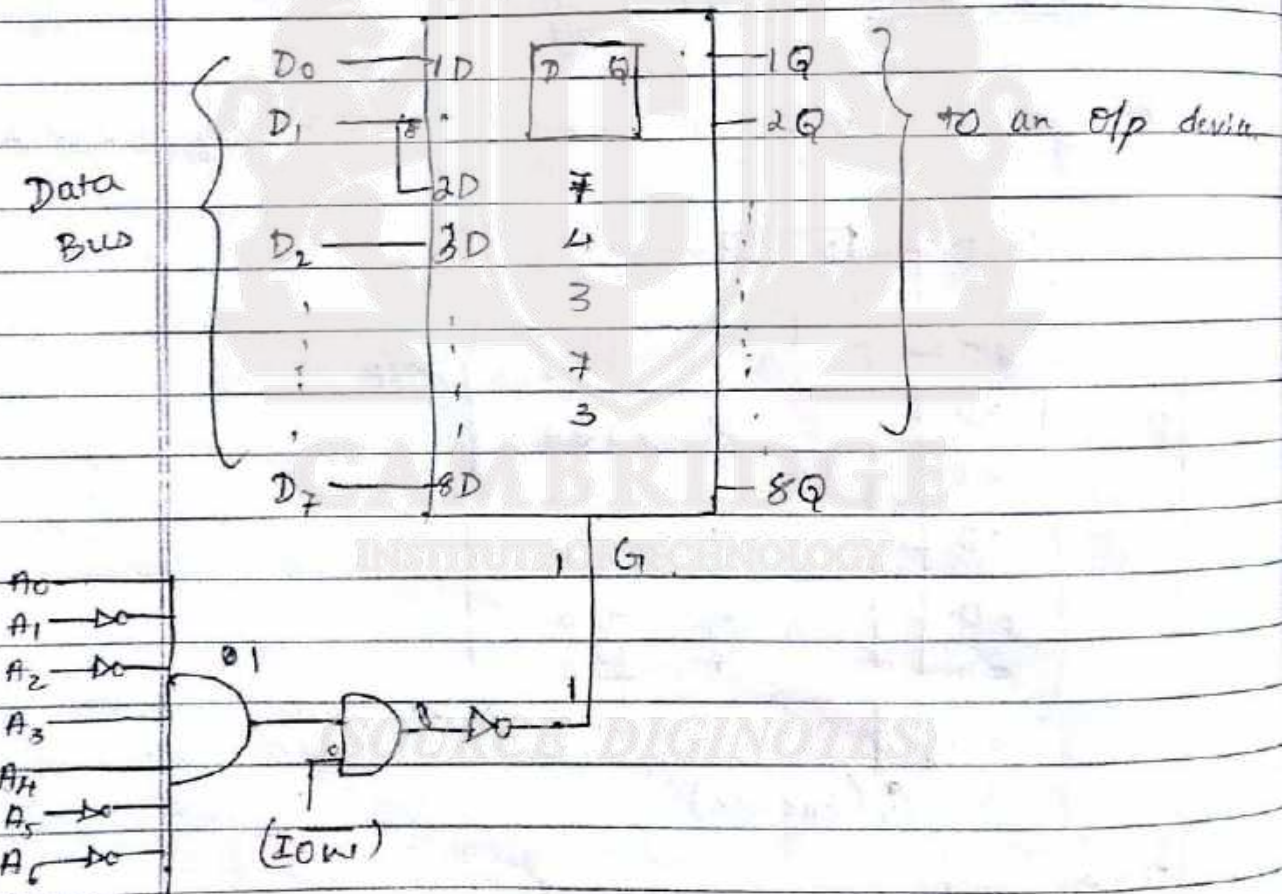
OUT

Design OUT 99H, AL

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	0	0	1	1	0	0	1	= 99H

$\overline{(IOWR)}$ · (I/O write control signal)

→ It is an O/P control signal from µP

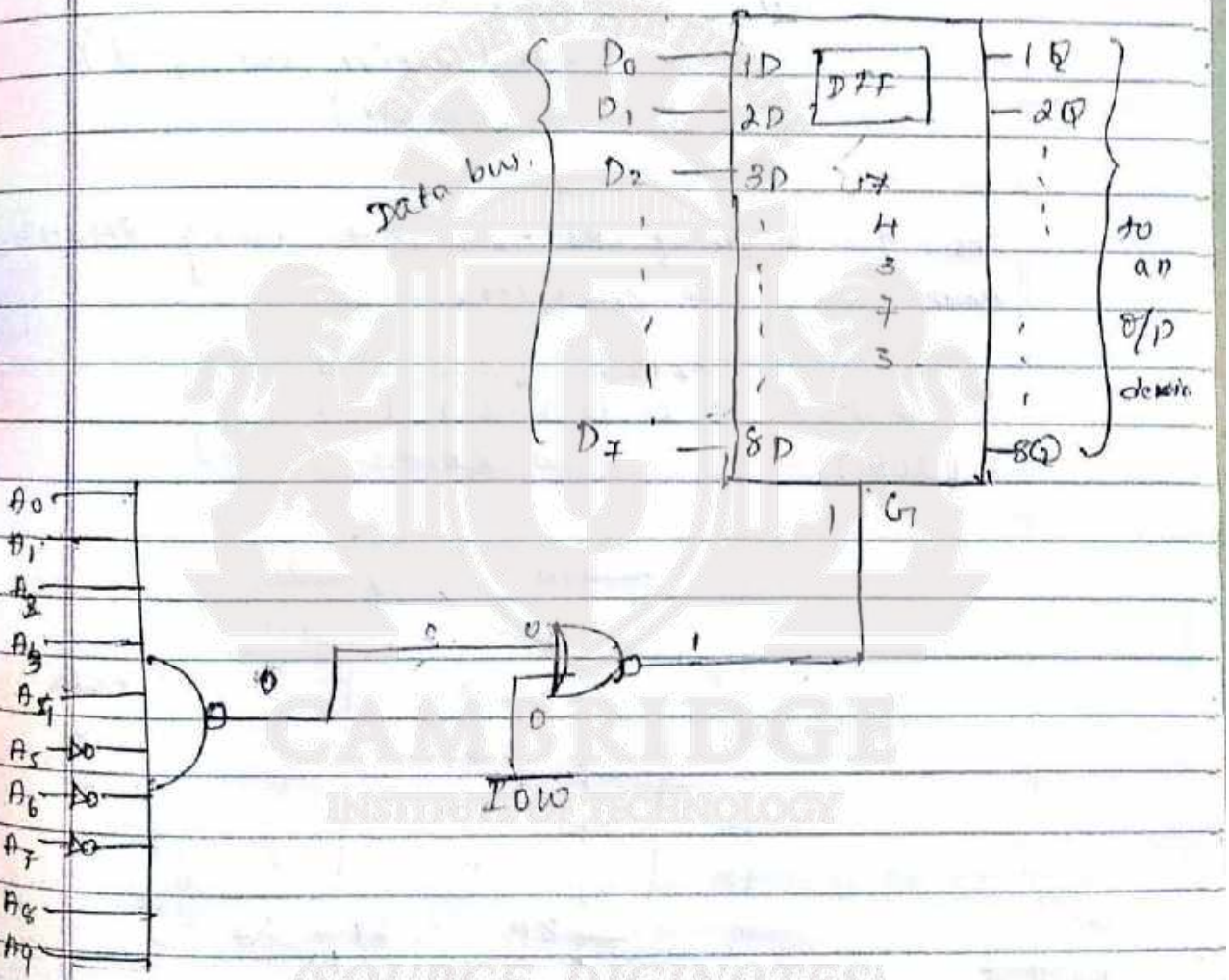


74373 is used only for out instruction

2 Show the design of an O/P port with an I/O address of 31FH using 74373 Latch and NAND Gate.

31FH = $A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0$ = 11 0001 1111

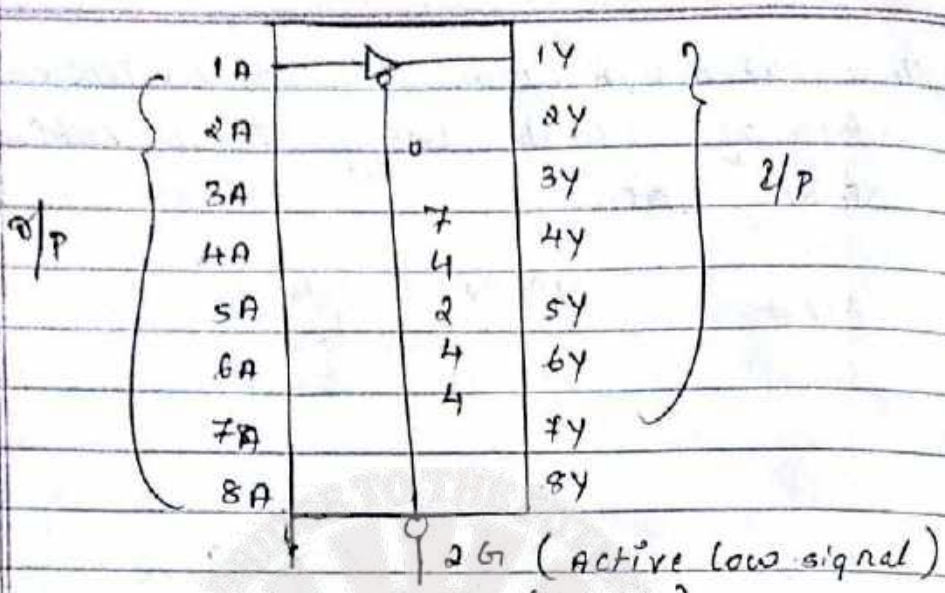
3 1 F



IC 74244 [usually its a Buffer]

P. T. O.

During Read ~~IOR~~ $\overline{IOR}$ is always @ 0

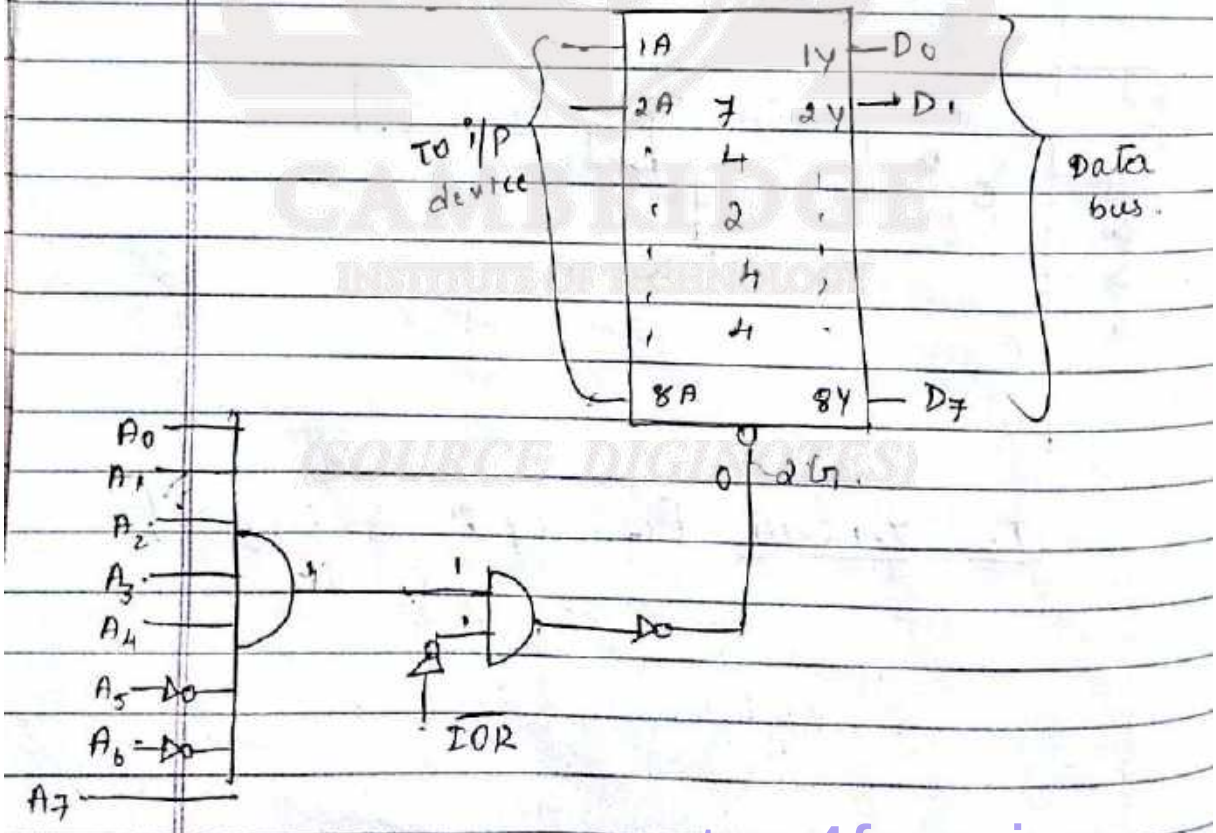


1. Show the design of IN AL, 9FH using 74244, AND gate and inverters.

$A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$

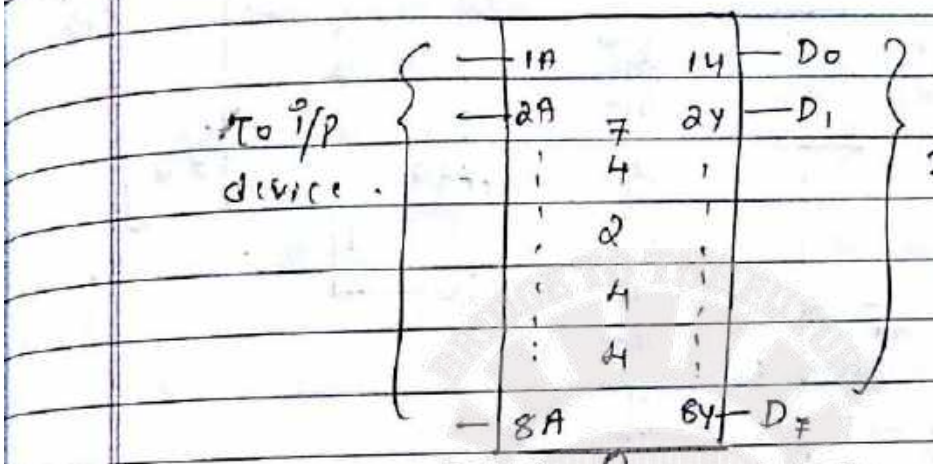
9FH = 1 0 0 1 1 1 1 1

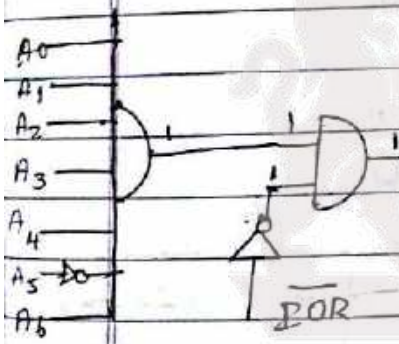
($\overline{IOR}$) → (I/O Read control signal)



