Date	Title	Page No.	Teacher's Sign / Remarks

classmate
Date 30/6/14
Page

CLASS NOTES

Overview of Subject

CO: Computer Organization

Books: ① Computer Architecture and Organization
— John P. Hayes
3rd edn

② Computer System Architecture
— Morris Mano

③ William Stallings — Computer System Architecture.

Chapters

1st sessional
1. History
2. Design Methodology ⟶ DPC concepts automatically included.

3. Processor Basics
   - user mode
   - Supervisor mode (Supervisor mod)

CPU
- Control Unit (ex. Brain)
- Datapath Unit (ex. Veins, Muscles, Registers, ALU)

2nd sessional
4. Design of Datapath unit
5. Control Unit

Hardwired C.U.
Programmed C.U.

3rd sessional
6. Memory Organization
7. System Organization

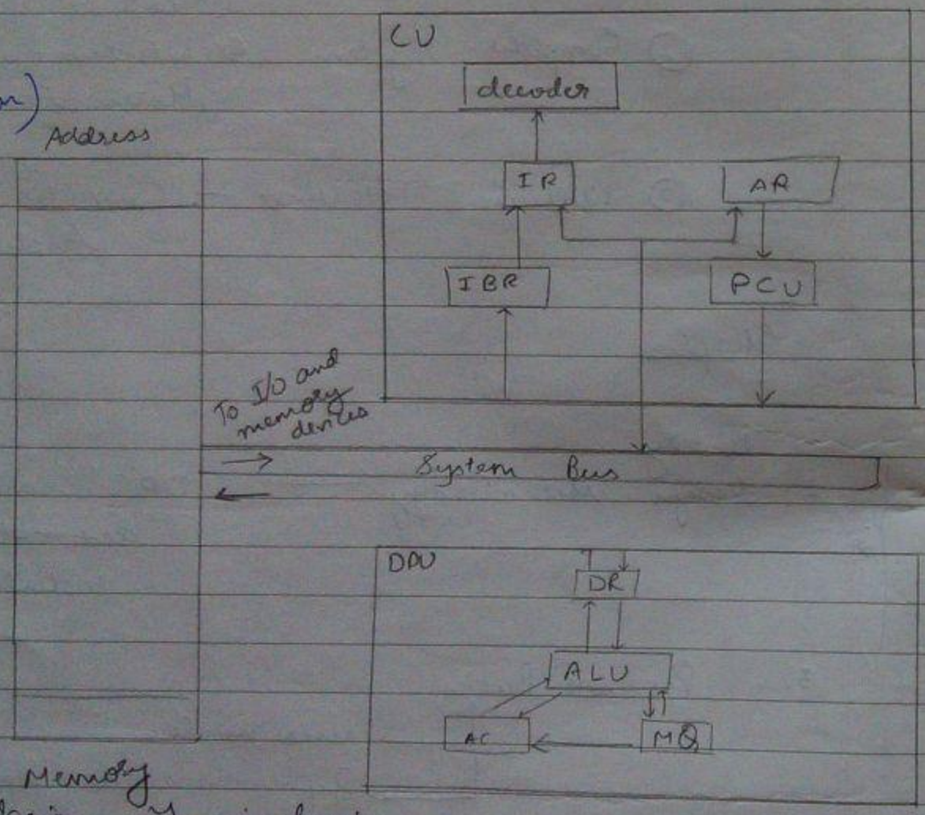
SSS

Chapter 1

History

IR → Instruction Register
PC → Program Counter
DPU →

IAS computers (1st generation)



(2^12)

Memory

Various Terminologies

PCU → Program Control Unit.
DPU → Data Processing Unit.

The function of PCU → fetch the instruction and decode that instruction.

AR → Address Register → stores the address of the current instruction

MQ → Multiplier – Quotient Register.
IR → Instruction Register
PC → Program Counter
DR → Data Register. (it stores data temporarily).
AC → Accumulator.
IBR → next-instruction buffer register.

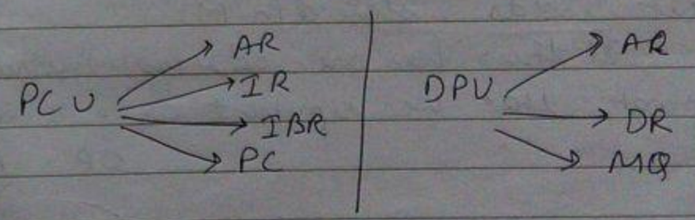
* PC is must because it stores the next instruction address.

* opt code is stored in IR.

System Bus

1. Address Bus [Unidirectional]
2. Data bus [Bidirectional]
3. Control Bus

IAS → Institute of Advanced Studies.



The IAS Computer →

The basic unit information in an IAS computer is a 40 bit word. (it is the standard packet information stored in memory location or transferred in one step between the CPU and the main memory M.)

✓ Each location in M is either a 40 bit single number or else a pair of 20 bit instructions.

✓ Number format is fixed-point, contains implicit binary point in some fixed position.
Numbers are usually treated as signed binary fractions lying between -1 and +1, but they can also be interpreted as integers.

01101000000 0000000000 0000000000
0000000000 = +8.125

↑
is a scaled number format.
(no need to remember)

✓ IAS instruction consists of an 8 bit opcode (operation code) followed by a 12 bit address A that identifies one of up to $2^{12} = 4k$ 40 bit words stored in M.
It thus has one-address instruction set format that we represent as

OP A.

* EDVAC has 4-address instruction format.
— So IAS format saves memory space.

Two key aspects of IAS 's design that have been incorporated into all computers are:
① The CPU contains a small set of high-speed storage devices called registers, that serve as implicit storage space for operands and results.

② A program's instructions are stored in memory M in approximately the same sequence in which they are executed.
— The address of the next instruction is usually that of the current instruction plus one.

EDVAC

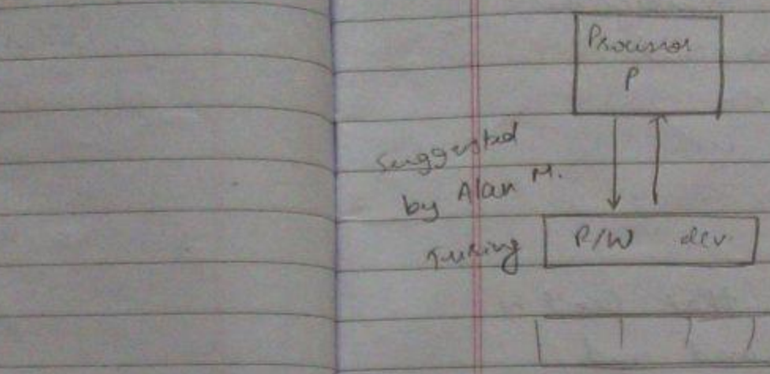
Instructions: | A1 | A2 | A3 | A4 | O/P |

Electronically Discrete Variable Computer

operands' address ② + ③ operation

OP stored the operation to be performed on the operands stored in A_1 and A_2. The result is stored in A_3. A_4 stores the address of the next inst.

Depending on the variants, the inst. address is either in A_3 or A_4.



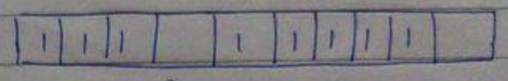
Suggested by Alan M. Turing

Abstract computer — what is a reasonable amount of time — should give o/p in a reasonable amount of time

- ✓ infinite memory comp.?
- ✓ has soln to only those for which algo. is given
- ✓ should process in a fixed amount of time & should give o/p.
- ✓ all possible sol" are not stored.

infinite memory

Addition of 3 and 5.



S_n → present state of
T_i → input
O_j → output
S_k →

- Start from left most 1 and go to the blank space.
- replace that blank space with a 1.
- now move left towards the first '1'.
- Remove that 1.
- sol" obtained.

Unsolvable problems → that can't be solved by any reasonable computer.

ex. Goldbach's conjecture → sum of any two prime numbers is always an even number.

→ Has proved to be true for each and every test case.

→ However, a generalised sol" (mathematical) has not been found or constructed.

Cache → SRAM
Main memory → DRAM

Undecidable problems → Result is ambiguous.
Some inputs give output Yes and
some inputs give output No.

Intractable problems → Problems that can't be solved using reasonable memory and and reasonable time are intractable problems.

Chapter 2

7/7/14

Computer → a system consisting of various components that are connected together, work in unison and give some o/p. (input - process - o/p)
Every system can be presented using a graph.

graph → nodes/vertices (represent 'data')
graph → edges (represent 'data transfer')

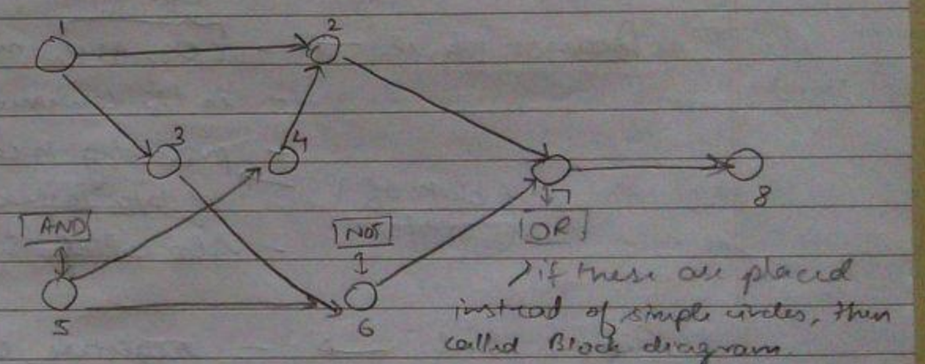
Every system has some → structure (graphs)
→ behaviour (for some specific i/p some specific o/p is obtained) → truth table

Structure represented using graph ⇒

Behaviour

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

$f(A,B) = A \oplus B$



⇒ Individually looking either at the structure or behaviour, we can't figure out what the machine has been designed for.

So, Block diagram is used → in which we not only represent simple circles, but also give the proper names of nodes.

To design a ckt →
1. Behaviour should be known.
2. Components available should be known [using which system is to be made].

✓ Thus, desirable system would be obtained.
If not obtained, then we need to redesign our ckt. until successful.

✓ Simulators should be used instead of physically checking the circuitry each and every time.

Levels of design → There are 3 different levels of design.

① Gate level
② Register level
③ Processor level. ↓

Processor level → – CPU (CU, ALU), memory, I/O devices
– is the lowermost level.
(blocks of words) → – processing is carried from blocks of data instead of bit-by-bit
– Also called System level.

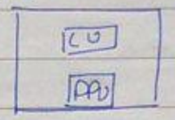
Register level → – registers, counter, multiplexers, decoders, encoders, PLA
– processing is done word by word.
(words)

Gate level → – most primitive form.
– done on individual gates.
– flip-flops, logic gates.
(bit) ← – processing done bit by bit (0 & 1)
– lowest level.

How actual designing process is carried out?

1. Designing should start at the processor level.

CPU



– Depending upon the behaviour, we need to design the (CPU).

Eg. for 4 bit processor,
we need to go to register level.

next we need 4-bit registers, so we need 4 bit device, (made at gate-level).

So, we need to go from
[processor → register → gate level]
–

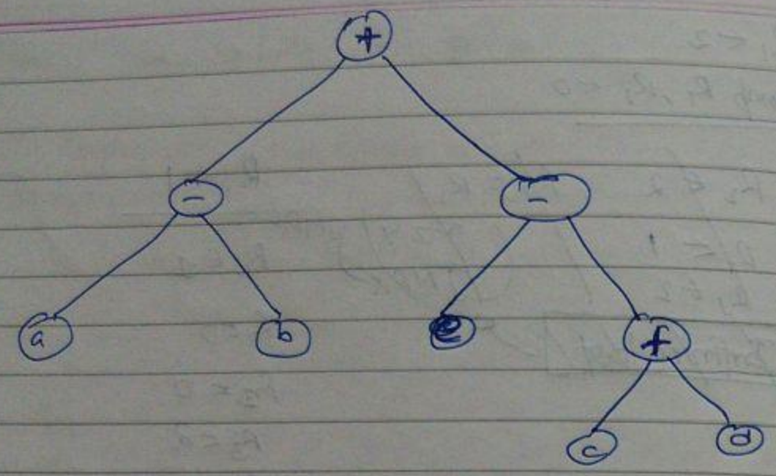
Adder → Behaviour — Adding two bits (A, B)
– generate a sum bit
– carry bit
Half

Truth Table

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

Full — Adding three bits (A, B, carry)
– generate sum bit
– generate carry bit

Ex



⇒ a, b, c, d, e are stored in memory. Load & Store architecture is used. Only registers are used for computation. Intermediary results can not be stored in the memory. Only reg. are to be used for computation. Find the minimum no. of reg. reqd. to perform the above operations

$$((a-b) + (e-(c+d)))$$

R_1 R_2 R_3

R_2 R_1

R_2

→ 3 minimum registers reqd. to perform the above operation.

Ex. $R_1 \rightarrow M[2050]$

$M[2050] \rightarrow R_2$

$M[2050] \rightarrow R_3$

WAP that gives the same o/p however requires less time / clock cycles.

