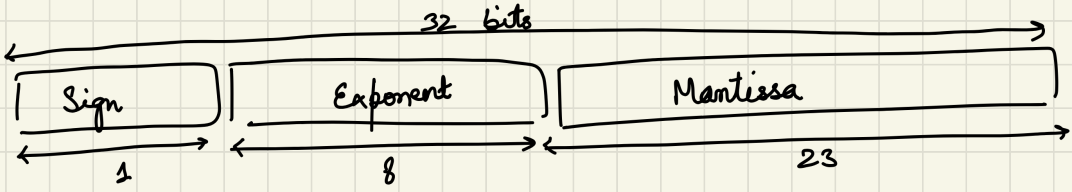
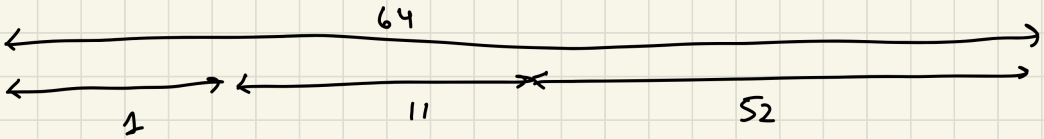
CAO

* Map → XOR, AND, Logic gates

- sign of mantissa : 0 → +
  1 → −





Lec 2-4 :

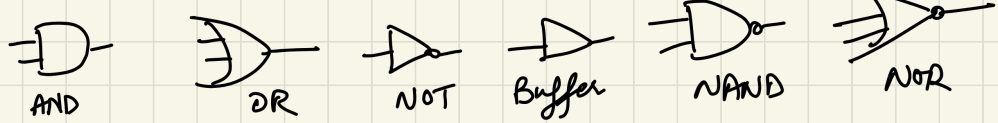
- Comp. Architecture — user's point of view (external view of comp.)

  Comp. Organisation — internal view of computer & roles various internal components play during program execution.

  Comp. design — Hardware design of computer

- Logic gates :

i)

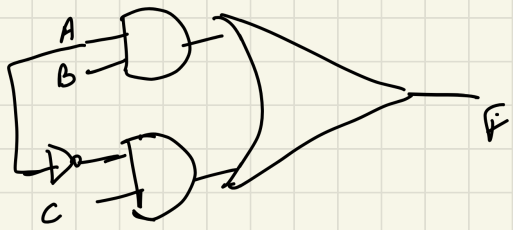


XOR = $A \oplus B = A'B + AB'$

XNOR = $(A \oplus B)' = A'B' + AB$

eg: $AB + A'C = F$

Soln

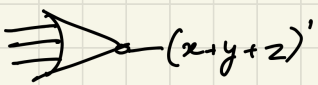


- Identities:

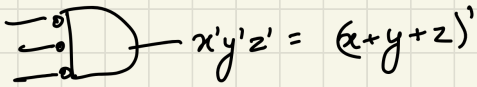
$(x+y)' = x'y'$

$x + yz = (x+y)(x+z)$

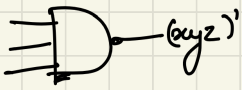
$(xy)' = x' + y'$



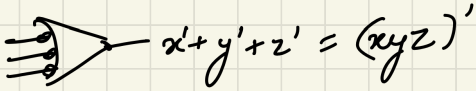
OR - Invert



Invert - AND



AND - Invert



- K-Map :

Minterm : for 1 output

A	B	C	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

A \ B,C	00	01	11	10
0	0	1	0	0
1	1	1	1	1

$f(A,B,C) = \Sigma(1,4,5,6,7)$

K-Maps :

2 variables

A \ B	0	1
0	0	1
1	2	3

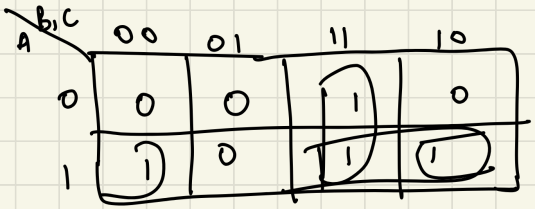
3 variables

A \ B,C	00	01	11	10
0	0	1	3	2
1	4	5	7	6

4 variables

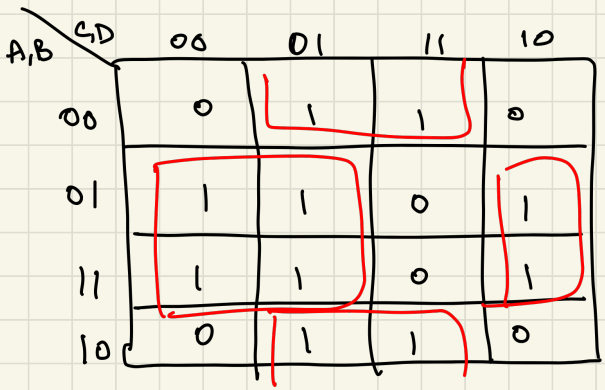
A,B \ C,D	00	01	10	11
00	0	1	3	2
01	4	5	7	6
10	12	13	15	14
11	8	9	11	10

eg.



$F = BC + AC' + AB$

eg. $F(A,B,C,D) = \Sigma(1, 3, 4, 5, 6, 9, 11, 12, 13, 14)$



sum of product = minterms

- Maxterms : (Product of sum - POS)

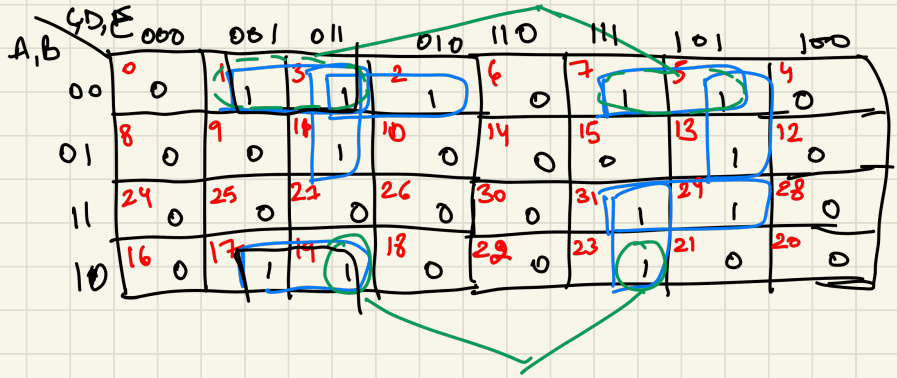
- for 0 output

- 5 variable K-map :

A=0

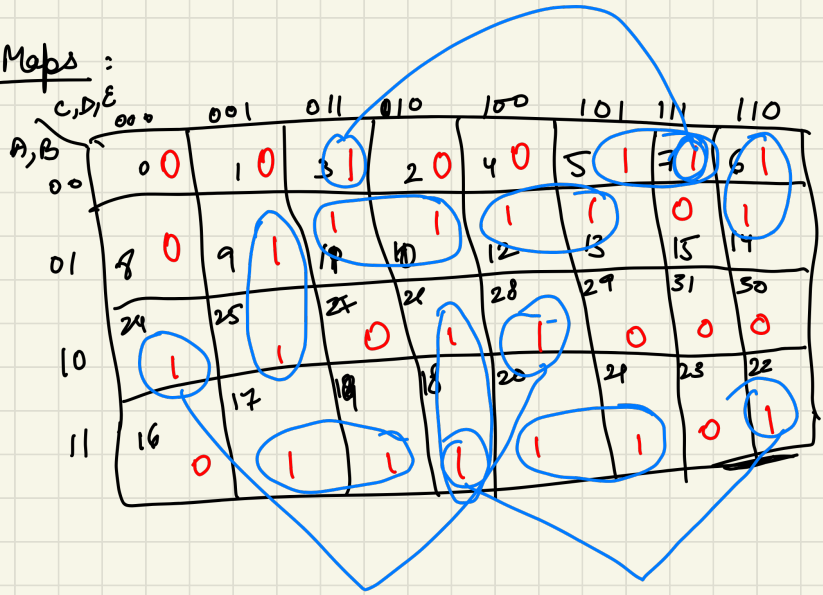
A=1

- 5 variable Reflection K-Map :

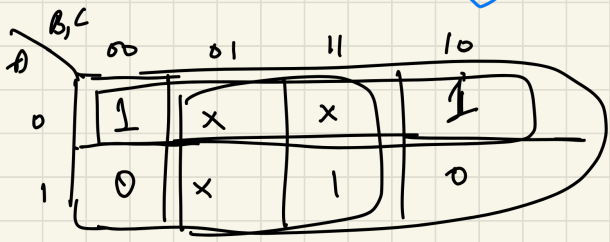


$B'C'E + A'B'C'D + A'C'DE + ABCE + A'B'E' + A'CDE' + AB'DE$

- Overlap Maps :



Eg



$F = \Sigma(0,2,7) + d(1,3,5)$

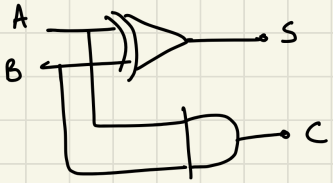
Sol $F = A' + C$

- **Combinational Circuits:**

Half adder : Adds two bits

Full adder : Adds three bits

Half Adder :



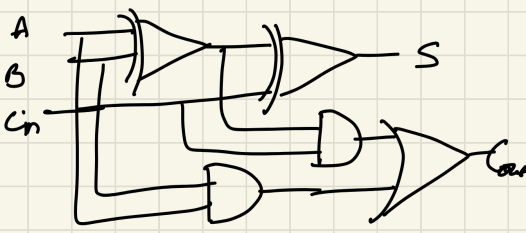
$S = A'B + AB' = A \oplus B$

$C = AB$

Input		output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder:

A	B	C_{in}	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$Sum = A \oplus B \oplus C_{in}$

$Carry = AB + (A \oplus B)C_{in}$

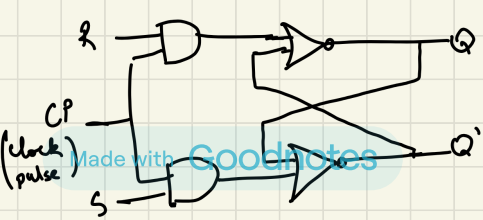
Lec 5 (Sequential logic – flip flops)

Flipflops

- SR-flipflop (basic)
- J-K flipflop
- D-flipflop (Delay)
- T flipflop (Toggle)

- Flip flops	latch
i) always have a clock signal.	doesn't have a clock signal.
ii) checks input & changes output only at times defined by clock signal.	– – – – changes the output immediately used to store either 1 or 0 at any given time.
iii) classified into synchronous & asynchronous.	no such classification
iv) edge triggered device	level triggered device

SR flipflop:



Truth table for SR flip flop

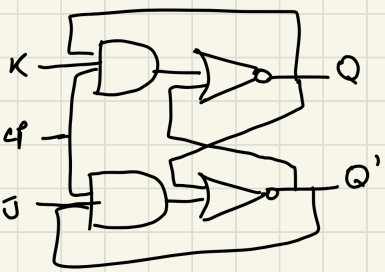
S	R	Q(N+1)
0	0	Q(N)
0	1	0
1	0	1
1	1	X

Characteristic table for SR flipflop

S	R	Q(N)	Q(N+1)	
0	0	0	0	same as Q(N)
0	0	1	1	same as Q(N)
0	1	0	0	depends on S
0	1	1	0	depends on S
1	0	0	1	depends on S
1	0	1	1	depends on S
1	1	0	x	when both S & R are on, states become indeterminate
1	1	1	x	when both S & R are on, states become indeterminate

$$Q(N+1) = S + R'Q(N)$$

JK flip flops : refinement of SR flip flop



J	K	Q(N+1)	
0	0	Q(N)	Hold
0	1	0	Clear to 0
1	0	1	set to 1
1	1	Q'(N)	Toggle

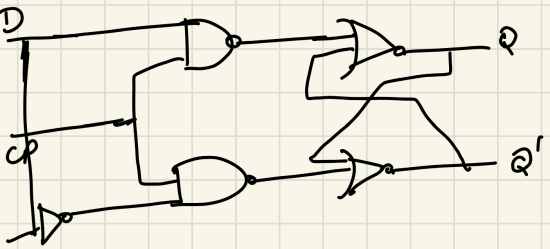
characterstic table of JK flipflop

J	K	Q(N)	Q(N+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$Q(N+1) = JQ'(N) + K'Q(N)$$

Delay flipflop (D flipflop):

also called transparent latch as input & output are equal
SR is converted to D by putting inverter b/w S & R.



D	Q(N+1)	
0	0	clear to 0
1	1	set to 1

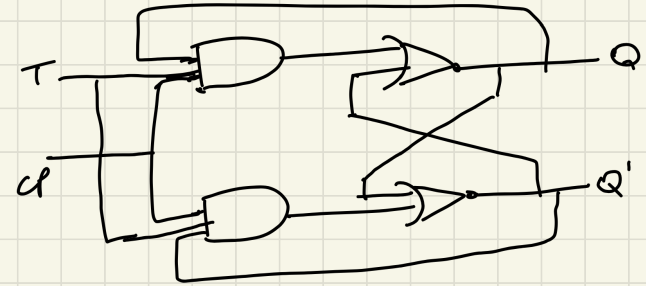
Characteristic table for D flipflop

D	Q(N)	Q(N+1)
0	0	0
0	1	0
1	0	1
1	1	1

$Q(N+1) = D$

Toggle flipflop (T flip flop):

obtained from JK flipflop. In JK flipflop when inputs J & K are connected to a single input by T



T	Q(N+1)
0	Q(N)
1	Q(N)

Characteristic table for T flip flop :

T	Q(N)	Q(N+1)
0	0	0
0	1	1
1	0	1
1	1	0

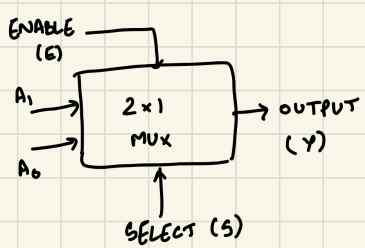
$$Q(N+1) = T \oplus Q(N) = T'Q(N) + T\,Q'(N)$$

Lec 6 : Multiplexer & Demultiplexer

- **Multiplexer :**

- 2^n inputs, single output

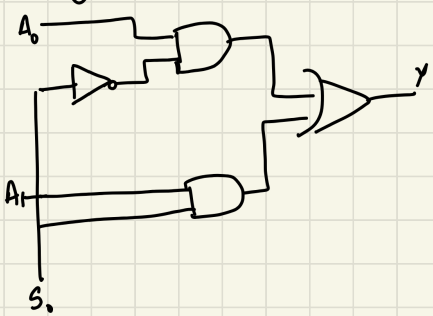
- **2×1 MUX :**



Truth table :

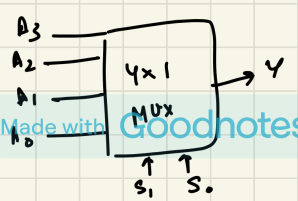
Input	Output
S_0	Y
0	A_0
1	A_1

Logical circuit :

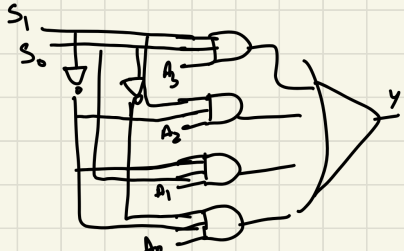


$$Y = A_0\bar{S}_0 + A_1S_0$$

- **4×1 MUX :**



S_1	S_0	Y
0	0	A_0
0	1	A_1
1	0	A_2
1	1	A_3



$$Y = S_1' S_0' A_0 + S_1' S_0 A_1 + S_1 S_0' A_2 + S_1 S_0 A_3$$

Ex: when input is 0,1,2,3 output is 1 greater than input
" " " 4,5,6,7 " " 1 less " ".

Design circuit.

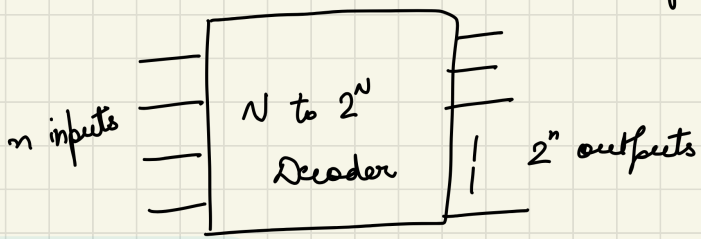
Sol^n

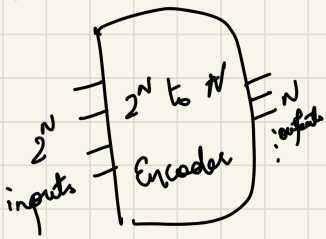
x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

A	B	C	S	C
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

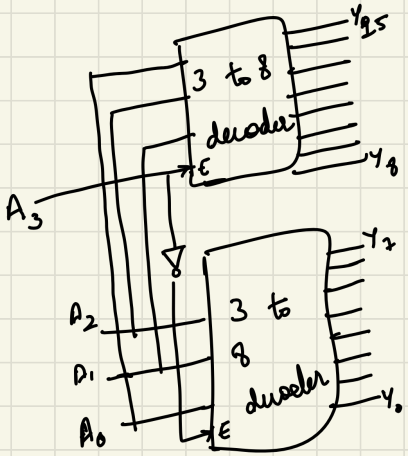
Draw the circuit using table. (full adder)

Lecture - 7 (Decoder & Encoder): with an additional input of enable

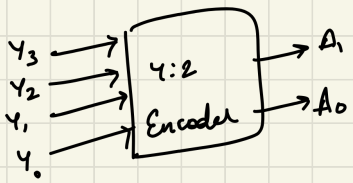




4 to 16 line decoder :



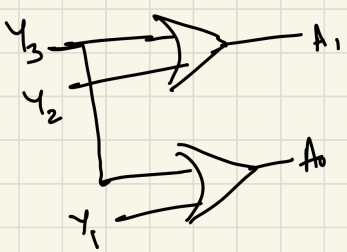
4 to 2 encoder



Inputs				Outputs	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

$A_1 = Y_3 + Y_2$

$A_0 = Y_1 + Y_3$



(ex 8 to 3 encoder = octal to binary)

Boolean Algebra :

A and ! Not

V or

Properties : ① $A + AB = A(1+B) = A$ or $A(A+B) = A$

② $A + A'B = A + B$ or $A' + A \cdot B = A' + B$

③ $(A+B)' = A' \cdot B'$ or $(A \cdot B)' = A' + B'$

for ②

$A + A'B$

$A + (1-A)B$

$A + B - AB$

eg: $F = (A+B+C)(A+B'+C)(A+B+C')$

$= (X+C)(X+C')(A+B'+C)$ $\quad (A+B = X)$

$= (X^2 + XC' + XC)(A+B'+C)$

$= (X^2 + X)(A+B'+C)$

$= X(A+B'+C)$

$= (A+B)(A+B'+C)$

$= A + AB' + AC + AB + BC$

$= A + AC + BC$

$= A + BC$

eg: $G = (A+B)(A+B')(A'+B)(A'+B')$

Sol' $G = (A + AB' + AB)(A' + A'B' + BA')$

$= (A)(A')$

$= 0$

midterm – 90s

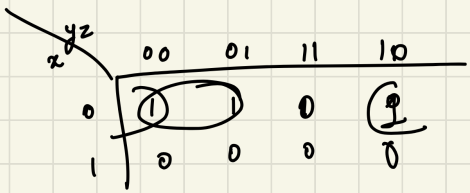
midterm – 30s

- Maxterm & minterm expressions
- Kmap

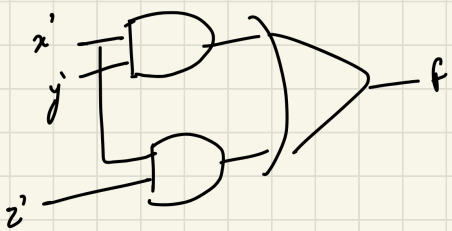
Numericals on full & half adder :

Q1.

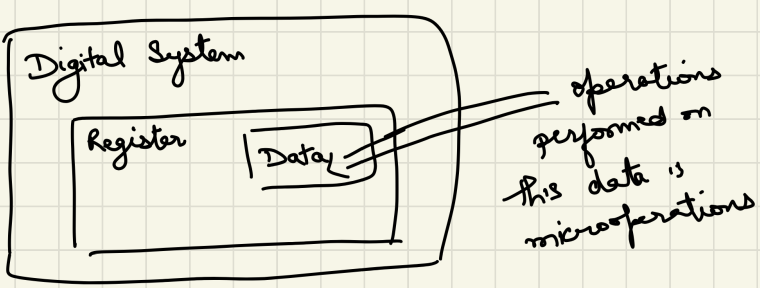
x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



$F = x'y' + z'x'$



Unit 2 :



A microoperation is an elementary operation performed on the info. stored in one or more registers.

eg. shift, load, clear, count

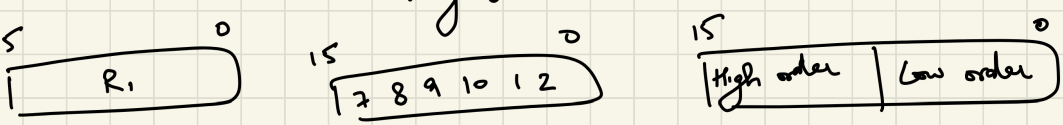
- Internal Hardware Organization of a digital computer

- Set of registers the computer contains
- Sequence of microoperations performed on the info. stored in the registers
- The control that initiates the sequence of microoperations

Register Transfer Language: Symbolic notation to denote microoperation transfer among registers

- Transfer ← availability of hardware logic circuit to perform the microoperations
  - Bus memory Transfer
  - Register Transfer

- Registers are denoted using MAR, PC, IR.
  - MAR → Memory access Register
  - PC → Program Counter
  - IR → Instruction Register



- Register transfer:

(1) Replacement operator

$R_2 \leftarrow R_1$

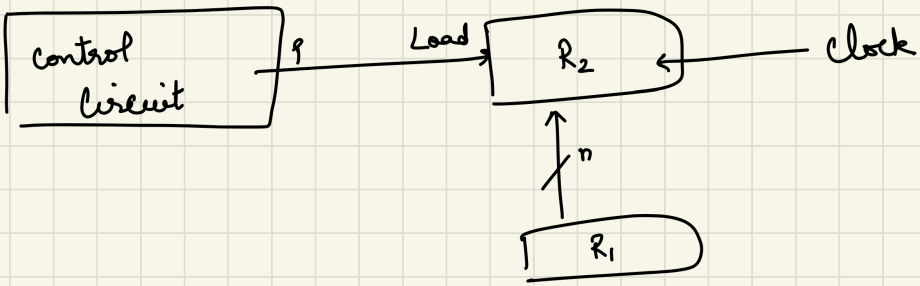
Transfer from R_1 to R_2

② Control signal

if (P=1) then $R_2 \leftarrow R_1$

or

P : $R_2 \leftarrow R_1$



when control variable P=1, n bits are transferred from R_1 to R_2.

Bus Transfer :

- communication lines are required for transfer b/w multiple registers & to ↓ the no. of lines common bus transfer is used.
- Bus transfer using 4 registers : n bits ⇒ n multiplexer

T : Bus ← C, R_1 ← Bus

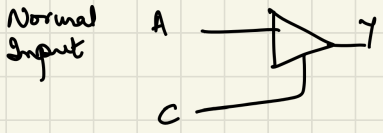
$R_1 \leftarrow C$

Three State Bus Buffer :

Rather than using MUX, we can use three state gates

States of a 3 state gate

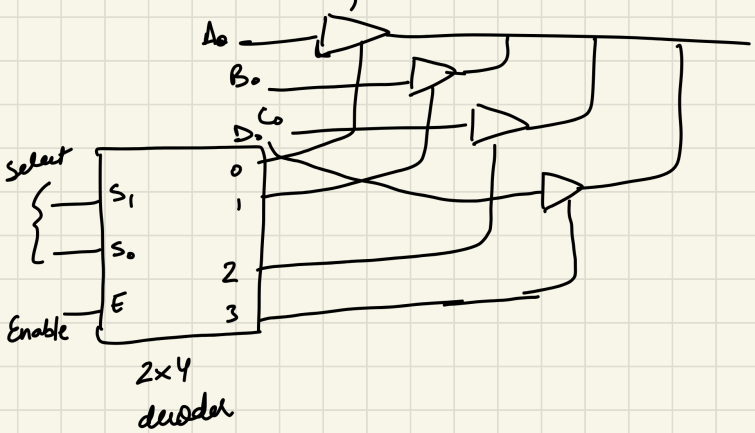
Logic 0 | Logic 1 | High-Impedance State (open circuit)



High-impedance if C = 0

Y = A if C = 1

Control Input



2×4 decoder

Memory Transfer:

- operations: Read, Write

Read : DR ← M[AR]

This causes a transfer of info. from memory word M selected by address in AR into DR.