2. IN AL, 5FH.

	A_6	A_5	A_4	A_3	A_2	A_1	A_0
5F =	1	0	1	1	1	1	1



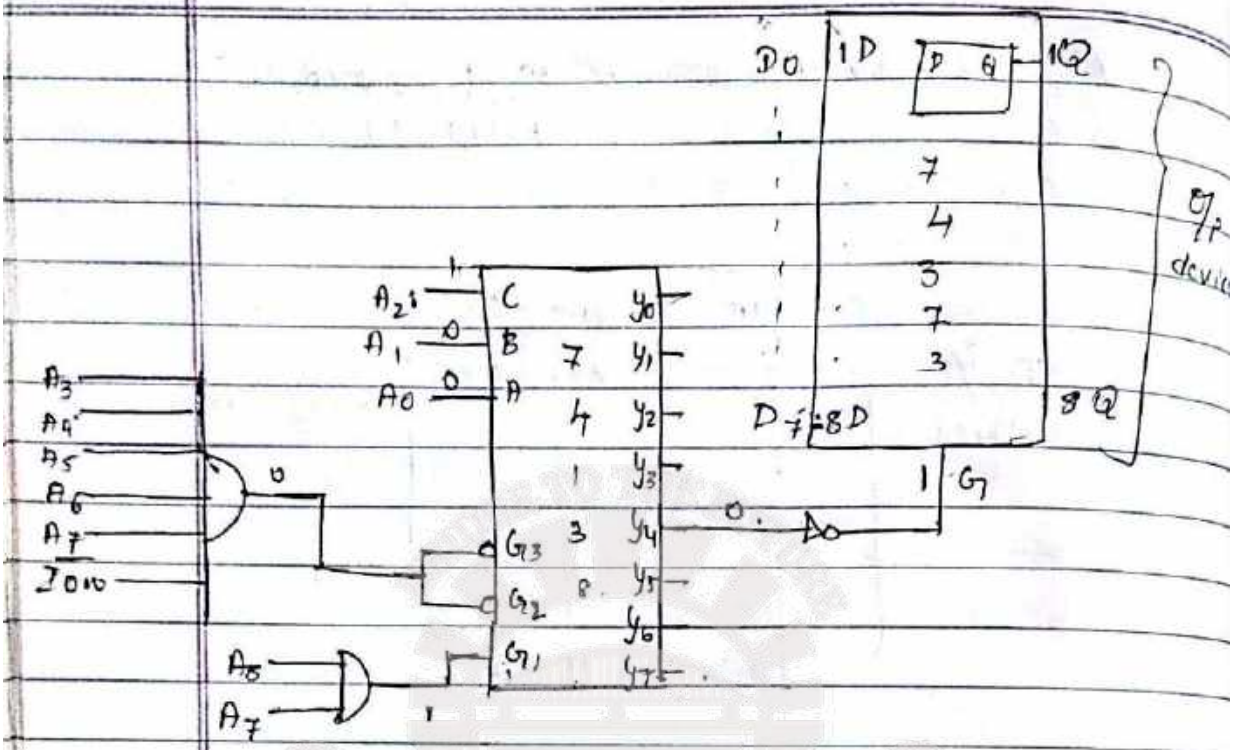


I/O Port design using 74138

1. OUT 304H, AL.

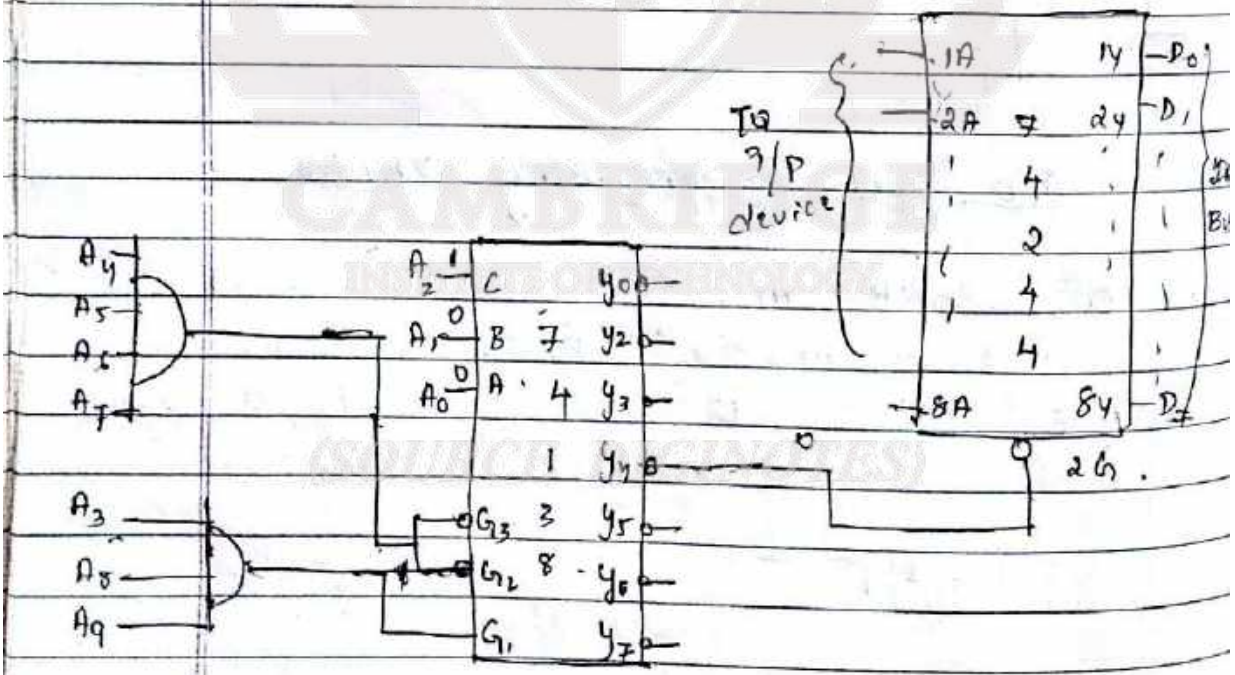
$\overline{IOW}$	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	0	0	0	0	1	0	0

P.T.O



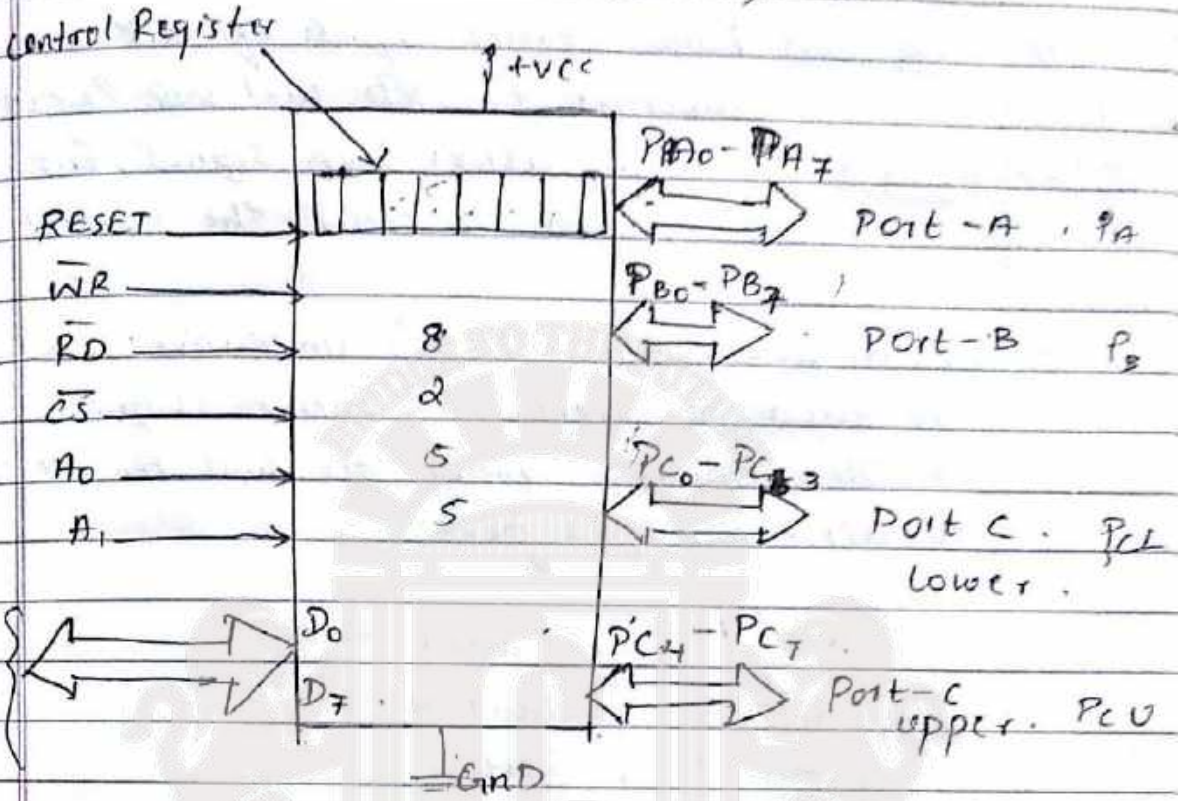
2. IN AL, 80CH

	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
IOR 0	1	1	0	0	0	0	0	0	0	0



26/4/17

PROGRAMMING And Interfacing (8255)



→ It is a peripheral IC used for interfacing.
→ This is programmable IC [programmable ports]
→ It is a 40-pin IC.

→ It has 3 8-bit ports.
→ Port A [PA0 - PA7] it is an 8-bit port and can be programmed as input or O/P.
→ Port B [PB0 - PB7] it is an 8-bit port and can be programmed as I/P or O/P.
→ Port C [PC0 - PC7] can be used all as I/P or O/P.
→ Port C can also be used as 2 4-bit ports ie Port C lower [PC0 - PC3] and Port C upper [PC4 - PC7]

→ $\overline{RD}$ and $\overline{WR}$ active low control signals are i/p's to 8255.

→ $\overline{IOR}$ and $\overline{IOW}$ control signals of uP are connected to $\overline{RD}$ and $\overline{WR}$ respectively.

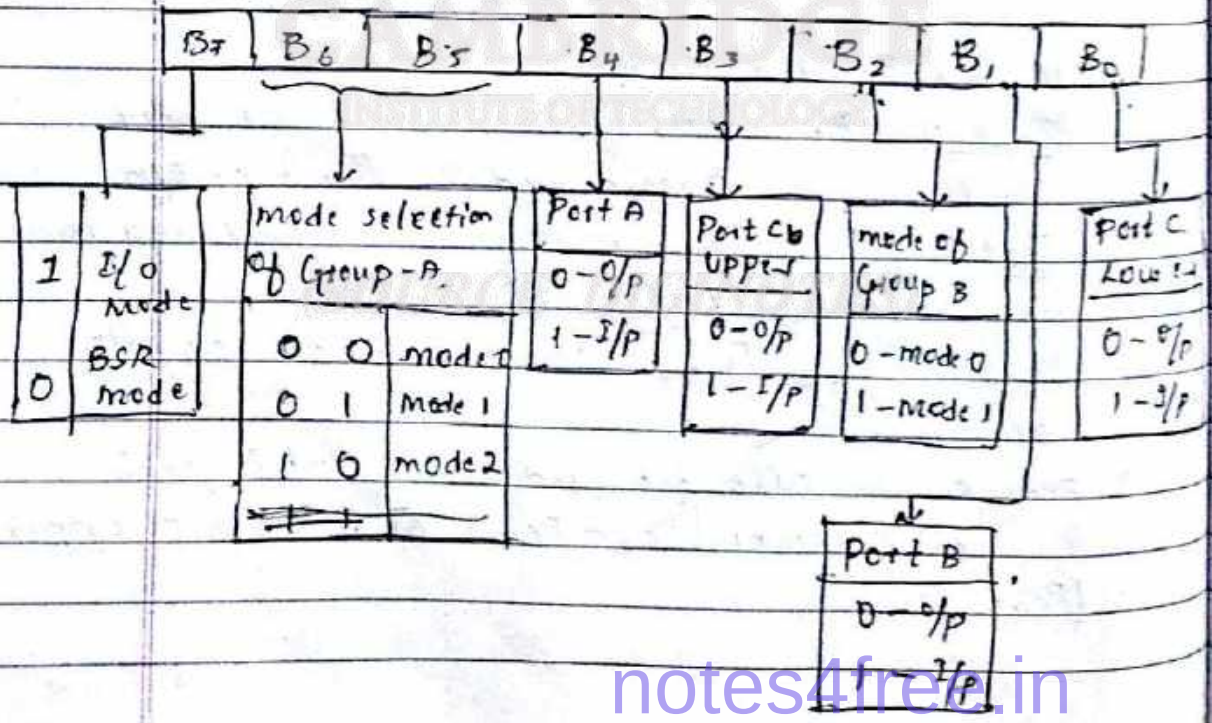
→ RESET It is an active high signal, input to 8255, used to clear the control reg.

→ A0, A1 and $\overline{CS}$:- $\overline{CS}$ is an active low signal is used to select the entire chip.

→ The Address signals A0 and A1 are used to select specified port

$\overline{CS}$	A1	A0	Selected Port
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register

Control Register format of 8255



Port-A and Port-C Upper → Group A.
Port-B and Port-C Lower → Group B.

I/O mode → mode 0 → simply send or receive data
→ ~~mode 1~~ → There is a handshake signal. [one way]
→ ~~mode 2~~ → Two way handshaking.