⇒

$R_1 \rightarrow M[2050]$	$R_1 \rightarrow M[2050]$
MOV R_2, R_1	$R_1 \rightarrow R_2$
MOV R_3, R_1	$R_1 \rightarrow R_3$ / $R_2 \rightarrow R_3$

faster (reg.) operations

(Ex). There are 2 level cache memory. Access time of level 1 cache is 1 nano second. Access time of level 2 cache is 10 ns, & that of main memory is 500 ns. Hit rate of level 1 and level 2 are 0.8 and 0.9 respectively. Find out average access time. ⇒

⇒ $t_A = Ht_{A_1} + (1-H)(t_{A_2} + t_{A_3})$

$H = \left(\frac{N_1}{N_1 + N_2}\right)$

$(0.8 \times 1) + (0.02 \times 500) + (0.18 \times 10)$

↑ 20% level 2 failure, 90% + 2 success

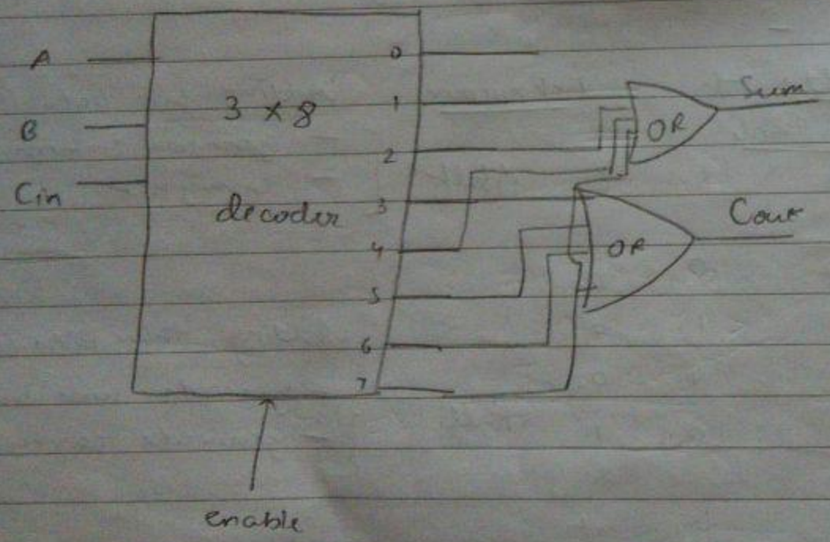
Truth Table for Full Adder

A	B	Cin	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

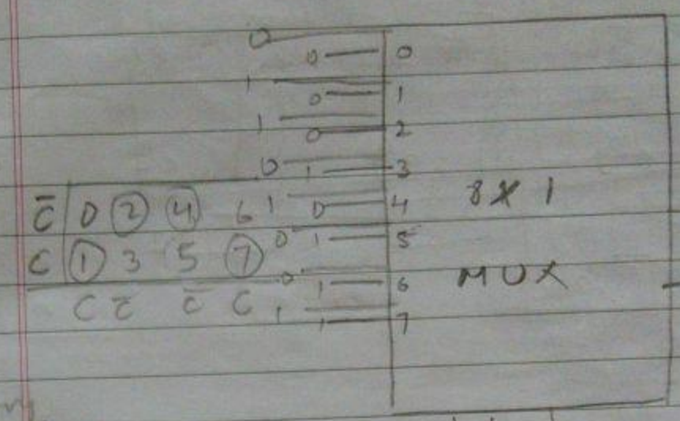
Sum → $A \oplus B \oplus Cin$

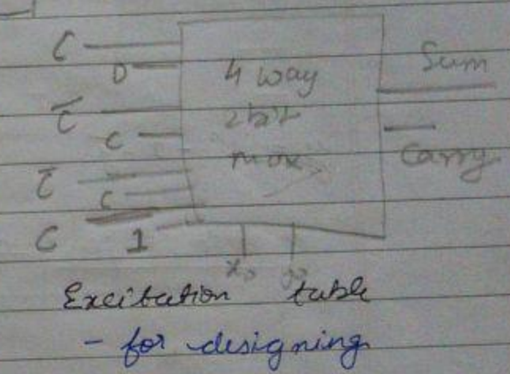
Carry → $A.B + (A \oplus B).Cin$

Full adder ckt using decoder + multiplexer.



12/7/14



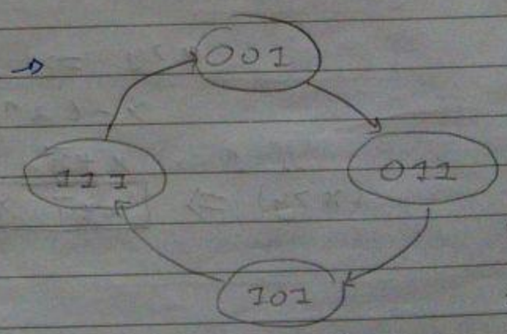


Characteristic table
- for analysis

Excitation table
- for designing

Q Design a counter that counts in odd sequence. using D-flip flop.
1, 3, 5, 7.

State digram →



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

D_A	D_B	D_C
0	1	1
1	0	1
1	1	1
0	0	1

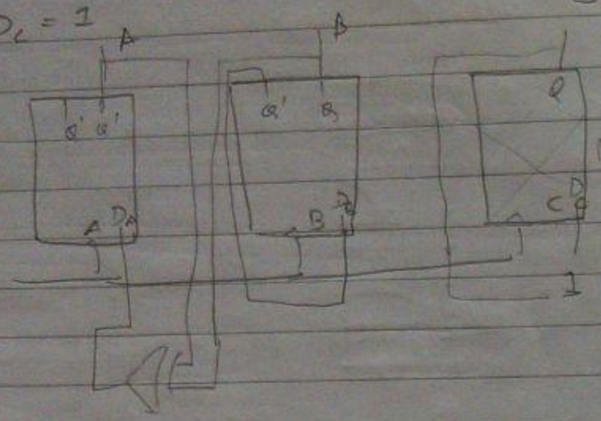
(D_A)

A \ BC	00	01	11	10
0	X		1	X
1	X	1		X

$D_A = A\bar{B} + \bar{A}A$

$= A \oplus B$

$D_0 = \bar{B}$ $D_1 = 1$



$y = 2^n - 1 - \bar{y}$

Eg: $y = 0110$

$\bar{y} = 1001$

$2^n = 2^4 = 16 - 1 = 15$

$$\begin{array}{r} 1111 \\ -\ 1001 \\ \hline 0110 \end{array}$$

Use adder ckt. for comparison →

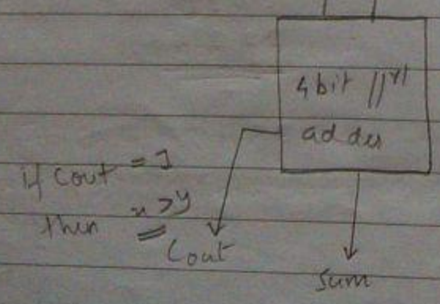
(1) $x > y$

$x > y \Rightarrow x - y > 0$

$x - (2^n - 1 - \bar{y}) > 0$

$x + \bar{y} > 2^n - 1$

$(x > y) \Rightarrow \boxed{x + \bar{y} > 1111}$



2. $x < y$?

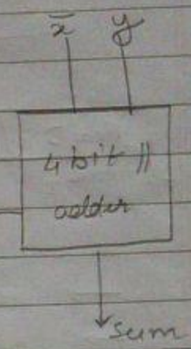
$x < y$

$y - x > 0$

$y - (2^n - 1 - \bar{x}) > 0$

$y + \bar{x} > 2^n - 1$

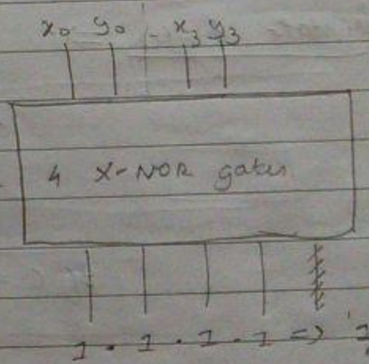
$\bar{x} + y > 1111$



3. $x = y$?

A⊙B for each bit

AND each output bit and check if equal to 1



Bus → unidirectional (address bus)

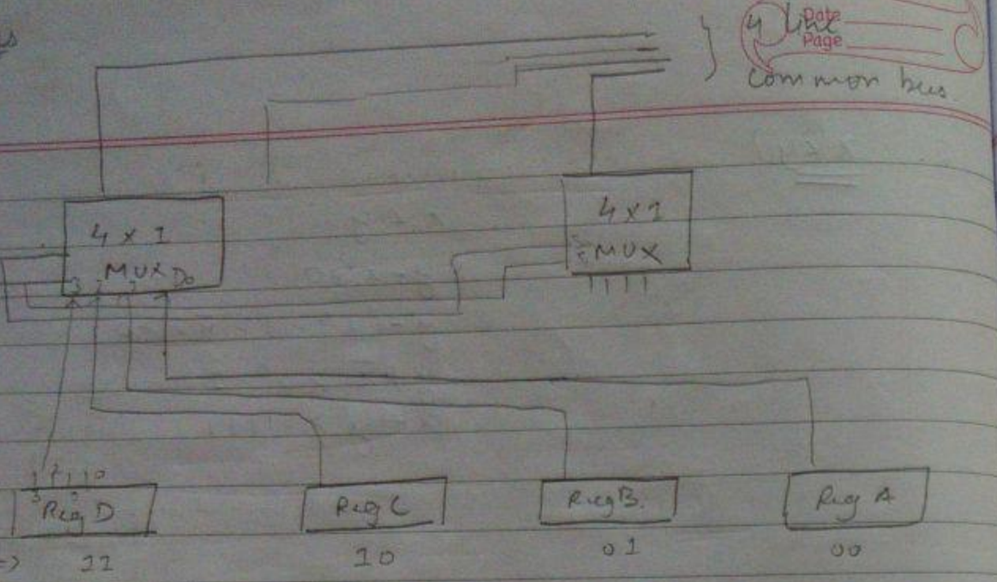
bidirectional (data bus)

shared
- many components can use this bus.

dedicated
- path betn 2 components only.

4 registers → A → 4 bits, B, C, D

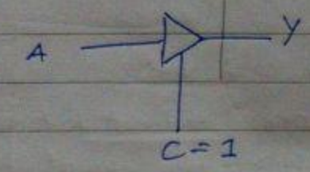
Some synchronization is present betn the reg. so as to transfer data to bus (from them/to them)



→ Select lines common for all multiplexers

→ only one reg gets selected at a time.

Tristate buffer ⇒



If C = 1, then

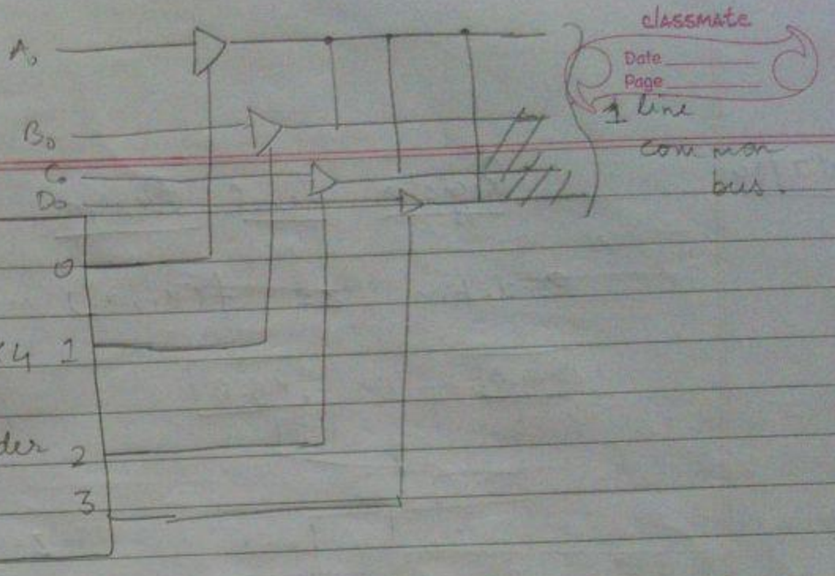
A	Y
0	0
1	1

If C = 0, then output would be in high impedance state.

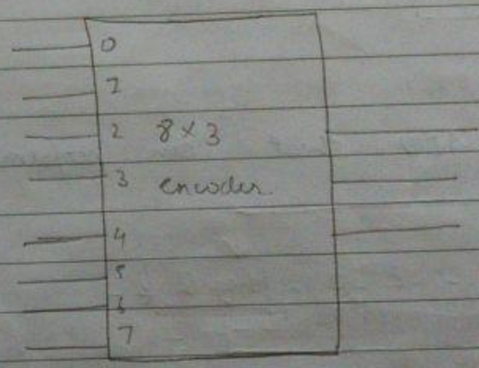
⇓

Open circuit

→ Y would be disconnected from the ckt. It can be pulled 'low' or 'high'.