Write : M[AR] ← R1

This causes a transfer of info. from R1 into memory word M selected by address in AR.

Microoperations :

$R_3 \leftarrow R_1 + R_2$ — content of R_1 plus R_2 transferred to R_3

$R_3 \leftarrow R_1 - R_2$ — Content of R_1 minus R_2 transferred to R_3

$R_3 \leftarrow \overline{R_2}$ — 1's complement of R_2

$R_3 \leftarrow \overline{R_2} + 1$ — 2's complement of R_2 (negate)

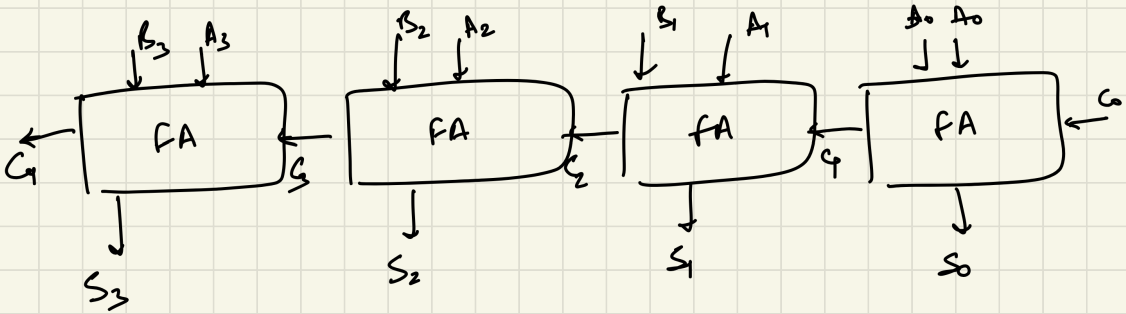
$R_3 \leftarrow R_1 + \overline{R_2} + 1$ — R_1 plus 2's complement of R_2 (subtraction)

$R_1 \leftarrow R_1 + 1$ — Increment content of R_1

$R_1 \leftarrow R_1 - 1$ — Decrement content of R_1

- 4 bit binary adder
  Binary adder - subtractor
  Binary Incrementer

Binary Adder : A binary adder is constructed with full adder circuits connected in cascade, output carry connected as input carry of next adder.



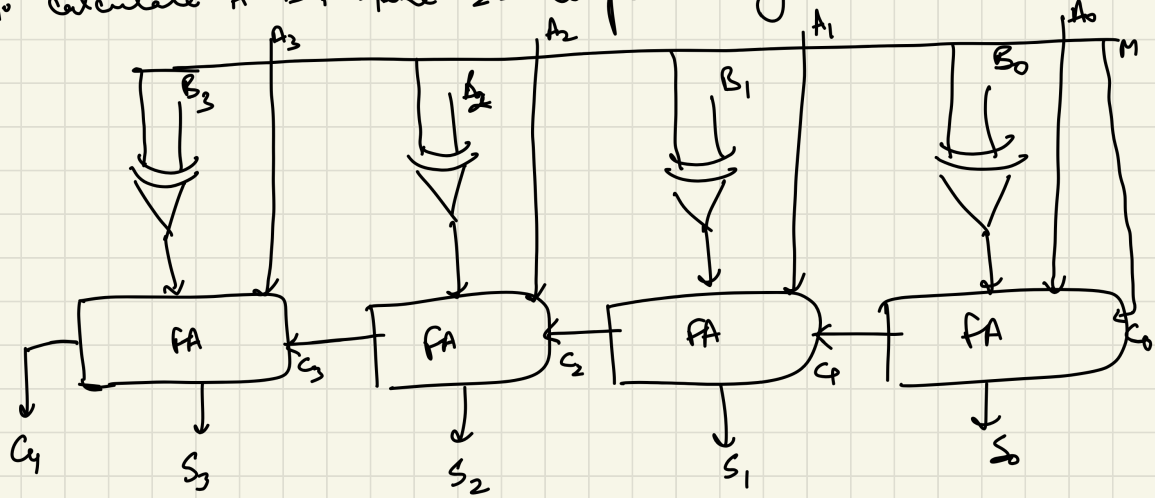
n-bit binary adder requires n full adders

Binary Adder - Subtractor:

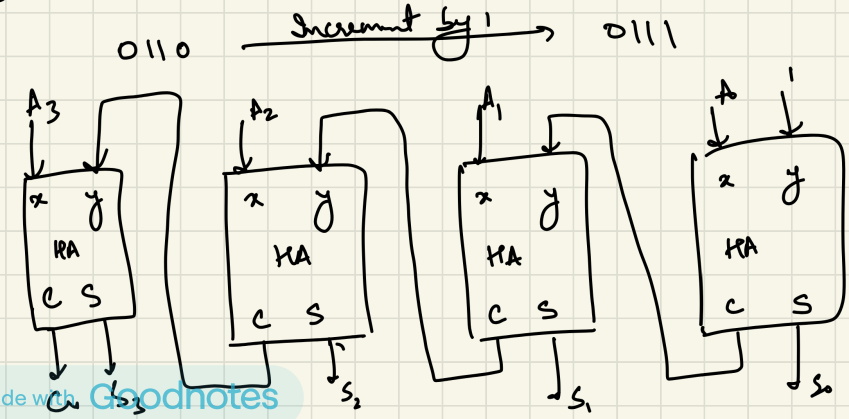
1's complement — can be implemented with inverters and added to sum through input carry

2's complement (subtraction) — Take 1's complement and add 1 to least significant pair of bits

To calculate A−B, take 2's complement of B and add it to A.



Binary Incrementer: Half Adders are used.



Arithmetic Circuit

Select S_1	Select S_0	Cin	Input Y	Output	Microoperation
0	0	0	B	$D = A + Y + C_{in}$	Add
0	0	1	B	$D = A + B$	Add with carry
0	1	0	$\bar{B}$	$D = A + B + 1$	Subtract with borrow
0	1	1	$\bar{B}$	$D = A + \bar{B}$	Subtract
1	0	0	0	$D = A + \bar{B} + 1$	
1	0	1	0	$D = A$	Transfer
1	1	0	1	$D = A + 1$	Increment A
1	1	1	1	$D = A - 1$	De
				$D = A$	Transfer

(*) '+' in control statement means 'OR', '+' in microop. denotes arithmetic sum.

$P + Q : R_1 \leftarrow R_2 + R_3 , R_4 \leftarrow R_5 \vee R_6$

OR ← Add ← OR

Applications of Logic Microoperations:

selective clear:

```
1010   A
1100   B
-----
0010   A (after)
```

$A \leftarrow A \wedge \bar{B}$

selective set:
set bits of A to 1 for corresponding 1's in B

selective complement:
complement portions of A where there are 1's in B.

Reset

```
1010
0011
-----
0010
-----
```

Mask operation : (AND)

```
1010   A
1100   B
1000   A (after)
```

Insert operation : ?

trying to mask the 4 leftmost bits

```
0110 1010   A (before)
0000 1111   B (mask)
0000 1010   A (after masking)
```

and then inserting the new value

```
0000 1010   A (after masking)
1001 0000   B (insert)
1001 1010   A (after insertion)
```

clear operation : $A \leftarrow A \oplus B$

The mask operation is AND microoperation and insert operation is OR. clear is achieved by XOR.

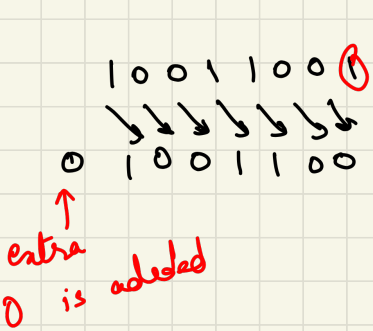
Shift Microoperations : Logical, Circular, Arithmetic

Logical

discard $R_1 \leftarrow shl\ R_1$

```
1 00110001   R1 (before)
0 01100010   R1 (after logical shift-left)
```

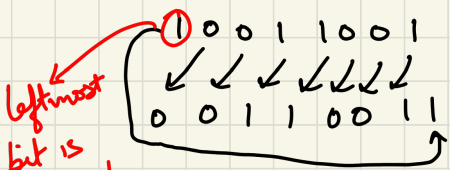
extra 0 is added



before

after right shift

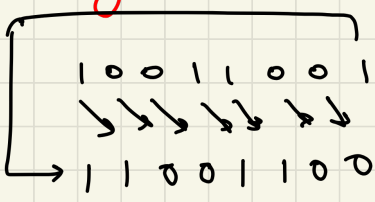
$R_2 \leftarrow shr\ R_2$



before

after

$R_1 \leftarrow cil\ R_1$



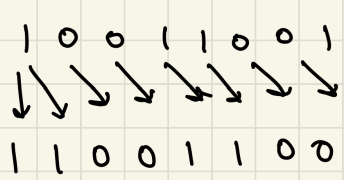
before

after

$R_2 \leftarrow cir\ R_2$

Arithmetic Shift : shifts a signed binary no. to left or right. The sign bit (i.e. the leftmost bit) must remain unchanged.

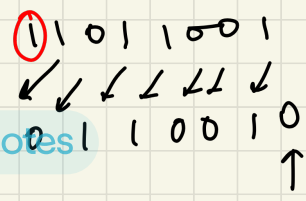
$R_1 \leftarrow ashr\ R_2$



R_1 (before)

R_1 (after arithmetic shift right)

$R_1 \leftarrow ashl\ R_1$



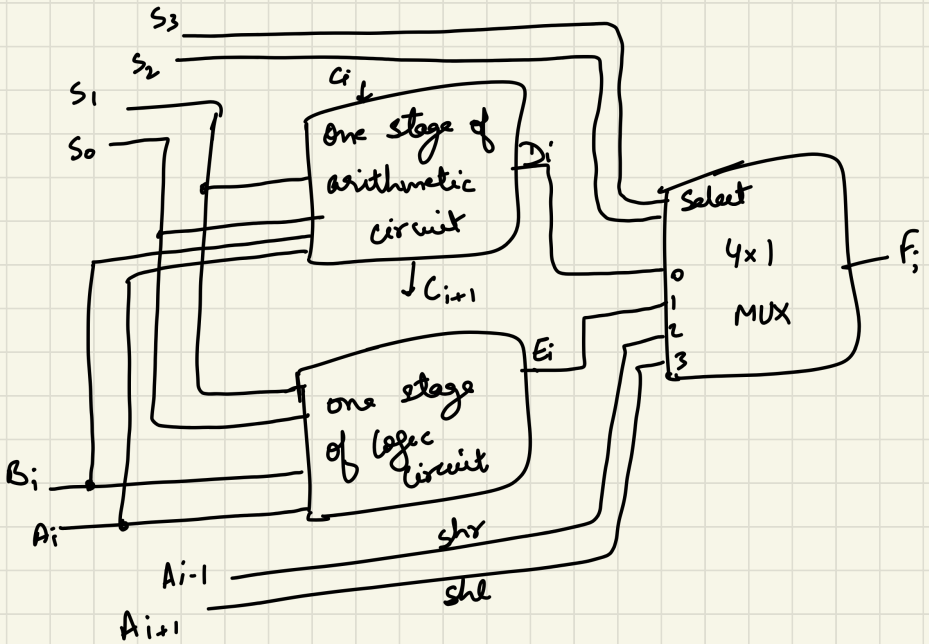
R_1 (before)

R_1 (after arithmetic shift-left)

In case of arithmetic left shift, the signed bit is lost resulting in erroneous results and this phenomenon is called "overflow".

Hardware Implementation of a Circuit Shifter ??

Arithmetic Logic Shift Unit : combination of all shift circuits

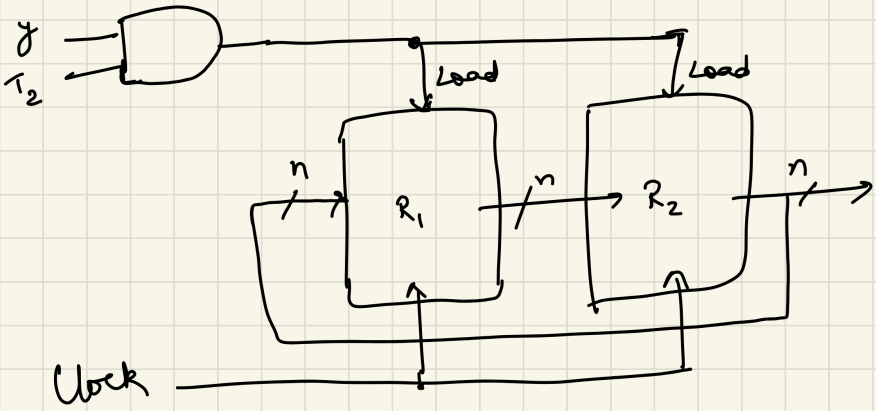


function table for arithmetic logic shift unit ???

questions :

q1 : yT2 : $R_2 \leftarrow R_1$, $R_1 \leftarrow R_2$

Sol :



q.2

$T_0 \leftarrow R_5 \leftarrow R_0$

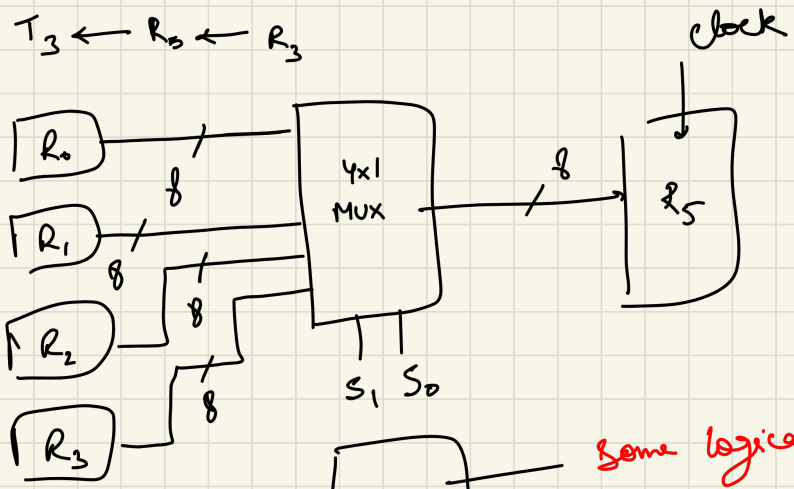
$T_1 \leftarrow R_5 \leftarrow R_1$

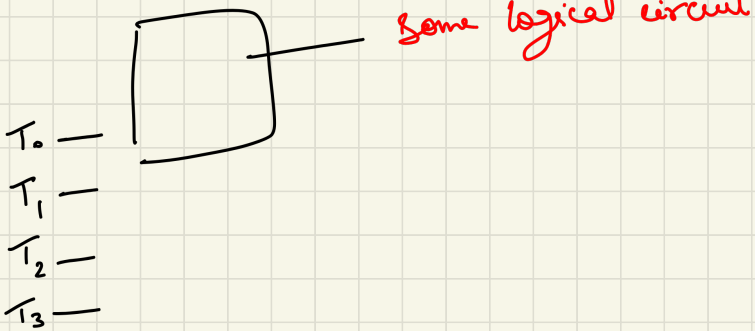
$T_2 \leftarrow R_5 \leftarrow R_2$

$T_3 \leftarrow R_5 \leftarrow R_3$

hardware = ?

Soln





our inputs to that logical circuit

Input		output	
S_0	S_1		
0	0	T_0	R_0
0	1	T_1	R_1
1	0	T_2	R_2
1	1	T_3	R_3

Time signals will be the inputs

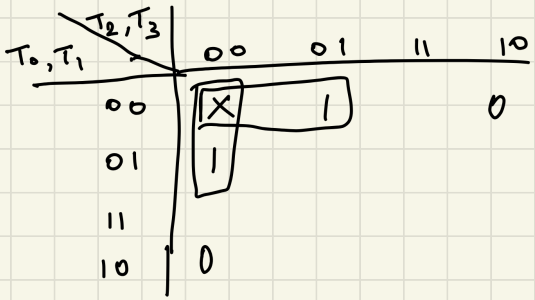
T_0	T_1	T_2	T_3	S_0	S_1	output of MUX
0	0	0	0	x	x	—
1	0	0	0	0	0	R_0
0	1	0	0	0	1	R_1
0	0	1	0	1	0	R_2
0	0	0	1	1	1	R_3

K-map for S_0

T_0, T_1 \ T_2, T_3	00	01	11	10
00	x	1	x	1
01	0			
11				
10	0			

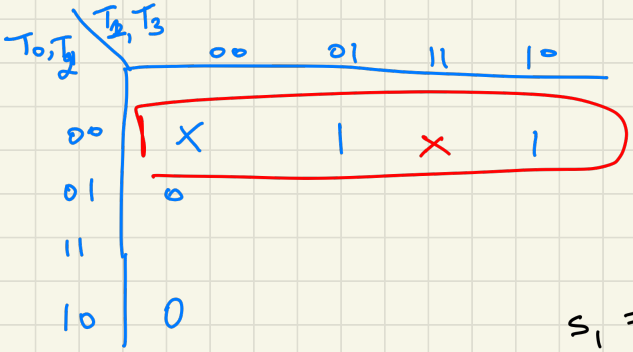
$S_0 = T_0' T_1'$

K-map for S_1

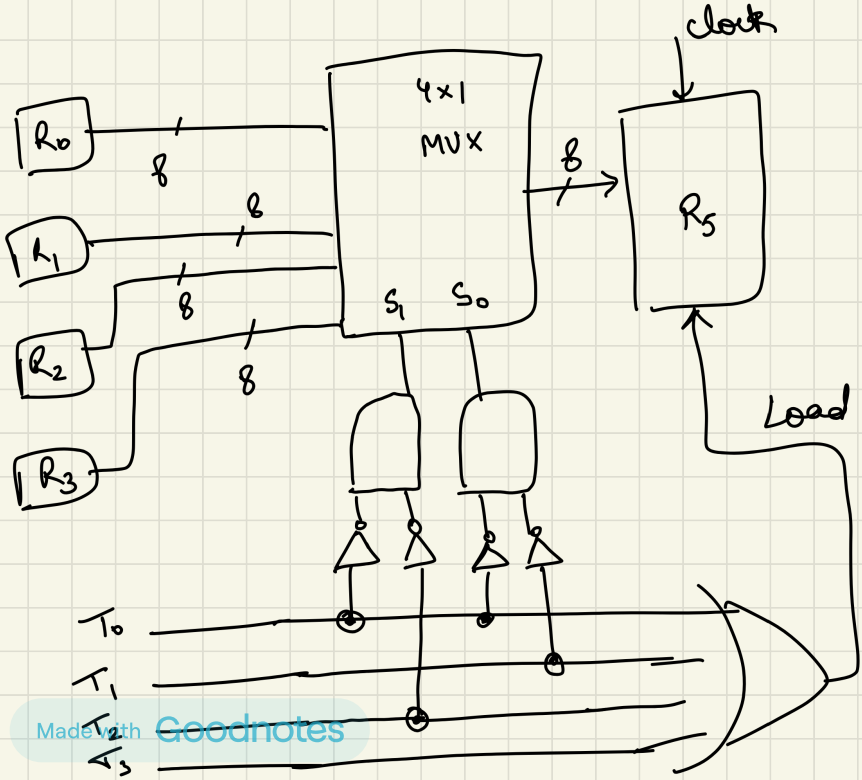


??

Alternative K-map for some reason



$S_1 = T_0' T_2'$



Load $= T_0 + T_2 + T_1 + T_3$

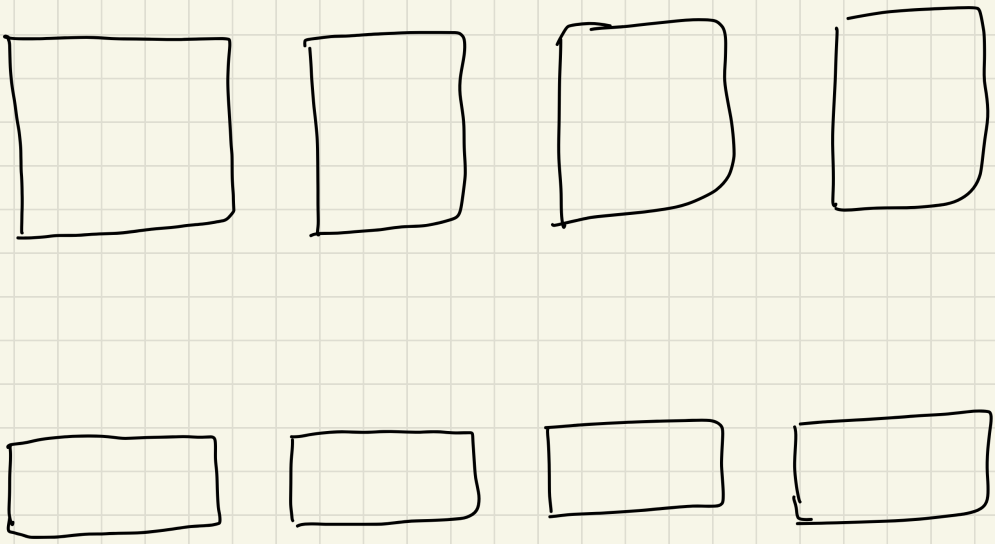
R_5 must be connected to all the time signals.

~~q3~~

P : $R_1 \leftarrow R_2$

P'Q : $R_1 \leftarrow R_3$

q4



q6 16 register of 32 bits each

Sol" a) 16 × 1 size of multiplexers

b) 4 selection lines ($16 = 2^4$)

c) 32 multiplexers (one bit for each register)

q7 a) $R_2 \leftarrow M[AR]$

b) $M[AR] \leftarrow R_3$

c) $R_5 \leftarrow M[R_5]$

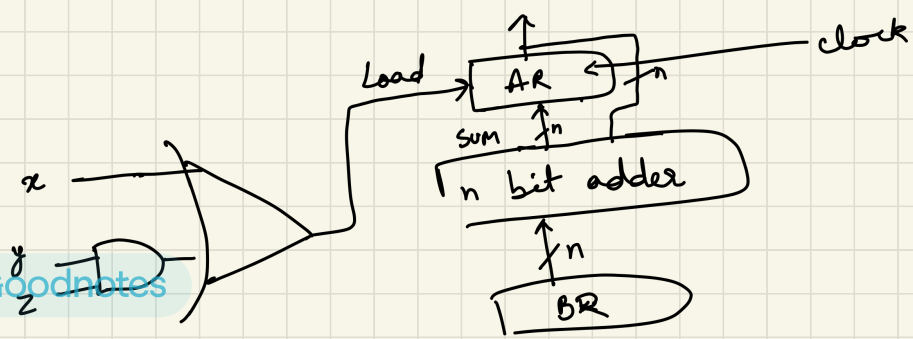
Sol" a) read the memory word specified by address AR into R_2.

b) write the content of R_3 into memory word specified by address in the AR.

c) same as a) (this will overwrite the prev. value of R_5)

q8 $x + yz$: $AR \leftarrow AR + BR$ AR & BR are n-bit registers

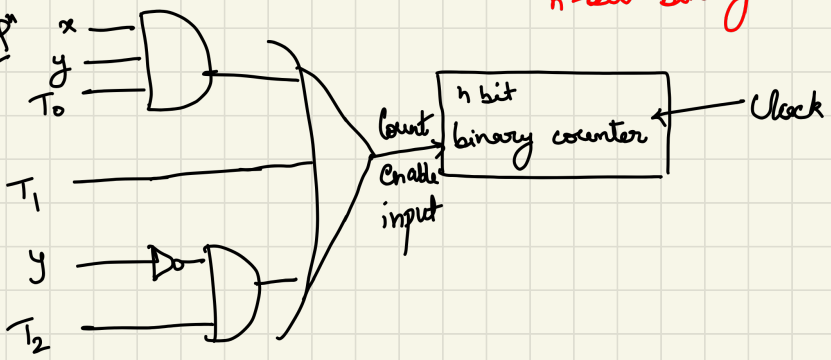
Sol"



q9 $xyT_0 + T_1 + y'T_2 : AR \leftarrow AR + 1$

n-bit binary counter

Sol"



q10 $xT : R_1 \leftarrow R_1 + R_2$

$x'T : R_1 \leftarrow R_2$

two 4 bit registers

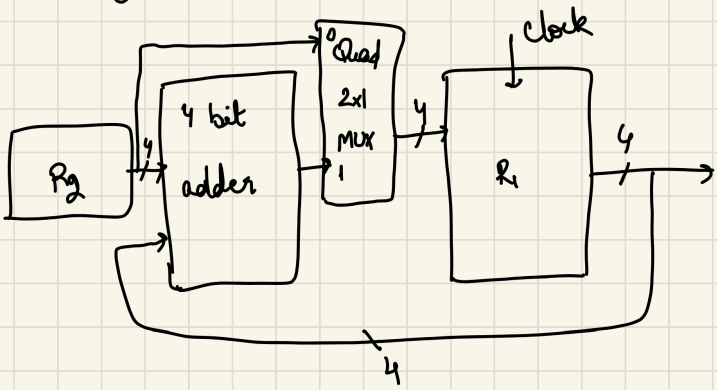
(use 4 bit adder & a quadruple 2-to-1 line mux.