BSR mode → will see later

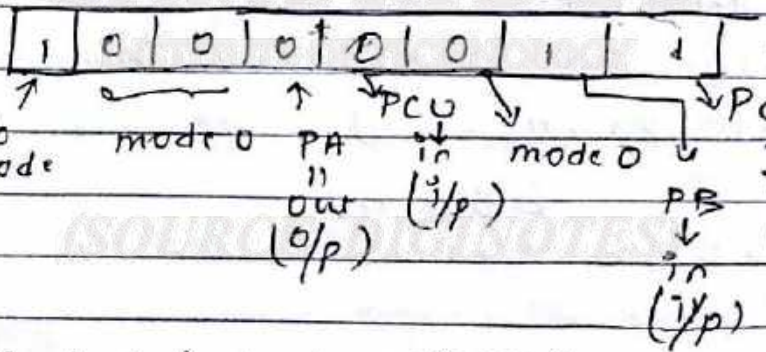
We are interested only in I/O mode, mode 0.

Example 1

1. (a) Find the control word if PA = out, PB = in
$PC_0 - PC_3$ = in, $PC_4 - PC_7$ = out.

(b) Program the 8255 to get data from Port B and send it to Port A
The Data from PCL is sent out to PCU. use the Port address of 300H to 303H for the 8255 chip

Sol a)



Control byte for 8255 is 83H

b)

P.T.O

1010

b

	A_9 A_8 A_7 A_6 A_5 A_4	A_3 A_2	A_1 A_0	
300H	1 1 0 0 0 0 0 0	0 0	0 0	Port A
301H	1 1 0 0 0 0 0 0	0 0	0 1	Port B
302H	1 1 0 0 0 0 0 0	0 0	1 0	Port C
303H	1 1 0 0 0 0 0 0	0 0	1 1	Control register

To be connected to A_1 & A_0 of 8255.

```
MOV DX, 303H   } Initialization of
MOV AL, 83H    } Control Register.
OUT DX, AL

MOV DX, 301H   } Receive data from PB
IN AL, DX

MOV DX, 300H   } Send data to PA.
OUT DX, AL

MOV DX, 302H   } Receive Data from PCL
IN AL, DX.
MOV CL, 4      }
ROL AL, CL     }
MOV DX, 302H   } Send Data to PCU
Out DX, AL
```

B_3 B_2 P_{C_1} P_{C_0}

0 0 0 0 1 0 1 0

P_{C_7} P_{C_6} P_{C_5} P_{C_4}

1 0 1 0.

2/4/17

1. 8255 shown in the following figure is configured as follows

Port A – i/p

Port B – o/p

and all the bits of Port-c has o/p.

i) Find the port address assigned to A, B, C and CR [control register]

ii) Find the control byte for the control Reg

iii) Program the ports to i/p Data from Port A and send it to Port B and Port C.

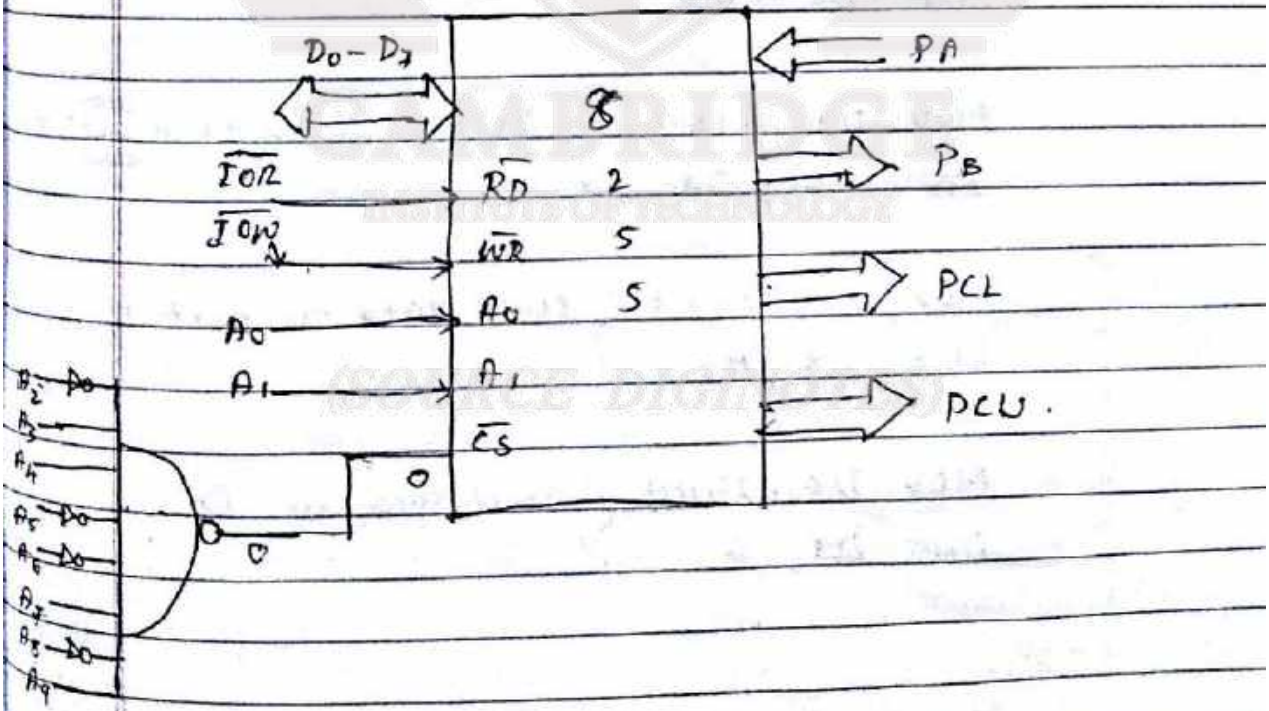
Sol

Port – A → i/p

B → o/p

PCL → o/p

PCU → o/p.

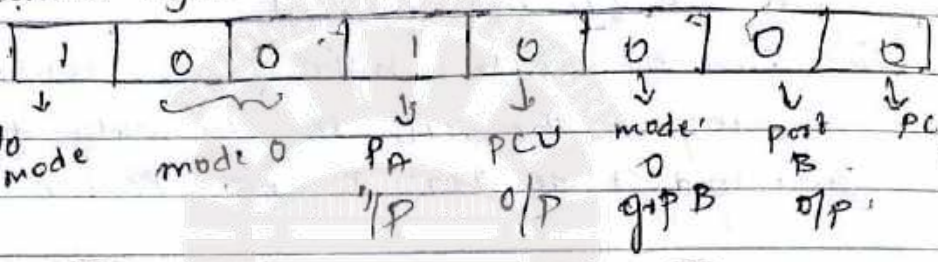


(i)

	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
298H =	1	0	1	0	0	1	1	0	0	0	Port A
299H =	1	0	1	0	0	1	1	0	0	1	Port B
29AH =	1	0	1	0	0	1	1	0	1	0	Port C
29BH =	1	0	1	0	0	1	1	0	1	1	Control Reg
	2		9						B		

(ii)

Control byte



= 90H.

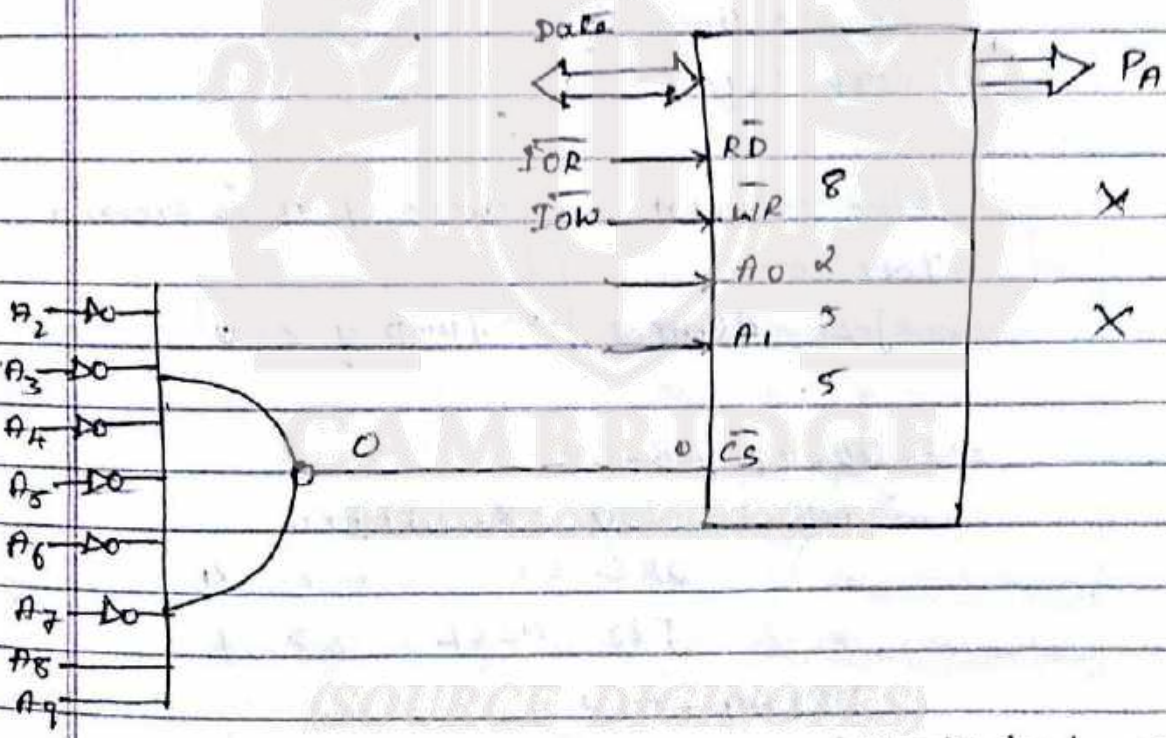
(iii)

```
MOV DX 29BH   } Initialization of
MOV AL 90H      Control Reg.
OUT DX, AL

MOV DX 298H   } Read Receive data from port A.
IN AL, DX

MOV DX, 299H  } send data to port B
OUT DX, AL

MOV DX, 29AH  } send data to PC.
OUT DX, AL
```

2. Show the Address decoding where Port-A of 8255 has an I/O address of 300H, and also write a program to toggle all bits of PA continously with a delay. Use INT 16H to exit if there is a key pressed

Sol

A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	0	0	0	0	0	0	0	0	= 300H
1	1	0	0	0	0	0	0	0	1	= 301H
1	1	0	0	0	0	0	0	1	0	= 302H
1	1	0	0	0	0	0	0	1	1	= 303H



X → dont care.

control word

1	0	0	0	X	0	X	X

P_A = 80H

```
AGAIN: MOV DX, 303H      } Initialize control
       MOV AL, 80H         reg.
       OUT DX, AL

       MOV DX, 300H      } Send AAH to PA
       MOV AL, AAH
       OUT DX, AL

       CALL DELAY

       MOV DX, 300H      } Send 55H to PA
       MOV AL, 55H
       OUT DX, AL
       CALL DELAY

       MOV AH, 01H       } check if A is pressed
       INT 16H
       JZ AGAIN          → Jump if Z = 0
```

AAH = 1010 1010

Delay

55H = 0101 0101

Delay.

```
DELAY PROC
NEXT: MOV CX, OFFFFH
      DEC CX        → 4
      JNZ NEXT.     → 7.
DELAY ENDP
```

Total delay

PAGE :
DATE : / /

```
DELAY PROC                    65536 = FFFF
.MOV SI, FFFFH
L2: MOV CX, FFFFH
NEXT: DEC CX.
JNZ NEXT
DEC SI
JNZ L2.
.DELAY ENDP.
```

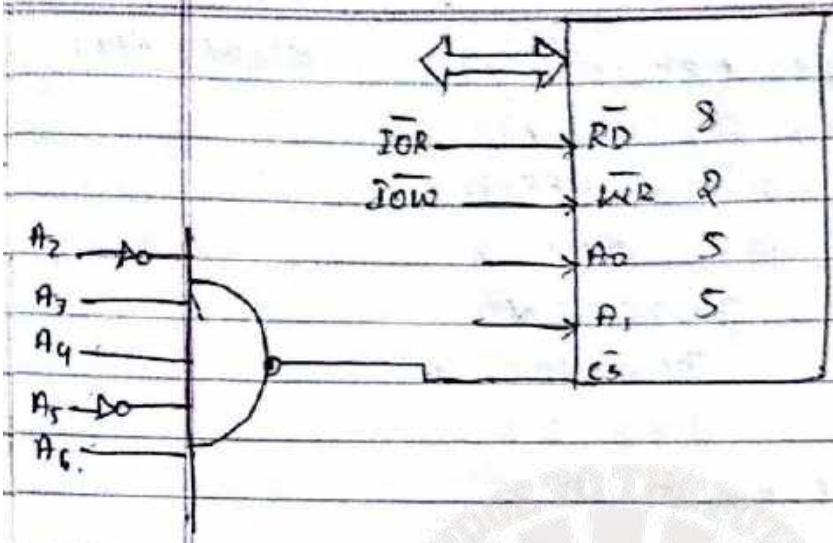
1. Show the decoding circuit for 8255 if we want port A to have an address 68H using NAND Gate inverters.
2. Write a P to monitor for a temperature of 100. if it is equal, it should be saved in Register BL, also send AAH TO Port B and 44H to port C. use the port addr of your choice
3. If 91H is the control word, indicate which port is I/P & which is O/P.

1:

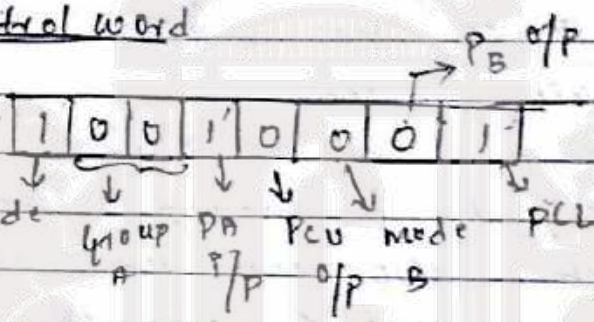
68H = 1011000

A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	0	1	1	0	0	0
1	0	1	1	0	0	1
1	0	1	1	0	1	0
1	0	1	1	0	1	1.

P.T.O.



3. Control word



Port A = i/p
Port B = o/p
Port C upper = o/p
Port C lower = i/p

2. Consider Port-A address = 22 H.

P_A = 22 H
P_B = 23 H
P_C = 24 H
CR = 25 H.