Encoder

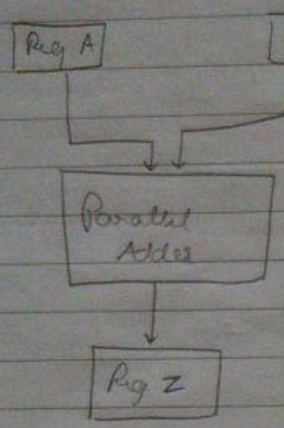


priority encoder	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	A	B	C
	1	0	0	0	0	0	0	0	0	0	0
	x	1	0	0	0	0	0	0	0	0	1
	x	x	1	0	0	0	0	0	0	1	0
	x	x	x	1	0	0	0	0	0	1	1
	x	x	x	x	1	0	0	0	1	0	0
	x	x	x	x	x	1	0	0	1	0	1
	x	x	x	x	x	x	1	0	1	1	0
	x	x	x	x	x	x	x	1	1	1	1

14/7/14

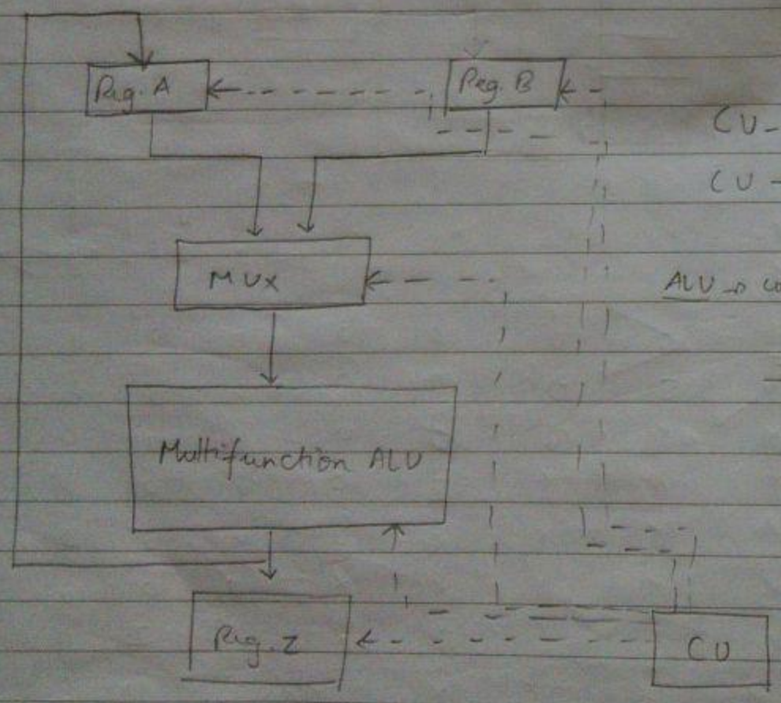
Register Level Design

Condition $z = f(x_1, x_2)$



$Z = A + B$

Multifunction ALU that can perform various ops.



CU - Reg → Clear

CU - MUX → Select i/p

ALU → consists of adder, subtractor etc.

→ All connected to a decoder

Partial products

$P_i = P_i + X_{7-i} Y_M$

$P_{i+1} = 2^{-1} \cdot P_i$ (shift right)

} Algo. for mult^n

2 'n' bits added ⇒ answer would be max. (n+1) bits.

2 'n' bits are multiplied ⇒ answer would be max. 2n bits.

Multiplier

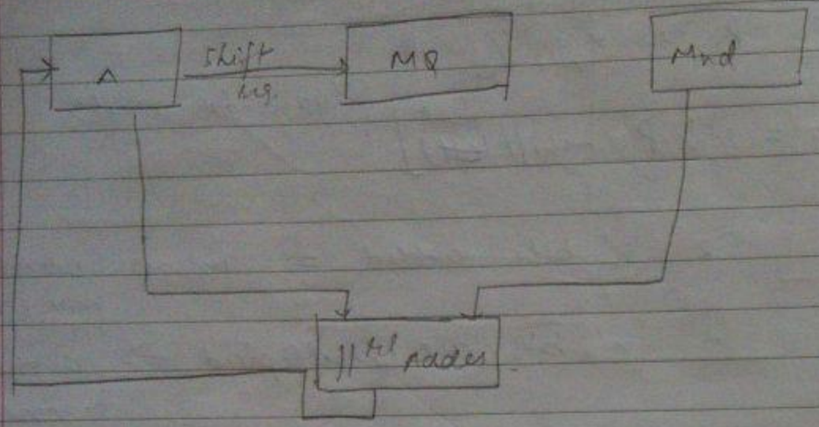
0101 × 1010	A	M Q	Multiplicand
	0000	1010	0101
Check the LSB. If it's 0, then do nothing, if it's 1, then add multiplicand to A	1st: 0000	0101	
	2nd: 0101	0101	
		1010	
	3rd: 0010	1010	
		0101	
	4th: 0110	0101	
	0011	0010	

To obtain sign of answer, we can XOR the sign bits of the multiplier & multiplicand.

0 0	0 —
0 1	1 — } -ve
1 0	1 —
1 1	0 —

+ve

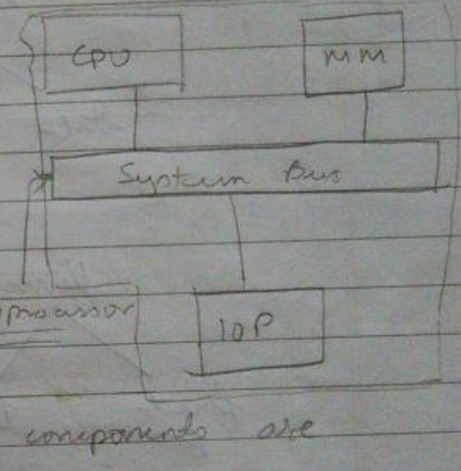
(Ex 5.2)



To design any processor
→ first design the Algorithm
→ next based on the algorithm, list out the components reqd.
→ Once the components have been decided, design the DPU.
→ Next step would be to design the CU.

Processor Level ⇒

✓ CPU
✓ IOP (processor)
deal with ✓ Memory
I/O devices. ✓ Interconnection
↳ So as not to interrupt the CPU.



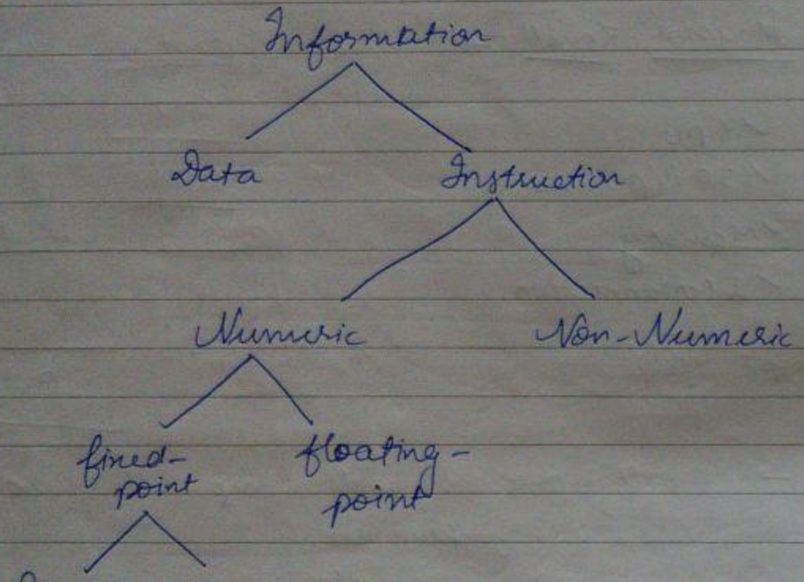
→ Usually processor level components are asynchronous
⇒ Components are highly interdependent. If all of them are synchronized, then there would be disparity betw their operating speeds.

Data Representation

19/9/14

Memory → stores the opcode, not the instruction.

Instruction	Opcode
MOV A, B	06 (say)



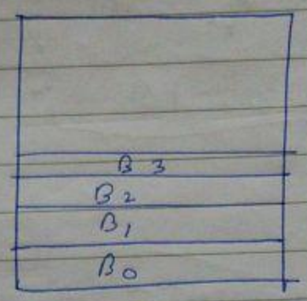
183
→ 1.83×10^2 represented as (M, E) ← Exponent
→ 18.3×10^1 ↑ Mantissa
(18.3, 1) (1.83, 2)

Instruction	Instruction size
MOV A, B	1 byte
MVI ~~A~~ A, 06	2 bytes
~~MV~~ LDA 2030	3 bytes

Q How to store data in memory?

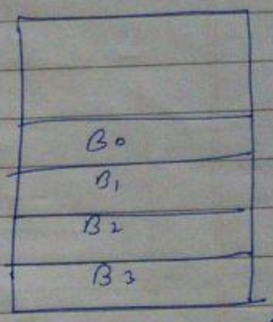
→ Can be stored in 2 ways →

little endian



MSB → higher add.
LSB → smaller add.

big endian



MSB → smaller add.
LSB → higher add.

Q How to detect overflow and underflow while performing addition of numbers? → signed

$X < Y$

$\overline{x_{n-1}} y_{n-1} \cdot \overline{c_{n-2}} + \overline{x_{n-1}} \cdot \overline{y_{n-1}} \cdot c_{n-2}$

```
  1001
+ 1101
-------
(1) 0110   Carry generated  C4 = 1
```

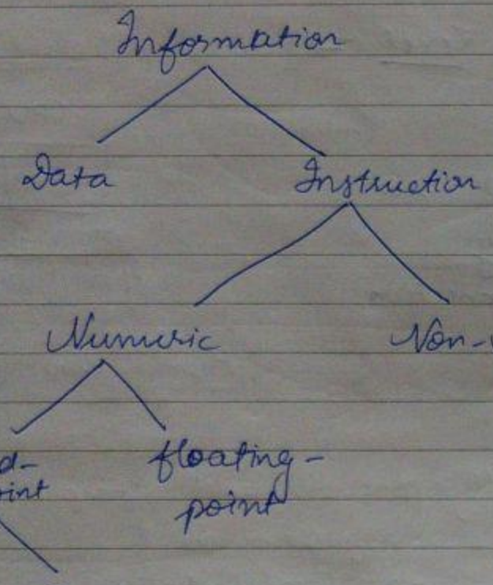
Q Design a ckt so as to set/reset Sign, Zero, Carry, Aux. Carry and Overflow flag.

Data Representation

19/7/24

Memory → stores the opcode, not the instruction.

Instruction	Opcode
MOV A, B	06 (say)



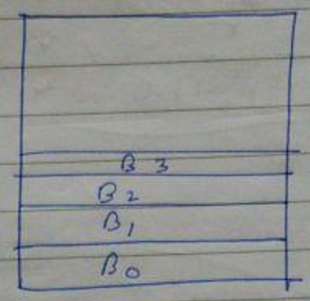
183
→ 1.83×10^2 represented as (M, E) ← Exponent
→ 18.3×10^1 ↑ Mantissa
(18.3, 1) (1.83, 2)

Instruction	Instruction size
MOV A, B	1 byte
MVI ~~A,~~ A, 06	2 bytes
~~LDA~~ LDA 2030	3 bytes

Q How to store data in memory?

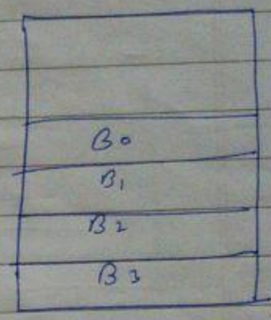
→ Can be stored in 2 ways →

little endian



MSB → higher add.
LSB → smaller add.

big endian



MSB → smaller add.
LSB → higher add.

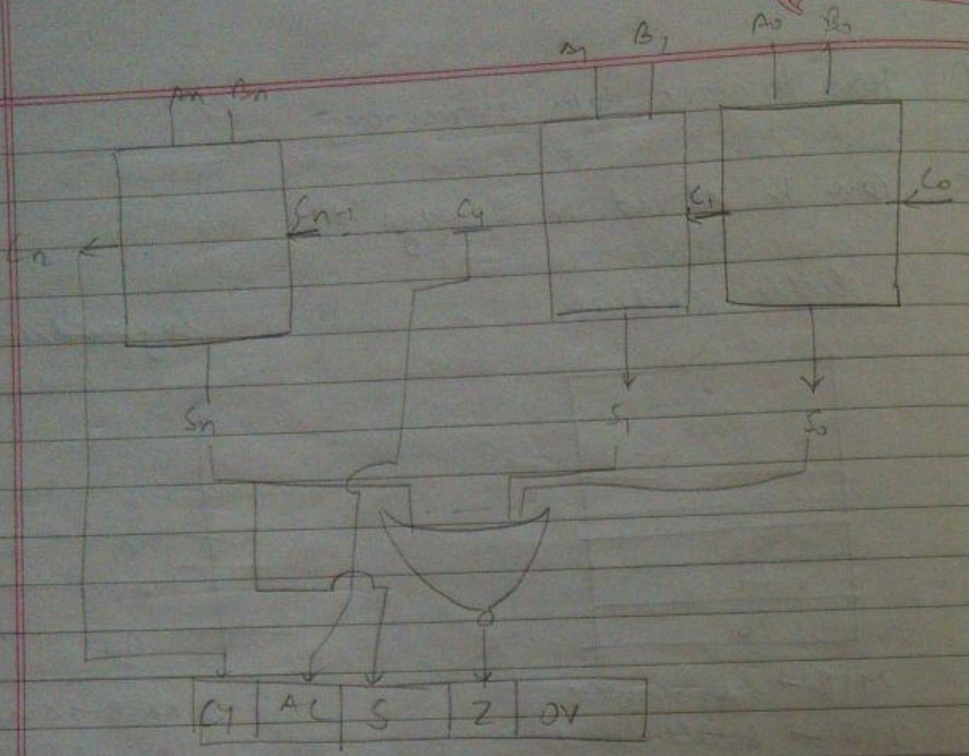
Q How to detect overflow and underflow while performing addition of numbers? → signed

$X < Y$

$\overline{x_{n-1}} y_{n-1} \cdot \overline{c_{n-2}} + \overline{x_{n-1}} \cdot \overline{y_{n-1}} \cdot c_{n-2}$

```
  1001
+ 1101
-------
(1) 0110   Carry generated  CY = 1
```

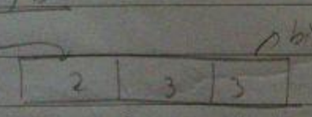
Q Design a ckt so as to set/reset Sign, Zero, Carry, Aux. Carry and Overflow flag.