Sol"

$T=1$

$R_1 \leftarrow R_1 + R_2 \quad x = 1$

$R_1 \leftarrow R_2 \quad x = 0$

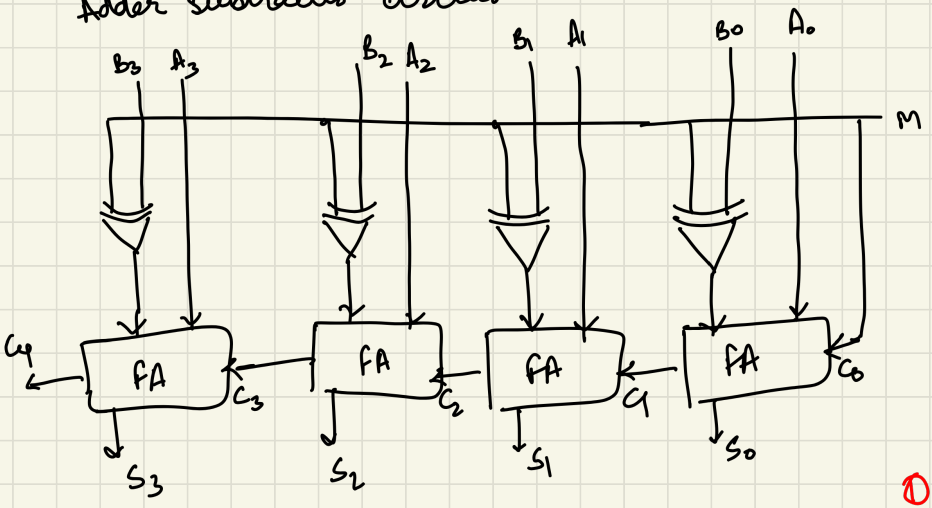


q12

Input Mode	Output Mode	
M	A	B
0	0111	0110
0	1000	1001
1	1100	1000
1	0101	1010
1	0000	0001

Sol^n

Adder subtractor circuit



when M = 0 — adder
M = 1 — subtractor

M	A	B	Sum/Diff	Carry (C_4)
0	0111	0110	1101	0
0	1000	1001	0001	1
1	1100	1000	0100	1
1	0101	1010	1011	0
1	0000	0001	1111	0

① 11 / 0111 / 0110 / 1101

② 1000 / 1001 / 0001

③ 1100 / 1000 / 0100

5 − 9 = 4

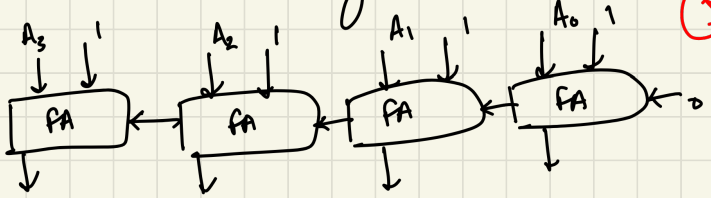
0010 / 1's 1010 / 0101 / 0001 / 1111

0 / 0101 / 1010 / 1011

(0 − 1 = 1) you take a borrow

Q13 Sol^n

4 bit decrementer using 4 full adder

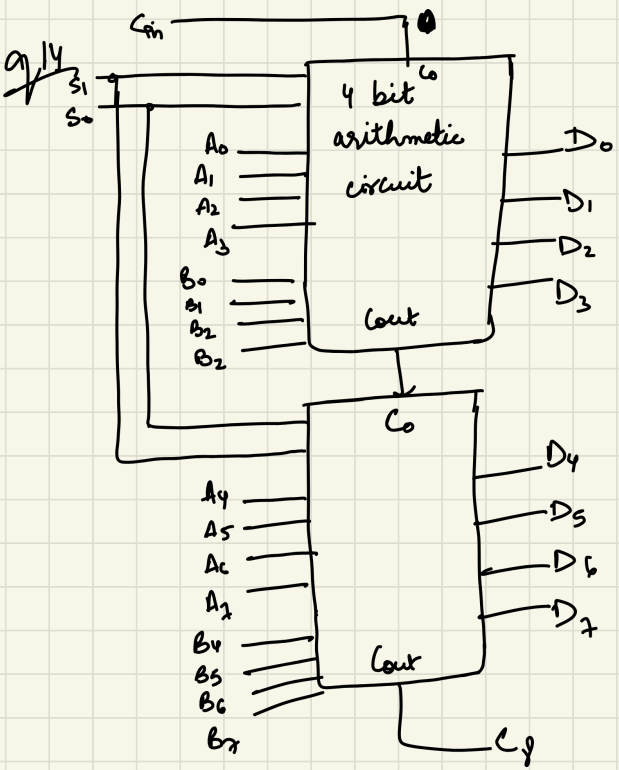


$A - 1 = A + (2'$ complement of $1)$

0001 → (1110) + 1 → 1111

⇒ $A_3 A_2 A_1 A_0$ + 1111

q14



we were given a 4 bit arithmetic circuit, it had to be connected to form an 8 bit circuit

q15

q-16, 17, 18 ⟶ easy

q19 : a) AR ← AR + BR

b) CR ← CR ∧ DR, BR ← BR + 1

Sol^n a)

```
  ①  1 1 1 1 1 1 1
     1 1 1 1 0 0 1 0
   + 1 1 1 1 1 1 1 1
   -----------------
AR = 1 1 1 1 0 0 0 1
```

b)

```
     1 0 1 1 1 0 0 1
AND  1 1 1 0 1 0 1 0
   -----------------
CR = 1 0 1 0 1 0 0 0
```

```
     1 1 1 1 1 1 1 1
  ①              1
   -----------------
   0 0 0 0 0 0 0 0 = BR
```

(c) AR ← AR − CR

```
  1 1 0 1 0   1 0   0   1 0
  1 1 1 1 0 0 1 0
- 1 0 1 1 1 0 0 1
-----------------
  0 1 1 1 1 0 0 1 = AR
```

∝ sequence main hai

```
  1 1 1 1 0 0 0 0
  1 0 1 0 1 0 0 0
-----------------
AR = 0 1 0 0 1 0 0 1
```

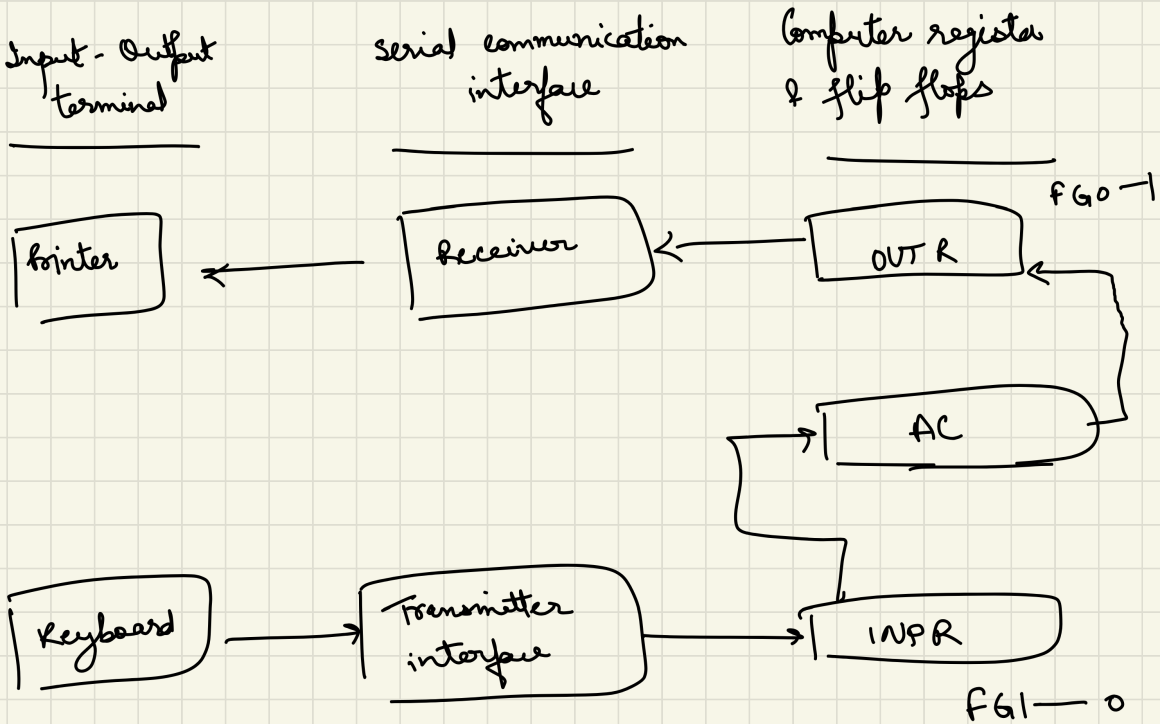
q-20

```
1 0 0 1 1 1 0 0
```

Arithmetic shift right

unit 3 :

- Input - Output Configuration ; flow of instruction transfer



- Initially, FGI is set to zero. When a key is pressed on Keyboard, an 8 bit alphanumeric code is shifted into INPR and FGI is set to 1.
- if FGI = 1, the info. from INPR is transferred in parallel to AC
- Initially, FGO is set to 1. The comp. checks the flag if flag is 1, the info. is transferred from AC to OUTR & FGO is cleared to 0.

- The printer accepts the coded info, prints the corresponding character, and when this is done.
FGO is set to 1.

- I/O instructions:

$D_3T_3 = p$ (common to all instructions)
$IR(i) = B_i$ (bit in IR (6-11) that specifies the instruction)

INP	p: SC ← 0 pB_{11}: AC ← INPR, FGI ← 0	Input character
OUT	pB_{10}: OUTR ← AC, FGO ← 0	Output character
SKI	pB_9: if (FGI = 1) then PC ← PC + 1	Skip on input flag
SKO	pB_8: if (FGO = 1) then PC ← PC + 1	Skip on output flag
ION	pB_7: IEN → 1	Enable Interrupt on
IOF	pB_6: IEN → 0	Enable " off

- Programmed Control Transfer: (PCT)

In this type of communication, the program checks the flag bit, & when it finds the bit is set, it initiates an info. transfer.

PCT is inefficient because of diff. of info. flow rates b/w processor and input/output device.

Program interrupt : A more efficient method is to let the i/o device inform the processor when its free and in the mean time the processor is free to do other task.

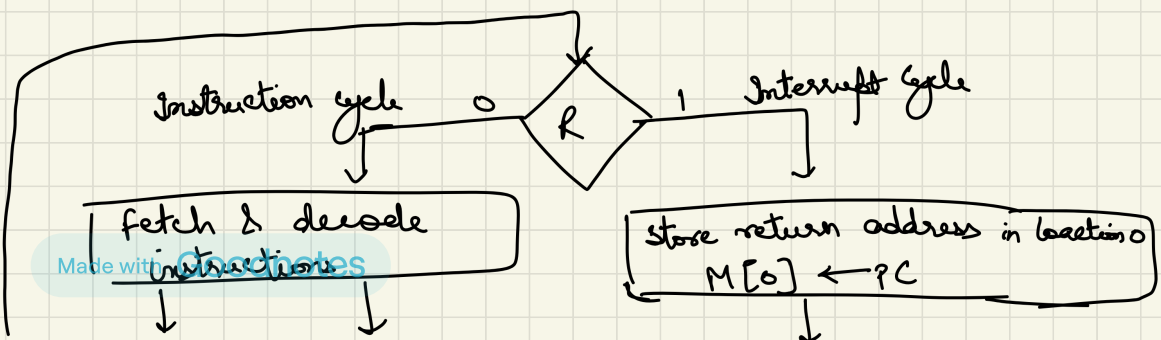
working :

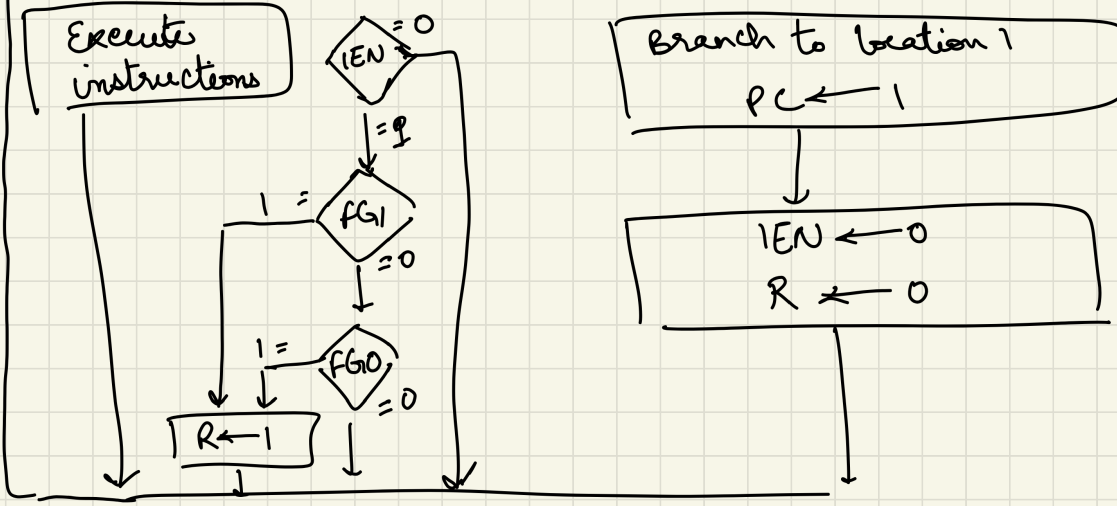
Step 1 — When comp. is running a program, it doesn't check the flags

Step 2 — When a flag is set, the comp. is interrupted from its current program and is informed that a flag has been set.

Step 3 — The comp. momentarily deviates from what it is doing to take care of I/O task & return to its original task.

Flowchart of Interrupt Cycle :





Microoperations for Interrupt Cycle :

1. $RT_0 : AR \leftarrow 0,\ TR \leftarrow PC$

During first timing signal, AR is cleared to zero. The content of PC is transferred to TR.

2. $RT_1 : M[AR] \leftarrow TR,\ PC \leftarrow 0$

During second timing signal, the return address is stored in memory location zero & PC is cleared to 0.

3. $RT_2 : PC \leftarrow PC+1,\ IEN \leftarrow 0,\ R \leftarrow 0,\ SC \leftarrow 0$

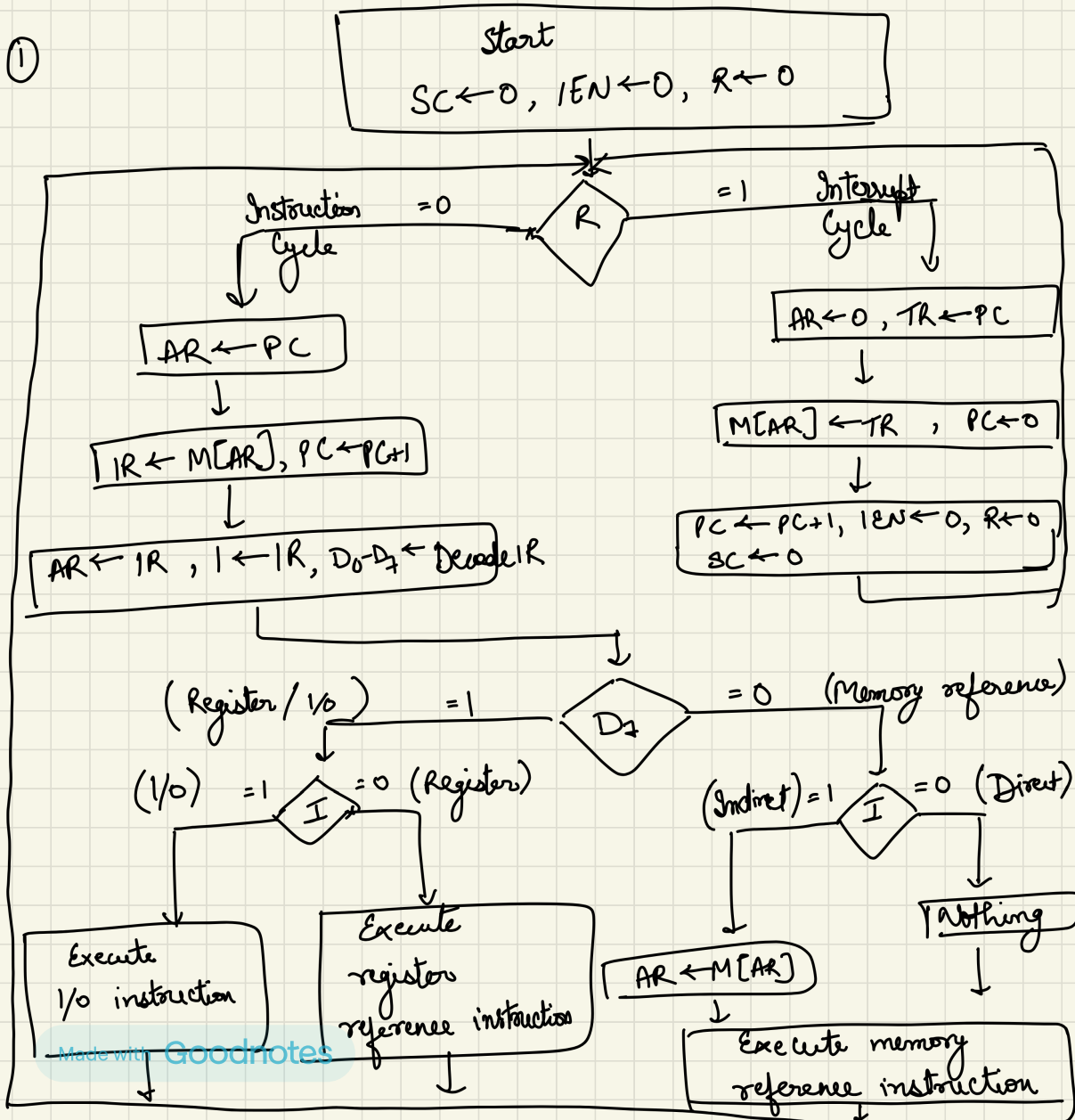
During the third timing signal, PC is increment. IEN, R, SC are cleared to zero.

Complete Computer Description:

① Flowchart for computer operation → Most Important

② Design of basic computer

①



(2) A memory unit with 4096 words with 16 bit each

9 register

7 flip flops (I, S, E, R, IEN, FGI, FGO)

2 decoder (3×8 op. decoder, 4×16 timing decoder

16 bit common bus

Control logic gates

Adder & logic circuit connected to input of AC

Exercise for unit 3 :

Q1 256K words of 32 bit each

Soln a) Indirect bit = 1

Opcode = $256K = 2^8 \times 2^{10} = 18$ bits

Register code = 64 registers = 2^6, 6 bits

Address code = $32 - 18 - 6 - 1 = 7$

b)

c) Data input = 32

Address input =

Q2 Direct → 2 read references (Addr, operand)

Indirect → 3 (, effective address, operand)

Q3

?

Q4

?

q5 a) $IR \leftarrow M[PC]$

$AR \leftarrow PC$

$IR \leftarrow AR$

b) $AC \leftarrow AC + TR$

$DR \leftarrow TR$

$AC \leftarrow AC + DR$

c) $DR \leftarrow DR + AC$ (AC doesn't change)

switch content of DR & AC — $DR \leftarrow AC$, $AC \leftarrow DR$

Addition — $AC \leftarrow DR + AC$

switch again — $DR \leftarrow AC$, $AC \leftarrow DR$

q6 a) $\underline{0001}\ \underline{0000\ 0010\ 0100} = (1024)_{16}$

1 — Add

0 2 4 — $(024)_{16}$

Add M[024] to AC

b) $\underline{1011}\ \underline{0001\ 0010\ 0100} = (B124)_{16}$

$2^3\ 2^1\ 2^0$ — 2^2 — B — $8+2+1$ — $(124)_{16}$

Store AC in M[M[124]]

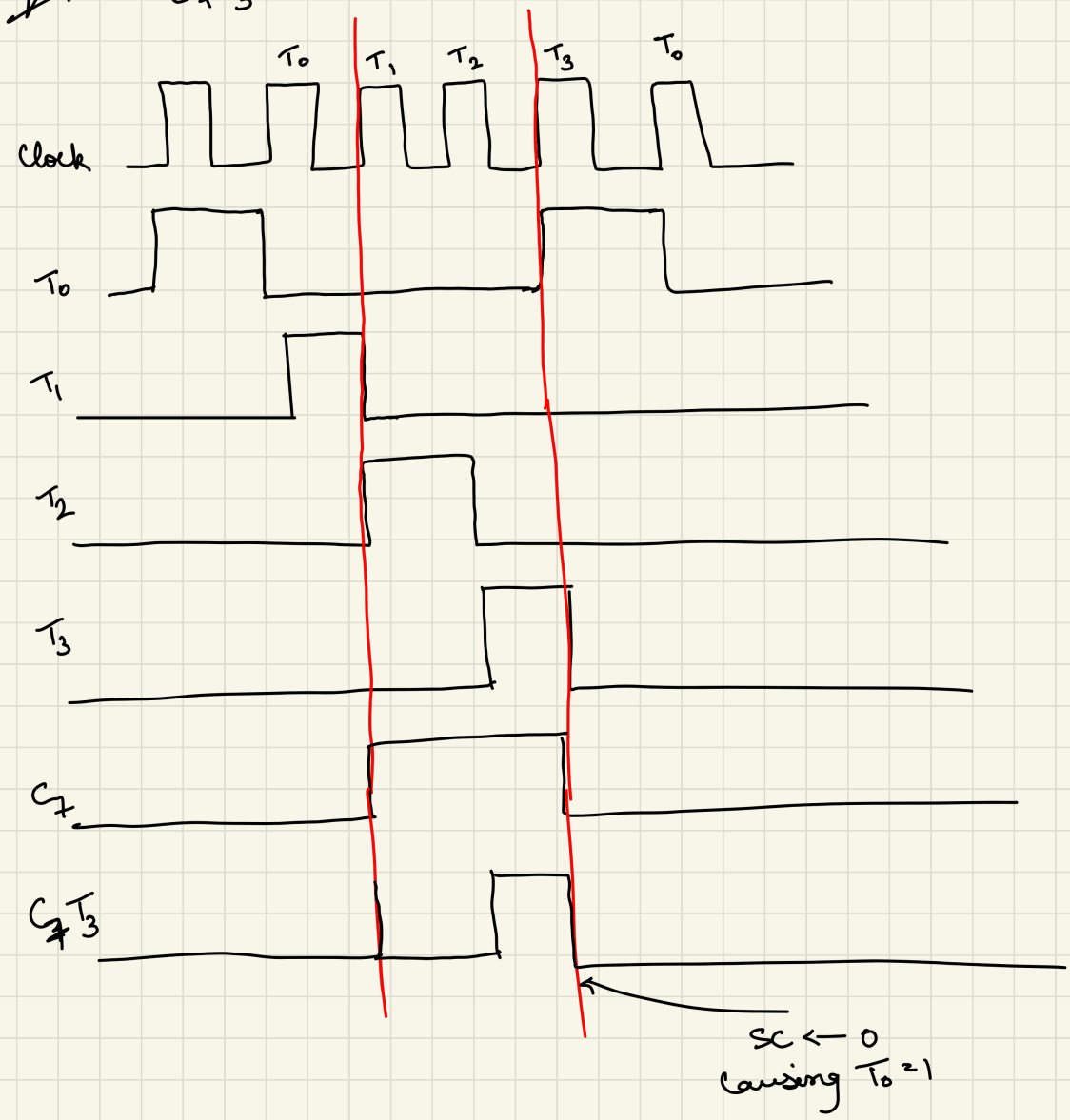
c) $\underline{0111}\ \underline{0000\ 0010\ 0000} = (7020)_{16}$

$1+2+4$ — Register reference — Increment AC

q7 CLE : Clear E
CME : Complement E

(Test set E → 1)

q8 C_7T_3 : SC ← 0



Q9 Register Reference Instructions

(5) Cir (AC)

AC = A937

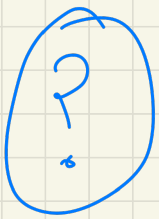
Mister glassman

1010 → 1101 ? → D

1001 → 0100 ? → 4

0011 → 1001 → 9

0111 → 1011 → B



(6) CIL(AC)

AC = A937

1010 → 0101 → 5

1001 → 0010 → 2

0011 → 0110 → 6

0111 → 1111 → F

(9) Skip to next (SNA) if AC is negative:

	E	AC	PC	AR	IR
Initial	1	A937	021	–	–
SNA	1	A937	023	008	7008

In case of if AC is +ve if AC is zero we incremented PC by 1 only

Q10 go over the instruction cycle

M[083] = BBF2

AC = A937

I = 0

	PC	AR	DR	AC	IR
Initial	021	-	-	A937	-
BUN	083	083	-	A937	4083

BUN : PC ← AR

	PC	AR	DR	AC	IR
Initial	021	-	-	A937	-
BSA	084	084	-	A937	5083

BSA : M[AR] ← PC, PC ← AR + 1

AR ← PC

ISZ : M[AR] ← M[AR] + 1

If M[AR] + 1 = 0 then PC ← PC + 1, SC ← 0

Q4.