PAGE :
DATE : / /

28/11/ * Memory cycle time and inserting wait states

→ To access memory devices, CPU provides fixed amount of time, referred as memory cycle time.
→ In 8088/86, memory cycle time takes 4-clock cycles
→ 80286 through pentium, the memory cycle time is 2-clock cycles.
→ If memory is slow, and its access time doesnt match with the memory cycle time of CPU, extra time can be requested from the CPU to extend the memory cycle time.
→ The extra time is called wait state

1. Calculate the memory cycle time of a 20MHz 80386 System with
i) 0 wait state
ii) 1 wait state
iii) 2 wait state.

Sol: For 80386 memory cycle time is 2-clock cycle

$f = 20MHz$.

clock period $= \frac{1}{f} = \frac{1}{20 \times 10^6} = 0.05 \times 10^{-6}$

$= 50 \times 10^{-9}$.

$T = 50ns$

	Memory cycle-time (a)	wait state (b)	memory cycle time with wait state (a+b)
i)	$2 \times T = 2 \times 50ns = 100ns$	0 wait state	100ns.
ii)	100ns	1 wait state $= 1 \times T = 1 \times 50ns = 50ns$	$= 150ns$.
iii)	100ns	2 wait state $2 \times T = 2 \times 50ns = 100ns$	200ns.

2. A Bus bandwidth :-

It is Rate of Data transfer.

Bus bandwidth = $\left[\frac{1}{\text{Bus cycle time}}\right]$ x Bus width in Bytes

2. calculate the memory Bus-bandwidth for the following up is the Bus speed is 20MHz
i) 80286 system with 0 wait state and 1 wait state (16-bit data bus)
ii) 386 system with 0 wait state and 1 wait state (32-bit data bus)

Sol

$$T = \frac{1}{f} = \frac{1}{20MHz} = 50ns.$$

i) 80286 System, memo

memory cycle time is 2 clock cycle.
Bus width = 16 bits = 2 bytes
Bus cycle time = Memory cycle time with wait state.

memory cycle time

i) 0 wait state
Bus cycle time = 2T + 0 wait state.
= 2 × T = 100 ns.

∴ Bus bandwidth = $\left[\frac{1}{\text{Bus cycle time}}\right]$ × Bus width in Bytes

$= \left\{\frac{1}{100 \times 10^{-9}}\right\} \times 2$

50

9.

$= 0.01 \times 10^{9} \times 2$. Bytes/sec.
$= 20 \times 10^{6}$ = 20MBytes/S.

P.T.O,

1 wait state

Bus cycle time = 2T + 1 wait state.

= 100 + 50

= 150 ns.

$$\text{Bus BW} = \left[\frac{1}{150 \times 10^{-9}}\right] \times 2$$

$= 6.66 \times 10^{3} \times 2 \times 10^{9}$

$= 0.043 \times 10^{6}$

13.3 MBps.

ii) 386 system

a) 0 wait state

$$\text{Bus bandwidth} = \left[\frac{1}{\text{Bus cycle time}}\right] \times \text{Bus width in bytes}$$

$$= \frac{1}{\frac{100}{25}\text{ns}} \times 4$$

= 40 MBps

b) 1 wait state

$$\text{Bus BW} = \left[\frac{1}{\text{Bus cycle time}}\right] \times \text{Bus width in byte}$$

$$= \frac{1}{150\text{ns}} \times 4.$$

$$= \frac{10^{8}}{15} \times 4.$$

$= 0.267 \times 10^{8}.$

$= 26.7 \times 10^{6}$ Bps

= 26.7 MBps

Microprocessor and Microcontrollers

Sub code: 15CS44 — IV Sem CSE/ISE

MODULE - 04

Chapter 4 : ARM Embedded Systems

Difference between Microprocessor & Microcontroller

* → Microprocessor has only CPU inside it. They don't have RAM, ROM and other peripheral on the chip. (Memory controller, Interrupt controller, Timer etc).
→ System designer has to add them externally to make them functional.

* → Microcontroller has a CPU, in addition with a fixed amount of RAM, ROM and other peripherals, all embedded on a single chip

* → Microcontrollers are designed for specific purpose/tasks. Specific means applications where the relationship of input and output is defined. Depending on the input, some processing needs to be done and output is delivered.
Ex: Washing machine, remote control, microwave oven, cars, telephone, etc.
– Since the applications are specific, they need small resources like RAM, ROM, I/O port etc. and hence can be

embedded on a single chip. This in turn reduces the size and cost.

* Microprocessor find applications where tasks are unspecific like developing software, games, photediting, etc in a single system. They need high amount of resources like RAM, ROM, I/O ports etc.

* Processing speed of microcontrollers is about 8 MHz to 50MHz, but the speed of general purpose microprocessors is about 1GHz, so it works much faster than microcontrollers

* Microprocessors cannot be used stand alone. They need other peripherals like RAM, ROM, buffer, I/O ports etc. and hence a system designed around a microprocessor is quite costly.

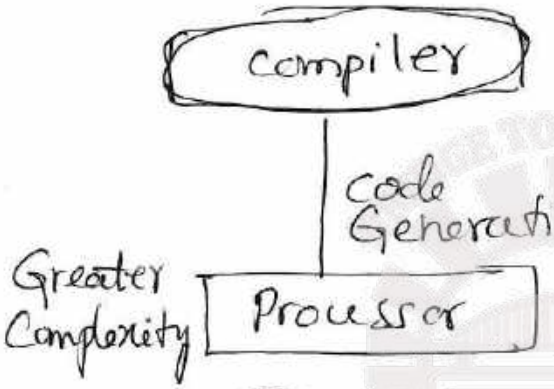
* In microcontroller program memory and data memory are seperate

In Microprocessor, program and data memory are stored in same memory module.

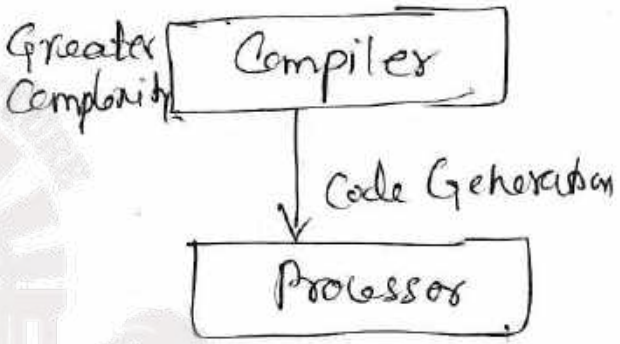
Difference between CISC & RISC

CISC	RISC
→ Complex Instructional Set computer	Reduced Instruction set computer
→ Large number of Complex instructions	Small number of instructions.
→ Instructions are of Variable number of bytes	Instructions are of fixed number of bytes
→ Instructions take Varying amounts of time for execution	Instructions take fixed amount of time for execution
→ Large number of Addressing modes, provides flexibility in choosing various ways of performing the data transfer, arithmetic and other operations	→ less number of Addressing modes, provides no flexibility in choosing the many different ways of performing the data transfer, arithmetic and other operations
→ Small amount of Cache and very few registers.	Large cache and large number of registers.

Emphasis on hardware ie, relies more on hardware for instruction functionality ∴ CISC instructions are more complicated



→ Emphasis on software ie, provide greater flexibility and intelligence in software rather than hardware. As a result RISC design places ~~more~~ greater demands on the compiler.



RISC Design philosophy

RISC philosophy is implemented with four major design rules:

1) Instructions: RISC processors have reduced number of instructions: these classes provide simple operations that can each execute in a single cycle.
→ compiler synthesizes complicated operations by combining several simple instructions.

2) Pipeling: the processing of instructions is broken down into smaller units that can be executed in parallel by pipelines.

3) Registers: RISC machines have a large general purpose register set. Any register can contain either data or an address.
→ CISC processors have dedicated registers for specific purposes

4) load-store architecture: The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory.
- This separates memory access from data processing, because memory access is costly

ARM Design philosophy:

~~Instruction set for Embedded Systems:~~

~~ARM~~

- There are a number of physical features that have driven the ARM processor design:
  - ↳ Low power consumption
  - ↳ Limited memory: High code density
  - ↳ Simple Hardware Execution Unit

ARM core is not a pure RISC architecture because of the constraints of its primary application - the embedded system.

Instruction set for Embedded System:

The ARM instruction set differ from the pure RISC definition in several ways. This makes the ARM suitable for embedded applications.

- Variable cycle Execution for certain instructions: Not every ARM instruction executes in a single cycle.
- More Complex instruction: This expands the capability of many instructions to improve the core performance and code density
- Thumb 16-bit instruction set: Second 16 bit instruction set called Thumb, that permits the ARM core to execute either 16 or 32 bit instructions. The 16-bit instructions improve the code density by 30%.
- Conditional Execution: Improves performance and code density by reducing branch.
- Enhanced instructions - The enhanced Digital Signal Processor (DSP) core added to the Standard ARM instruction set to support 16x16 bit multiplier operations.

Embedded System Hardware

Embedded Systems can control many different devices from small sensor to real time control systems.

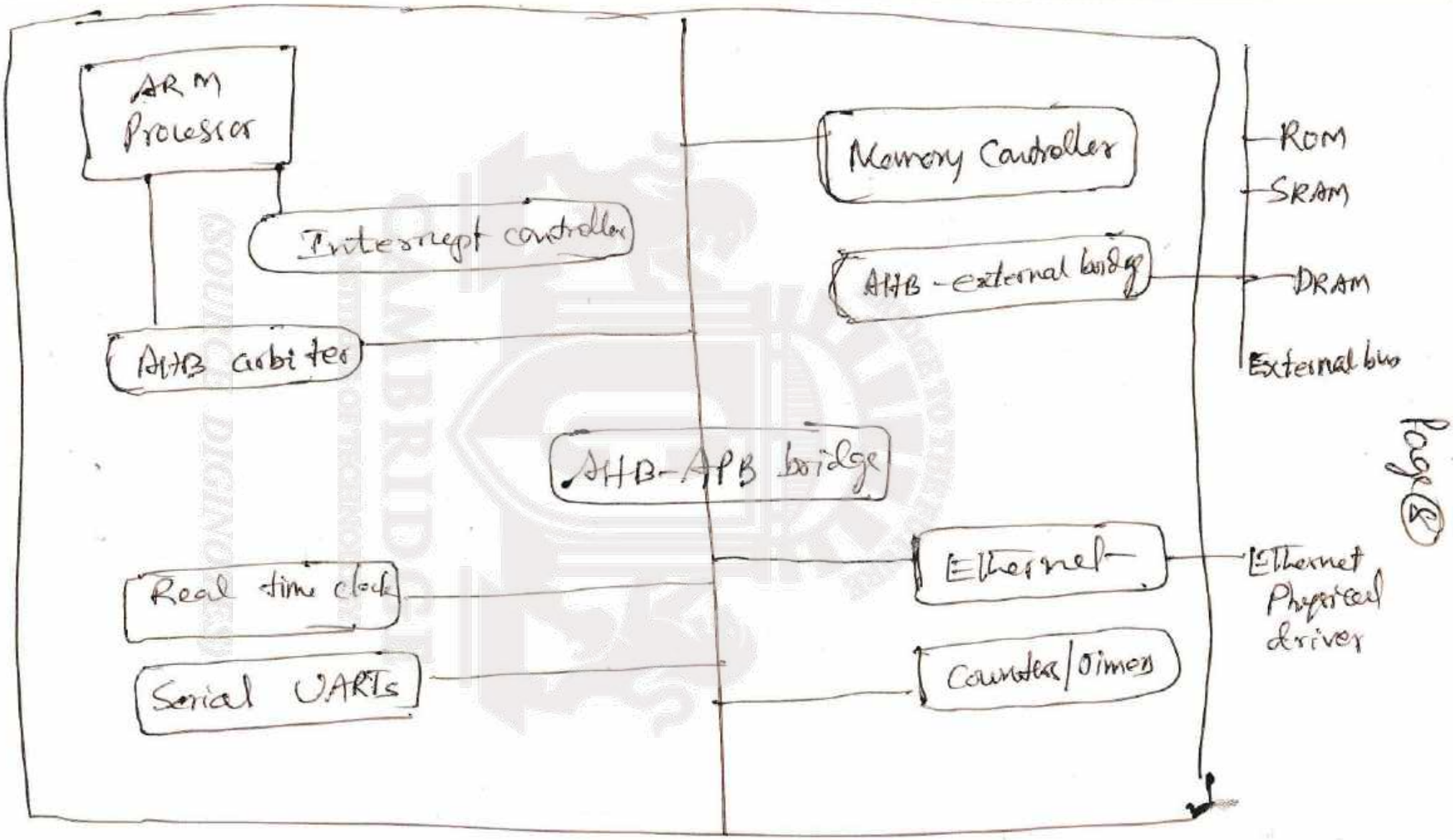
Following fig shows a typical embedded device based as an ARM core.

* Each box represents a feature or function. The lines connecting the boxes are the buses carrying data

We can seperate the device into four main components

- ARM processor: controls the embedded device
- Controllers: coordinate important functional blocks (eg interrupt and memory controller)
- Peripherals: USB, LCD, etc.
- Bus: is used to communicate between different parts of the device.

Sandeep Kumar
Dept of CSE
Cambridge



ARM Bus Technology :

↳ Embedded system uses different bus technologies than those designed for X86 PC

→ Embedded device use an on chip bus

→ Core is a master who initiates data transfer

↳ A bus has two architecture levels

* Physical level that covers the electrical characteristics and bus width (16, 32 or 64 bits)
* Second level deals with protocol — the logical rules governing the communication between processor and peripheral.

AMBA Bus protocol :

AMBA (Advanced Microcontroller Bus Architecture)

↳ 1996, it's introduced and widely adopted as the on chip bus architecture for ARM processors.

↳ The first AMBA buses introduced were
* ASB : ARM System bus and
* APB : ARM peripheral bus.

↳ Later, ARM introduced another bus design
* AHB : ARM high performance bus

Using AMBA

↳ Peripheral designers can reuse the same design on multiple projects (with different processor Architecture)

↳ Plug and play.