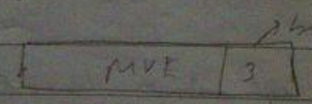
Q How is the size of any instruction det.?

→ It is determined using Huffman code.
Inst. that are frequently used are smaller in size than those which are (lesser) rarely used. or less frequently used.

MOV A,B



MVI A,45



21/7

Mantissa → precision
Exponent → range

3 bits (M) 3 bits (E)

011, 101

$3 \times 2^3 = 24$ (highest)

111, $-3 \times 2^3 = -24$

100, 100 -4×2^{-4}

Dec	Unsigned	1's comp.	2's comp.	Sign. mag.
~~+4~~				
+3	011	011	011	~~011~~ 011
+2	010	010	010	010
+1	001	001	001	001
+0	000	000	000	000
-0	000	111	(1)000	100 (-4)
-1	001	110	111	101
-2	010	101	110	110
-3	011	100	101	111

Tag field → (3 bit) (special bits used for recognizing if the word is simple data, inst., its precision, etc.

The disadvantage is → memory usage increases.

However, nowadays they are not used.
- the very 1st word is considered as inst. as processor.

Check bits

parity → fails for even no. of errors

secded → single Error Correction
Double Error Detection

$2^C \geq n + C + 1$

(C: check bits; n: data bits)

for $n = 8$ the above eqn is satisfied for $C = 4$

0 1 1 ①	0 1 1 0	0 1 0 ①	0 1 0 0	0 0 1 ①	0 0 1 0	0 0 0 ①	0 0 0 0
D_4	D_3	D_2	C_2	D_1	C_1	C_0	D_0
1 0 0 ①	1 0 1 ①	1 0 1 0	1 0 0 ①	1 0 0 0			
D_7	D_6	D_5	C_3				

$C_0 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$ (LSB)
$C_1 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$ (2nd bit)
$C_2 = D_2 \oplus D_3 \oplus D_4$ (3rd bit)
$C_3 = D_5 \oplus D_6 \oplus D_7$

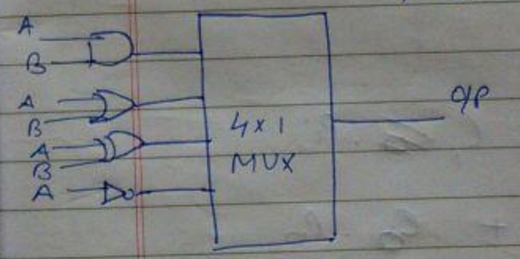
4/8

2nd Sessional

Logic Unit

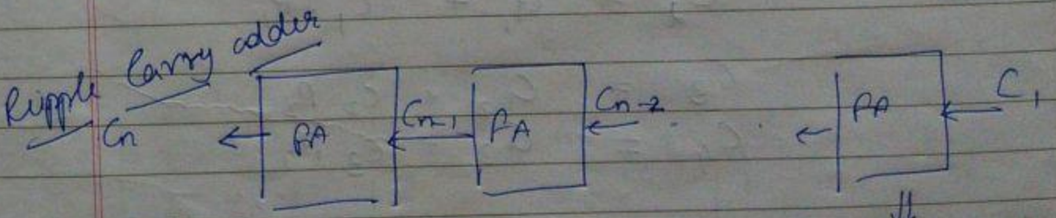
A AND B
A OR B
A XOR B
COMPLEMENT / NOT A

1st bit and so on for n-bits.



Reg A: An . . . A1
Reg. B: Bn . . B1

Ripple Carry adder



disadvantage
n x (delay)

Carry Look Ahead Adder

$G_i = x_i \cdot y_i$ only when $(x_i = y_i = 1)$

(Carry gen. signal)

$P_i = (x_i + y_i)$

(Carry propagate signal.)

$Sum = x_i \oplus y_i \oplus C_{in}$

$Carry = x_i y_i + x_i C_{in} + y_i C_{in}$

$(G_i) + (P_i \cdot C_{in})$

$C_0 = C_{in}$

C_1

C_2

C_3

$C_0 = C_{in}$

$C_1 = x_1 y_1 + x_1 C_{in} + y_1 C_{in}$

$S =$

$C_{(i+1)} = G_i + C_i \cdot P_i$

$G_0 = x_0 y_0$

$C_1 = G_0 + C_0 \cdot P_0$

$C_2 = G_1 + C_1 \cdot P_1$

$C_3 = G_2 + C_2 P_2$

$C_4 = G_3 + C_3 P_3$

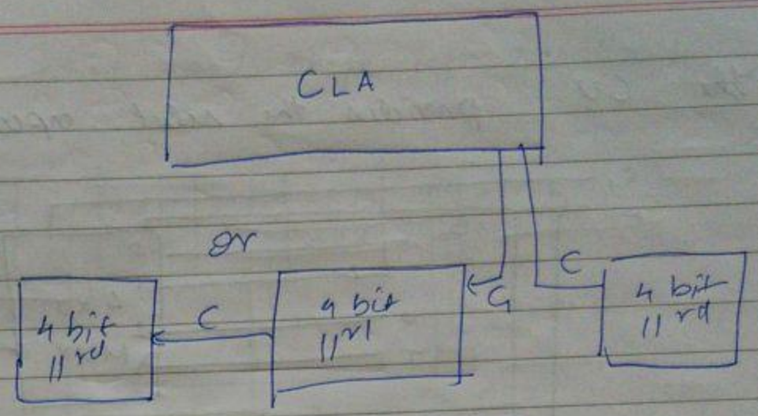
$= G_3 + (G_2 + C_2 P_2) \cdot P_3$

$= G_3 + P_3 G_2 + (G_1 + P_1 \cdot C_1) \cdot P_3 \cdot P_2$

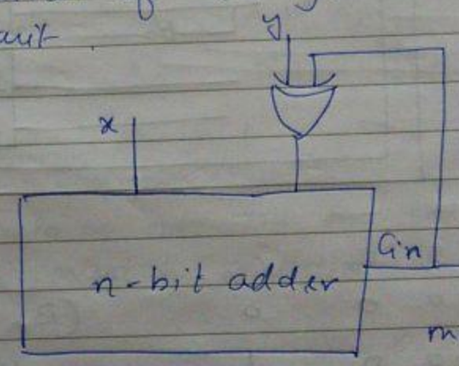
$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 \cdot P_1 \cdot C_1$

$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 \cdot P_2 \cdot P_1 \cdot (G_0 + C_0 P_0)$

$C_{out} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 = C_{in}$

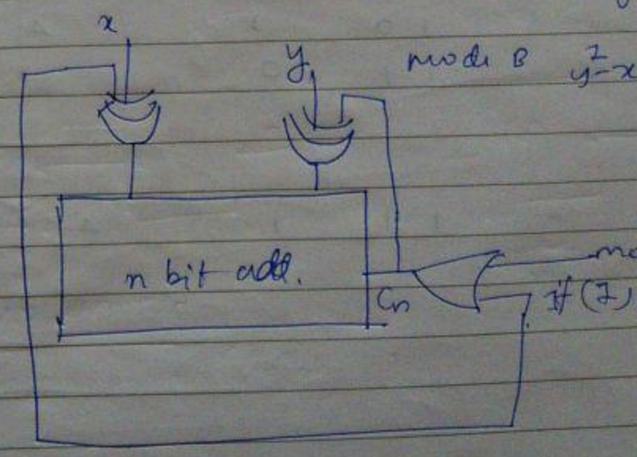


=> implimentation of $(x+y)$ & $(x-y)$ in the same circuit.



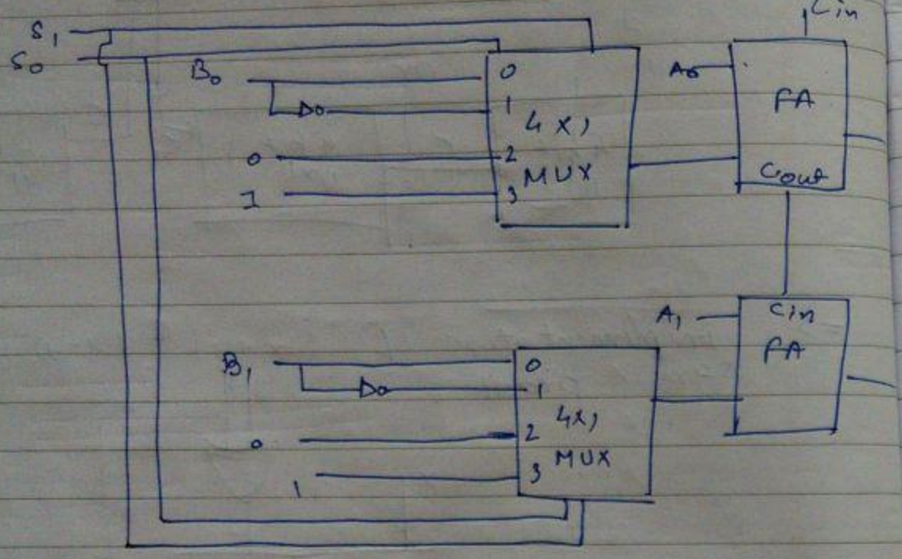
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

$\begin{bmatrix} x+y \\ x-y \\ y-x \end{bmatrix}$



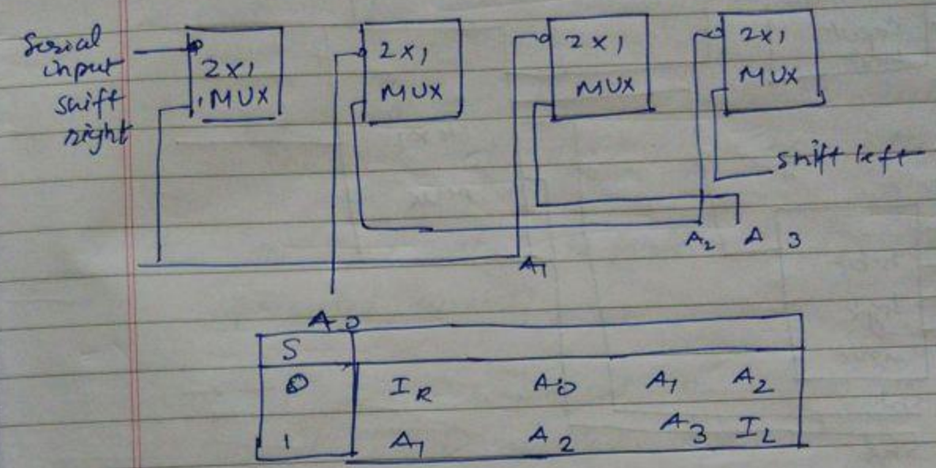
8/8/14

=> The CU provides the select inputs.



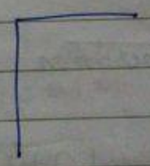
S_1	S_0	C_{in}	operⁿ		
0	0	0	$(B_0 \mid B_1)$	A+B	ADD
0	0	1	"	A+B+1	ADC
0	1	0	$(\bar{B}_0 \mid \bar{B}_1)$	$A+\bar{B} \to A-B$ (-1)	SUB
0	1	1	"	$A+\bar{B}+1$	SUB
1	0	0	A	transfer A	
1	0	1	A	A+0+1	INR
1	1	0		A+2's complement of 1 => A-1	DCR
1	1	1	A	A-1+1=A	transfer of A.

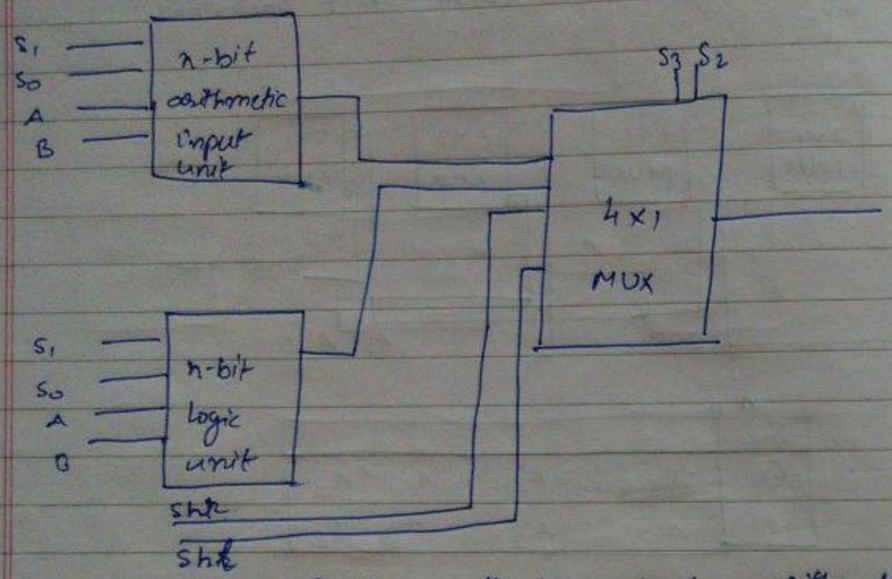
Shift right and shift left operation:



For rotate operation,
(RoR) → A_3 to be connected to 'shift right' input.
(RoL) → A_0 must be connected to 'shift left' input.

n-bit arithmetic input →





S_3 S_2 → choose that particular arithmetic operation.
S_1 S_0 (depends on these select lines)

0 0 → ⟶ → performs arithmetic operation.
0 1 → performs logical operation (depends on S_1 & S_0. which one to perform)

1 0 → shift right operation.
1 1 → shift left operation.