PC = 7FF

M[7FF] = EA9F

M[A9F] = 0C35

M[C35] = FFFF

?

9.12 PC = 3AF M[3AF] = 932E M[9AC] = 8B9F
AC = 7EC3 M[32E] = 09AC

Soln a)

Memory

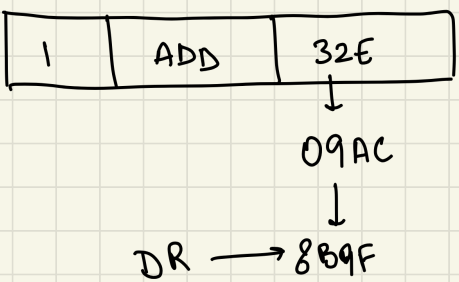
3AF	932E
32E	09AC
9AC	8B9F

PC has 3AF and at 3AF we have 932E.

9 = 1001

1001 32E = | 1 | ADD | 32E |

b) AC = 7EC3



9 A B C D E F

```
  1 1 1
  7EC3
  8B9F
  ----
  0A62
```

(1)

3+F = 18 = (16)+(2)

goes in carry

Add 7EC3 and 8B9F, we get 0A62.

E = 1

c) | 1 | ADD | 32E | → 09AC → 8B9F → DR

E = 1, I = 1, SC → 0

PC = 3AF+1 = 3B0

AR = 9AC

AC = 0A62

IR = 932E

DR = 8B9F

```
  1
  3AF
  1
  ---
  3B0
```

Q13

	Symbol	opcode	Symbolic designation
i)	XOR	000	$AC \leftarrow AC \oplus M[EA]$
iii)	ADM	001	$M[EA] \leftarrow M[EA] + AC$
iii)	SUB	010	$AC \leftarrow AC - M[EA]$
iv)	XCH	011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$
v)	SEQ	100	If $(M[EA] = AC)$ then $PC \leftarrow PC+1$
vii	BRA	101	If $(AC > 0)$ then $PC \leftarrow EA$

? ?

Soln i)

original

$AC \leftarrow AC + M[AR]$,

$E \leftarrow Cout$

D_1T_4: $DR \leftarrow M[AR]$

D_1T_5: $AC \leftarrow AC + DR$,

$E \leftarrow Cout$, $SC \leftarrow 0$

New Instruction

$M[EA] \leftarrow M[EA] + AC$

D_1T_4: $DR \leftarrow M[AR]$

D_1T_5: $DR \leftarrow AC$, $AC \leftarrow AC + DR$

D_1T_6: $M[AR] \leftarrow AC$, $AC \leftarrow DR$, $SC \leftarrow 0$

Q14 $S_2S_1S_0 = 000$

LDC Address CTR ← M[Address]

(Replace ISZ instruction that loads a no. into CTR)

Add a register reference ICSZ

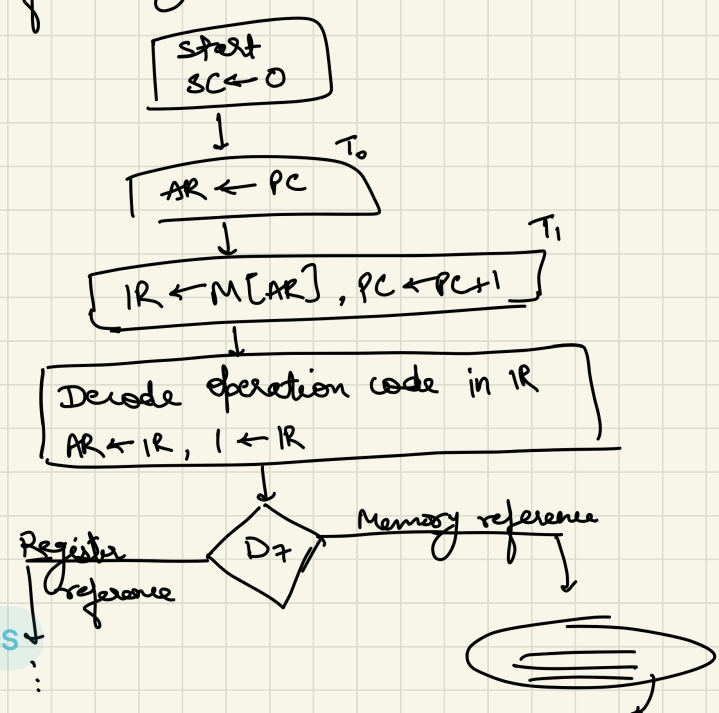
Sol^n Conversion from memory reference instruction to register reference instruction.

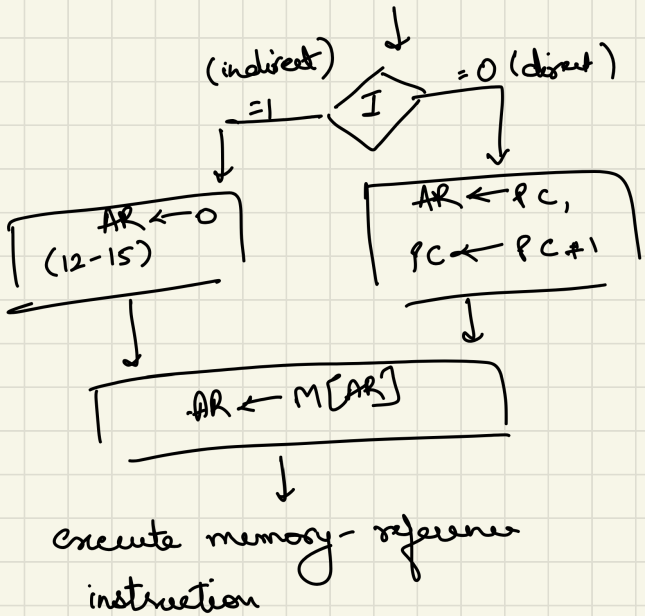
The new instruction ICSZ can be executed at T_3 instead of T_6.

Q15 65536 × 16, (I=1) address part of instruction residing in position 0 through 11

I=0, the address of instruction is given by 16 bits in next word following the instruction.

Sol^n



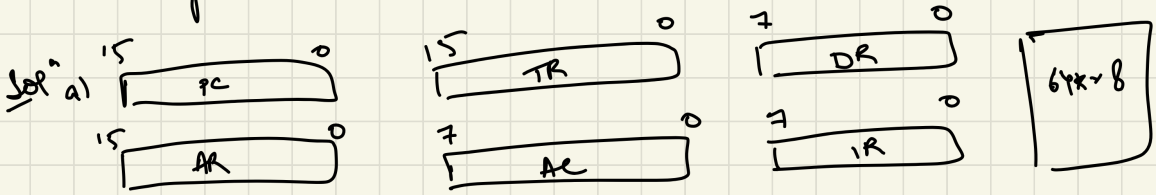


q16 PC, AR, TR, AC, DR, IR , 8 bit opcode, 16 bit address

16 8

All operands are 8 bits. No indirect bit

Sol^n a)



b)

c) T_0 : IR ← M(PC), PC ← PC+1

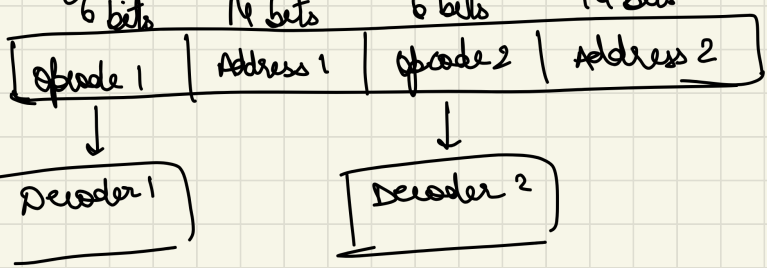
T_1 : AR ← M[PC], PC ← PC+1
(0-7)

T_2 : AR(8-15) ← M[PC], PC ← PC+1

T_3 : DR ← M[AR]

q17 16384 words, 40 bits per word, 6 bits for operation part, 14 bits for address part, 2 instructions are packed in 1 memory word, 40 bit IR is available.

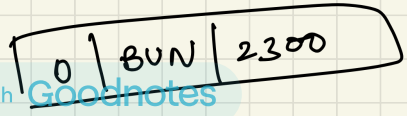
soln



1. Read 40 bit double instruction from memory to IR, increment PC.
2. Decode opcode 1
3. Execute instruction 1 using address 1.
4. Decode opcode 2.
5. Execute instruction 2 using address 2.
6. Go to step 1.

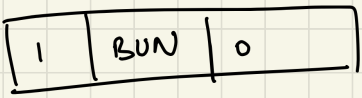
q18 FGO = 1, EN = 1

soln

a) 

?

b)



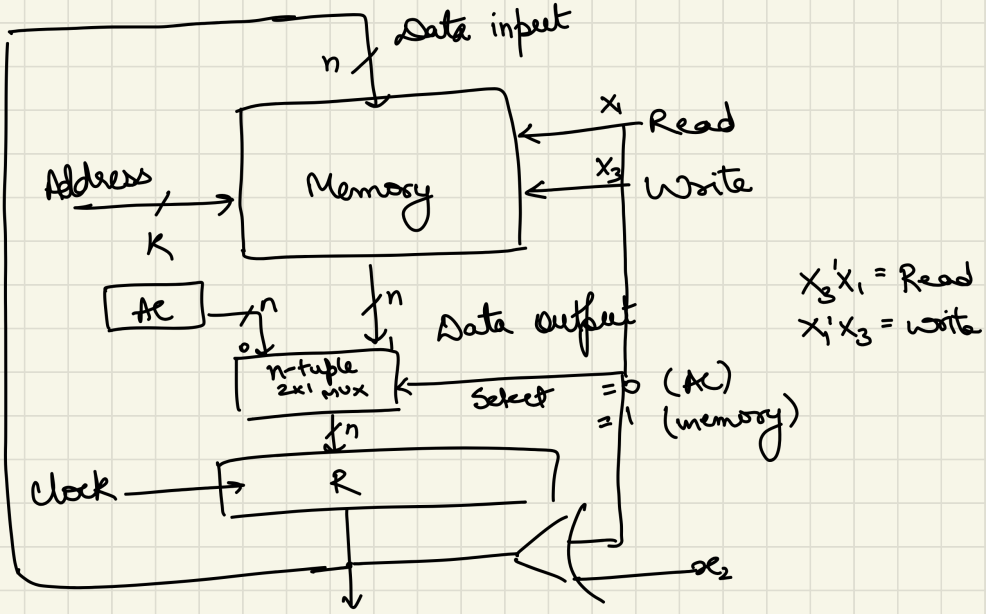
q.19

$x_3'x_1$: $R \leftarrow M[AR]$ — read memory word into R

$x_1'x_2$: $R \leftarrow AC$ — transfer AC to R

$x_1'x_3$: $M[AR] \leftarrow R$ — write to memory

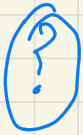
soln



$x_3'x_1$ = Read

$x_1'x_3$ = write

q.22, 23, 25



Unit - 3 :

- Computer instruction → instruction code = opcode + address

↓

operation to be performed

4096 x 16 memory → 4096 = 2^{12} ⇒ 12 bits to specify an address

16 bit memory word = 2^4 ⇒ 4 bit opcode

=

↓

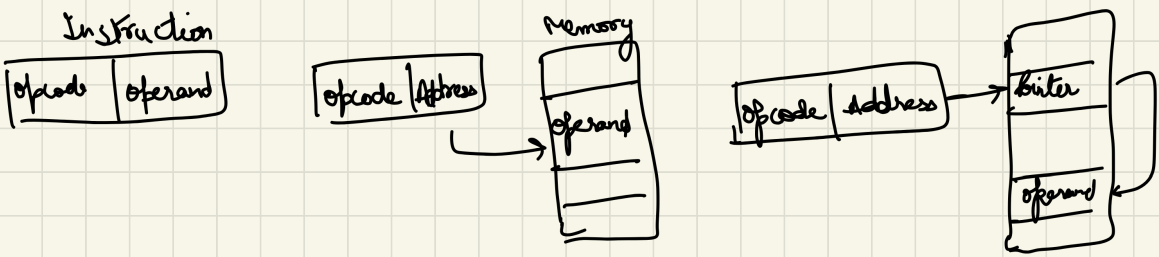
16 operations can be performed

- Addressing modes : Immediate, Direct, Indirect

↓ specifies operand

↓ specifies address of operand

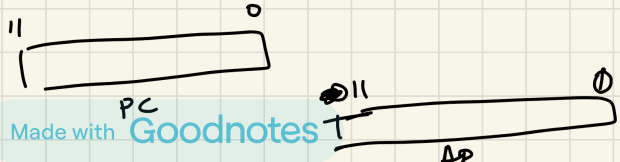
↓ code contains address of memory in which address of operand is present



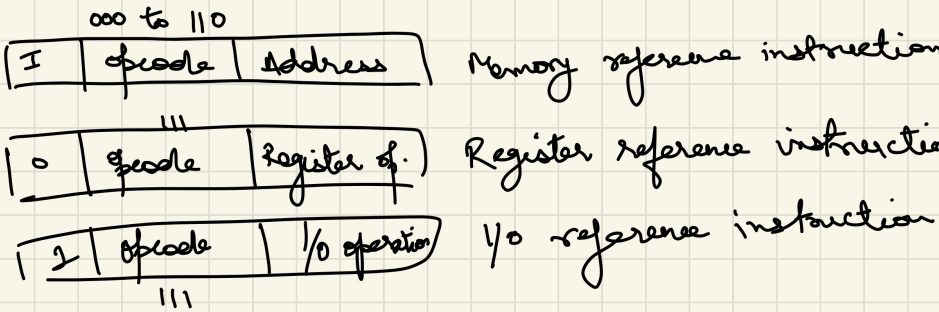
Immediate

Direct

Indirect

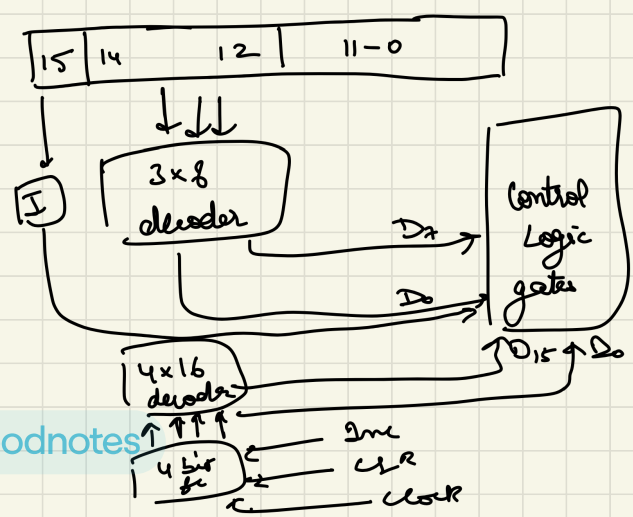


PC	12	program counter	Holds address of instruction
AR	12	address register	Holds address for memory
IR	16	Instruction register	Holds instruction code
TR	16	Temporary register	Holds temporary data
DR	16	Data register	Holds memory operand
AC	16	Process register (accumulator)	Process register
INPR	8		Holds input character
OUTR	8		Holds output character



- Timing of all registers is controlled by a Master Control Generator.

Block diagram of typical control unit



- Relationship b/w Data transfer, Clock Transition, Timing Signals

T_0 : $AR \leftarrow PC$

- Instruction cycle : Fetch the instruction from memory
Decode the instruction
Read effective address from memory
Execute the instruction

diagram for fetch phase

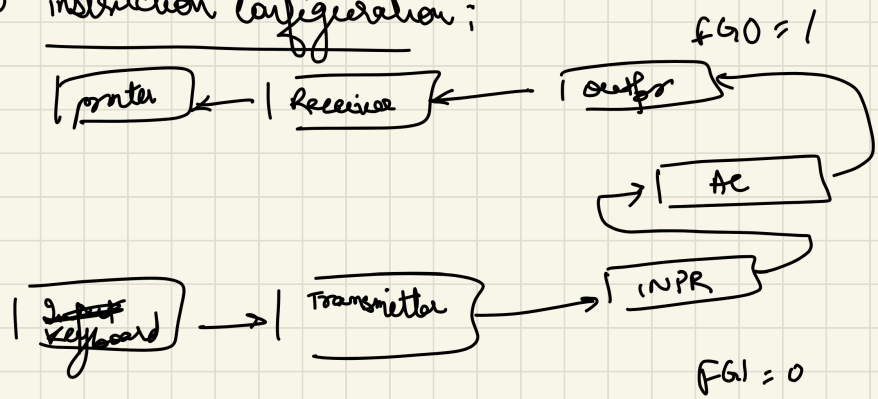
- Memory reference Instructions :

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow Cout$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BSA	D_4	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_5	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$
BUN	D_6	$PC \leftarrow AR$

execution of memory reference starts with T_4.

BSA : M[AR] ← PC, PC ← PC+1

I/O instruction configuration:



Lec-1 : (Unit-1)

- Binary to octal

$$\underset{6}{110}\ \underset{5}{101} = 65$$

- Binary to hexadecimal (form groups of 4)

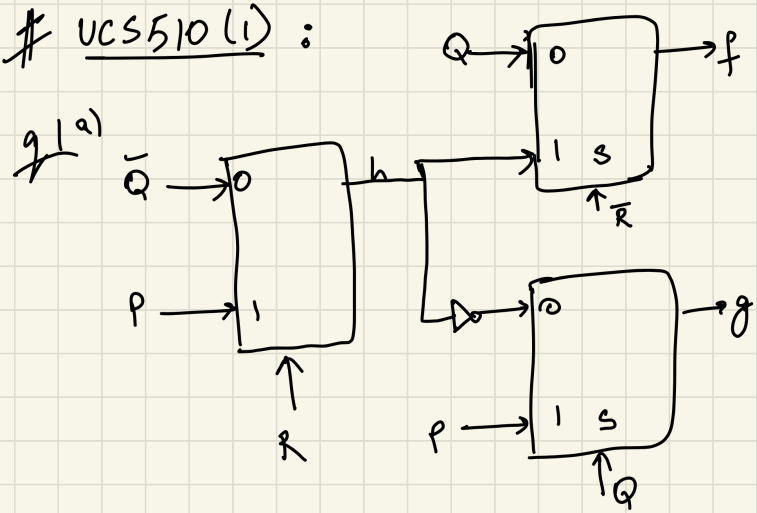
for integer — right to left
for fractional — left to right

- Octal to hexadecimal : use binary as intermediate

- 0 → me
  1 → ue

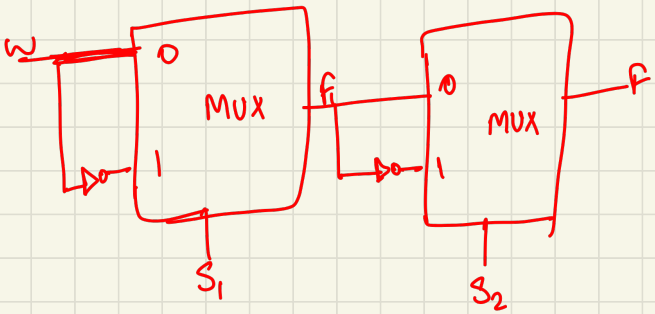
PYQ

UCS510 (1) :



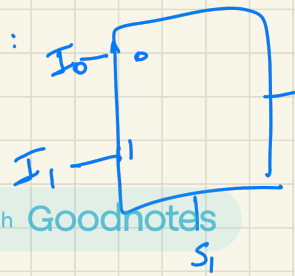
g = ?

Sol: $g = PQ + \bar{h}\bar{Q}$, $h = PR + \bar{Q}\bar{R}$



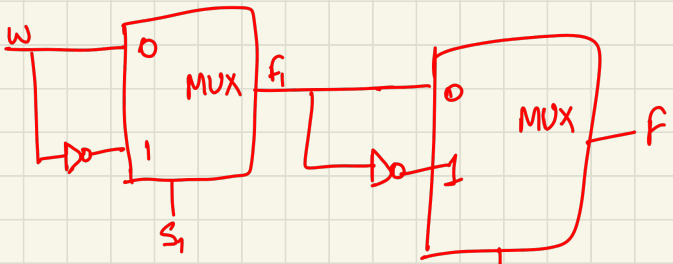
F = ?

Concept :



$f = I_0\bar{S_1} + I_1S_1$

Analysis :



$$f_1 = W\bar{S_1} + \bar{W}S_1 = W \oplus S_1$$

$$F = f_1\bar{S_2} + \bar{f_1}S_2 = f_1 \oplus S_2 = W \oplus S_1 \oplus S_2$$

$$\boxed{F = W \oplus S_1 \oplus S_2}$$

$$g = PQ + \bar{h}\bar{Q}$$
$$h = PR + \bar{Q}\bar{R}$$

$$\Rightarrow g = PQ + \overline{(PR + \bar{Q}\bar{R})}\,\bar{Q}$$

$$= PQ + (\overline{PR})(QR)(\bar{Q})$$

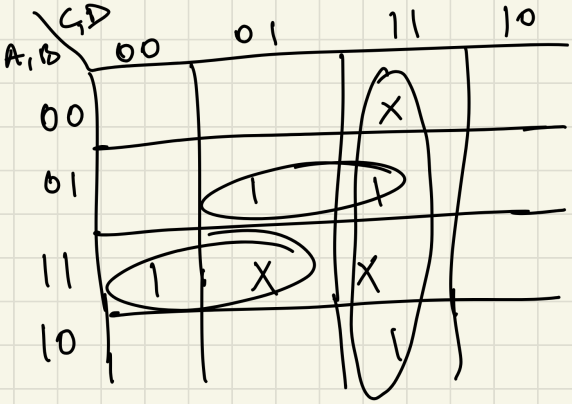
$$= PQ + 0$$

$$\boxed{g = PQ}$$

q2 b) $f(A, B, C, D) = \Sigma(5, 7, 11, 12) + d(3, 11, 13, 15)$

sol Minimal exp. = SOP

A,B \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10



$$f = CD + ABC + A'BD$$

S	$C_{in} = 0$	$C_{in} = 1$
0	D = A+B	D = A-B
1	D = A+1	D = A-1

2 bit input ⟶ A, B

carry input ⟶ $C_{in} = 0$, $C_{in} = 1$

selection line ⟶ S

Soln a)

reference :

S	$C_{in}=0$	$C_{in}=1$
0	$D = A+B$	$D = A+1$
1	$D = A-1$	$D = A+B'+1$

for add^n → Adder , carry → full adder
all outputs have A → A should be one of inputs
we use a multiplexer to take care of the selection variable ⇒ one of the inputs from mux into adder.