AHB provides higher data throughput than ASB. Because

* It use a centralized multiplexed bus scheme.
* This change allows the AHB bus to run at higher clock speed.
* It is 64/128 bit width

MEMORY :

- An Embedded System has to have some form of memory to store and Execute Code.
- We have to consider the price, performance and power consumption of the memory.
- Specific memory characteristics hierarchy, width type

Hierarchy :

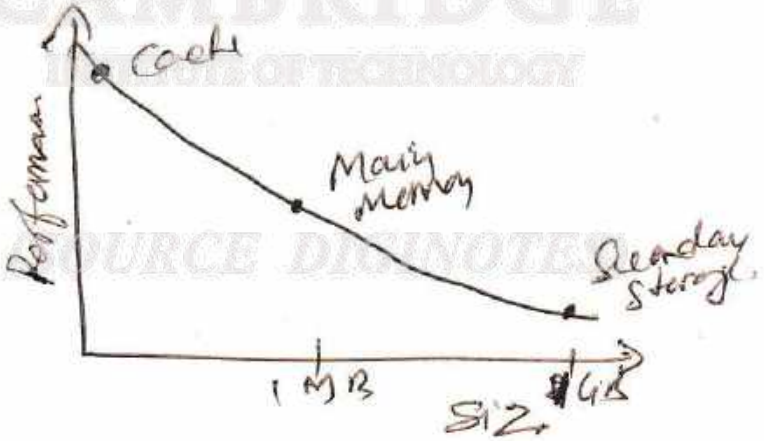
Following fig shows the memory trade-offs

Cache : · used to speed up data transfer between core and Main Memory
- located near the ARM core.
- Provides an overall increase in performance but with a lot of predictable execution time.

Types :

DRAM - The most commonly used RAM for devices.
- Need to have its storage cells refreshed and given a new electronic charge every few milliseconds.

SRAM : faster than DRAM



Width : The memory width is the number of bits the memory returns on each access.
→ Typically 8/16/32 or 64 bits.

Peripherals :

Embedded System that interact with the outside word need some form of peripheral device

↳ Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.

* Controllers are specialized peripherals that implement higher level of functionality within an embedded system.

* Two important types of Controllers are
  → 1) Memory controller
  → 2) Interrupt Controller

Memory Controllers :

Connect different types of memory to the processor bus.

→ On power up a memory controller is configured in hardware to allow certain memory device to be active. These memory devices allow the initialization code to be executed.

→ Some memory devices must be set up by software.

Interrupt Controller :

→ When a peripheral or device requires attention, it raise an interrupt to the processor.

→ An interrupt controller provides a programmable governing policy.

→ There are two types of interrupt controller available for the ARM processor

1) Standard interrupt controller
   - ↳ Sends an interrupt signal: can be programmed to ignore or mask an individual device.
   - ↳ Its interrupt handler determines which device requires service.

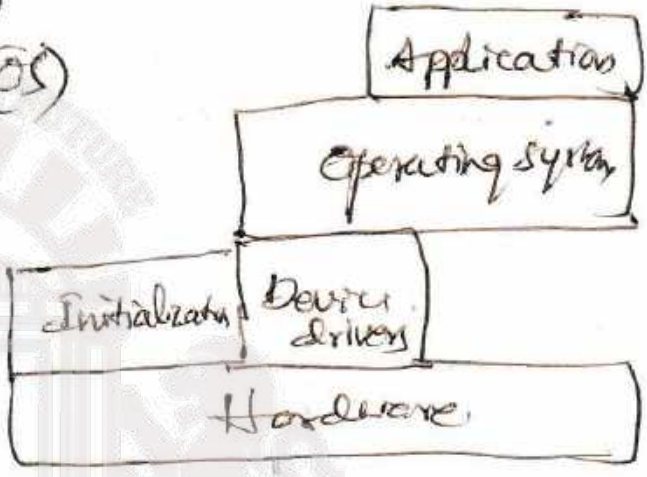
2) Vector Interrupt controller (VIC)
   - ↳ Associate a "priority" and a "handler address" to each interrupt
   - ↳ ~~Depending on its Size~~

Embedded System Software

→ An embedded system needs software to drive it.

→ There are four typical software components required to control an embedded device.

- Initialization code
- Operating system (OS)
- Device drivers
- Application



i) Initialization or Boot code:

→ takes the processor from the reset state to state where the OS can run.

→ It usually configures the memory controller and initializes some devices.

→ Handles a number of administrative tasks prior to handing control over an Operating System.

→ three different tasks
- * Initial hardware configuration
- * Diagnostics — Fault identification & isolation.
- * Booting — loading OS image

2) Operating System (OS)

↳ OS organizes the system resources
  * peripherals
  * Memory
  * Processing time

↳ So that these resources can be efficiently used by different applications.

The main OS ARM supports is ~~RT~~
↳ RTOS
↳ platform OS.

3) Applications:

↳ OS schedules applications
↳ ARM processor are found in the applications include networking, automotive, mobile devices, consumer devices etc.

Chapter 1: ARM Embedded Systems.

Question Bank

1) Explain the difference between microprocessor and microcontrollers
2) Explain the difference between CISC and RISC processors.
3) Explain the design rules on which RISC philosophy is implemented.

4) Explain the ARM Design philosophy (Physical features that have driven the ARM processor design)

5) Explain the embedded system hardware based on a ARM core with a neat diagram

OR

6) Explain the following w.r.to ARM System design
   i) ARM Bus Technology
   ii) AMBA Bus protocol
   iii) MEMOR
   iv) Memory controller
   v) Interrupt controller

7) Explain Embedded System software (Initialization code, operating system and applications)

SANDEEP KUMAR
DEPT. OF CSE
CAMBRIDGE IT

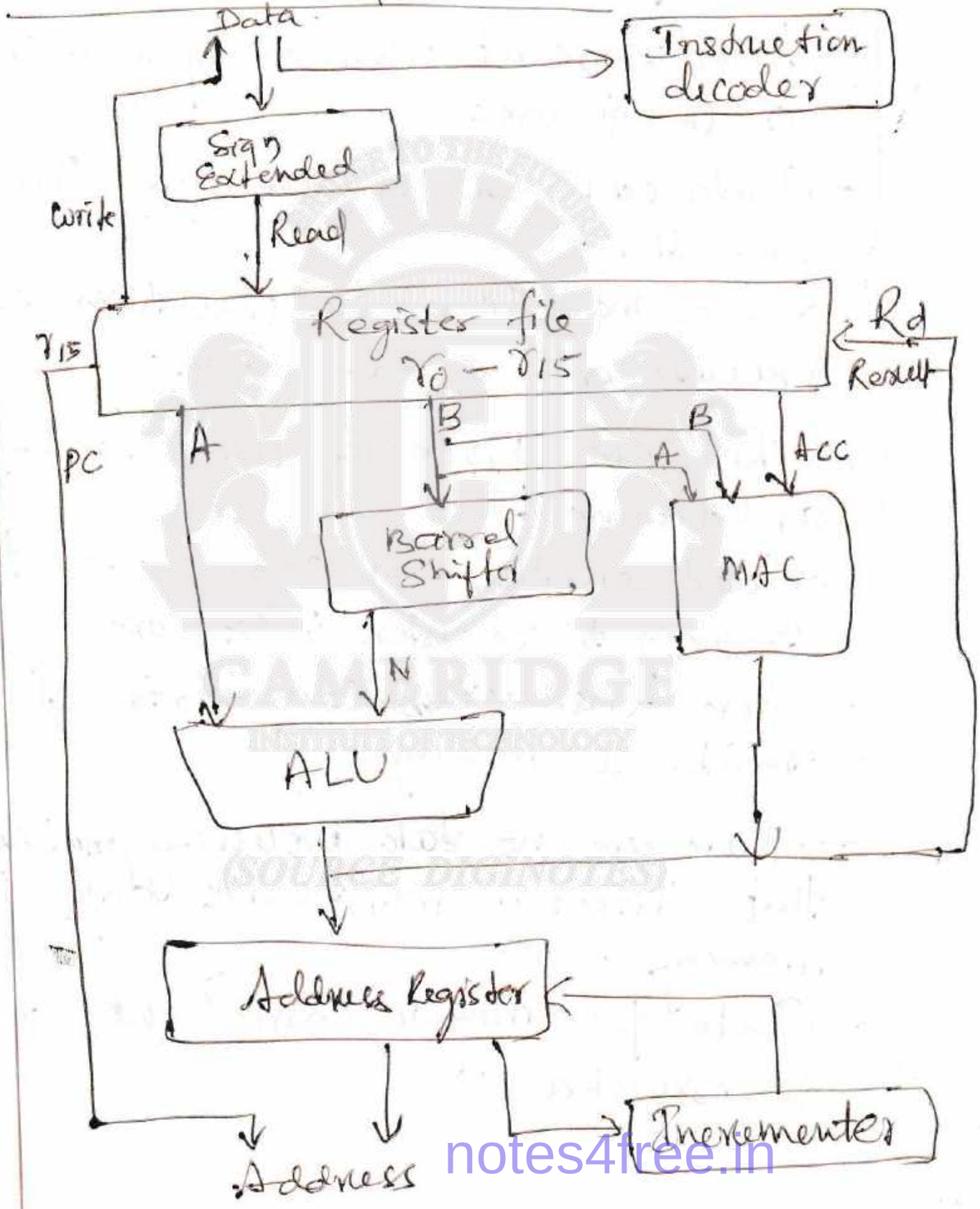
SANDEEP KUMAR
DEPT OF CSE
CAMBRIDGE IT

Module - 4
Chapter - 2

M.M
IV CSE/ISE
Subcode: 15CS45

ARM PROCESSOR Fundamentals

ARM Core dataflow model:



- Fig shows the components that makeup an ARM core -

→ A programmer can think of an ARM core as functional units connected by data buses.

→ Boxes represent either an operation unit or storage area

→ Data enters the processor core through data bus.

→ Data may be an instructions to execute or a data item.

→ the ARM processor uses load-store architecture -

→ load instruction copy data from memory to registers in the core

- Store instructions copy data from registers to memory.

→ There are no data processing instructions that directly manipulate data in memory.

→ Data processing is carried out only in registers

The operational units are

1) Register file :
   - A storage bank made up of 32-bit registers
   - Since the ARM core is a 32 bit processor, most instructions treat the registers are holding signed or unsigned 32 bit values.

2) Sign extended hardware :
   - It converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

3) ARM instructions :
   - Typical have two source registers Rn and Rm and a single destination register Rd.
   - Source operands are read from the registers file using the internal buses A and B respectively. The result in Rd is written back to register file using result bus.

4) The ALU (Arithmetic and Logic Unit) or MAC (Multiply accumulate unit):

↳ takes the register values Rn and Rm from the A and B buses and compute a result.

↳ Data processing instructions write the result in Rd directly to the register file.

↳ Load and Store instructions use the ALU to generate an address to be held in the address register and broadcast on the address bus

5) Barrel shifter:

The register Rm can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate the wide range of Expressions and addresses.

6) Incrementer:

↳ For load and store instructions the incrementer updates the address register before the core reads or write the next register value from or to the next sequential memory location.

Features of ARM Processor

↳ ARM is a 32 bit Architecture

↳ When used in relation to the ARM
- Byte means 8 bits
- Halfword means 16 bits (two bytes)
- Word means 32 bit (four bytes)

↳ Most ARMs implement two instruction sets
- 32 bit ARM instruction set
- 16 bit Thumb instruction set

↳ ARM has seven basic operating modes
1) - User: Unprivileged mode under which most task run
2) - FIQ (Fast Interrupt request): entered when a high priority interrupt is raised
3) - IRQ (Interrupt Request): Entered when low priority interrupt is raised.
4) - Supervisor: Entered on reset and when a software interrupt instruction is executed.

5) Abort: Used to access memory access violations

6) Undef: Used to handle undefined instructions.

7) System: Privileged mode using the same registers as user mode.

↳ ARM has three operand instruction: Two source operand registers and one result register.

Registers of ARM processor

ARM has 37 registers, all 32-bits long.

There are upto 18 active registers:

↳ 16 data registers ($r_0 - r_{15}$)

↳ cspr (Current program status Reg)

↳ SPSR (Saved program Status Register)

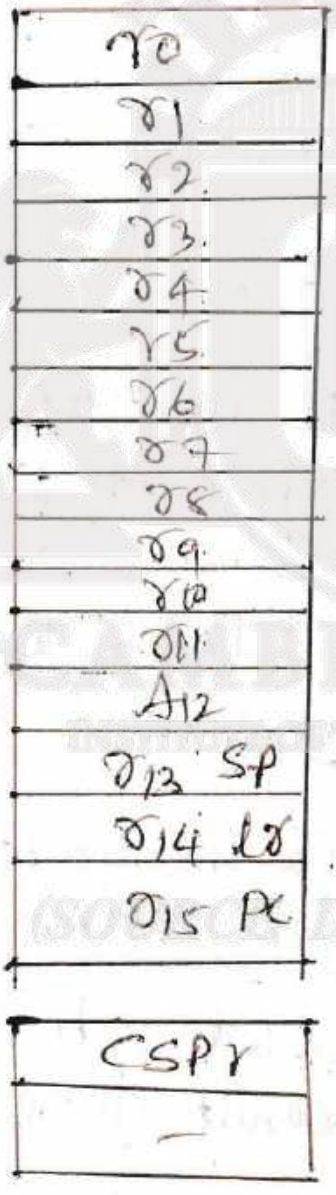
Three registers r_{13}, r_{14}, r_{15} are assigned a particular task.

r_{13} — Stack pointer — store the address of the top of the stack

r_{14} - Link register : Store the return address whenever it calls a subroutine.

r_{15} - Program counter - contains the address of next instruction to be fetched by the processor

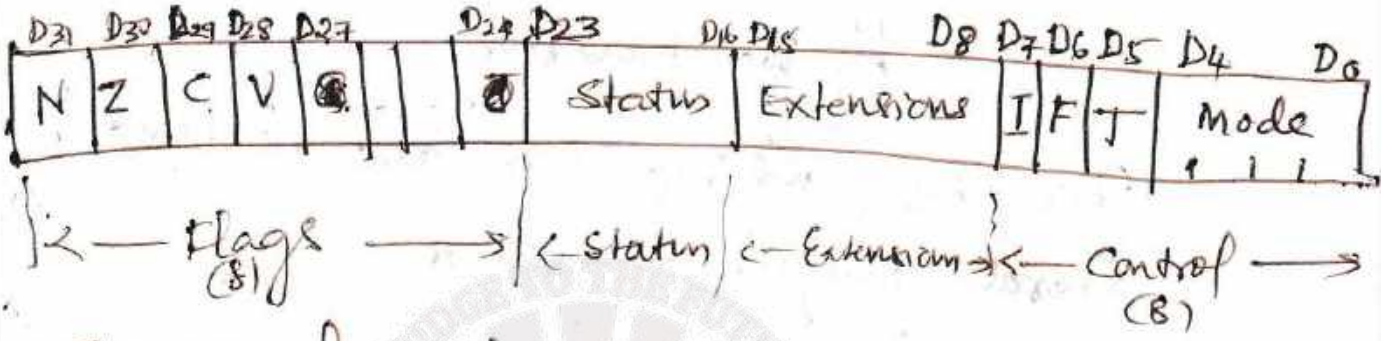
r_{13}, r_{14}, and r_{15} can also be used as general purpose registers



Register available in User mode

Current Program Status Register (CPSR)

- dedicated 32 bit register resides in the register file



CPSR has four fields.