* Perform multiplication operation → shifting and adding.

```
  1010   Multiplicand
x 0110   Multiplier
  0000
 1010x
1010xx
0000xxx
--------
0111100
--------
```

For unsigned multiplication of 'n' bit numbers →

→ use a register of size 2n bits.

→ we can use the same thing for magnitude part for signed multiplication.

→ for sign of the resultant multiplication would be 'xor' of the signs of the two numbers.

→ Can the same steps be performed for 2's complement represented numbers?

→ for 2's complement rep., is the number is +ve, then it can be applied. However, if the number is -ve, then this method cannot be used.

- We need a common rep. of both +ve & -ve of +ve & -ve numbers.

+5 0101
↳ (1010) + 1 = 1011 (2's comp. of +5)

1101
-5 → (0011)

```
 111
  +1
----
1000
----
```

$X = x_{n-1} \cdot x_{n-2} \cdots x_0$

2's comp $X = \overline{x_{n-1}}\ \overline{x_{n-2}} \cdots \overline{x_0} + 1$

$-\overline{X} = 1 - X$

$= (1 - x_{n-1})(1 - x_{n-2}) \cdots (1 - x_0) + 1$

$= 1111 \ldots 1 - x_{n-1}\ x_{n-2} \cdots x_0 + 1$

n times

$111 \ldots 1 + 1$

$= \underline{2^n}$

$2^3 = 8$

111

$+ \quad 1$

$\underline{2^3} \leftarrow \underline{1000}$

$= 2^n - x_{n-1} x_{n-2} \ldots x_0$

$-x = 2^n - x$

$\Rightarrow x = -2^n + x$

$= 2^n - [0 x_{n-2} \ldots x_0 + 1000 \ldots 0]$

(x_{n-1}) if x is (-ve).

$= 2^n - 2^{n-1} - x_{n-2} \ldots x_0$

$$-x = 2^{n-1} - \sum_{i=0}^{n-2} 2^{n-2} x_i - x$$

$$x = -2^{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

Common eqn: $x = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i$.

(for +ve numbers).

(same for -ve numbers)

Booth's Algorithm

A, Q_0, Q_{-1}, M

Consider the example of multiplying 6×2

M = 0110 Q = 0010

Q_0	Q_{-1}	
0	0	} shift right
1	1	
0	1	} A = A+M, shift right
1	0	A = A−M

10000_{10}

$- \quad 0110$

1010

1101

0110

$\overline{0011}$

M = 0110

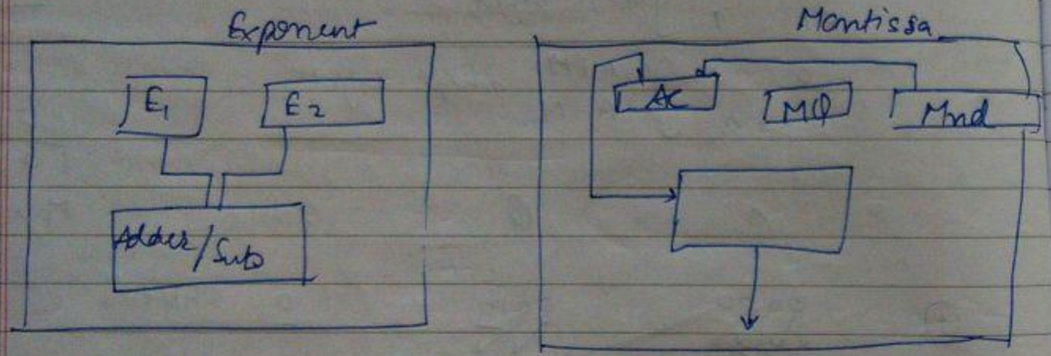
	A	Q	Q_{-1}	
	0000	0010	0	shift
①	0000	0010	0	subtract
	1010	0001	0	
②	1101	0000	1	shift
				add
	0011	0000	1	shift
③	0001	1000	0	
				shift
④	0000	1100	0	

6×2 = 12

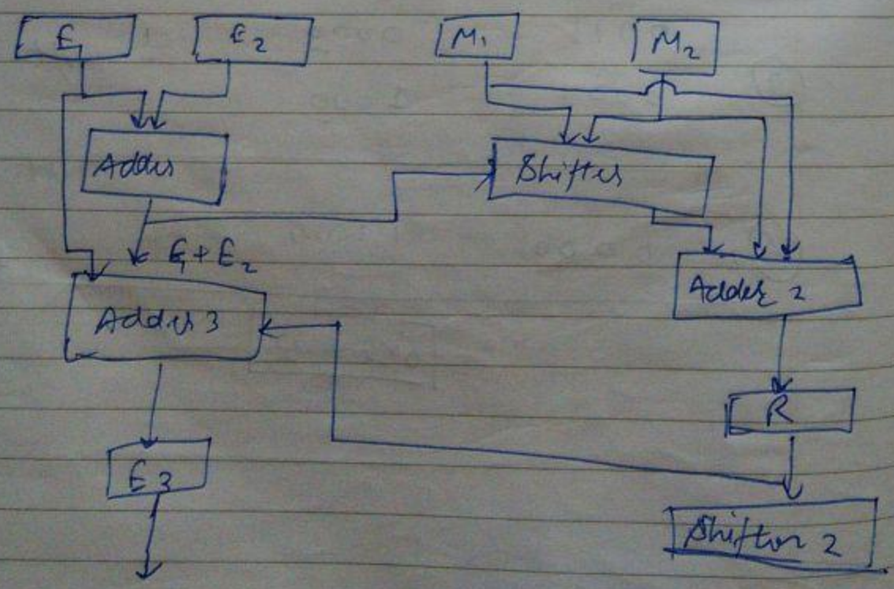
25/8

Floating - point (exponents) ALU.

$$X+Y = X_m \times 2^{X_E} + Y_M \times 2^{Y_E}$$
$$X-Y = X_m \times 2^{X_E} - Y_M \times 2^{Y_E}$$
$$X \times Y = X_m \times Y_m \times 2^{X_E + Y_E}$$
$$X/Y = X_m / Y_m \times 2^{X_E - Y_E}$$



normalized → number represented in 2's complement form.



Ch 4

Pipelining

$X \pm Y \Rightarrow$ Shift the exp
Shift the mant.
Add mant.
Normalized

Chapter 4 — Datapath Design

instruction processor → DPU, CU

Fixed - Point arithmetic → + - * /

Addition & Subtraction

Full adder → $Z_i = x_i \oplus y_i \oplus c_{i-1}$

$$C_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$

Full subtractor →

$$Z_i = x_i \oplus y_i \oplus b_{i-1}$$
$$b_i = \bar{x}_i y_i + x_i b_{i-1} + \bar{y}_i \cdot b_{i-1}$$

30/8 $-9/5 \rightarrow$ requires 5 bits for respresentation

A	Q	M
11111	10111	00101
		SL
11111	01110	
11111	01110	$Q_0 \leftarrow 0$, restore A
		SL
11110	11100	
11110	11100	$Q_0 \leftarrow 0$, restore A
		SL
11101	11000	
11101	11000	$Q_0 \leftarrow 0$, restore A
		SL
11011	10000	
11011	10000	$Q_0 \leftarrow 0$, restore A
		SL
10111	00000	
11100	00001	

Sign bit remains same → $Q_0 \leftarrow 1$ & $A \leftarrow A+M$

2's complement of Q → 00001
11110
+1
11111 = (-1)

11111 + 00101 = 1|00100 | 11110 + 00101 = 00011 | 11101 + 00101 = 00010 | 11011 + 00101 = 100000 | 10111 + 00101 = 11100 | 11011

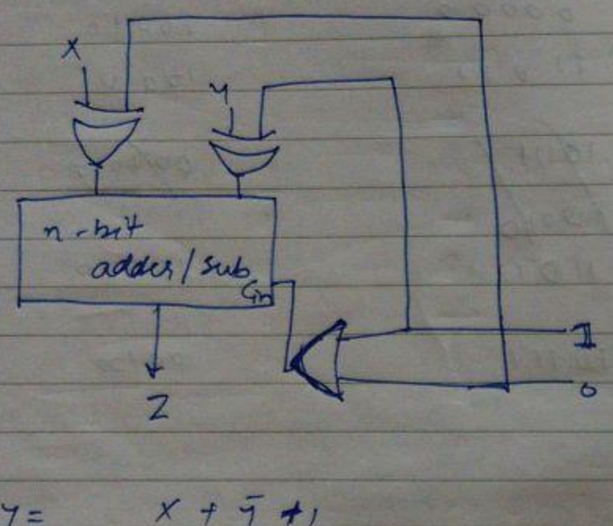
9/5

A	Q	M
000000	01001	00101
		SL
00000	10010	
11011	10010	00101
		SL
10111	00100	
10010		
	10010	000101
11011		$Q_0 \leftarrow 0$, restore A
		SL
10111	00100	

10111
00101
11100

Design 2's complement adder / subtractor so that it can complete the operations,

$X+Y$, $X-Y$, $Y-X$ as specified by a control input.



$X+Y$

$X-Y = X + \bar{Y} + 1$

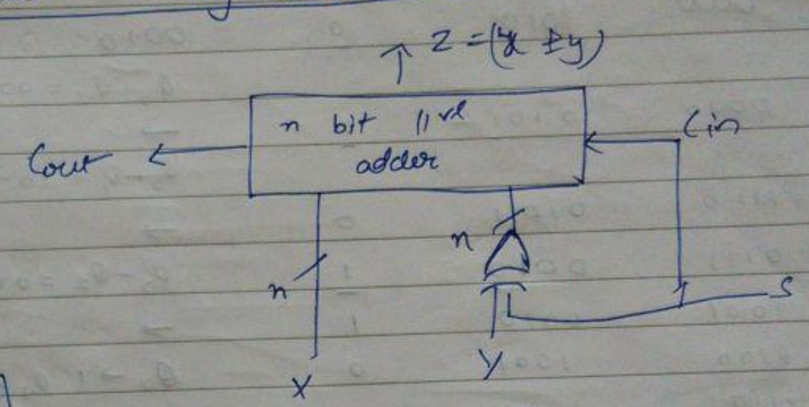
$Y-X = Y + \bar{X} + 1$

$X \oplus S = X$ if $S=0$ $\quad \bar{X}.0 + X.1 = X$

$X \oplus S = \bar{X}$ if $S=1$ $\quad \bar{X}.1 + X.0 = \bar{X}$

$Y-X = Y + (-X)$ → $(1+\bar{X})$ in 2's complement rep.

Subⁿ using adder ckt. →



(X ± Y)

overflow → add 2 +ve or 2 -ve nos

-6 = 1010

-ve Multiplication using Booth's algo.

-6 × 2

A	Q	Q_{-1}	M
0000	1010	0	0010
			$Q_0 - Q_1 = 00$
0000	0101	0	→
			$Q_0 - Q_1 = 10$ → Sub.
1110	0101	0	→
0111	0010	1	$Q_0 - Q_1 = 01$ → Add
1001	0010	1	→
0100	1001	0	$Q_0 \to 1$ $Q_{-1} = 1$ → Add
0110	1001	0	→
0011	0100	0	

now take 2's complment

=> 1011 + 1 = 1100 => -(12)

-3 × 11 => (11 × -3)

+11 → 01011

-3 → 11101

00010
11100
+1
11101

A	Q	Q_{-1}	M
00000	01011	0	11101
			$Q_0 Q_1 = 10$ Sub
00011	01011		→
00001	10101	1	→
2 00000	11010	1	$Q_0 Q_1 = 01$ → Add
3 11101	11010	1	→
01110	11101	0	$Q_0 Q_1 = 10$ →
10001	11101	0	→
01000	11110	1	$Q_0 Q_1 = 01$ →

01110
- 11101
10001

	A	Q	Q_{-1}	
	00101	11110	1	→
4	00010	11111	0	$Q_0 Q_1 = 10$ → Sub
5	00101	11111	0	→
	00010	11111	1	→
	00001	01111	1	

Sub

Add

01111
↓
10000
+1
10001

-4 × 3

0100
1011 → 1100
+1

$Q_0 Q_{-1}$	
00, 11	→
01	A, →
10	S, →

A	Q	Q_{-1}	M
1111	1100	0	0011
			→
1111	1110	0	→
1111	1111	0	Sub
1100	1111	0	→
1110	0111	1	→
1111	0011		

1100
+1
1101

A	Q	Q_{-1}	M
0000	1100	0	0011
~~0011~~ 0000	0110	0	→
0000	0011	0	Sub
1101	0011	0	→
1110	1001	1	→
1111	0100		

1011
+1
(-1) × 1100

-8 × 3

8 → 01000
10111
+1
01111
+1
11000

A	Q	Q_{-1}	M
1 — 00000	11000	0	00011
→			→
2 — 00000	01100	0	→
			→
3 — 00000	00110	0	→
00000	00011	0	Sub.
4 — 11101	00011	0	→
11110	10001	1	→
5 — 11111	01000		