Now converting table,

S	C_{in}	X	Y
0	0	A	B
0	1	A	0
1	0	A	1
1	1	A	B'

concept : $sum = A \oplus B \oplus C_{in}$

$C_{out} = AB + (A \oplus B) C_{in}$

q.2b)

eg. $(45.45)_{10} = (\quad)_2$ using IEEE 754 format

Sol^n $(45)_{10}$ = 32 16 8 4 2 1 = $(101101)_2$

1 0 1 1 0 1

$(0.45)_{10}$ = 101101.01110011 = $1.0110101110011 \times 2^5$

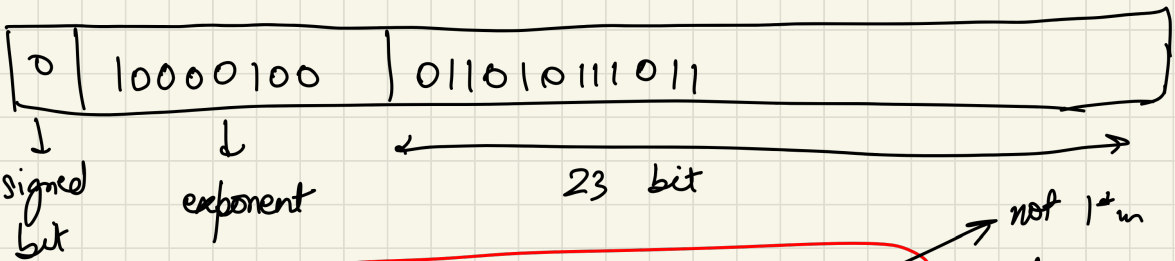
01110011

$127 + 5 = 132$

0	0.45 × 2	0	0.20 × 2
1	0.90 × 2	0	0.40 × 2
1	1.80 × 2	1	0.80 × 2
1	1.60 × 2	1	1.60
1	1.20		

$132 =$ 128 64 32 16 8 4 2 1

1 0 0 0 0 1 0 0



Binary to decimal : $(-1)^{s} \times 2^{e-127} \times 1.m$ → not 1^{st} m

$1.m$

s = signed bit

e = exponent number = 10000100 = (in decimal)

m = mantissa = ____ (in decimal)

eg: $(85.125)_{10} = (\quad)_2$ using IEEE

$(85)_{10}$ = 64 32 16 8 4 2 1 = $(1010101)_2$

1 0 1 0 1 0 1

$(0.125)_{10}$ =

0	0.125	
	× 2	
0	0.250	
	× 2	
1	0.500	
	× 2	
0	*.000	
	× 2	
	0.000	

= 0010̄

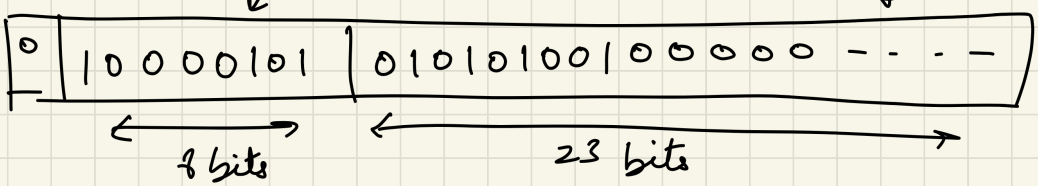
$85.125 = 1010101.001$

$= 1.010101001 \times 2^6$

$6 + 127 = 133$

$133 = 128\ 64\ 32\ 16\ 8\ 4\ 2\ 1$

$1\ 0\ 0\ 0\ 0\ 1\ 0\ 1$

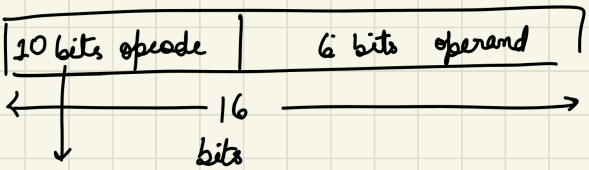


$0100\ 0010\ 1001\ 0100\ 1000\ 0000\ 0000\ 0000$

$4\ 2\ A\ A\ 4\ 0\ 0\ 0$

Q3 b) K two operand instruction

L zero operand instruction



two operand instructions $= 2^{10} = 1024$

number of one operand instructions $= \dfrac{2^{16} - L - 2^{12}K}{2^6}$

2^{16} = total encoding possible

L = zero operand instruction

$2^6 \times 2^6 \times K$ = two " "

Q4 a) PC = F2C → E78F → 895C → FFF0

ISZ : Increment and skip if zero

$M[AR] \leftarrow M[AR] + 1$

if $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

execution : D_6T_4 : $DR \leftarrow M[AR]$

D_6T_5 : $DR \leftarrow DR + 1$

D_5T_6 : $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC+1$, $SC \leftarrow 0)$

Timing Signal	PC	AR	DR	IR
Initial	F2C			
T_0	F2C	F2C		
T_1	F2C	F2C		E78F
T_2	F2C	78F		E78F
T_3	F2C	95C		E78F
T_4	F2C	95C	FFF0	E78F
T_5	F2C	95C	FFF1	E78F
T_6	F2C	FF1	FFF1	E78F

Address	Memory
F2C	E78F
78F	895C
95C	FFF0

Q) PC — X_1 — X2 — X3 — X4

UCS510(2) :

Q1(a)

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$F = A'BC + AB'C + ABC' + ABC$

$= BC + A(B \oplus C)$

b) 1 10000011 010... (22 zeroes) = ( )$_{10}$

$2^7 + 2 + 1$

131

$2^{22} = 4194304$

(-) $131 - 127 = 4$

2^4

$-1.\underline{4194304} \times 2^4$

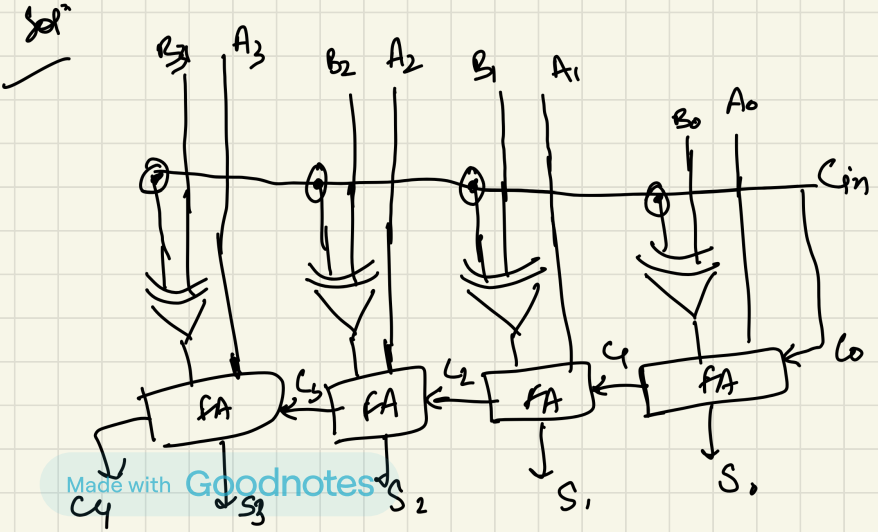
$-1.4194304 \times 16 = -22.7108864$

Q2. Adder/subtractor with C_{in} & two inputs A & B.

when $C_{in} = 0 \longrightarrow A + B$

$C_{in} = 1 \longrightarrow A - B$

Sol^n



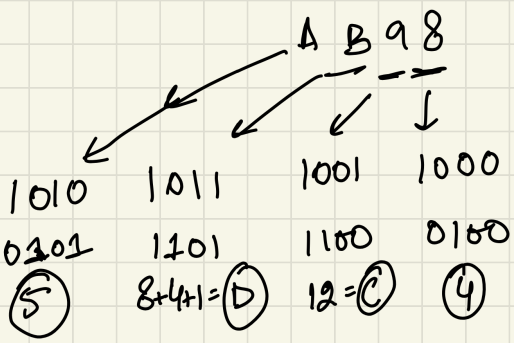
03 b) AC ← AB98 , E = 0 , IR ← 7200

PC = 2FF

	AC	IR	AR	E	
Initial	AB98	7200		0	
CMA	5467	7200	200	0	complement AC
CME	AB98	7100	100	1	complement E
		7080	080	0	

CIR

CIR, CIL ?



05 a) "load to AC" , Address = 300, 301 , address = 600

first word — second word — address field

Mode	Address

PC = 300, R1 = 500 XR = 200

M[499] = 550 M[600] = 900 M[800] = 650

M[500] = 800 M[700] = 1000 M[802] = 350

M[900] = 400 M[902] = 700

Sol" i) Direct address:

Instruction specifies an address field = 600

⇒ EA (effective address) for = 900
direct address mode

AC is loaded with content of memory location 600.

⇒ AC = 900

ii) Immediate Operand:

In immediate mode, second word of instruction is the actual data.

AC is loaded with address part directly.

AC = 600.

iii) Indirect operand:

M[600] = 900

M[900] = 400 ⇒ AC = 400

iv) Relative Address:

Address part in instruction + PC = 600 + 300 = 900

M[900] = 400 ⇒ AC = 400

v) Indexed Address
(using XR)
address field = 600

XR = 200
AC is loaded with sum of address part and content of XR.

600 + 200 = 800, M[800] = 650 ⇒ AC = 650

vi) Register Indirect:
In this mode, the address part of the instruction contains the address of memory location that contains the effective address

R1 = 500
M[R1] = 800
M[800] = 650
⇒ AC = 650

UCS510 (3):

Q3. Explain various addressing modes with eg.

(2)

PC	AC	AR	DR	IR	SC

?

PC = FFF , AC = 7EC3

→ 8BCD

→ B420

→ FFFF

Sol^n Memory reference instructions, six more times

i) AND → AND memory word to AC

	PC	AR	DR	AC	IR
Initial	FFF			7EC3	
AND	1000				

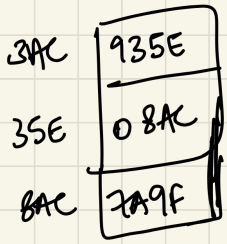
DR ← M[AR]

AC ← AC ∧ DR

FFF + 1 = 1000

UCS310(4) :

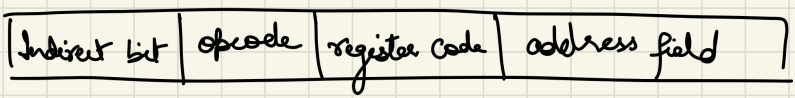
Q1 a) PC = 3AC, AC = 7EC3



opcode of ADD = 001
AND = 010

Sol" - what instruction will be fetched & executed next.

The instruction that will be fetched & executed next depends on content of memory at 3AC. The current value of PC is 3AC and content of memory at address 3AC is 935E. The opcode bits of ADD and AND is 001 and 010. Therefore instruction at memory address 3AC is not AND and ADD instruction.



935E = 1001 0011 0101 1110

- 1001 → Indirect bit code
- 001 → opcode
- 1 → register code
- 1110 → address field

Sol"

a) PC contains address of next instruction to be executed

⇒ instruction at 35C will be fetched & executed

b) content of 35C = 935E

935E = 1001 0011 0101 1110

3 5 E

- 1 → Indirect mode
- 001 → opcode ⇒ ADD
- 0011 0101 1110 → address

Address of operand = 35E

Address of operand at 35E is 08AC ⇒

operand is at 08AC ⇒ operand is 7A9F

DR ← M[AR]

AC ← AC + DR

AC = 7EC3

M[AR] = 7A9F

```
  7EC3
+ 7A9F
------
  3FCE
```

3+F = 12 (in hex) = C

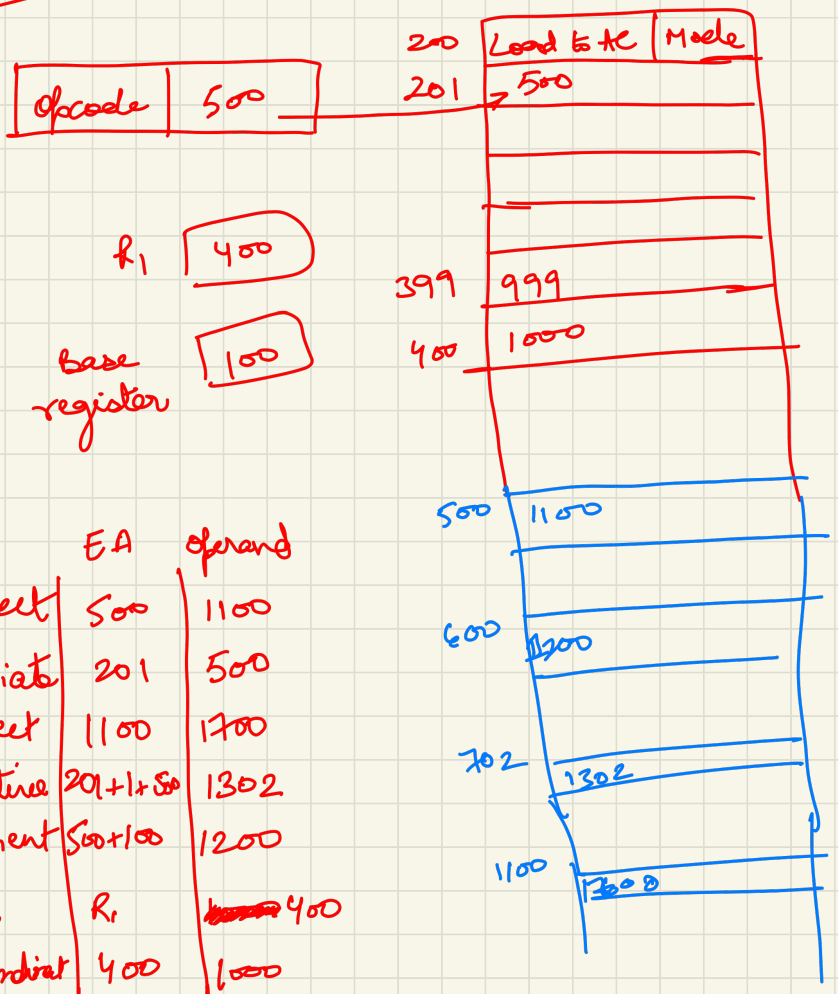
C+9 = 15 = F

E+A = 18 = 15+3 = 3

7+7 = E

12+9
21
16+5
10+14=24
16+8=18

Q2 Reference Q :



		EA	Operand
a)	Direct	500	1100
b)	Immediate	201	500
c)	Indirect	1100	1700
d)	PC relative	201+1+500	1302
e)	Displacement	500+100	1200
f)	Register	R_1	400
g)	Register Indirect	400	1000
h)	Autoindexing with inc. using R_1	400	1000

R_1 401

Q → diff b/w mem. reference instruction & register reference instruction using eg.

diff. b/w interrupted I/O and programmed I/O

Q: A = 01000001
B = 10000100

Sol: i) A = 65, B = 132 } unsigned

ii) A = 65, B = −124 } signed

10000100 = 01111011 + 1
= 01111100
= 124
⇒ (−124)

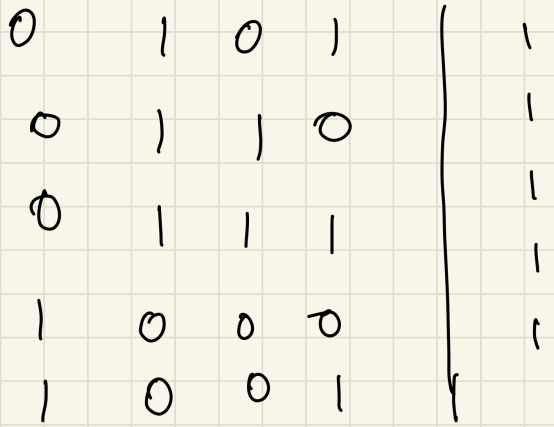
C, S, V, Z bit
- C → carry out of leftmost bit
- S → sign bit of result
- Z → whether result is zero
- V → whether there was an overflow

There is an overflow if the sum of two +ve no.s is −ve and sum of two −ve no.s is +ve.

UCS510(5)

Q1. a)

A	B	C	D	Output (Y)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0



(5)

b) PC = 16 bits, AC, DR, IR = 8 bits

Q2. a) R_1 = 1100

R_2 = 1010

1001 0 V=1

b) K-map

Q3 a) 64 registers and uses 16 bit instruction format

(B) I type : opcode + Register + 4 bit immediate value

(A) R type : opcode + 2 n names

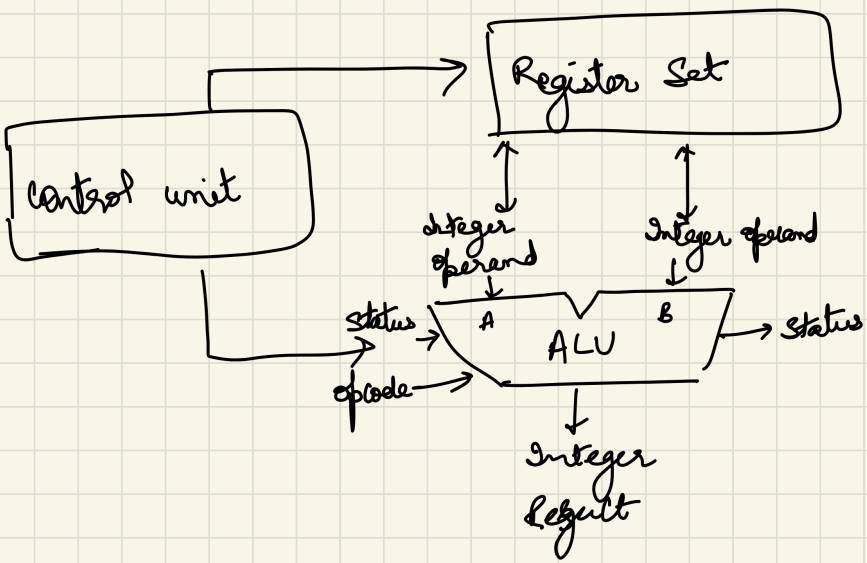
I type uses → 6 bit for opcode

$8 \times 6 = 56$ remaining opcodes

R type uses → 4 bit for opcode

$$\frac{56}{4} = 14$$

Unit- 4 Central processing unit :



Register set stores the intermediate data during the execution of instructions.

ALU performs the required micro-operations for executing the instructions.

The control unit supervises the transfer of information among the registers and instructs ALU as to which operation to perform.

- Advantage of having many registers:

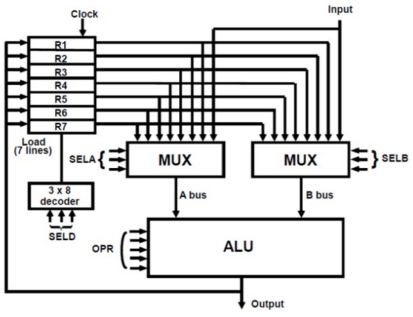
① Transfer b/w registers within processors is really fast.

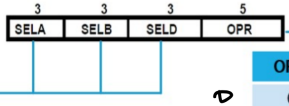
② Going "off the processor" to access memory is slow.

Bus organization for 7 CPU Registers: (slide 8)

A BUS ORGANIZATION FOR SEVEN CPU REGISTERS

- The control unit directs the information flow through ALU by
  - Selecting various components in the system.
  - Selecting the Function of ALU.
- Example: R1 ← R2 + R3
  1. MUX A selector (SELA): BUS A ← R2.
  2. MUX B selector (SELB): BUS B ← R3.
  3. ALU operation selector (OPR): ALU to ADD.
  4. Decoder destination selector (SELD): R1←Out Bus
- The entire information is passed through the **Control Word**.





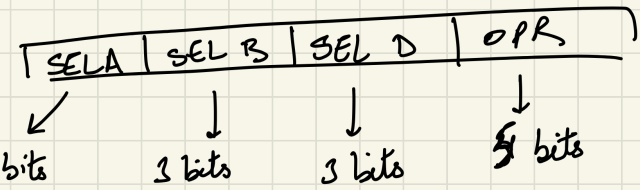
ALU operations encoding

Register Selection Field Encoding

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

	OPR Select	Operation	Symbol
0	00000	Transfer A	TSFA
1	00001	Increment A	INCA
2	00010	Add A + B	ADD
5	00101	Subtract A - B	SUB
6	00110	Decrement A	DECA
8	01000	ADD A and B	AND
10	01010	OR A and B	OR
12	01100	XOR A and B	XOR
14	01110	Complement A	COMA
16	10000	Shift right A	SHRA
24	11000	Shift left A	SHLA

- Control word format : 14 bits



Binary code	SELA	SELB	SEL D
000	R_1	R_1	R_1
001	R_2	R_2	R_2
⋮	⋮	⋮	⋮

OPR Select	Operation	Symbol
00000	Transfer A	TSF A
⋮	⋮	⋮

- Stack Org. : (push/pop operation)

ordered linear list, LIFO

Stack pointer = (top pointer of stack) pointing to the register storing the address of topmost element

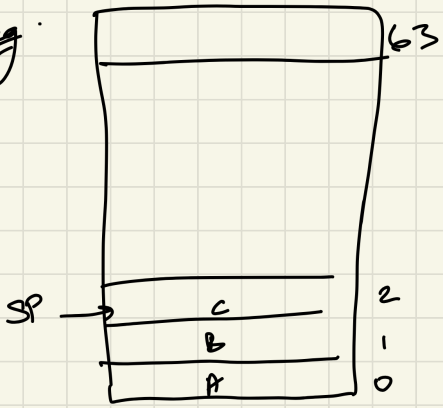
Types of stack org :

Register stack : It is built using a register.

Memory stack : It is logical part of memory allocated as stack. The logically partitioned part of RAM is used to create stack.

- Register Stack Org :-

eg.



stack of registers

64 registers

address = (0, 1, 2 ... 63)

denoted by

SP = stack pointer of 6 bits

$\because 2^6 = 64$

made of flip-flops ← Flags = FULL EMPTY

Full = 1, Empty = 0 ⟶ stack is full

DR is the data register through which data is transferred to and from the stack.

zero address instructions are used in register stack org. i.e. the address that doesn't contain address of operands.

- Push operation & Pop operation code

PUSH Operation

For the PUSH operation initially,

SP ← 0

Full ← 0

EMTY ← 1

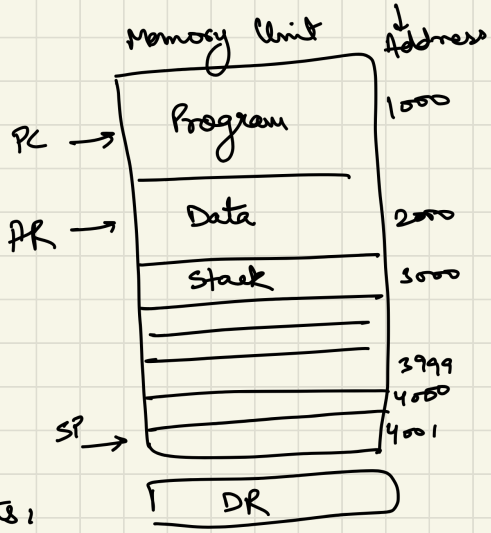
SP ← SP+1	Increment stack pointer
M[SP] ← DR	Write on top of the stack
If (SP = 63) then (FULL ← 1)	Check if the stack is full
EMTY← 0	Stack is not empty

POP Operation

DR ← M[SP]	Read from the top of the stack
SP← SP-1	Decrement the stack pointer
If (SP = 0) then (EMTY←1)	Check if the stack is empty
FULL ← 0	Stack is not full

- Memory Stack Org. :

a processor register is used as a stack pointer. Stack pointer is a CPU register that specifies the stack's initial memory.



RAM is divided into three parts:

① Program — part of RAM where programs are stored

② Data — " " " " data is "

③ Stack — part of " used to implement stack

- Most computers don't provide hardware to check overflow / underflow, so upper limit / lower limit registers are used. (software)

PUSH Operation	
SP ← SP -1	Decrement stack pointer
M[SP] ← DR	Write on top of the stack

POP Operation	
DR ← M[SP]	Read from the top of the stack
SP← SP + 1	Increment the stack pointer

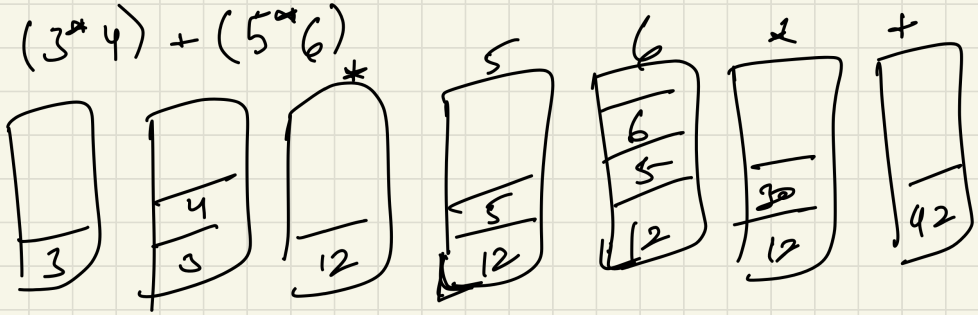
- Reverse Polish Notation:

Post fix notation are used by stack org. & prefix are used by general register org:

eg: $(A+B)^* (C^*(D+E)+F)$

postfix : AB + CDE * + F * +

- $(3^*4) + (5^*6)$



- Processor Organization:

1) Single register organization (Accumulator)
2) General register organization → Any register can be used as source or dest.
3) Stack organization (Hardware stack) → All operations are done using hardware stack

- Instruction format :

Machine instructions contain no. of bits. These bits are grouped together & called fields

- Single Accumulator org : use only one address field

eg. $AC \leftarrow AC + M[x]$

the instruction for arithmetic add is defined by an instruction "ADD".

General Register Org. : uses two or three address fields

An instruction symbolized by ADD, R_1, X specifies the operation $R_1 \leftarrow R_1 + M[x]$

This instruction has two address fields : R_1 and x.

Stack Organization : Doesn't require address field as the operation is performed on the two items on top of stack using push and pop operations

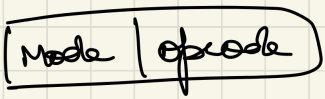
- Instruction format representation :

← Machine instruction

Addressing mode	Opcode	Operand

- Instruction format type : 0, 1, 2, 3 address inst format

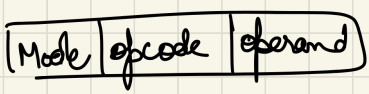
- Zero address instruction format :



Stack organization supports this
doesn't have operand field, they are implicitly defined
arithmetic exp. ⟶ RPN

One address instruction format : uses only one address field

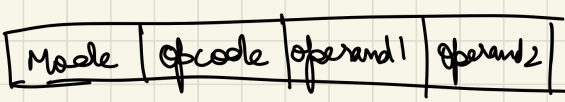
One operand is in the accumulator and other is in memory location.