Flags, Status, Extension, Control.

Conditional Code flags:

C (Carry flag): To indicate the ALU operation generated the Carry.

V (Overflow): To indicate the ALU operation overflowed

Z (Zero flag): To indicate the Zero result from ALU

N (Negative): To indicate the negative result from ALU.

Interrupt disable bits

I = 1: Disable IRQ (Interrupt Request)

F = 1; Disable FIQ (Fast interrupt)

T – bit T = 0: Processor in ARM state
         T = 1: Processor in Thumb state

Mode bits : Specify the processor mode

Bank Registers:

- Following fig shows all 37 registers in the register file
- of those, 20 registers are hidden from a program at different times. These registers are called banked registers.
- They are available only when the processor is in a particular mode

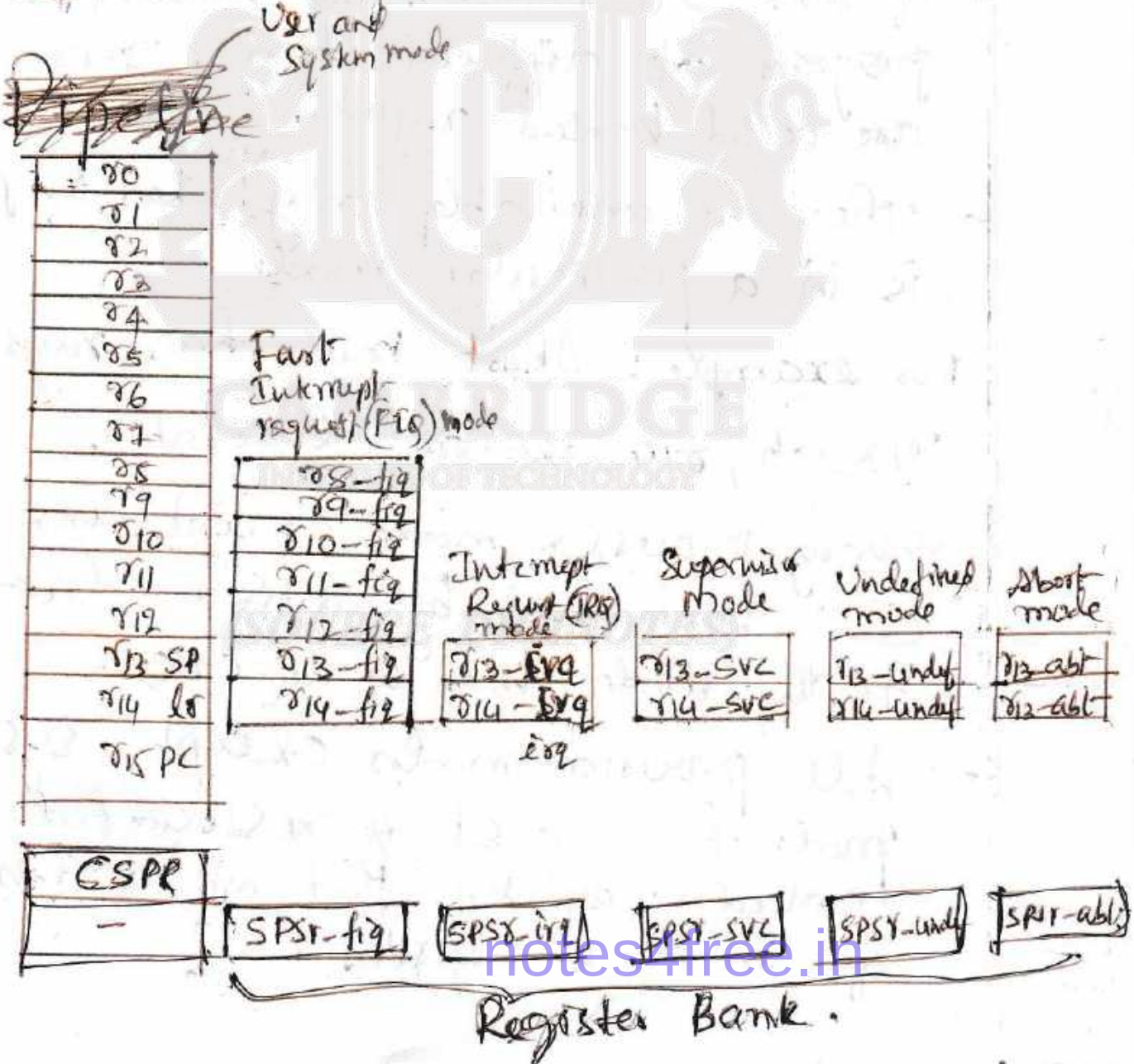
For example: Abort mode has registers r13_abt, r14_abt and spsr_abt.

→ Every processor mode except user mode can change mode by writing directly to the mode bits of the CPSR.

→ All privileged modes except system mode have a set of associated banked registers that are a subset of main 16 registers.

- If you change the processor mode, the banked registers from the new mode will replace an existing register.
  (Maps one to one to User mode register)

Example - When the processor is in the Interrupt request mode, the instructions you execute still uses registers r_{13} and r_{14}. However, then registers are the banked registers r_{13}_irq and r_{14}-irq, User mode registers r_{13} and r_{14} not affected.



PIPELINE

↳ ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor

↳ Allows several operations to be undertaken simultaneously, rather than serially.

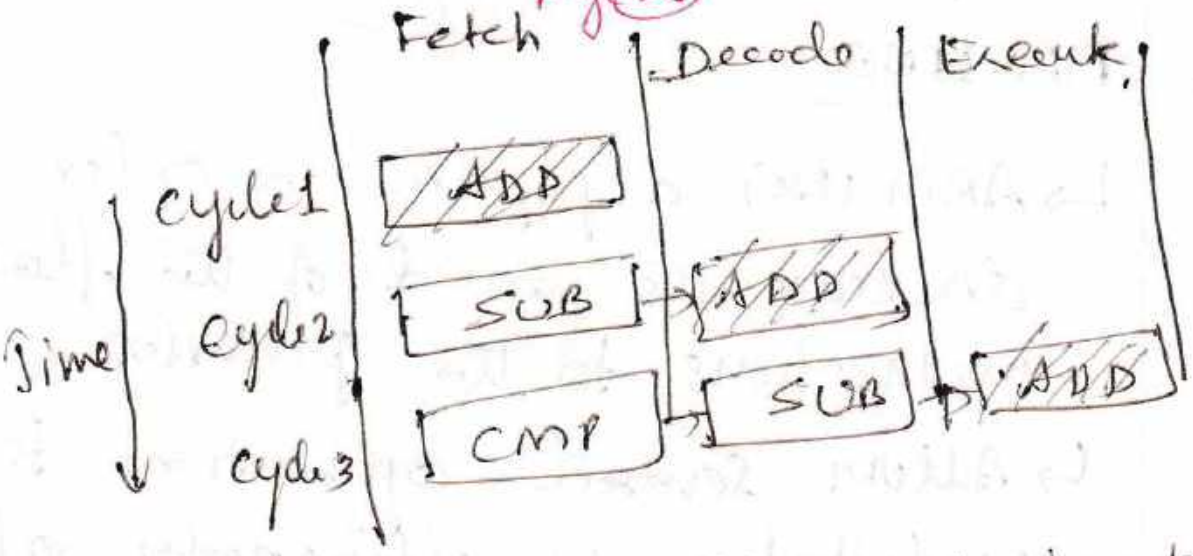
Following fig shows the three stage pipeline

→ Fetch loads an instruction from memory

→ Decode identifies the instruction to be executed.

→ Execute processes the instruction and writes the result back to a register

- Following fig shows the pipeline using a simple example.
- It shows the three instructions (ADD, SUB & CMP) being fetched, decoded and executed by the processor



In cycle 1, core fetches ADD instruction from memory

In cycle 2, the core fetches the SUB instruction and decodes the ADD instruction

In cycle 3, ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.

This procedure is called filling the pipeline

(2)

Exceptions, Interrupts, and the Vector Table.

- When an exception or interrupt occurs, the processor sets the PC to a specific memory address. The address is a special address range called the vector table.
- The entries in the vector table are instructions that branch to specific routines (Interrupt Service Routine) designed to handle interrupt.
- The memory map address 0x0000 0000 is reserved for the vector table.
- When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table.
  - → Each vector table entry contains a form of branch instructions pointing to the start of a specific routine
- When an exception occurs, the ARM
  - → copies CSPR into SPSR_<mode>
  - → sets appropriate CSPR bits
    - change ARM state
    - → change to Exception mode

- Disable interrupts.

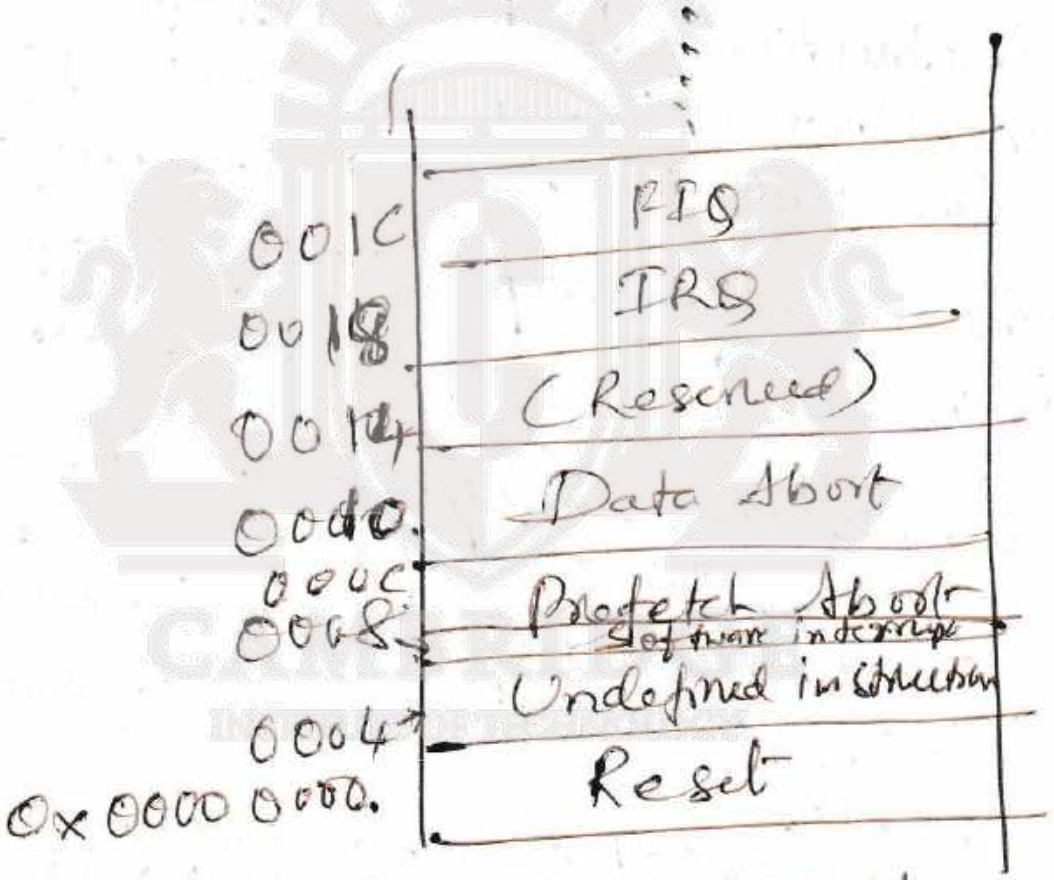
→ Store the return address in link register

→ Set PC to Vector address (ie Subroutine address)

To Return, exception handler needs to

↳ Restore CSPR from SPSR_<mode>

↳ Restore PC from Link register



Vector Table

Reset Vector: location of the first instruction executed by the processor when power is ON.

→ Undefined instruction vector : is used when the processor cannot decode an instruction.

→ Software interrupt vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

→ Prefetch Abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions.

→ Data Abort vector is raised when an instruction attempts to access data memory without the correct access permissions.

→ Interrupt Request Vector (IRQ) is used by external hardware to interrupt the normal execution flow of the processor.

→ Fast Interrupt request vector is similar to the interrupt request but it is reserved for hardware requiring fast response times.

Core Extensions of ARM

→ There are three hardware extensions around core, placed next to the ARM core.

→ They improve performance, manage resources, and provide extra functionality.

→ Three hardware extensions are
- 1) Cache and tightly coupled Memory
- 2) Memory management
- 3) Coprocessor interface.