00000
-00011
11101

00010
-01011
10111

-3 × 11

A	Q	Q_0	M
0000	1101	0	1011
0101	1101	0	Sub 1 →
0010	1110	1	→
1101	1110	1	Add → 2
0010	0110	1111	0 → Sub
0011	1111	0	→ 3
0110	0001	1111	1 →
0011	0000	1111	→ 4

0011 → 1101

0000
-1011
0101

0010
+1011
1101

0110
-1011
0011

-3 × 11 → 11 ⇒ 01011
-3 ⇒ 11101

A	Q	Q_0	M
00000	11101	0	01011
00000	10101	0	Sub 1 →
-01011			
10101	11101	0	→
01010	11110	1	Add → 2
-01011			
00010	11111	0	→ Sub
00010	11111	0	→ 3
-01011			
01001	11111	1	→ 4
-01011			
11110	11111	1	→ 5

00011
11100
+1
11101

-9 × -12

12 → ~~01100~~ 11100
0001 1
+1
00100

9 → 11001
00110
+1
00111

A	Q	Q_0	M	
00000	00100	0	00111	
00000	00010	0	→	1
00000	00001	0	→	2
01001	00001	0	Sub	
00100	10000	1	→	3
01011	10000	1	Add.	
00101	11000	0	→	4
00010	11100	0	→	5

00000
-00111
01001

00100
00111
01011

-2 × -12

A	Q	Q_0	M	
00000	~~00100~~ 11110	~~0~~ 0	~~11110~~ 00100	
00000	01111	0	→	1
~~11100~~	~~11110~~	~~0~~	~~00100~~	
11100	01111	0	Sub	2
11110	00111	1	→	
11111	00011	1	→	3
11111	10001	1	→	4
11111	11000		→	5

01100
0001 1
+1
00100

00010
11101
+1
11110

00000
-00100
11100

00010
11101
+1
-2 ← 11110

01100
10011
+1
-12 ← 10100

-2 × -12

A	Q	Q_0	M	
00000	10100 ~~11110~~	0	11110	
			→	1
00000	01010	0	→	2
00000	00101	0	Sub	3
00010	00101	0	→	
00001	00010	1	Add	
11111	00010	1	→	4
11111	10001	0	Sub	5
00001	10001	0	→	
00000	11000	1		

(16 + 8) = (24)

00000
-11110
00010

00001
+11110
11111

11111
-11110
00001

0111
1001

-7/3

A	M	D
1111	1001	0011
		←
1111	0010	Add.
Add		Q_0 ← 0, Restore A

1111
0011
10010

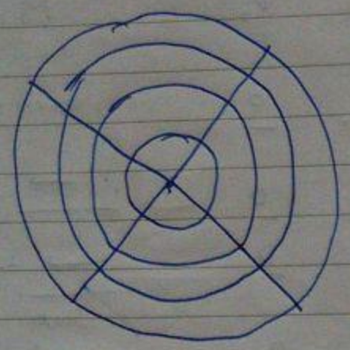
First left shift

Enter m → if m is (-ve) then AC → 111...

Enter divisor →

13/9/14

Serial Access
Random Access
Semirandom Access.



Between the tracks → Random Access memory.

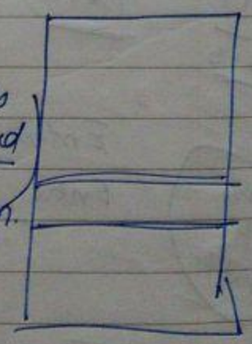
In / On the tracks → Serial Access memory

DRO / NDRO → Destructive / Non-destructive read out memory.

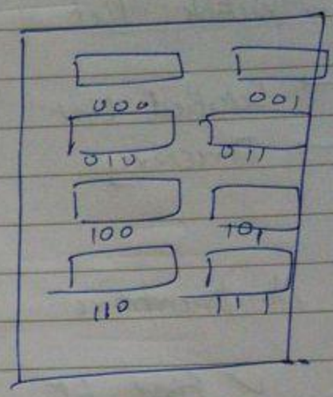
Dynamic / Static → memory. → (EEPROM)

Volatile / Non-volatile → memory.

Data transfer rate → (no. of bits transferred / time taken.)

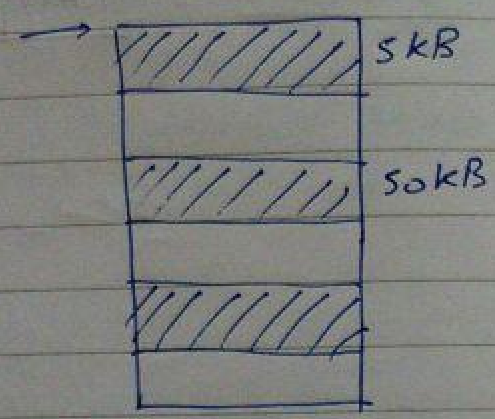


RAM



22/9/14 Available list

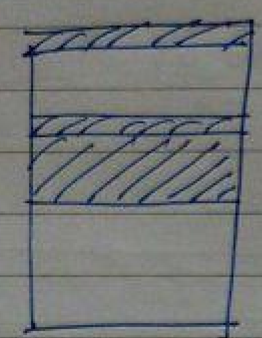
→ Occupied list
→ Directory



Non preemptive

✓ First fit
✓ Best fit

Preemptive allocation →



✓ demand swapping

$t_A = t_B + t_{A1}$

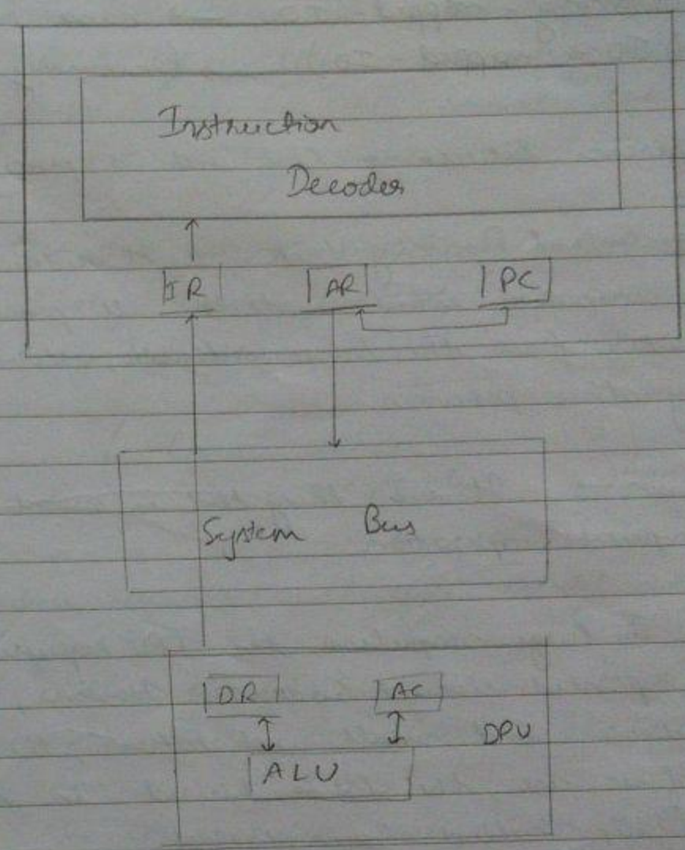
gcc -std=c99 x.c

(helpercode/new)

std:: ~~scanf()~~ --gcd()

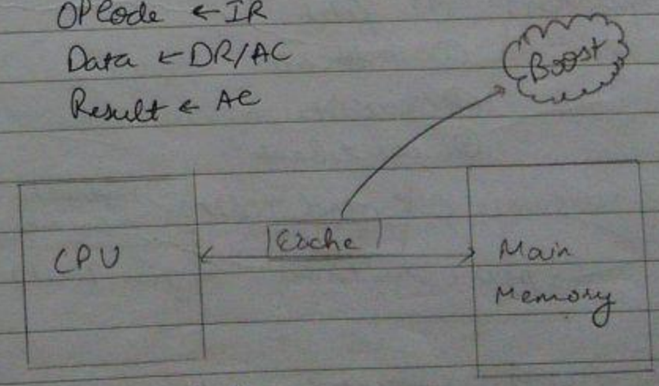
_m_ Euclid

22/



< Accumulator based CPU >

OPcode ← IR
Data ← DR/AC
Result ← AC



M: External Memory
MM: Main M [Also RAM]
CM: Cache memory [for faster memory oper. often mounted on the same die as the processor.]

Memory - mapped - IO → same instruction used for (IO - mapped - IO) → (memory and IO)

Q Difference between a CPU and a processor.

CPU → Central Processing Unit → It is the prios element in a computer which controls all processes of the computer from the basic arithmetic, I/O, logical and many other processes.

Processor → It is a thing that is meant to process computer operation.

In large computers, the CPU requires one or more printed circuit boards. However, in personal computers and small workstations, the ones which most of us use, the CPU is housed in a single silicon chip called a microprocessor.

CPU's follow a 4 step process while performing any operation: ① fetch
② decode
③ execute
④ writeback

CPU → CU (Control Unit) → extracts instructions from memory & decodes them
→ ALU (Arithmetic Logic Unit)

A microprocessor incorporates all of the function of a computer's CPU on a single integrated IC
→ The internal working of the microprocessor depends on the design and purpose of the microprocessor.

It is limited by the number of transistors that can be placed on the chip, the terminals that can connect to other parts of the computer.

Due to the high similarity in usage, it is easy to understand why both of these words have become synonymous

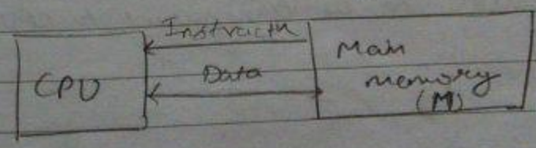
(CPU ⟷ processor is acceptable now-a-days.)

Procedure execution is therefore carried out as follows:

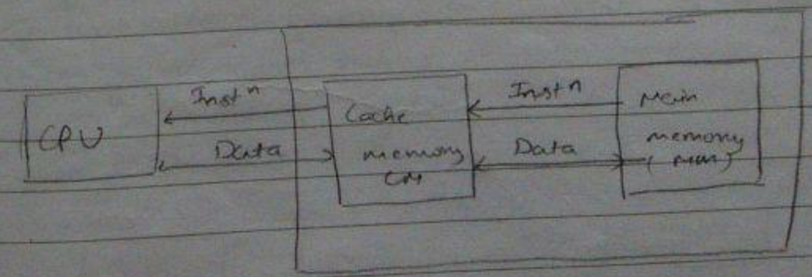
1. The CPU transfers instructions, their input data from main memory to registers in the CPU.

2. The CPU executes them instructions in their stored sequence (except) if altered by a branch instruction.

3. CPU transfers the results from its registers to the main memory thereafter.



(without cache)



(with cache)

Difference between cache and main memory

Cache Memory	Main memory
1) High speed	1) Relatively lower speed
2) less amount of memory	2) more amount of memory
3) does work in one instruction cycle.	3) takes ~~too~~ many instn cycles to perform the same work.
4) integrated in the processor to reduce the average time to access the main memory.	4) can be placed independently.
5) can work without cache.	5) cannot work without main memory.
6) L1, L2 and L3 cache (inside processor)	6) different sizes available.

→ A particular memory address is referred to as M(adr) or simply adr.

→ The cache is designed to be transparent to CPU's instructions.

The CPU communicates with IO device in the same way as it communicates with external memory.

The IO devices are associated with addressable registers called IO ports to which the CPU can store a word (an op operation) or from which it can load a word (for i/p operation)

→ All IO data transfers are implemented by memory-referencing instructions, and approach called memory-mapped IO.

↳ (memory referencing)

The sine-qua-non condition is that the IO (port devices) and memory locations share the same set of addresses.

In memory mapped I/O, the peripheral devices are treated as memory locations. This means that memory addresses are assigned to I/O ports. The CPU just reads and writes data to a memory location, and the IO controller automatically maps the updating part, (ie) transferring data from port to memory and from memory to port

Advantage :- Thus mapping allows the use of full instruction set of CISC computer directly on peripherals, because to the CPU (or program), they are just simple memory locations.

Such systems donot have I/O specific instructions.

Is interrupt part of instruction cycle?

→ Yes, because before the instruction has been completed the check for interrupt is performed in machine cycles.

(instruction cycle) ⇒ machine cycle (1 instruction executes completely)
↳ (clock cycles)

The second method to map I/O system in a computer system is

⇒ I/O mapped I/O

In I/O mapped I/O, the peripheral devices are addressed directly by the CPU using the port addresses. The length of port addresses is generally less than the length of a memory address. There are specific instructions for carrying out input and output tasks at the ports.

→ This method of mapping does not facilitate the use of memory oriented complex instruction set directly on port addresses.

CPU ⟶ I/O devices

User and Supervisor modes ⇒

programs executed by general-purpose computers fall in 2 broad categories → user programs
→ supervisor programs.

User/Application program handles a specific application such as a word processing program, that are of interest to computer users.

On the other hand, supervisor programs manage the routine functions of the computer system on behalf of the users. Ex: computer OS.

Instruction cycle → sequence of operations performed to finish one instruction.

$T_{clock} = (1/f)$ — further divided into clock cycles.

Examples of supervisory functions → ✓ controlling graphics interface
✓ transferring data between secondary and main memory.