It has only one operand.
It has two special instruction LOAD and STORE.

Load : transfer data to accumulator
store : move data from accumulator to memory

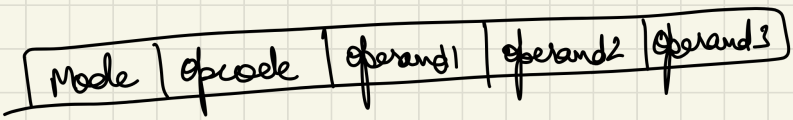
two address instruction :



has two operands
requires shorter assembly language instructions
result can be stored in different locations

MOV : instruction to transfer the operands to the memory from process registers

Three address instruction :



The operand1 and operand2 contain the data or address that the CPU will operate. Operand 3 contains the result's address

- Addressing Modes:

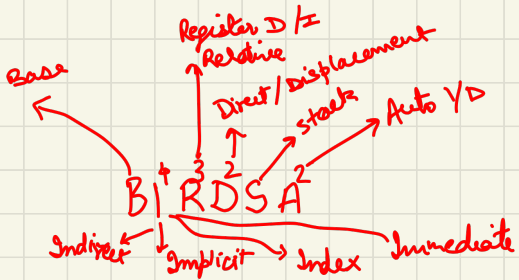
Addressing modes define the rules and mechanisms by which the processor calculates the effective memory address or operand location for data operations.

The purpose for using addressing modes :

① to provide versatility to the user
② to reduce the number of bits in addressing field of inst.

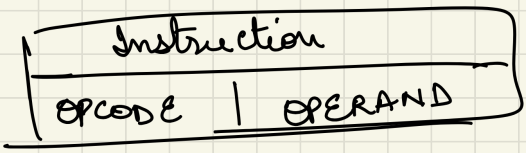
Types of Addressing Modes :

① Implied / Implicit : Stack
eg. CMA, CLA, PUSH, POP

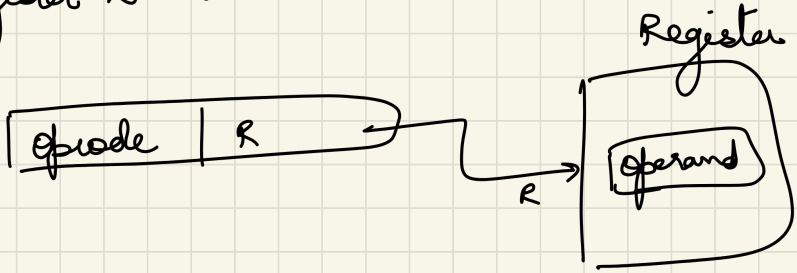


(2) Immediate :

eg. ADD 8 will increment the value stored in the accumulator by 8.

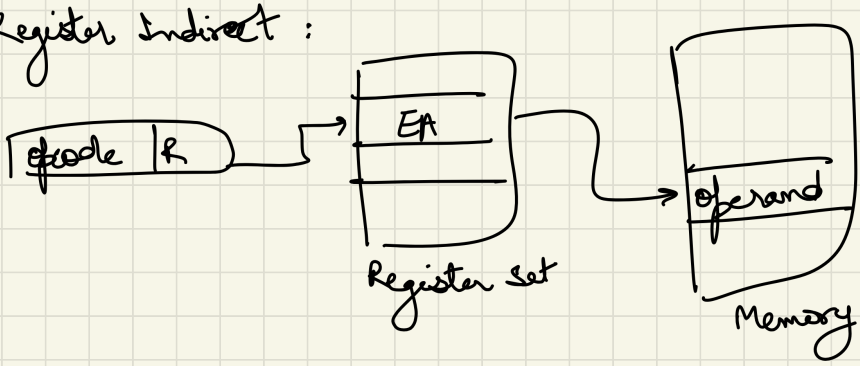


(3) Register Direct :



eg. MOV R1, 35H

(4) Register Indirect :

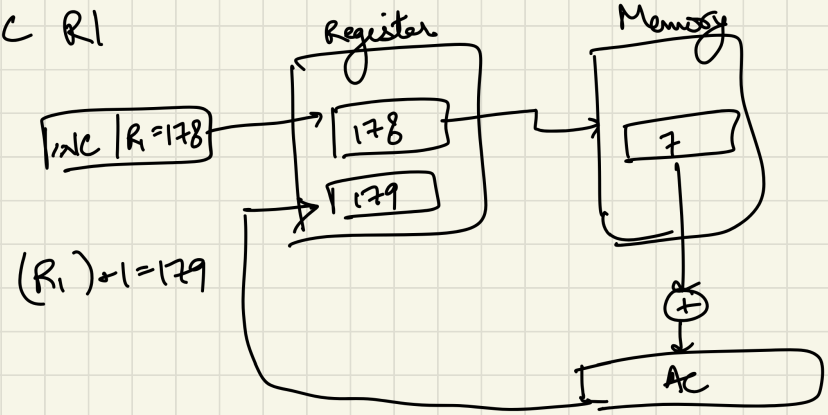


eg. MOV R1, [R2]

(5) Auto-increment :

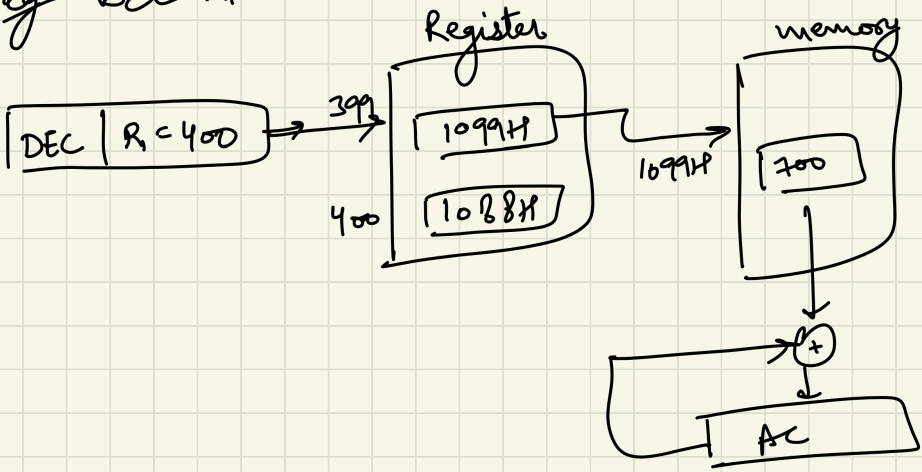
follows a post increment approach

eg: INC R1



(b) Auto-Decrement : pre decrement approach

eg. DEC R1



(7) Direct add. mode : eg: ADD, [1000H] (1 reference)

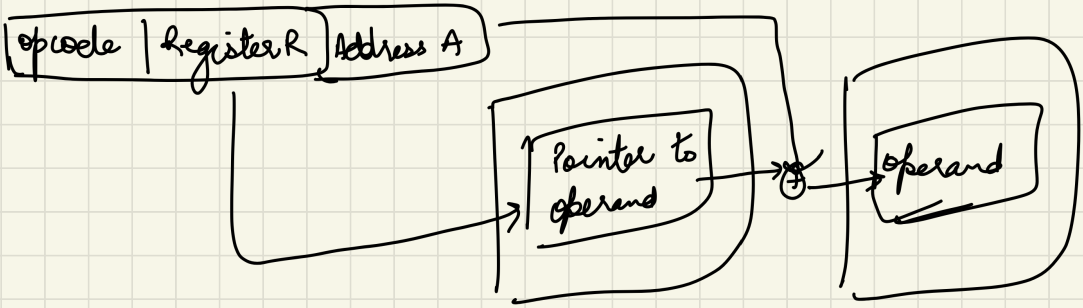
(8) Indirect add. mode : eg. ADD M (2 references) slow

⑨ Displacement Addressing Mode :

eg: MOV R1 , [1000H + 09H]

EA = A + R

R : displacement value



⑩ Relative addressing mode :

eg: MOV R1 , [PC + Address field]

EA = A + PC

⑪ Indexed Addressing Modes : The index register's content is added to instruction's address to obtain the effective address.

eg: MOV R1 , [S1 + 1000H]

EA = content of index register + instruction address part

⑫ Base Register : EA = A + (R)

A : Inst. add.

R : pointer to base address

⑬ Stack Addressing modes

- Applications, adv. & disadv. of addressing modes

APPLICATIONS OF ADDRESSING MODES

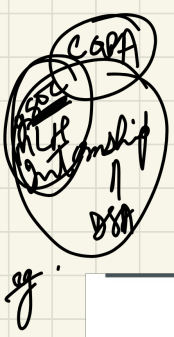
Addressing Mode	Applications
Immediate Addressing Mode	Initialize the register to a constant value.
Direct Addressing Modes Register Direct Addressing Mode	Helps access static data and implement variables.
Indirect Addressing Modes Register Indirect Addressing Mode	Helps implement pointers and pass arrays as parameters.
Relative Addressing Mode	Helps in program relocation at runtime. And in changing the sequence of instructions during execution.
Index Addressing Mode	Helps in the array and record implementation.
Base Register Addressing Mode	Helps in writing relocatable code and handling recursive procedures.
Auto-Increment Addressing Mode Auto-Decrement Addressing Mode	Helps implements loops and stacks.

Adv : improve efficiency, reduce latency

used to implement complex ds

program sizes can be reduced

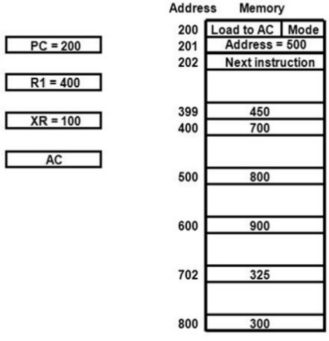
gives you diff. way of specifying address of operands



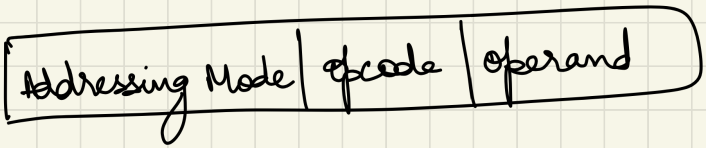
eg.

EXAMPLE: ADDRESSING MODES

Addressing Mode	Effective Address	Content of AC
Immediate AM	201	500
Register AM	–	400
Register Indirect AM	400	700
Auto-Increment AM	400	700
Auto-Decrement AM	399	450
Direct AM	500	800
Indirect AM	800	300
Relative AM (EA=PC + Address Field)	702	325
Base Register AM (EA=XR + Address Field/)	600	900
Indexed Register AM (EA=XR + Address Field/)	600	900



- Data Transfer and Manipulation :



Types of instructions :

1. Data Transfer Instructions
2. Data Manipulation Instructions
   - Arithmetic
   - Logical & bit manipulation
   - Shift
3. Program Control Instruction

- Data transfer inst. : They move data from one location to another. also called copy inst. (without changing binary content)

- Program control :

Program control instructions modify or change flow of a program.

It changes value of PC leading to change in execution.

features :

i) conditional / unconditional

ii) flags / control flow & jump inst. called jumps, calls, returns

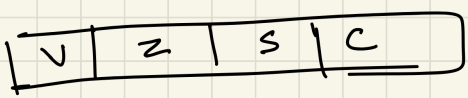
iii) Subroutine & subroutine handling

iv) Loop & loop handling inst.

PROGRAM CONTROL INSTRUCTIONS

Program Control Instructions	Description
Branch (BR)	Branch which means it is an unconditional jump. It is unconditional branching wherever we specify the address we need to branch.
Skip (SKP)	Skip instructions is used to skip one(next) instruction. It can be conditional or unconditional. It does not need an address field. In the case of conditional skip instruction, the combination of conditional skip and an unconditional branch can be used as a replacement for the conditional branch.
Jump (JMP)	The jump instruction transfers the program sequence to the memory address given in the operand based on the specified flag.
Compare (CMP)	The Compare instruction performs a comparison via a subtraction, with difference not retained. CMP compares register sized values, with one exception.
CALL and RETURN	The CALL and RETURN instructions interrupt the flow of a program by passing control to an internal or external subroutine. An external subroutine is another program. The RETURN instruction returns control from a subroutine back to the calling program and optionally returns a value.
TEST	TEST instructions perform the AND of two operands without retaining the result, and so on.

- status bit / flag / processor status word :

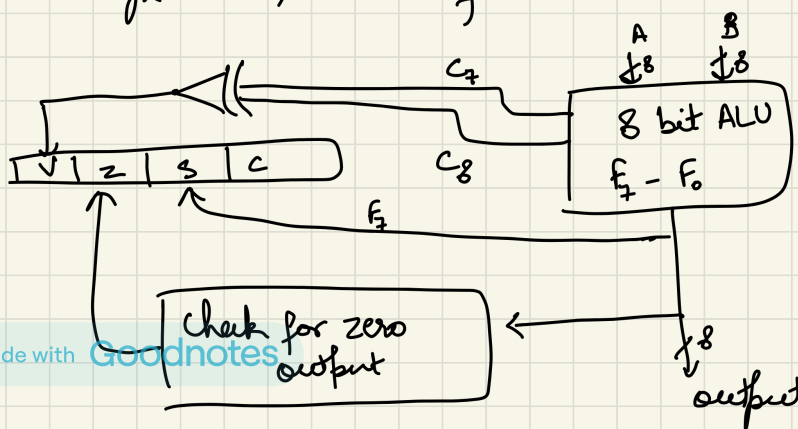


V = overflow

Z = if ALU output = 0 then Z = 1 otherwise Z = 0

C = if ALU output gives carry " C = 1 " C = 0

S = Sign bit , S = 0 if +ve



- Conditional branch inst : check the condition and branch

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed compare conditions (A - B)	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B

- Subroutine call and return :

perform a self-contained sequence of instructions that performs a computational task.

names for instruction that transfer program control : Call subroutine, jump to subroutine, branch to subroutine, branch & save the address

- Most imp. operations : ① branch to the begining of subroutine
② Save the address (return) to get address of location in calling program upon exit from subroutine

- locations for storing return addresses : in subroutine, in memory, processor register, ✓memory stack

eg. In memory stack

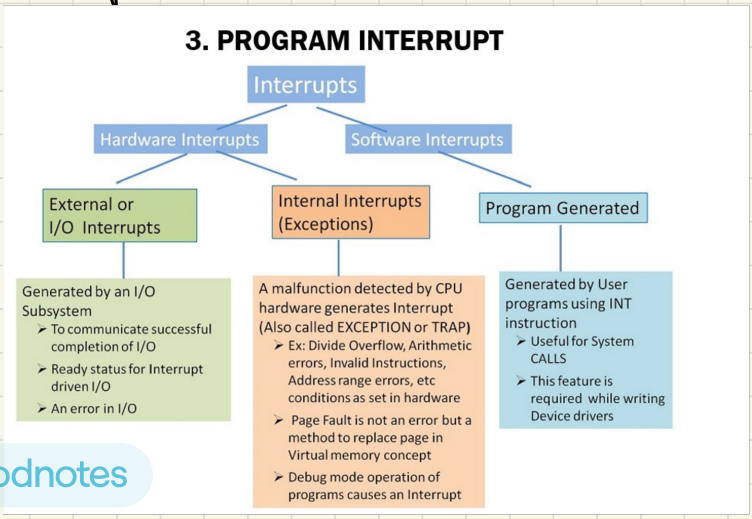
```
CALL
  SP ← SP-1
M[SP] ← PC
  PC ← EA

RTN
  PC ← M[SP]
  SP ← SP+1
```

- Program Interrupt :

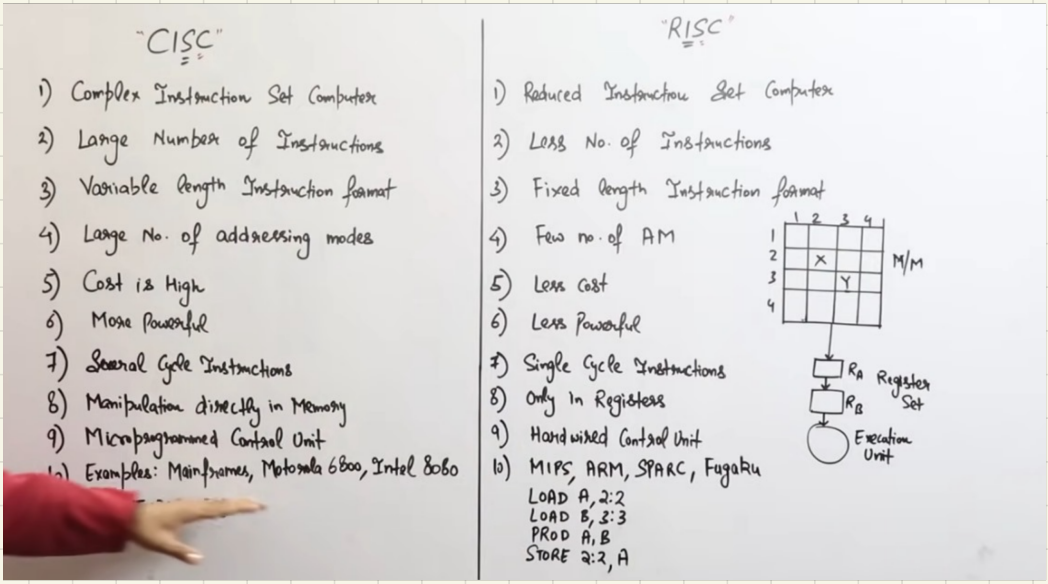
An interrupt is a signal emitted by device attatched to a computer, which temporarily stops or terminates a service of the current process

- Types of interrupts : External, Internal, Software



- Interrupt Procedure:

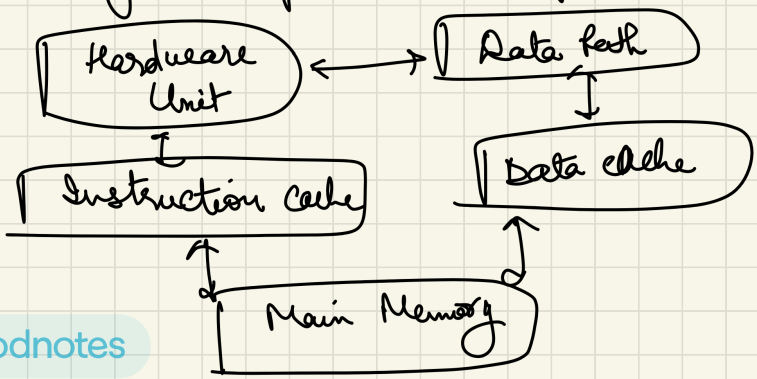
– RISC and CISC :



– RISC :

RISC is a microprocessor architecture that uses a simple set of instructions than can be substantially modified.

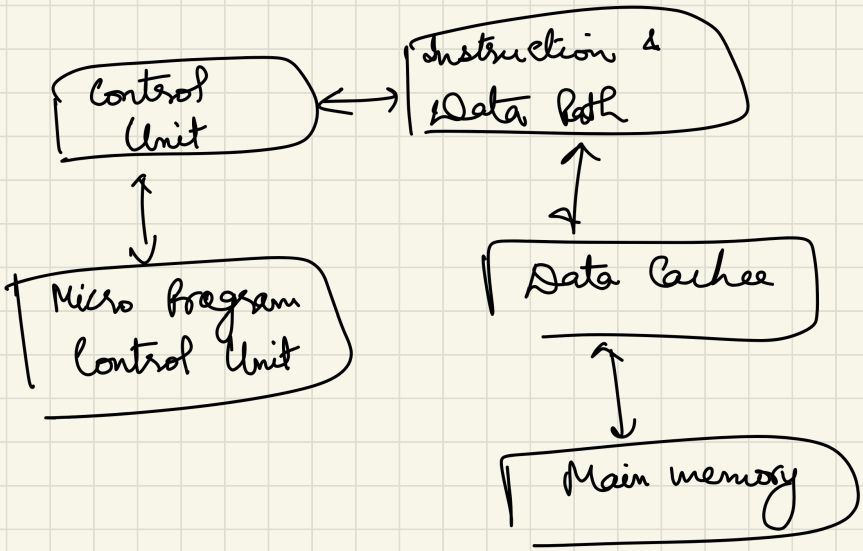
each instruction cycle has one clock per cycle and each cycle consists of three parameters : fetch, decode, execute.



features of RISC processors:

1. RISC use 1 CPI (1 clock per instruction cycle). each CPI comprises of fetch, decode, execute.
2. Multiple registers in RISC allow them to hold instruction safely fast to computer.
3. RISC uses pipelining to execute multiple parts of instruction efficiently.
4. RISC has simple addressing mode & fixed instruction length for pipeline execution.
5. Uses LOAD and STORE to access memory location.

- CISC:



features :

- CISC may take longer than a single clock cycle to execute code.
  Length of code is short, so it requires minimal RAM.
  Provides more accessible programming in assembly language.
  focuses on creating instructions on hardware rather than software, ∴ they are faster to develop.
  It comprises of fewer registers and more addressing modes, typically 5 to 20.

DIFFERENCE BETWEEN RISC AND CISC

	RISC	CISC
1	**Instructions of a fixed size**	Instructions of **variable size**
2	Most instructions take **same time to fetch**.	Instructions have **different fetching times**.
3	Instruction set **simple and small**.	Instruction set **large and complex**.
4	**Less addressing modes** as most operations are register based.	**Complex addressing modes** as most operations are memory based.
5	**Compiler design is simple**	**Compiler design is complex**
6	**Total size of program is large** as many instructions are required to perform a task as instructions are simple.	**Total size of program is small** as few instructions are required to perform a task as instructions are complex & more powerful.
7	Instructions use a **fixed number of operands.**	Instructions have **variable** number of operands.
8	Ideal for processors performing a **dedicated operation**.	Ideal for processors performing a **verity of operations**.
9	Since instructions are simple, they can be decoded by a **hardwired control unit**.	Since instructions are complex, they require a **Micro-programmed Control Unit**.
10	Execution speed is **faster** as most operations are register based.	Execution speed is **slower** as most operations are memory based.
11	As No. of cycles per instruction is fixed, it gives a **better degree of pipelining**	Since number of cycles per instruction varies, **pipelining has more bubbles** or stalls.
12	E.g.:: **ARM7**, **PIC 18** Microcontrollers.	E.g.: Intel **8085, 8086** Microprocessors.

Pipelining (Unit-4) :

In non-pipelined (sequential) architecture, all the instructions of a program are executed sequentially one after the other
Pipelining is referred as
A technique in which a given task is divided into a number of subtasks that need to be performed in sequence.
One of the processes of arranging the hardware so that simultaneous execution of multiple instructions takes place, thus, improving the overall performance.
The main advantage of pipelining is the simultaneous execution of various subtasks, which improves the system's throughput.

In pipelined execution, the first instruction will take T clock cycles to process, but the other instructions will take just 1 more clock cycle.
The speedup can be calculated as:
Speedup (S) = Cycles of non-pipelined processor/ cycle of pipelined processor
$S = (n*t\_n)/((k+n-1)\ [\![\ *t]\!]\ \_p\ )$
n = number of tasks
k = k = number of segments pipeline
tn= clock cycle time (non-pipelined)
tp= clock cycle time (pipelined)

$$S = \frac{n t_n}{(n+k-1) t_p}$$

As the number of tasks increases,
Take the limit as n approaches to infinity, (n+k-1) approaches to n, resulting in theoretical speedup (max) of:
$\lim_{\top}[\![\ n \to \infty]\!]\quad Smax = t\_n/t\_p$
If time it takes to process a task is the same in pipelined and non-pipelined circuits then,
$Smax = (k*t\_p)/t\_p = k$

$$S_{max} = \frac{t_n}{t_p}$$

Instruction pipelines are used to divide the task of executing a stream of instructions into subtasks to be executed in different pipeline segments to improve the throughput of the computer system.
For example, if we have a stream of instructions, then one segment of the pipeline can read the instructions while another segment can decode the previous instruction. In this way, more than one instruction will be handled simultaneously by the computer system which will improve its throughput. The instruction pipeline will be more efficient if the instructions are divided into equal-duration segments.

A typical example of an instruction pipeline used by computer systems consists of the following segments:
Segment 1: This segment will fetch the instruction from the memory
Segment 2: This segment will decode the instruction and find out the effective address
Segment 3: This segment will fetch the operands from the memory
Segment 4: This segment will execute the instruction

Types of pipelining — ① Instruction
② Arithmetic

↓
read from slides
(do the diagram)

- There are several stages of processing an instruction. A pipeline can be of three, four, five, or six stages.