1) Cache and tightly coupled Memory:

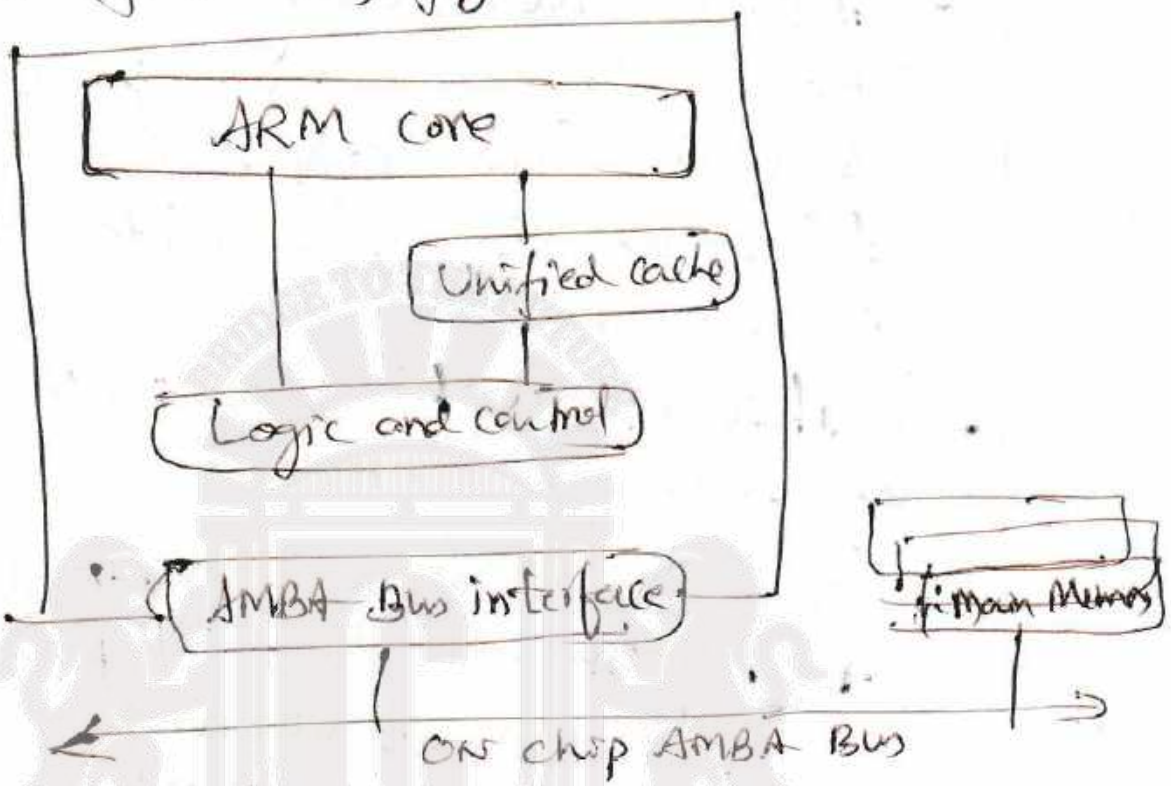
→ Cache is a block of fast memory placed between main memory and the core.

→ With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.

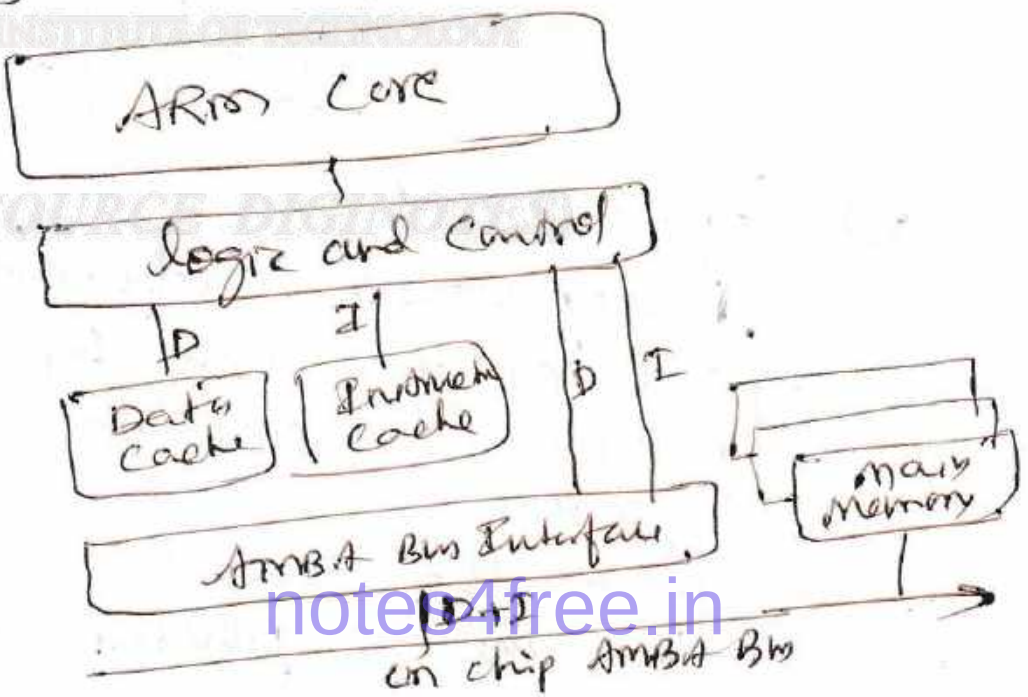
→ Most ARM based embedded system uses a single level cache internal to the processor

→ ARM has two forms of cache;

→ First type combines both data and instruction into a unified cache shown in the following fig



→ Second type, has seperate caches for data and instructions shown in the fig.



2) Memory Management:

- Embedded systems use multiple memory devices
- → It is necessary to organize these devices and protect the system from applications trying to make inappropriate accesses to hardware
- → This is achieved with the assistance of memory management devices/hardware.
- → ARM core has three types of memory management hardware
  - → No extension providing no protection
  - → Memory protection unit providing limited protection
  - → Memory management unit providing full protection.

3) Coprocessors:

- A coprocessor extends the processing features of a core by extending the instruction set or providing configuration registers.
- → More than one processor can be added via the co-processor interface

→ The coprocessor can be accessed through a group of dedicated ARM instructions that provide load-store type interface.

→ The coprocessor can also extend the instruction set by providing a specialized group of new instructions.

Chapter 2 : Question Bank

1) Explain with a neat diagram, ARM Core data flow model
2) Explain the seven basic operating modes of ARM.
3) Explain with a neat diagram, Registers of ARM processor.
4) Explain Current-Program Status Register (CPSR) of ARM / Explain Conditional code flags.
5) Explain with neat diagram, the Register Bank of ARM core
6) Explain the pipeline execution of ARM instructions with an example
7) Draw, Explain the Interrupt Vector Table of ARM core and also explain the steps taken by the ARM core when an exception/Interrupt occurs.
8) Explain the Core extensions of ARM with diagrams.

Data processing instructions — ARM Instruction set

- MOVE Instructions
- Arithmetic instructions
- Comparision Instructions.
- Logical Instructions.
- Multiply Instructions

Conditional Execution

→ ARM instruction can be made to execute conditionally by post fixing them with the appropriate conditional code field.

→ This improves code density and performance by reducing the no of forward instructions.

ex

ADD r_0, r_1, r_2 ; $r_0 = r_1 + r_2$

Same ex can be written as

ADDEQ, r_0, r_1, r_2 ; If Z = 1, then perform $r_0 = r_1 + r_2$

The following table shows the possible conditional codes with postfix

Postfix	Description	Flag tested.
EQ	Equal	Z = 1
NE	NOT Equal	Z = 0.
CS/HS (CMP)	Unsigned Higher or Same	C = 1
CC/LO	Unsigned Lower	C = 0
MI	Minus	N = 1
PL	positive or zero	N = 0
VS	Overflow	V = 1
VC	No - overflow	V = 0
HI	Unsigned Higher	C = 1 & Z = 0
LS	Unsigned Lower or same	C = 0 or Z = 1
GE	Greater or Equal	N = V
LT	Less than	N != V

GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V

Note

→ By default data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S" postfix

→ But, By default CMP instruction will affect the flag after the execution.

Ex ADD $r5, r6, r10$; $r5 = r6 + r10$ do not affect the flag after the execution.

ADDS $r5, r6, r10$; $r5 = r6 + r10$ Do affect the flag after the execution.

Move instructions < MOV, MVN

Syntax :- <Mnemonic>{<condition>}{S}.R_d, N

N → is eigther a register or an immediate data

Ex

(i) MOV R_5, R_8 ; $R_5 \leftarrow R_8$

(ii) MOV R_5, #23; $R_5 \leftarrow$ #23

MOV :- Move a 32-bit data into a register

MVN :- Move a NOT 32 bit value into a Register

(iii) MVN R_5 #23; 23 = 10111

23 → 00000000000000000000000000010111

1's comp 23 → 11111111111111111111111111101000

F F F F E 8

$R_5 \leftarrow$ 0xFFFFFFE8

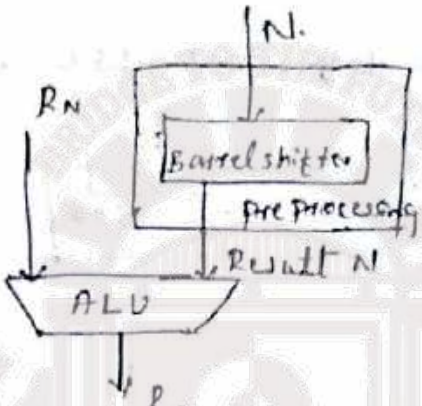
Barrel shifter

The ARM doesnot have actual shift instruction. Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.

Data processing instructions are processed within ALU.

ARM processor shifts the 32-bit binary pattern in one of the source registers left or right by a specific no of positions before it enters ALU.

This shift increases power & flexibility of many data processing operations.



Eg:- MOV Rd, RN, N [without barrel shift]

MOV Rd, RN LSL #5 [with barrel shift]
N

Before execution

$R_7 = 5$

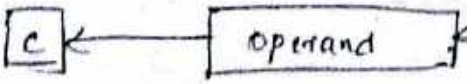
$R_8 = 8$

After execution : $R_8 = 8$

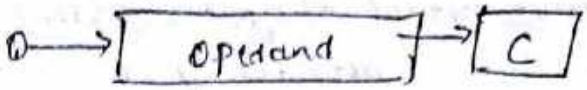
$R_7 = 5 \times 4 = 32$

The following shift & Rotate instructions supported in Barrel shifter

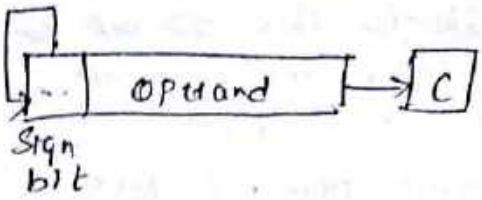
1) LSL : Logical shift left [multiply the power of two]



2) LSR :- Logical shift right [Divides by power of two]



3. Arithmetic shift Right : ASR [Divide by powers of 2]
& reserves sign bit.



4. Rotate right :- ROR

Similar to ASR but LSB is shifted to MSB

9/5/17

Arithmetic Instructions

Syntax : <Mnemonic>{<Condition>}{S} R_d, R_N, N

N → can be an Immediate data, Register or barrel shifted reg. data

Mnemonic	Description	Operation
1. ADD	Add two 32-bit vertices	$R_d = R_N + N$
ADC	Add two 32-bit values with carry flag	$R_d = R_N + N +$ carry flag
SUB	Substract two 32-bit value	$R_d = R_N - N$
SBC	Substract with carry two 32-bit value.	$R_d = R_N - N -$ carry flag
RSB	Reverse Substract two 32-bit value	$R_d = N - R_N$
RSC	Reserve Subtract with carry two 32-bit value	$R_d = N - R_N -$ carry flag.

N → can be an Immediate data, Register, or a barrel shifted data.

Ex

i) ADD r_1, r_4, r_5 ; $r_1 = r_4 + r_5$. } Flags are not updated

ii) ADDEQ r_1, r_4, r_5 ; if $r_4 = r_5$, ie $Z = 1$ } Flags are not updated
$r_1 = r_4 + r_5$

iii) ADDS r_1, r_4, r_5; $r_1 = r_4 + r_5$; conditional flags are updated.

iv) SUB $r_1, r_4, \#5$; $r_1 = r_4 - 5$

v) SBC r_1, r_4, r_5 ; $r_1 = r_4 - r_5 -$ carry flag.

vi) RSB r_1, r_4, r_6 // $r_1 = r_6 - r_4$

vii) RSC r_1, r_4, r_6 // $r_1 = r_6 - r_4 -$ carry flag.

8. PRE [Before Execution]

: r_0 = 0x00000000
r_1 = 0x0000 0002
r_2 = 0x0000 0001

SUB r_0, r_1, r_2

$r_0 = r_1 - r_2$

Post POST [After Execution]

: r_0 = 0x0000 0001
r_1 = 0x0000 0002
r_2 = 0x0000 0001 & flags are not updated

9. PRE r_1 = 0x0000 0001

SUBS $r_1, r_1, \#1$

$r_1 = r_1 - 1$

r_1 = 0x0000 0001
1 - 0x0000 0001
Post r_1 = 0x0000 0000 } flags are modified

flags : Z = 1, C = 0, N = 0, V = 0.

Using Barrel shifter with ~~shifter~~ Arithmetic Instruction

PRE: r_0 = 0x0000 0000
r_1 = 0x0000 0005

ADD $r_0, r_1, (r_1)$, LSL #1 Logical shift Left by 0f r_1 by 1

Barrel shifted data.

r_1 = 0x0000 0005
r_1 = 0000 0000 0000 0101
A r_1, LSL #1 0000 0000 0000 1010

$\therefore r_0 = r_1 + r_1, LSL\ \#1$

$1010 = r_1, LSL\#1$

$0101 = r_1$

1111

F

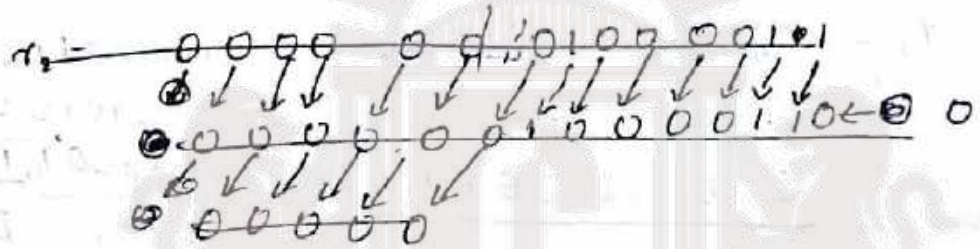
$\therefore r_0 = 0x0000\ 000F$

2. $r_1 = 0x0000\ 0008$.

$r_2 = 0x0000\ 0043$

ADD $r_0, r_1, r_2, LSL\ \#2$.

i) $r_2\ LSL\ \#2$



$r_2 = \ldots 0\ 4\ 3$

0000 0100 0011

0000 1000 0100.

$R_0 r_2 = 0000\ 0000\ 1100$ after 2 shift

1. 0 C.

$\therefore r_2 = 0x0000\ 010C$

$r_0 = r_1 + r_2, LSL\ \#2$

$r_1 = 0x0000\ 0008$

$r_2, LSL\#2 = \dfrac{0x0000\ 010C}{0x0000\ 0011}$ 4

$\therefore r_0 = 0x0000\ 0114$

3. $r_4 = 0x0000\ 0431$

$r_5 = 0x0000\ 0290$

SUB r_2, r_4, r_5, LSR #2

i) r_5, LSR #2.