Interrupts → It is generally useful to design a CPU so that it can receive requests for supervisor service directly from secondary memory units and other I/O devices. Such a request is called 'interrupt'

(There are high-priority requests)

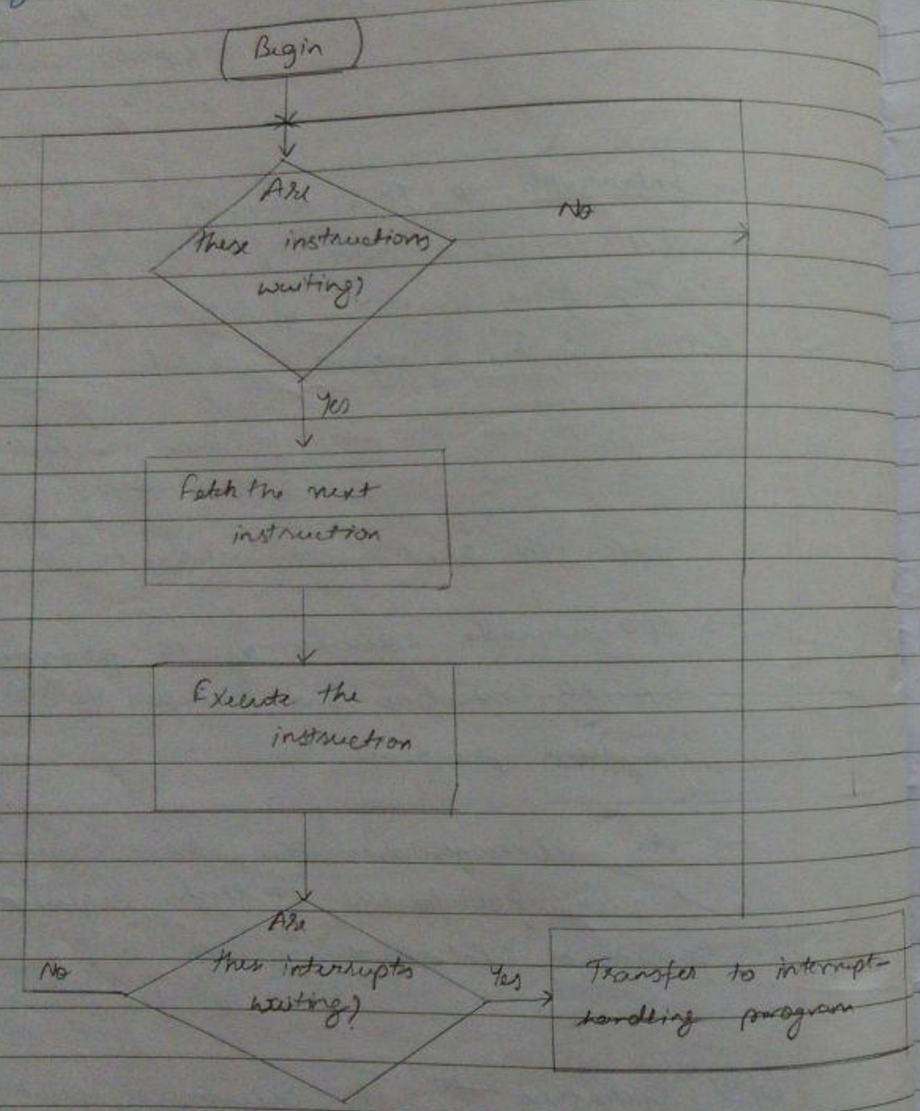
In the event of an interrupt;

→ CPU suspends execution of the program that it is currently executing and transfers to an interrupt handling program.

→ As interrupts from I/O devices require rapid response from the CPU, it checks frequently for the presence of interrupt requests.

CPU Operation → The sequence of operations performed by the CPU in processing an instruction constitutes an 'instruction' cycle. While the details of the instructions vary with the type of instruction, all instructions require 2 major steps: ① fetch step during which new inst. is read from external memory M
② execute step

⇒ To check for pending (interrupts requests) is usually included in the instruction cycle.



⇒ Overview of CPU structure

Accumulator-based CPU ⇒ CPU design continues to be based on the premise that the CPU should be as fast as the available technology and overall design will allow.

Since cost generally increases with design complexity, the number of CPU components is kept relatively small.

IAS computer proposed by Von Neumann & his colleagues follows basis for subsequent designs.

→ It comprises a small set of registers and the circuits needed to execute a functionally complete set of instructions.

→ In many early designs, the Accumulator, a CPU register
- ✓ stored input or output operand
- ✓ stored result.

in many instruction executions.

Register based basic CPU (early)

- ✓ used by first generation and typical low cost microcontrollers.
- ✓ word size is fixed in bit (assumptions)
- ✓ instructions can be adequately specified by means of register-transfer operations in our HDL.
- ✓ Instructions fetched by PCU, which has ~~it~~ main register PC or Program counter.
- ✓ instructions are executed in DPU.

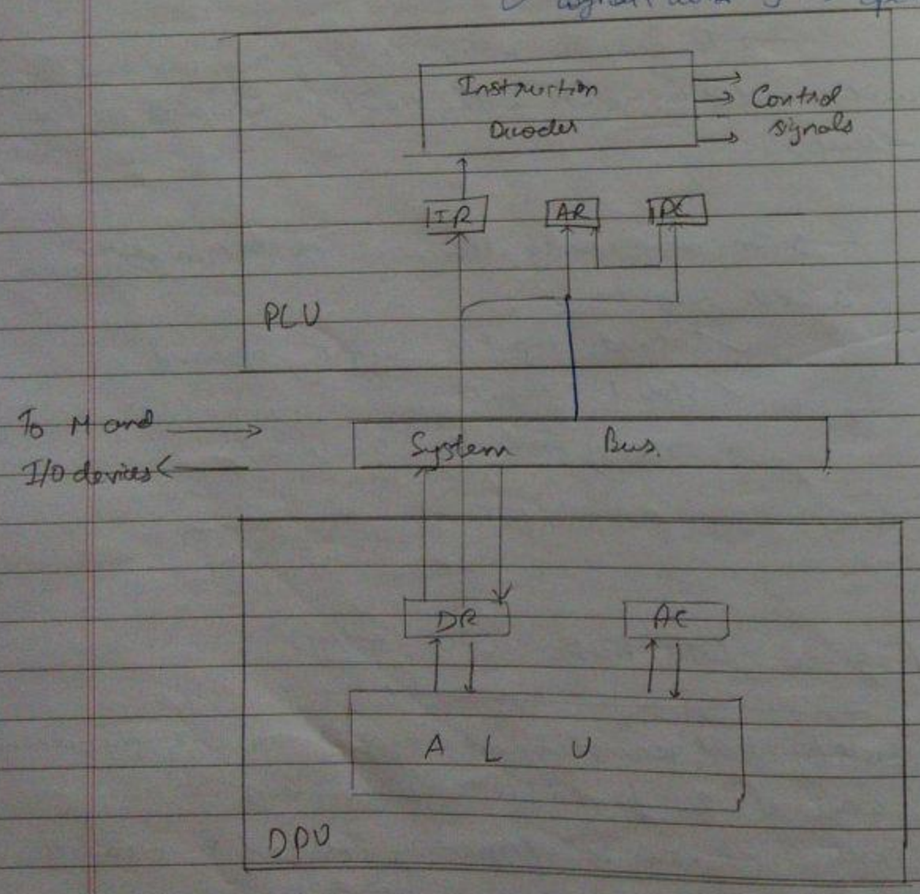
✓ DPU consists of 1 bit 'ALU' and 2 data registers DR and AC.

✓ Most instructions ~~executed~~ are of the form

$$X1 := f_i(X_1, X_2)$$

X_1 & X_2 denote a CPU register (PC, AC or DR), or an external memory M(adr).

f_i → operation performed by the ALU are limited to fixed-point (integer) : adder and subtraction,
✓ shifting
✓ logical (word-gate) operations.



A small accumulator based CPU.

Program Control Unit (PCU)

AR : Address Register
IR : Instruction Register
PC : Program Counter

Data Processing Unit (DPU)

AC : Accumulator register
DR : Data Register

Memory addresses are stored in 2 add. registers in PCU

① the program counter PC (for instruction add)
② general purpose data register. (AR). (address)

I must refer to data word in M → external memory location adr.

has 2 parts
✓ opcode 'op'
✓ memory address 'adr'

$I = (op \cdot adr)$

Each instruction cycle begins with the instruction fetch operation.

IR . AR := M(PC);

→ transfers the int. word I from M to the CPU.

The 'op' is loaded into PCU's instruction register IR, and address 'adr' is loaded into address register AR.

IR := op , AR := adr

Instructions that dont reference M dont use AR, this opcode part specifies CPU register to use, as well as fi to be carried out.

Once, opcode of I has been placed in the IR, the CPU proceeds to decode and execute it.

Note: CPU can increment PC in order to obtain the address of the next instruction.

Two imp. memory addressing instructions are

- load — LD adr — AC := M(adrs)
- store — M(adrs) := AC; → ST adr

HDL	Assembly lang.	Narrative
AC := M(X);	LD X	Load x from M into acc. AC
DR := AC;	MOV DR, AC	Move contents of AC to DR
AC := M(Y);	LD Y	Load Y into acc AC
AC := AC + DR;	ADD	Add DR to AC
M(2) := AC	ST Z	Store contents of AC in M

↑ The above program is using the load/store feature and is hence called [Load/store architecture]

8/7/14

The CPU design in the previous diagram can be designed to implement

AC := fi(AC, M(adrs))

Thus, we can optimize the prev. code fragment

HDL	Assembly lang.	Narrative
AC := M(X)	LD X	Load x from M in AC
AC := AC + M(Y)	ADD Y	Load Y in DR & add AC
M(Z) := AC	ST Z	Store content of Y in M.

→ The ADD Y instruction can be expected to take longer to execute than the original ADD instr. that refers only to CPU registers. (Memory ref. complicate instr-decoding logic in CPU)

→ However, overall exe. time is reduced as we have reduced LD & MV. (eliminated).

- RISC → Reduced Instruction Set Computing follows (Load & store) Architecture
- CISC → Complex Instruction Set Computing.

(minimal & easy & reduced)

1 instr. may complete the whole task ✓ interval decoding of inst. is complex.

8085 microprocessor
- → 8 bit processor (each reg. is of 8 bits)
- → address bus is of 16 bits
- → instruction size can vary.

SP → stack pointer → stores stack memory (of 16 bit as add. bus is of 16 bit)

Registers available are

8085

- A
- B C →
- D E →
- H L →

← general purpose reg

SP →
PC →

Flag → tells some conditions after execution of instruction.
C, S, Z, P, AC' — sign, carry, auxillary carry, parity, zero
↓ used in decimal operations

Q What is Program Status Word in reference to the 8085 microprocessor ? (PSW)
(Is it same as flag register?)

⇒ A register containing status flag values + AL register content

Arithmetic and Data Transfer Instructions

Commands

MOV Rd, Rs ← dest. reg / ← source reg

MOV A, B

→ In reality data is being copied from B to A.

Size of this instruction in opcode would be → 1 byte.

⇒ This entire thing would be represented by a single opcode.
(Instruction)

(Opcode) ↓ opcode

MOV A, B ⇒ 1 byte only (78)

MOV B, A ⇒ 1 byte only (47) ← opcode

MOV A, C ⇒ 1 byte only (79) ← opcode

MVI B, 47 H ⇒ move immediately data specified by '47' opcode into register B.

(move immediately to the specified address)

(06 / 47) ← opcode

Say so is the data, then it would be stored (copied) to register B.

⇒ Again its a 2 byte instruction.

MVI B 57H → 06 (opcode) / 57

⇒ 2 byte instruction.

Opcode → gives a shorter instruction to the processor.

MVI D 47H (opcode) (16 / 47) → 2 byte instruction

Add Register

ADD R
ADD B
↳ A = A + B → Add content of B to content of A and store the result in A.

⇒ 2 byte instruction set

Simply
ADD C ⇒ AC := AC + C
→ Add content of register C to the content of the accumulator and store the result in the accumulator.

⇒ 2 byte instruction.

Instruction	Opcode
ADD A	87
ADD B	80
ADD C	81
ADI C6	58 → C6

ADI C6 ↳ Add content of C6 to AC & store in AC (immediately)
↳ Add immediately to and store in AC.

(+)
No ambiguity ⇒ ADI C6 — C6 ← } 2 byte instr.
on which C6 or C6 ←
C0 to be selected?

ADI C0 — C0 ← } 2 byte instr
(+) C0 ←

reason { opcode knows the size of instruction so it would know that 'C0' [and one] is to be used not the first one

SUB A → Subtract content of A from content of AC and then store in AC.

AC := AC − A. — 1 byte instruction.

SUB A, B → A := A − B
2 byte instruction

→ Subtract content of B from the content of A and store it in A.

1 byte instruction { INR Rg → Increment register by 1 ; INR B → " " B " ; INR C → increment → (C := C + 1) }

ADI A, 01 → 2 byte instruction.

⇒ INR is preferable as it is 1 byte instruction.

Logical Instructions ⇒

ANA A ⇒

A → 0000 0100
B → 0011 1100 AND
0000 0100 → A

ANA B ⇒
'And' content of A with content of B. and store in A.

ANI B ⇒ 'And' content of AC with content of B. and store in AC.

ORA Rg ⇒ 'Or' content of Rg with AC and store in AC.

XRA Rg ⇒ 'Xor' content of Rg with that of AC and store in AC.

CMA ⇒ Compliment ~~comp~~ contents of AC and store in AC.