3-stage
Fetch
Decode
Execute

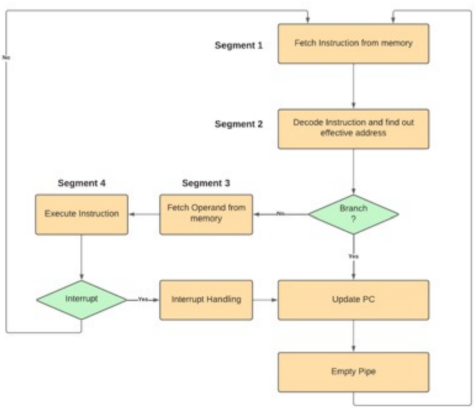
4-stage
Fetching the instruction
Decoding the instruction
Executing the instruction
Write Back

5-stage
Fetching the instruction
Decoding the instruction
Memory Access for operands
Executing the instruction
Write Back

6-stage
Fetching the instruction
Decoding the instruction
Calculate the effective address of the operand
Memory Access for operands
Executing the instruction
Write Back

Doubt!!

INSTRUCTION PIPELINING



Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction 1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
(Branch) 3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

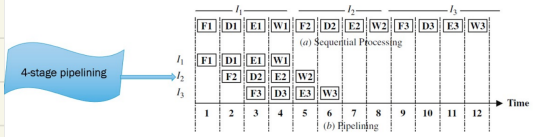
Pipelining Hazards: situations that prevent the next instruction in the instruction stream from executing during its designated clock cycles.

- Data Hazards
  Control Hazards or Instruction hazards
  Structural Hazards

- Sequential Vs pipelined execution of instructions :

SEQUENTIAL VS. PIPELINED EXECUTION OF INSTRUCTIONS

- Consider that there are three instructions- I1, I2, and I3 and there are 4 stages of execution – Fetch (F), Decode (D), Execute (E), and Write back (W).
- It takes 12 machine cycles to execute these three instructions in **sequential processing** and only 6 machine cycles in **pipelining**.



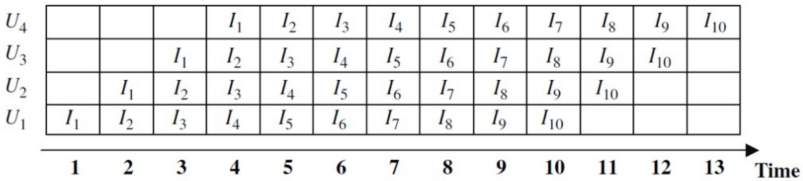
Pipelining versus sequential processing

Performance Measures for goodness of a pipeline :

PERFORMANCE MEASURES FOR THE GOODNESS OF A PIPELINE

- In order to formulate the performance measures for the goodness of a pipeline in processing a series of instructions
  - A space-time chart (called the Gantt's chart) is used.
  - In this chart, the vertical axis represents the segments (four in this case) and the horizontal axis represents time (the time (T) taken by each subunit to perform its task is the same, therefore, known as unit time)

13 time units are required to execute 10 instructions using 4-stage pipelining



The space–time chart (Gantt chart)

- Data Hazard :

It occurs when either source or destination operands of an instruction are not available at the time expected in the pipeline.

A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result, some operation has to be delayed, and the pipeline stalls.
When the execution of an instruction is dependent on the results of a prior instruction that's still being processed in a pipeline, data hazards occur. If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.

Flow/True Data Dependency [RAW (or Read after Write)]:
This is when one instruction makes use of data from a previous instruction.
Example,
ADD X0, X1, X2
SUB X4, X3, X0

Anti-Data Dependency [WAR (or Write after Read)]
When the second instruction is written to a register before the first instruction is read, this is known as a race condition. In the case of a simple structure of a pipeline, this is uncommon. WAR, on the other hand, can occur in some machines having complex and specific instructions.
Example,
ADD X2, X1, X0
SUB X0, X3, X4

Output data dependency [WAW (or Write after Write)]
This is a situation where two simultaneous instructions must write the same register in the same sequence they were issued.

Output data dependency [WAW (or Write after Write)]
This is a situation where two simultaneous instructions must write the same register in the same sequence they were issued.
Example,
ADD X0, X1, X2
SUB X0, X4, X5

WAW and WAR hazards can only occur when instructions are executed in parallel or out of order. These occur because the same register numbers have been allotted by the compiler although avoidable.
This situation is fixed by renaming one of the registers by the compiler or by delaying the updating of a register until the appropriate value has been produced.
Modern CPUs not only have incorporated Parallel execution with multiple ALUs but also out of order issues and execution of instructions along with many stages of pipelines.

Structural Hazard

Hardware resource conflicts among the instructions in the pipeline cause structural hazards. Memory, a GPR Register, or an ALU might all be used as resources here.
When more than one instruction in the pipe requires access to the very same resource in the same clock cycle, a resource conflict is said to arise.
In an overlapping pipelined execution, this is a situation where the hardware cannot handle all potential combinations.

Solution: For a portion of the pipeline, instructions must be performed in series rather than parallel.

Control Hazard

Control hazards are called Branch hazards and are caused by Branch Instructions. Branch instructions control the flow of program/ instructions execution. Recall that we use conditional statements in the higher-level language either for iterative loops or with conditions checking (correlate with for, while, if, and case statements). These are transformed into one of the variants of BRANCH instructions. It is necessary to know the value of the condition being checked to get the program flow.
Thus a Conditional hazard occurs when the decision to execute an instruction is based on the result of another instruction like a conditional branch, which checks the condition's resultant value.
The branch and jump instructions decide the program flow by loading the appropriate location in the Program Counter(PC). The PC has the value of the next instruction to be fetched and executed by CPU. Consider the following sequence of instructions.

Solution For control Hazards :

Stall:
SPRD
Stall the given pipeline as soon as any branch instructions are decoded. Just don't allow IF anymore. Stalling reduces throughput as it always does. According to statistics, at least 30% of the instructions in a program are BRANCH. With Stalling, the pipeline is effectively operating at 50% capacity.

Prediction:
Consider a for or a while loop that is repeated 100 times. We know the program would run 100 times without the given branch condition being met. The program only exits the loop for the 101st time. As a result, it's better to let the pipeline run its course and then flush/undo when the branch condition is met. This has less of an impact on the pipeline's throttle and stalling.

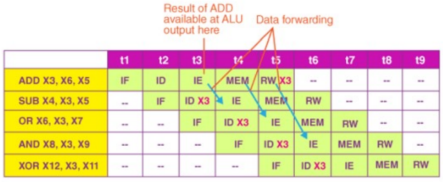
pt

Dynamic Branch Prediction :
A history record is maintained with the help of Branch Table Buffer (BTB). The BTB is a kind of cache, which has a set of entries, with the PC address of the Branch Instruction and the corresponding effective branch address. This is maintained for every branch instruction that occurs.

Reordering Instructions:
Delayed branching entails reordering the instructions to move the branch instruction later in the sequence, allowing safe and beneficial instructions that are unaffected by the result of a branch to be brought in earlier in the sequence, delaying the fetch of the branch instruction. If such instructions are not available, NOP is used. The Compiler is used

1. DATA HAZARDS

- **Solution 1:** At the IF stage of the SUB instruction, add three bubbles. This will make it easier for SUB – ID (Instruction Decoder) to work at t6. As a result, all subsequent instructions in the pipe are similarly delayed.
- **Solution 2:** Forwarding of Data – Data forwarding is the process of sending a result straight to that functional unit that needs it: a result is transferred from one unit's output to another's input. The goal is to have the solution ready for the next instruction as soon as possible.



Reordering instructions: Delayed branching entails reordering the instructions to move the branch instruction later in the sequence, allowing safe and beneficial instructions that are unaffected by result of a branch to brought in earlier in the sequence, delaying the fetch of the branch instruction. If such instructions are not available, NOP is used. The compiler is used to implement this delayed branch.

Numericals :

eg. No. of clock cycles to process 200 tasks in a six-segment pipeline.

Sol" $(n + m - 1) = (6 + 200 - 1) = 205$

eg.

Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

Solution:

Segment	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8					
2		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8				
3			T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8			
4				T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8		
5					T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
6						T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8

(n + m – 1) = 6 + 8 – 1 = **13 cycles**

eg. t_n (non-pipeline) = 50ns

t_p (pipeline) = 10 ns

$n = 100$ (no. of task)

$k = 6$ (segments)

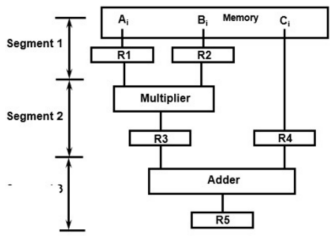
Sol^n

$$\text{Speedup}(S) = \frac{n^* t_n}{(n+k-1)t_p} = \frac{100 \times 500}{105 \times 10} = 4.76$$

$$\text{max}^m \text{ speedup } (S_{max}) = \frac{t_n}{t_p} = \frac{50}{10} = 5$$

eg. 05

QUESTION 5



Solution:

(a) $t_p = 45 + 5 = 50$ ns k = 3

(b) $t_n = 40 + 45 + 15 = 100$ ns

(c) $$S = \frac{nt_n}{(k+n-1)\,t_p} = \frac{10 \times 100}{(3+9)50} = 1.67$$ for n = 10

$$= \frac{100 \times 100}{(3+99)50} = 1.96$$ for n = 100

(d) $$S_{max} = \frac{t_n}{t_p} = \frac{100}{50} = 2$$

Unit - 5 (Memory Organization) :

Memory heirarchy

Main memory

Aux Memory

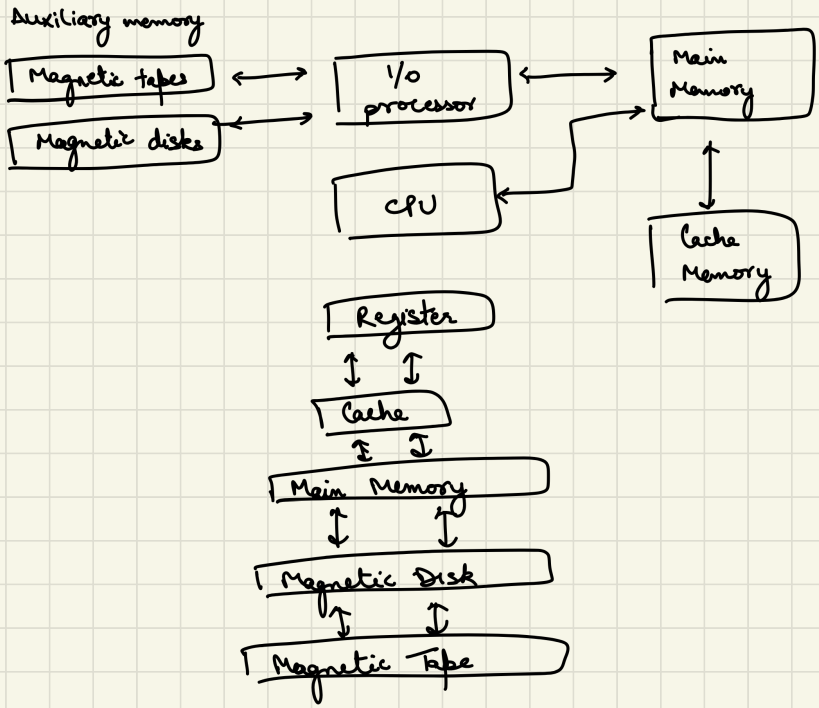
Assosciative Memory

Cache Memory

Virtual Memory

Memory Management Hardware

- Memory Heirarchy : Memory heirarchy is to obtain highest possible access speed while minimizing the total cost of memory system.



Main memory : (Primary mem / Volatile memory) is used to communicate directly with the CPU. CPU is mainly used to execute any task in the computer. CPU can execute only that program which will be present into the main

memory. If we are going to save any program by default it get saved into the secondary memory. but during execution DMA / OS transfers the program to the main memory.

Auxiliary memory: (Secondary mem. / Non-volatile memory) is used to store the data permanently into the comp. Eg are magnetic disk & tapes. CPU can't access the data of auxiliary memory.

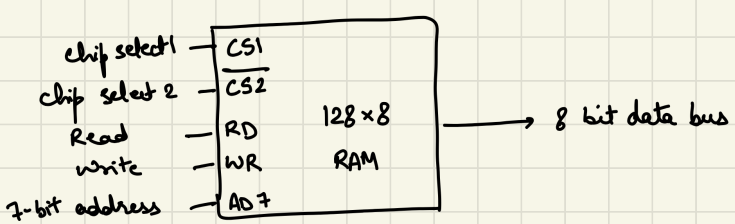
Cache memory: Small and faster memory which has higher accessibility.
used to store data which is frequently used.
Main memory access time is slow but cache memory access time is fast.
So if there are frequently used instructions then OS / I/O transfers the instructions of main memory to cache memory so that access will be fast.

I/O processor: is mainly used to transfer data from secondary memory to main memory.

- **Main memory:**

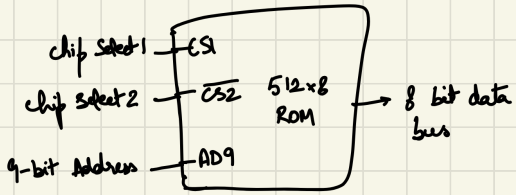
① Typical RAM chips

Types of memory
i) Static
ii) Dynamic



CS1	$\overline{CS2}$	RD	WR	Memory fn	State of data bus
0	0	x	x	Inhibit	High Impedence
0	1	x	x	Inhibit	High Impedence
1	0	0	0	Inhibit	High Impedence
1	0	0	1	write	Input data to RAM
1	0	1	x	Read	output data from RAM
1	1	x	x	Inhibit	High Impedence

Typic ROM chip



Uses of ROM

① Bootstrap Loader
② Computer startup

The **static RAM** consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. It is very fast but the cost is high. Cache memory is designed with the help of SRAM. The **dynamic RAM** stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles. Main memory is designed with the help of DRAM.

RAM is a volatile memory when we switch off the computer the entire contents of memory will be lost but ROM is a permanent memory that means it stores contents permanently. Most portion of the memory is RAM only but less is ROM.

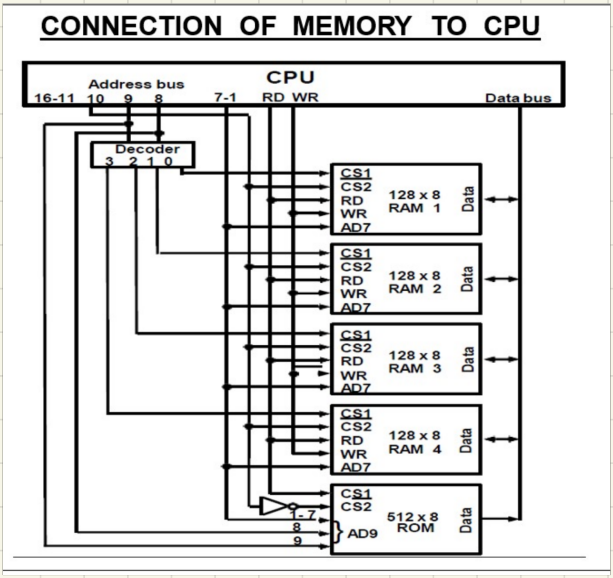
Bootstrap loader: Among other things, the ROM portion of main memory is needed for storing an initial program called a bootstrap loader. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use. RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024 x 8 memory constructed with 128 x 8 RAM chips and 512 x 8 ROM chips.

- Memory Address Map :

memory connection to CPU

- RAM and ROM chips are connected to a CPU through the data and address buses
- the low-order lines in the address bus select the byte within the chips & other lines in the address bus

select a particular chip through its chip select inputs



Auxiliary Memory:

① Info org. on magnetic tapes

② Organization of disk Hardware — Moving Head disk, fixed head disk

- characteristics of Auxiliary memory:

① Access mode

② Access Time

③ Transfer rate

④ Capacity

⑤ Cost

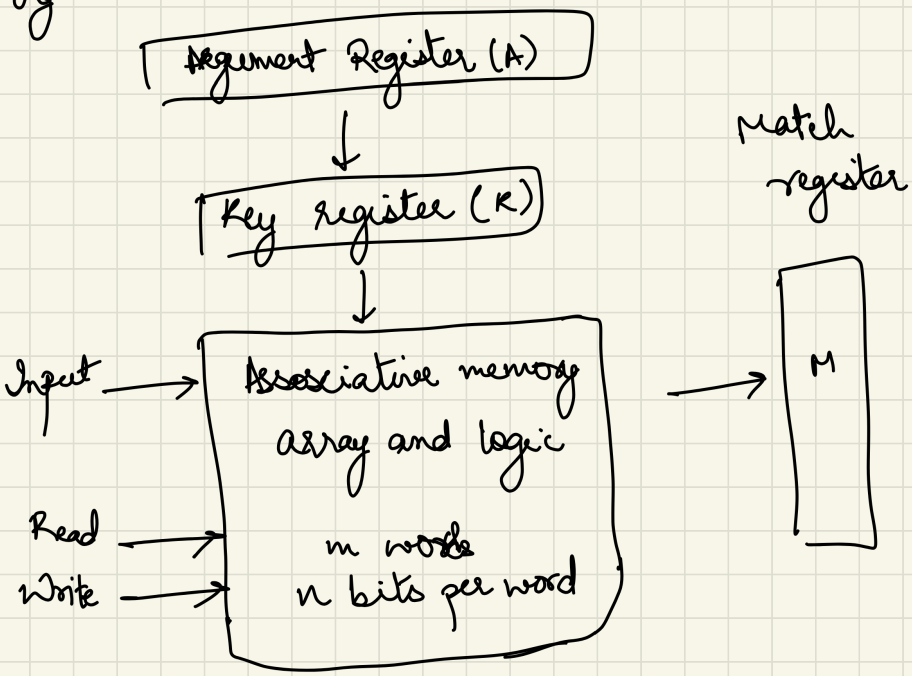
- Access Time : ① Seek Time

② Transfer Time

Associative Memory:

- Accessed by content of data rather than by an address
- Also called Content Addressable Memory (CAM)

Hardware org.



- Cache memory :

Locality of Reference

- The references to memory at a given time interval tend to be confined with localized areas.
- This area contains set of info. & the membership changes gradually over time.

Temporal locality : The info. which will be used in near future is likely to be in use already. (eg. Reuse of info. in loops)

Spatial Locality : If a word is accessed, near words are likely to be accessed soon (eg. Related data items (arrays) are usually stored together; instructions are executed sequentially)

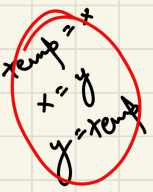
The property of Locality of References make Cache System work.

Performance of Cache :

① Memory Access — All the memory accesses are directed first to Cache.
If the word is in Cache, access Cache to provide it to CPU.
If " " " not ", bring a block including that word to replace one of the existing blocks.

② Performance :

$$T_e = T_c + (1-h)\,T_m$$



h = hit ratio

T_e = effective memory access time in cache memory

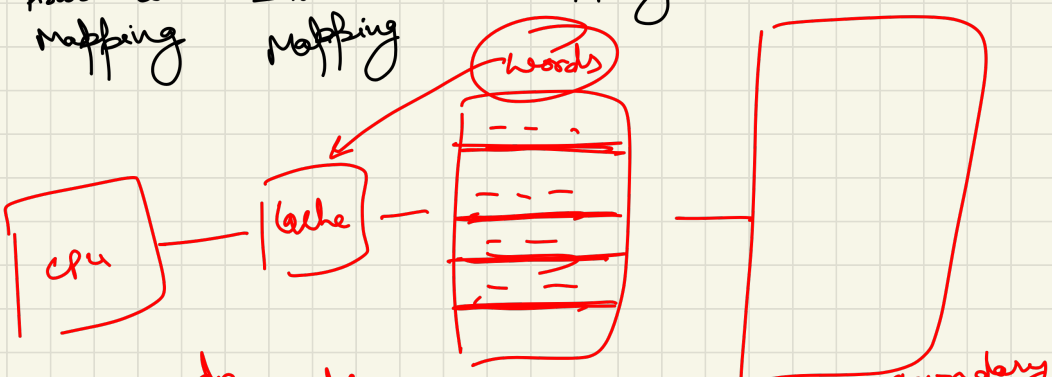
T_m = main memory access time

T_c = Cache access time

Memory and Cache Mapping :

Mapping fⁿ : specification of correspondance b/w main and memory blocks and cache blocks

- Associative mapping
- Direct Mapping
- Set-Associative Mapping

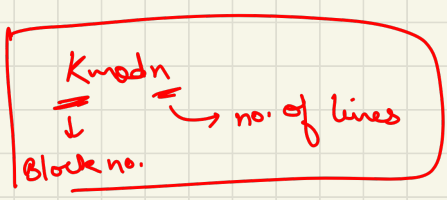


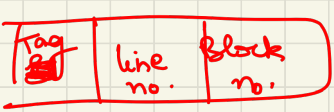
where do you place the words in cache → Mapping

→ hardware clock

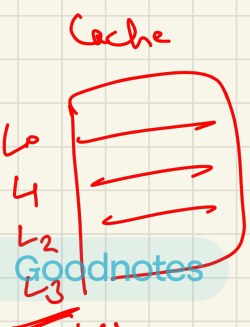
size of line = size of block

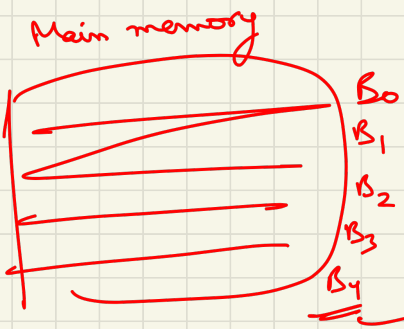
Direct mapping :





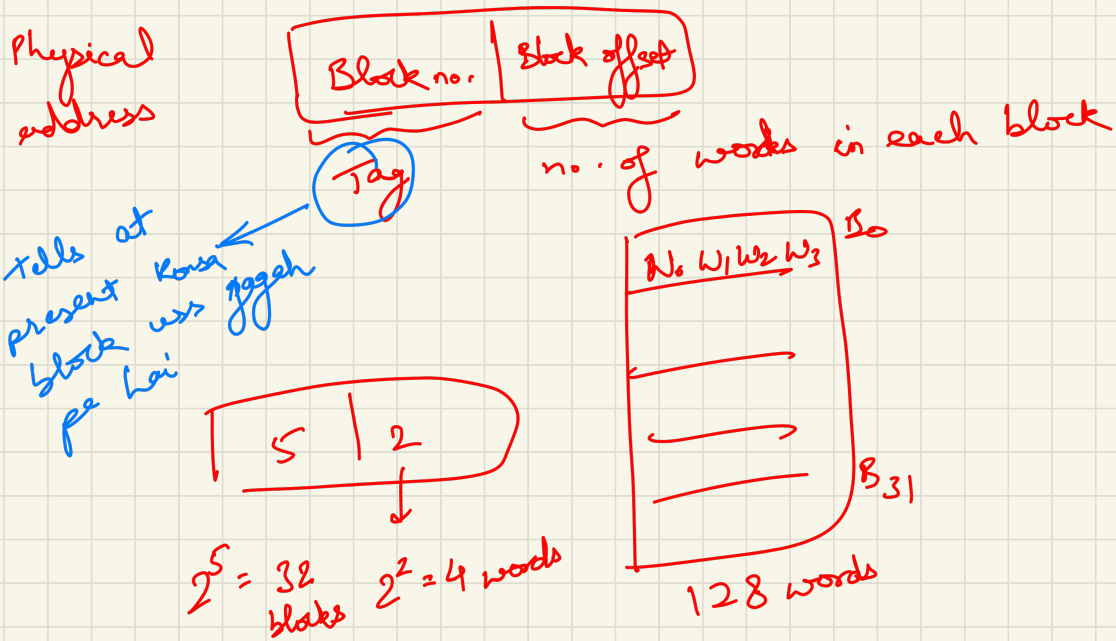
direct → no. of conflict miss is high





Fully associative mapping → koi bhi block kahi bhi aa sakta hai

↳ adv : No conflict miss (no. of hits ↑)



↳ disadv. : Tag bit length has increased.
Also no. of comparisons ↑, as any block can be anywhere. (comparison = no. of tag bits)

Set associative

Block replacement policy — LRU
Cache write — Write through, write back (not updated)
Virtual mem —

no. of blocks in memory = m
no. of pages in virtual address space = n

page replacement algo : FIFO, OPT, LRU

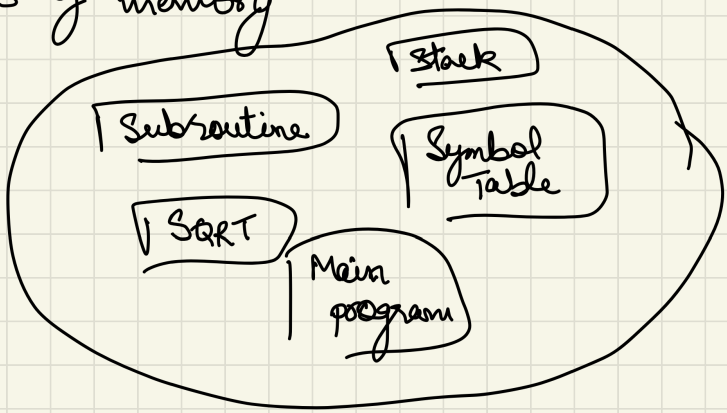
- Counter → replace which has not been used for the longest time
- Stack → time-of-use register

- Memory Management Hardware :

① Dynamic Storage Relocation – mapping (Basic f^n's of MM)
② Provision for sharing common info.
③ Protection of info. against unauthorized access

- Segmentation : set of logically related instructions or data elements associated with a given name.