$r_5 = \ldots\ldots 0290$

0000 0010 1001 0000

0000, 0000, 1010, 0100

0 0 A 4.

$r_5 = 0x0000\ 00A4$.

$\therefore$ $r_2 = r_4 - r_5$, LSR #2.

$0x0000\ 0431$

$0x0000\ 00A4$

$r_2 = 0x0000\ 059D$

0100 0011 0001

0000 1010 0100

0100 1001 1101

5 9 D.

X

Ans = 38D

10/5/17

Logical Instructions.

Syntax: <Mnemonic>{<condition>}{S} (S: flag updation), R_d, R_N, N

N → can be immediate data, or a barrel shifted data.

	Description.	Operation
AND	Logical bitwise AND of two 32-bit value	$R_d = R_N$ Bitwise AND. N
ORR	Logical bitwise OR of two 32-bit value.	$R_d = R_N$ Bitwise OR N
EOR	Logical bitwise XOR of two 32 bit values	$R_d = R_N$ Bitwise XOR N

BIC logical bit clear. (AND NOT)

$R_d = R_N$ AND NOT R_S

Right to Left Pattern

Ex 1.

PRE: $r_3 = 0x0000\ 0000$

$r_4 = 0x1020\ 3040$

$r_5 = 0x0307\ 0509$

ORR r_3, r_4, r_5 // $r_3 = r_4$ Bitwise r_5.

AND OR

& flags are not updated.

r_4 0001 0000 0010 0000 0011 0000 0100 0000.

r_5 0000 0011 0000 0111 0000 0101 0000 1001

0001 0011 0010 0111 0011 0101 0100 1001

1 3 2 7 3 5 4 9.

$\therefore r_3 = 0x1327\ 3549$.

Ex 2.

PRE: $r_3 = 0x0000\ 0000$

$r_4 = 0x1020\ 3040$.

$r_5 = 0x0307\ 0509$.

ANDS r_3, r_4, r_5 // $r_3 = r_4$ Bitwise r_5.

AND

updated.

$r_3 = 0x0000\ 0000$

$Z = 1$, $C = 0$, $V = 0$, $S = 0$

Ex 3

PRE: $r_3 = 0x0000\ 0000$.

$r_4 = 0x1020\ 3040$

$r_5 = 0x0307\ 0509$.

BIC r_3, r_4, r_5 // $r_3 = r_4$ AND [NOT r_5]

r_5 = 8

	0000	0011	0000	0111	0000	0101	0000	100							
NOT r_5 =	1111	1100	1111	1000	1111	1010	1111	0110							
r_4 AND	0001	0000	0010	0000	0011	0000	0100	0000							
	0001	0000	0010	0000	0011	0000	0100	0000							
	1	0	2	0	3	0	4	0							

r_3 = 0x10203040

4. r_1 = 0b1011. } binary value
r_2 = 0b0110 }

EOR r_0, r_1, r_2 // r_0 = r_1 XOR r_2.

r_1 = 1011
r_2 = 0110
r_0 = 1101

5. r_4 = 0x00014538.
r_5 = 0x00023409

AND r_0 r_4, r_5, LSL #1

r_5 = 0000 0010 0011 0100 0000 1001

r_5,LSL #1 : 0000 0100 0110 1000 0001 0010.

r_4 : 0000 0001 0100 0101 0011 1000

AND : 0000 0000 0100 0000 0001 0000

∴ r_0 = 0x00004010.

AND r_0 = r_4 AND r_5 LSL #1.

Comparision Instruction

Syntax : <Mnemonic>{<condition>}{S} R_N, N

→ N : Immediate data or a Register or a Barrel shifted da

→ By default flags are updated, S postfix not applicable //

	Conclusion	Operation
CMP	$R_N - N$ Result not written but flags are updated	$R_N - N$
CMN (compare negative)		$R_N + N$
TEQ	R_N Bitwise Result not written but flags are updated.	R_N Bitwise OR N
TST	"	R_N Bitwise AND N

Ex $r_1 = 0x5$

$r_4 = 0x5$

CMP r_1, r_4 // $r_1 - r_4$

$0x5 - 0x5$ $Z=1$

12/5/17

Multiply Instructions

→ multiply the contents of a pair of registers and accumulate the result in another register.

→ The long multiply accumulate the result in a pair of registers representing a 64-bit value.

→ The final result is placed in a destination register or a pair of registers.

1) Small multiply [32-bit product]

Syntax :- MUL{<condition>}{S} R_d, R_M, R_S

MLA{<condition>}{S} R_d, R_m, R_s, R_N

Eg	Description	Operation
MUL	multiply.	$R_d = R_M * R_s$
MLA	multiply and accumulate.	$R_d = R_m \times R_s + R_n$

Ex: PRE · $r_1 = 0x0000\ 0008$ ·
$r_4 = 0x0000\ 0003$.
MUL · $r_5\ r_1\ r_4$ // · $r_5 = r_1 \times r_4$.

$8 \times 3 = (24)_{10}$

16 | 24 | 8
1 | | = (18)H

2 | 2 | 1
1 | | 0

$r_1 = 0x0000\ 0008$
$r_4 = 0x0000\ 0003$
$\therefore r_5\ 0x0000\ 0018$

2. Long multiply [64-bit product]

i) Syntax:- ~~UMULL~~

UMULL {<Condition>} {S} $R_{dLO}, R_{dHI}, R_M, R_S$.
↑ Unsigned ↓ Long

ii) UMLAL {<Condition>} {S} $R_{dLO}, R_{dHI}, R_M, R_S$

Eg	Description.	Operation
UMULL	long multiply	$(R_{dHI}, R_{dLO}) = R_M \times R_S$.
UMLAL	long multiply & accumulate	$(R_{dHI}, R_{dLO}) = (R_M \times R_S) + (R_{dHI}, R_{dLO})$

Ex PRE: $r_1 = 0x0000\ 0000$
$r_4 = 0x0000\ 0001$
$r_5 = 0x0000\ 0003$
$r_6 = 0x0000\ 0002$

UMULL r_1, r_4, r_5, r_6.

R_{dLO} R_{dHI} R_m R_s.

$(R_{dHI}, R_{dLO}) = (R_m \times R_s)$

$(r_4, r_1) = (r_5 \times r_6)$

POST : $\therefore r_4 = 0x0000\ 0000$ } 64-bit product
$r_1 = 0x0000\ 0006$ }
$r_5 = 0x0000\ 0003$
$r_6 = 0x0000\ 0002$.

B BRANCH INTRUCTION

It changes the flow of Execution (or) used to call a SubRoutine

↳ sub routine is not called using call instruction in micro controller.

Syntax

(i) B{<cond>} label

(ii) BL{<cond>} Label

(iii) BX{<cond>} Rm

iv) BLX{<cond>} Label/Rm

Mnemonic	Description	Operation
B	Branch	PC = Label
BL	Branch with link [used for subroutine call]	PC = Label Lr = address of the next instruction after BL

BX	Branch with Exchange	PC = Rm.
BLX	Branch Exchange with Link.	PC = Rm LR = address of the next Ins^{n} after BLX ie LR = PC PC = Rm/Label

Ex:- B SKIP ← Label
ADD r_0, r_2, r_3
SUB $r_3, r_0, \#3$
SKIP: ADC $r_4, r_5, \#1$

} Forward jump [Branch] skip 2 ins^{n}

Ex BL DELAY ← subroutine name

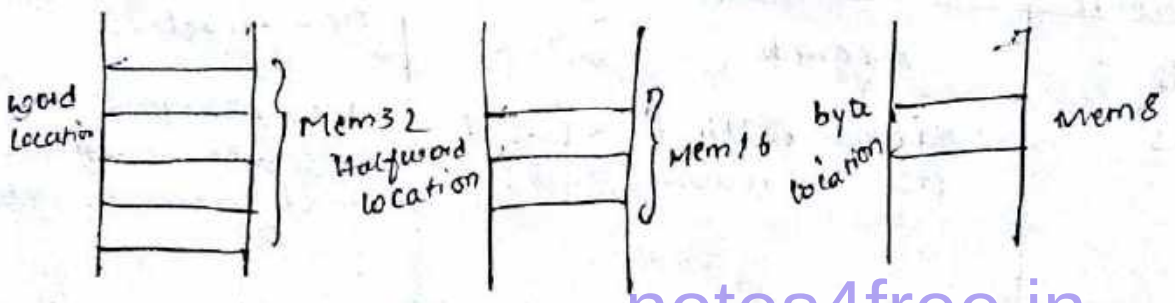
DELAY } Subroutine instruction.
MOV PC LR.

12/5/17

LOAD STORE Instruction

LOAD :- Memory to processor register
STORE :- Processor register to Memory

We can move
Byte - 1 Byte (8-bits)
Halfword - 2 Bytes (16-bits)
Word - 4 Bytes (32-bits)



LOAD operation:

Syntax: LDR / LDRH / LDRB / LDRSB / LDRSH . R_d, addressing

- LDR → word
- LDRH → Half word
- LDRB → Byte
- LDRSB → signed Byte
- LDRSH → signed Half word.

	Description	Operation
LDR.	load word into a register	$R_d \leftarrow$ mem 32 [addr]
LDRH	Load half word into a register	$R_d \leftarrow$ Mem 16 [addr]
LDRB	Load Byte into a register	$R_d \leftarrow$ Mem 8 [addr]
LDRSB	Load signed Byte into a register	$R_d \leftarrow$ Sign Extended (mem 8 [addr])
LDRSH	Load signed half word into a register	$R_d \leftarrow$ sign Extended (mem 16 [addr])

Example

PRE:- $r_0 = 0x0000\ 0000$

$r_1 = 0x0FFF\ EEEE$

LDR r_0, [r_1]

POST:-

12	0x0FFF EEEE
13	0x0FFF EEEF
14	0x0FFF EEF0
15	0x0FFF EEF1

$r_0 \leftarrow [r_1]$

$r_0 \leftarrow [0x0FFF\ EEEE]$

$r_0 \leftarrow$ $51413 12

POST: $r_0 = 0x51413 12$

STORE operation

Syntax

STR / STRH / STRB . Rd, addressing .

P.T.O

Mnemonic	Description	Operation
STR	Store/save a word from register specified in the instruction to memory	Mem32[addr] ← R
STRH	Store/save a Half word from register specified to memory.	Mem16[addr] ← R
STRB	Store/save a Byte from register specified to memory	Mem8[addr] ← R

Ex

PRE :- $r_0 = 0x0000\ 0000$

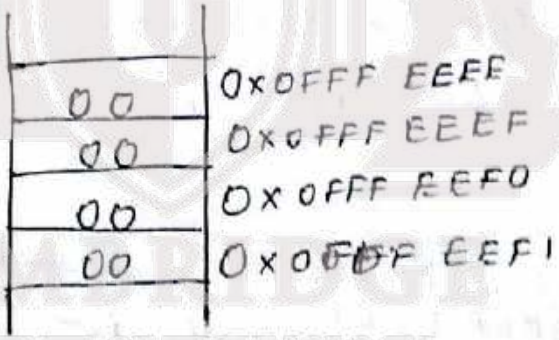
$r_1 = 0x0FFF\ EEEE$

STR $r_0, [r_1]$

POST :- $[r_1] \leftarrow r_0$

$[0x0FFF\ EEEE] \leftarrow r_0$

$[0x0FFF\ EEEE] \leftarrow 0x0000\ 0000$

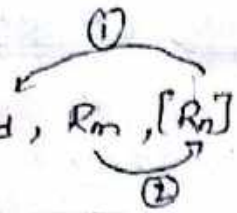


SWAP Instruction

It swaps the content of memory with the contents of a Register

syntax :-

SWP / SWPB $R_d, R_m, [R_n]$



	Description.	Operation.
SWP	Swap a word between memory and a register.	$R_d \leftarrow mem32[R_n]$ $mem32[R_n] \leftarrow R_m$
SWPB	Swap a Byte between memory & a register	$R_d \leftarrow mem8[R_n]$ $mem8[R_n] \leftarrow R_m$

8

Assignment Submission 16/5/17 8:30 AM.

1. Write the Data flow model of ARM processors & explain.
2. Write the Register file of ARM based processors and Explain the purpose of each register.
3. Draw CSPR of ARM-based processors & Explain Each ~~bit~~ purpose of each bits.
4. Assume the content of the registers before execution

PRE $r_0 = 0x12403400$

$r_1 = 0x01123435$

$r_2 = 0x01125345$

$r_3 = 0x0001F3BC$

$r_4 = 0x0005A8CDF$

Show the contents of the registers after the execution of each of the following instructions.

1) MOV R0, R1, LSR, #3

2) ADD R3, R4, LSL #2.

3) ADDS R3, R4, R2.

4) ADC R4, R1, LSR #1

4) SUB r0, r1, r3 LSR #3

5) EOR r0, r1, LSL #1

ARM

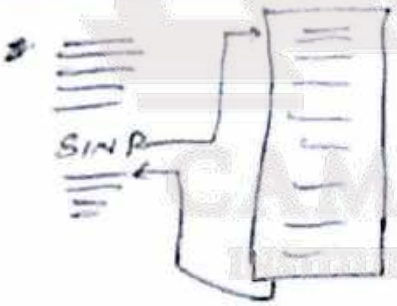
Software interrupt instructions

It causes a software Interrupt exception, which provides a mechanism for Applications to call an OS subroutines

Syntax

SWI{<cond>} Interrupt number

	description	Operation
SWI	Software Interrupt	lr_svc = address of the Instruction follow SWI Spsr_svc = CPSR PC = Vectors + 0x8 In CSPR make I = 1 (Disable IRQ Interrupt)
	SVC is the name of the register in a mode.	



PROGRAM Status Register Instructions

It Transfers the content of CPSR/SPSR to Register. Register to CPSR/SPSR.

There are two instructions.

MRS	CPSR/SPSR to Register	Rd ← SPSR/CPSR
MSR	Register to CPSR/SPSR	CPSR/SPSR ← Rd

Syntax :-

MRS {<cond>} Rd, CPSR / SPSR.

MSR {<cond>} CPSR / SPSR , Rd.

LOADING CONSTANTS

used to move a 32-bit constant value into a Register

syntax :- LDR Rd, = constant.

ADR Rd, Label.

LDR	Load constant numbers	Rd = 32-bit constant.
ADR	Load address	Rd = 32-bit relative address

Ex:- LDR R1, = Num.

ADR R3, [PC, = Next].