- User view of memory



- Implementation of Page and Segment Tables

- Two memory accesses are needed to access a word, one for pg table, one for the word

- Logical Address | segment | page | word |

  Physical Address | Block | word |

- Memory protection:

  protection info. can be stored in the segment table or segment register

format of typical segment descriptor

| Base address | Length | Protection |

↓

specifies Access Rights to a particular segment

- full read & write
- read only
- execute only
- system only

FRES

Q1 i) $\frac{2048 \times 8}{128 \times 8} = 16$

128 × 8 RAM chip to 2048 byte

ii) $2048 = 2^{11}$ ⟶ 11 lines

$128 = 2^{7}$ ⟶ 7 common lines

iii) 11 − 7 = 4 lines for decoding

4 × 16 decoder

Slide 15-29 :

- Associative memory mapping

CAM

Address	Data
01000	3450
02777	6710

Mapping table is associated in associative memory → fast, very expensive

Mapping table stores both address and the content of memory word

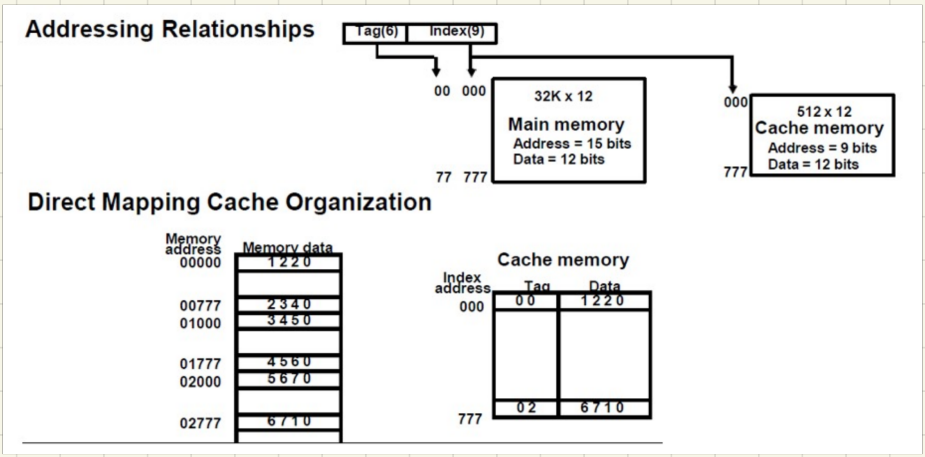
- Direct Mapping : (Rmodn gives unique)

① Each memory has only one place to load in Cache

② Mapping Table is made of RAM instead of CAM.

③ n-bit address contain 2 parts : K bit index, n-K tag field

④ n bit address is used to access main memory, K bit index for Cache



Operation:

- CPU generates a memory request with (Tag, Index)

  Access cache using Index;

  (tag, data) compare Tag and tag

- If matches → Hit

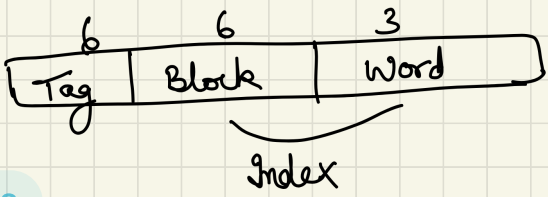
  Provide Cache[Index](data) to CPU

- If not match → Miss

  M[tag, Index] ← Cache[Index](data)

  Cache[Index] ← (Tag, M[Tag, Index])

  CPU ← Cache[Index](data)



- Set Associative Mapping :

Each memory block has set of locations in the cache to load

← Set Ass. mapping cache with set size two

Operation :

- CPU generates a memory address (Tag, Index)
- Access cache with index (Cache word = (tag 0, data 0); (tag 1, data 1))
- Compare Tag and tag 0 and then tag 1
- If tag i = Tag → Hit, CPU ← data i
- If tag i ≠ Tag → Miss

  Replace either (tag 0, data 0) or (tag 1, data 1)

  Assume (tag 0, data 0) is selected for replacement

  M[tag 0, Index] ← Cache [Index] (data 0)

  Cache [Index] (tag 0, data 0) ← (Tag, M[Tag, Index])

  CPU ← Cache [Index] (data 0)

Block Replacement Policy:

BLOCK REPLACEMENT POLICY

Many different block replacement policies are available

LRU(Least Recently Used) is most easy to implement

Cache word = (tag 0, data 0, *U0*);(tag 1, data 1, *U1*), Ui = 0 or 1(binary)

Implementation of LRU in the Set Associative Mapping with set size = 2

Modifications

Initially all U0 = U1 = 1
When Hit to (tag 0, data 0, U0), U1 <- 1(least recently used)
(When Hit to (tag 1, data 1, U1), U0 <- 1(least recently used))
When Miss, find the least recently used one(Ui=1)
If U0 = 1, and U1 = 0, then replace (tag 0, data 0)
M[tag 0, INDEX] <- Cache[INDEX](data 0)
Cache[INDEX](tag 0, data 0, U0) <- (TAG,M[TAG,INDEX], 0); U1 <- 1
If U0 = 0, and U1 = 1, then replace (tag 1, data 1)
Similar to above; U0 <- 1
If U0 = U1 = 0, this condition does not exist
If U0 = U1 = 1, Both of them are candidates,
Take arbitrary selection

Cache Write:

Write Through
When writing into memory
If Hit, both Cache and memory is written in parallel
If Miss, Memory is written
For a read miss, missing block may be overloaded onto a cache block
Memory is always updated
-> Important when CPU and DMA I/O are both executing
Slow, due to the memory access time

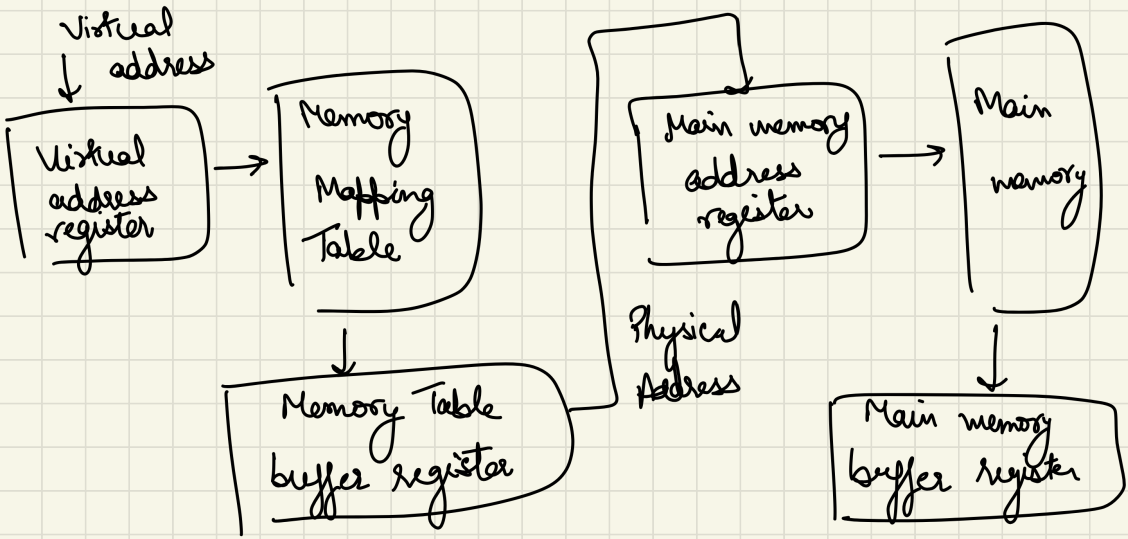
Write-Back (Copy-Back)
When writing into memory
If Hit, only Cache is written
If Miss, missing block is brought to Cache and write into Cache
For a read miss, candidate block must be written back to the memory
Memory is not up-to-date, i.e., the same item in Cache and memory may have different value

Write Through
When writing into memory
If Hit, both Cache and memory is written in parallel
If Miss, Memory is written
For a read miss, missing block may be overloaded onto a cache block
Memory is always updated
-> Important when CPU and DMA I/O are both executing
Slow, due to the memory access time

Write-Back (Copy-Back)
When writing into memory
If Hit, only Cache is written
If Miss, missing block is brought to Cache and write into Cache
For a read miss, candidate block must be written back to the memory
Memory is not up-to-date, i.e., the same item in Cache and memory may have different value

- Virtual memory : Logical address

memory mapping Table for virtual address



Address Mapping : Representation of the above diagram for an eg.

- Associative memory page table :

Assume that
Number of Blocks in memory = m
Number of Pages in Virtual Address Space = n

Page Table
- Straight forward design -> n entry table in memory
Inefficient storage space utilization
<- n-m entries of the table is empty

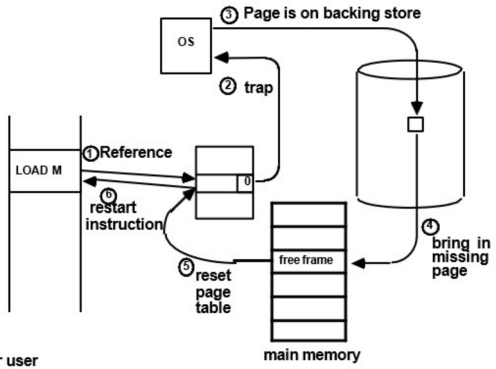
- More efficient method is m-entry Page Table
Page Table made of an Associative Memory m words; (Page Number:Block Number)

Page fault : page no. can't be found in page table

- Page Fault :

PAGE FAULT

1. Trap to the OS
2. Save the user registers and program state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the backing store(disk)
5. Issue a read from the backing store to a free frame
   a. Wait in a queue for this device until serviced
   b. Wait for the device seek and/or latency time
   c. Begin the transfer of the page to a free frame
6. While waiting, the CPU may be allocated to some other process
7. Interrupt from the backing store (I/O completed)
8. Save the registers and program state for the other user
9. Determine that the interrupt was from the backing store
10. Correct the page tables (the desired page is now in memory)
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, program state, and new page table, then resume the interrupted instruction.



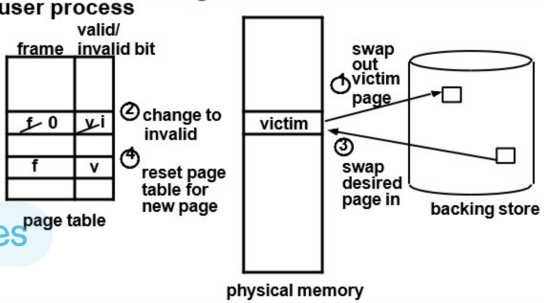
Processor architecture should provide the ability to restart any instruction after a page fault.

- Page Replacement :

Decision on which page to displace to make room for an incoming page when no free frame is available

Modified page fault service routine

1. Find the location of the desired page on the backing store
2. Find a free frame
   - If there is a free frame, use it
   - Otherwise, use a page-replacement algorithm to select a *victim* frame
   - Write the victim page to the backing store
3. Read the desired page into the (newly) free frame
4. Restart the user process



- Page Replacement Algo. :

① FIFO : selects page that has been in the memory the longest time
uses queue
easy to implement
may result in frequent pg fault

② Optimal Replacement : Lowest page fault rate of all algos
Replace the pg which will not be used for longest period of time

③ LRU : OPT is difficult to implement since it requires future knowledge. Replace the pg which has not been used for longest period of time

LRU may require substantial hardware assistance
The problem is to determine an order for the frames defined by the time of last use

LRU replacement methods : (Slide 28)

① Stack
② Counter
③ LRU Approximation

Segmentation :

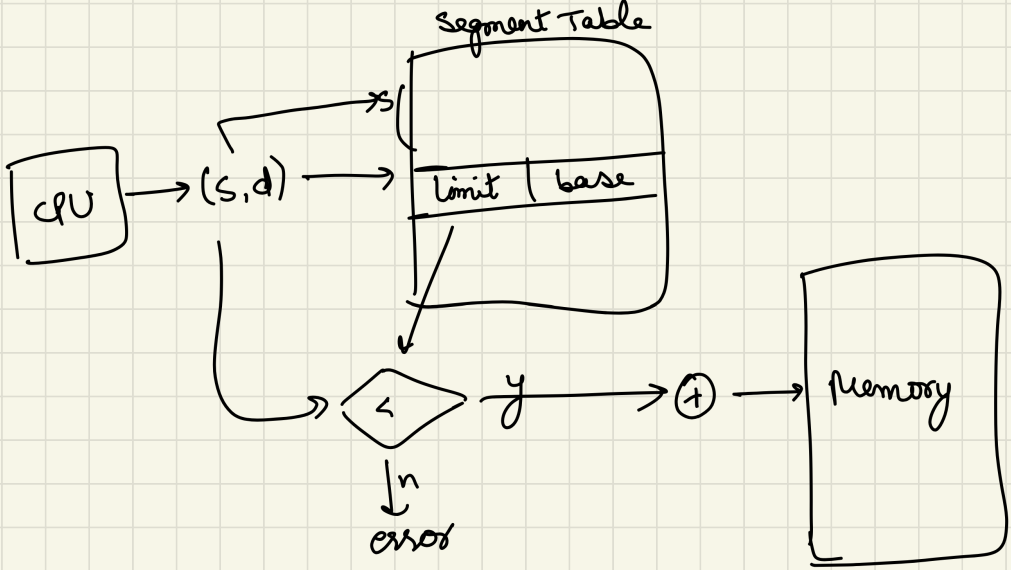
A memory management scheme which supports user's view of memory

A logical address space is a collection of segments

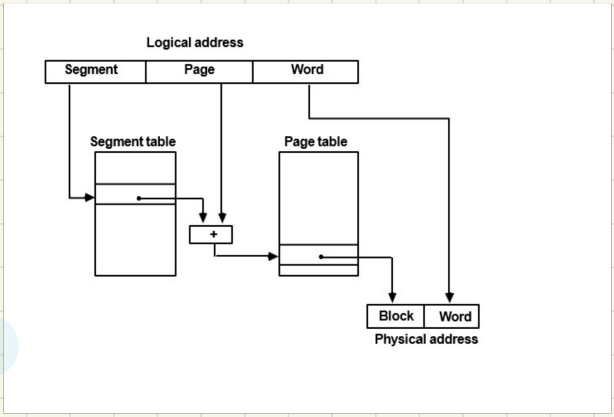
Each segment has a name and a length

Address specify both the segment name and offset within segment

for simplicity of implementation, segments are numbered



Segmented fg System :



Implementation of page and segment tables :

Implementation of the Page Table

- Hardware registers (if the page table is reasonably small)
- Main memory
  - Page Table Base Register(PTBR) points to PT
  - Two memory accesses are needed to access a word; one for the page table, one for the word
- Cache memory (TLB: Translation Lookaside Buffer)
  - To speedup the effective memory access time, a special small memory called associative memory, or cache is used

Implementation of the Segment Table

Similar to the case of the page table

Q2: i) $\frac{1024 \times 8}{1024 \times 1} = 8$ chips

ii) $\frac{16 \times 2^{10} \times 8}{1024 \times 1} = 128$

iii) $16K = 2^{14}$

14 total lines

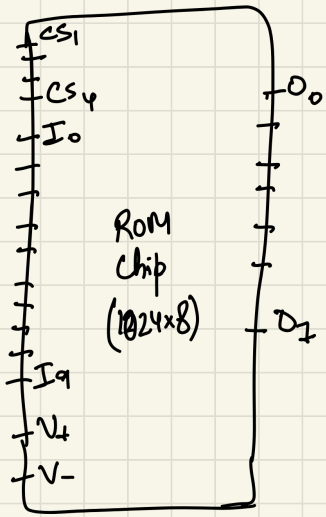
$1024 = 2^{10}$ —— common lines → 10 lines specify chip address

4 decoder lines

↳ 4 lines are decoded into 16 chip select lines

Q3 1024×8 ROM chip has 4 select line inputs, 5V power supply

Sol^n



⓪ output lines = 8

① 1024 × 8

2^{10} → 10 common lines

② 4 select lines

③ Power pin = 2

Total pins = 8 + 10 + 4 + 2
= 24

Q4 RAM chips = 256 × 8
ROM chips = 1024 × 8
Need = 2K RAM, 4K ROM

4 interface unit
4 register

Sol^n

$\frac{2 \times 2^{10} \times 8}{256 \times 8} = 2^3 = 8$ RAM chips

$\frac{4 \times 2^{10} \times 8}{1024 \times 8} = 4$ ROM chips

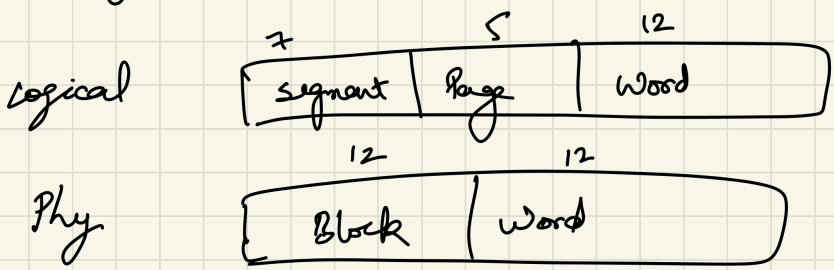
4 × 4 = 16 register ($2^4 = 16$)

Component	Address	16 15 14 13 12 11 10 9	8765	4321
RAM	0000-07FF	0 0 0 0 0 ← 3×8 decoder	xxxx	xxxx
ROM	4000-4FFF	0 1 0 0 ← 2×4 decoder → xx	xxxx	xxxx
Interface	8000-800F	1 0 0 0 0 0 0 0	0000	xxxx

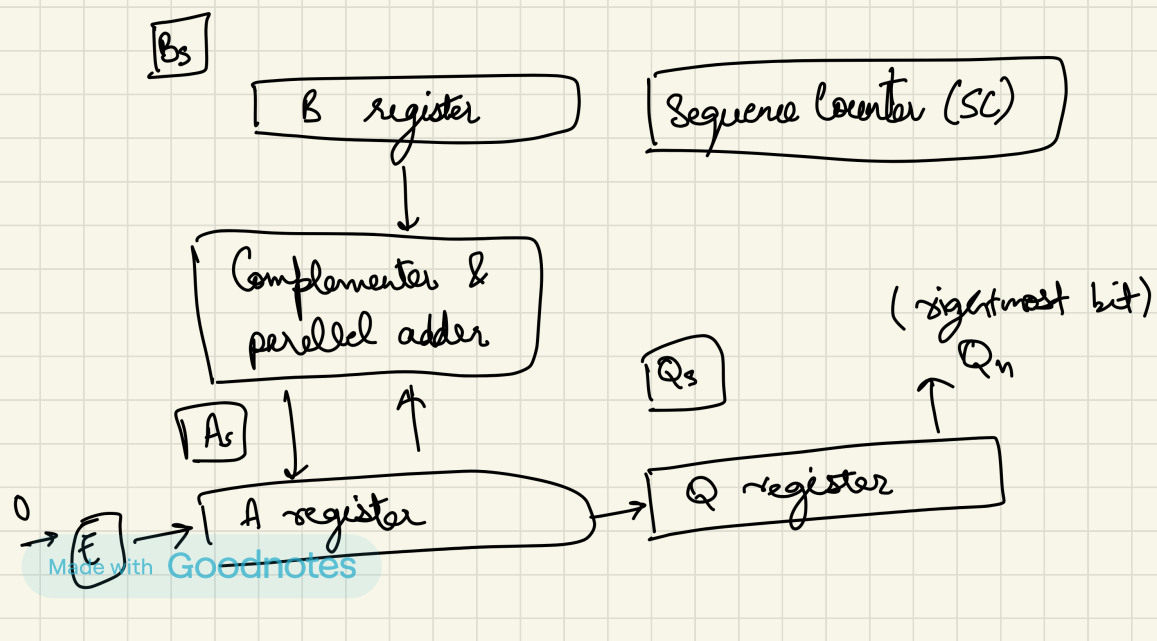
Q5 128 segments, Each seg. — 32 pages, 4K words
4K blocks of 4K words in each, logical & phy. address format?

Sol" Logical address space = $128 \times 32 \times 4K = 2^7 \times 2^5 \times 2^2 \times 2^{10} = 2^{24}$

Phy = $4K \times 4K = 2^{12} \times 2^{12} = 2^{24}$



Multiplication Algorithm:



eg: $7 \times 3 = 21$

Sol^n

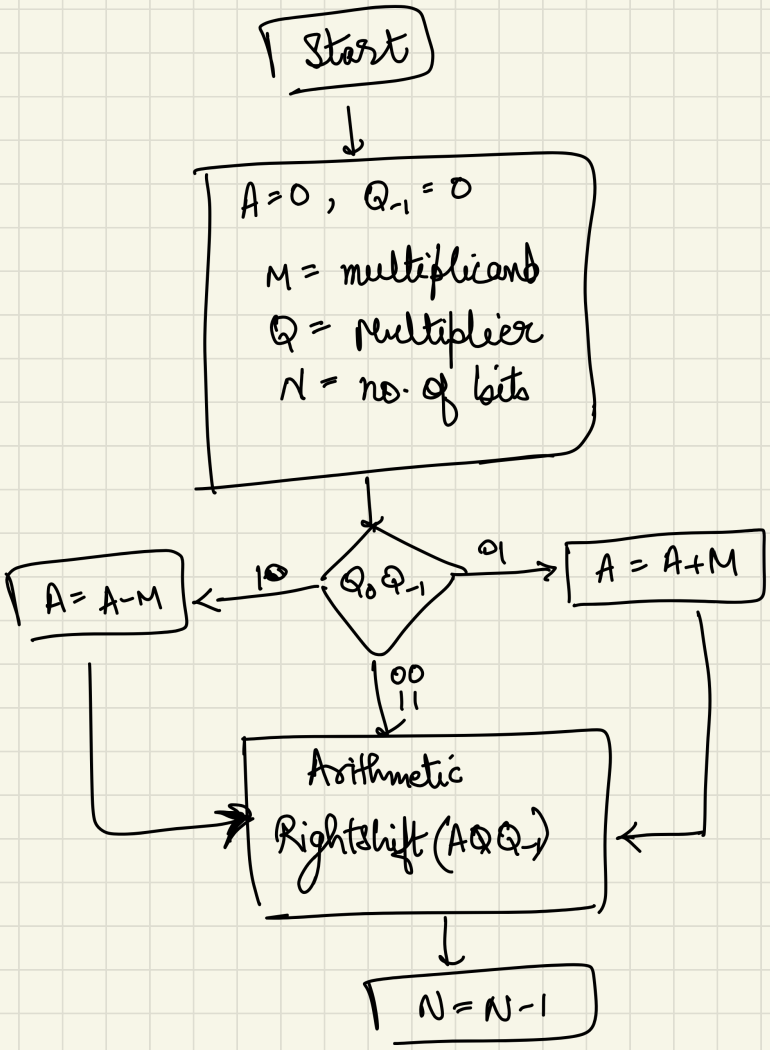
$7 = 0111$ (M)

$3 = 0011$ (Q)

	A	Q	Q_{-1}	operations
1)	0000	0011	0	A−M ⇒ A+2's comp of M 1001
				0000 + 1000 + 1
2)	1001	0011	0	
	1100	1001	1	shift operation
3)	1110	0100	1	A+M ⇒ 1110 + 0111 = 1 0101
	0101	0100	1	
	0010	1010	0	
4)	0010	1010	0	shift operation
	0001	0101	0	

N=0

$7 \times 3 = 0001\ 0101 = 21$



eg: $-9 \times -13 = ?$

Sol^n

$-9 =$ 11001 (M) | 10111

$-13 =$ 11101 (Q) | 10011

	A	Q	Q-1	operation
①	00000	10011	0	$00000 + \overline{10111} + 1 = 01001$ (SUB)
	01001	10011	0	shr

01100 11001 !

② 00100 11001 1 Shr

00010 01100 1

③ 00010 01100 1 Add^n

11001 01100 1 Shr

11100 10110 0

④ 11100 10110 0 Shr

01110 01011

1 1

00010

10111

11001

DMA :

- Input/Output Interface :

Provides a method for transferring info. b/w internal storage and external I/O devices