11/7/14

Branch Instructions ⇒

JMP ⇒ 16-bit memory address.
↳ jump to following memory add.

1000	06
1001	02
1002	C3
1003	10
1004	20

MVI B, 06 (2 byte inst.)

JMP 2010

⇓

3 byte instruction
→ 1 to specify jump
→ 2 bytes for memory add. (10, 20)

JMP BDDA
16 bit data

JC ⇒ 16-bit memory add.
↳ jump on carry (3 byte instruction)

JNC ⇒ 16-bit memory add.
↳ jump on no carry (3 byte instructions)

similarly for JZ, JNZ, JPE, JPO → All have 16-bit memory add. + 3 byte instr.
JZ → zero
JNZ → non-zero
JPE → even parity
JPO → odd parity

reg. B is always with reg. C.

LXI B, 5060 ⇒ (immediately)
↑ Register pair
B ← 50, C ← 60 } → contents of '50' & '60'.

NOP → No operation.

HLT → Halt

ADD A → Add content of register A to accumulator (AC) and store in AC. (1 byte inst.)

MVI A, 20 → 2 byte inst.
inst. → 1 byte
and storing memory add. → +1 byte.

For storing 05 in register A →

✓ MVI A, 05 (8 bit) ⇒ correct. (used for storing 8-bit data directly in the specified register)

~~X LDA A, 05~~

LDA 3050 (16 bit) → means use data stored at location 3050 and store in AC. (Say its 01) then (AC ← 01)
(Load Accumulator)

LXI B, 3050 → store immediately 50 in B and its pair (i.e.) C should store 60.

LXI B – C
LXI D – E
LXI H – L
(Register pairs)

STA 1111 → store the contents of accumulator at the specified memory location.

=> All these instructions are specific to the Intel 8085 microprocessor.

LXI Rp, 16-bit memory address

eg: LXI B, 3010 (3 byte instruction)

MOV R, M ← Eg: MOV D, M

(memory add. from 'HL' pair)

indirectly take memory add. from registers → HL [1050] ← 16 bit add

1050 [30] ← 8 bit value.

D ← 30 ← 8 bit reg. D

Q Store 50 in register B. using the instruction MOV B, M.

Assume all memory addresses are available.

```
STA   3050(V)
STA   M(X)
MOV   Rp, M(V)
MOV   A, 3050(X)

MVI   A, 50
STA   1050
LXI   H, 1050
MVI   B, M
```

MOV M, A → whatever is the content of accumulator [50] should be stored at the memory add. pointed by HL. In this case 1050.

intially retrieved from HL

1050 → 1050 [50]

At memory locn 1050 ↓ so would be stored.

MVI M, 8 bit add → 2 byte inst.

retrieves data from add. (say 20)

retrieves add from HL reg.

address → [20]

=> retrieves data from 8-bit add. and stores it in the memory add. referenced by the 'HL' register pair.

INX B → 1 byte

B	C
00	01
00	02
00	FF
01	00

INR A → 1 byte inst

no new add specified

increment value of register pair by 1.

DCX B → 1 byte => decrement value of reg. pair by 1.

B	C
00	03
00	02
01	00
00	FF

Load => Accumulator ← recieves data from memory

LDAX B } → retrieve the content of the memory add. pointed by
LDAX D } the register pair 'B-C' and store it in the accumulator.

↳ similarly consider the reg. pair D-E and store in the accumulator.

Store => Accumulator → data transferred to memory

STAX B → 0607 → 05 stored from A
STAX D → 0809 → 05 stored from A

A → 05
B → 06
C → 07
D → 08
E → 09

0607 → 03 (stored initially)
0809 → 07 (")

LDA 1676 → load content from this memory add. in A
↑ memory add.

INX SP
↳ (stack pointer)

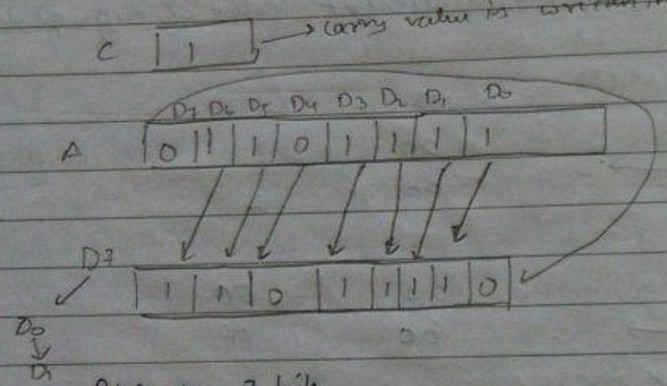
Rotate functions

RLC | → work on the left shifting operation.
RAL | → rotate left

Rotate right

RRC | → work on right shifting operation
RAR |

~~RLC~~
~~RAL~~ → Rotate Accumulator left through carry
RAL



RLC => 8-bits.

RAL => 9-bit data.

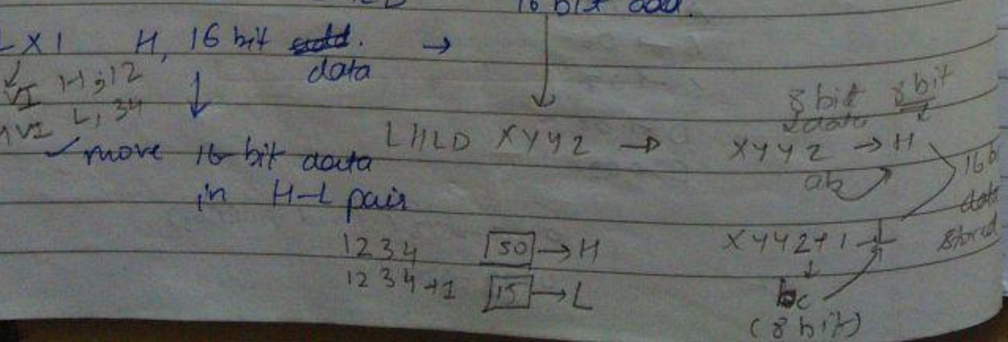
RRC => rotate right by shifting carry.
RAR => rotate right thru carry.

15/7/14

LHLD → directly stores data from the specified 16 bit memory address in HL registers

Ex: LHLD 16 bit add. →

~~same~~ LXI H, 16 bit ~~add~~. data →
MVI H, 12
MVI L, 34
↓ move 16 bit data in H-L pair



15/7/14

RISC Architecture

LD x
ST x
MOV DR, AC
MOV AC, DR
ADD
SUB
AND
NOT
BRA addr
~~BR~~ BZ addr.

→ Negate the value stored in Accumulator.

~~MOV DR, AC~~
{ MOV DR, AC
SUB
SUB

→ Perform the multiplication AC = AC x N
Multiply Y with N. ↳

AC + AC + AC + AC } N times

~~ST 1234~~

~~MOV AC, 0000H~~
~~LD 0000H~~
~~ST MOV DR, AC~~

BZ end
LD 0000H
MOV DR, AC
LD 0001H
~~MOV~~ SUB
ST 0001H
LD 0002H
MOV DR, AC
ADD

ST 0003H
BRA multi
~~BZ~~
end:

0000)
0001 N
0002 Y
0003 Product

Institute of Advanced Science

18/7/14

The IAS Computer

Data bus
Address of 12 bits
40 bits x 4k
Add. Bus
= memory = no of addresses
= 4k (2^{12} ($2^{10} = 1024 \approx 1k$, $2^2 = 4$)

data bus = 12

Addn features

→ Architecture Extensions

1. Multipurpose register set in storing data & add.
2. Additional data, instruction & address types.
3. Register to indicate computation status
4. Program control stack
5. Speed up techniques — Caching
   — Pipelining

Q Which are the diff. address types?
— for some instructions, operand is and add.; for some add. of operand is spec. in the program instr.

Q A machine has a 32 bit architecture with 1 word long instruction. It has 64 registers each of which 32 bits long. It needs to support 45 instr. which have an immediate operand in addition to two register operands. How many bits can be used for immediate operand? → 14 bits

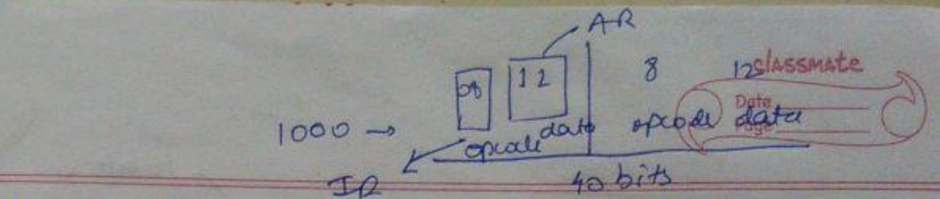
→

6 bits	6	6	14
R_1	R_2	R_3	32 bits

Immediate operand

$45 \leq 2^6$ ∴ 45×2^6

0 - A
1 - 3
2 - C
:
:
63 registers



PSW →

7	6	5	4	3	2	1	0
S	Z	X	AC	X	P	X	CS
Sign	Zero		Aux. Carry		Parity		Carry status

Unidentified / Undefined

Combn of these 8 bits is called PSW.

→ PSW and AC are treated as 16 bit-units for stack operations.

22/7/14

	8085	8086 (8 bits)	IAS
Data bus	8	16	8
Address bus	16	20	40

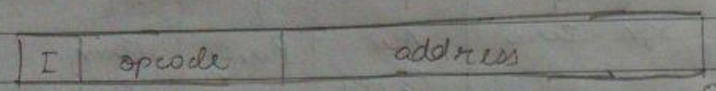
(8 bits) on each memory add.

BASIC computer (Mano machine)

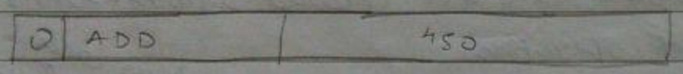
- one accumulator
- 12-bit address bus
- 16-bit data bus
- 8-bit i/o bus
- 25 instructions
- PC; AR → 12 bits
- DR, AC, IR, TR → 16 bits
- INPR, OUTR - 8 bits
- Interrupt flags →
- Single bit reg. → Carry flag & halt

[450]

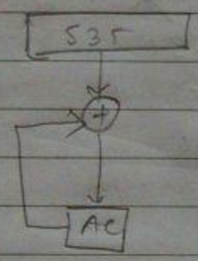
All instructions are 1 word long. → 16 bit.



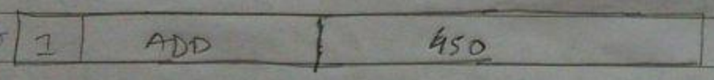
I=0 → direct addressing mode [LDA 0450/STA 0450]
I=1 → indirect addressing mode [MOV A,M]

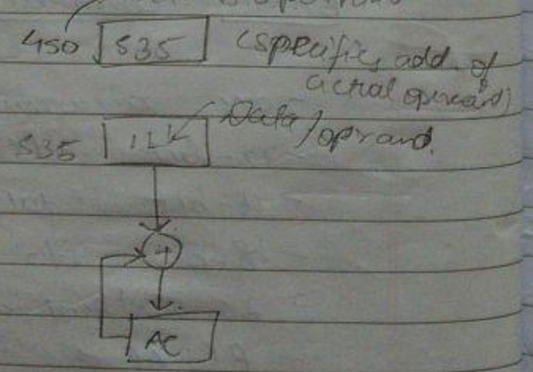


Direct

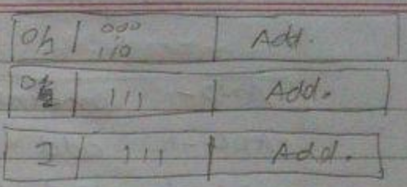


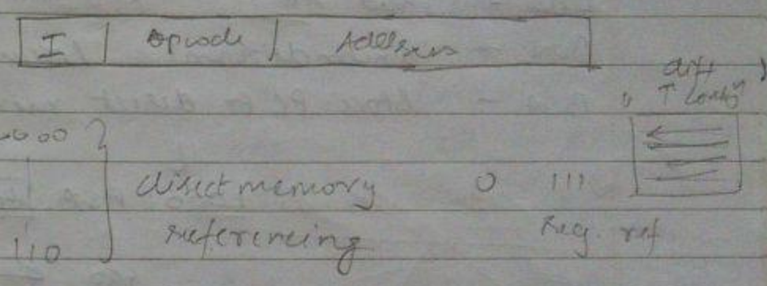
Indirect





1. Memory reference Instrs.
2. Register " "
3. I/O " "





15	14 13 12	11 ... 0	Mnemonic
0	0	Direct address	AND
0	1	"	ADD
0	2	"	LDA
0	3	"	STA
0	4	"	BUN
0	5	"	BSA
0	6	"	ISZ
0			

PC → 8 bits
Accumulator → 16 bits

Description

AND - anding AC with content from direct add.
ADD - add " " " "
LDA - load AC " " " "
BUN - unconditional branch to following address.
BSA - Store PC to direct memory and branch to following add.

1000 BSA 450

450 ← 1000 (→ content of PC)
PC ← 451 go to (PC+1)

ISZ - increment value in direct memory & skip to next inst. if sum is 0.

ISZ 012

~~INC~~.

M[AR] ← M[AR] + 1
if M(AR) = 0
PC = PC+1

1000 ISZ 4000 ~~if 1000 ISZ ffff~~

[4000] → 50
+1
51 ≠ 0

if 4000 → ffff
+ 1
0

INC PC

25/7/14 ⇒

15	14-13-12	11 — 0	
0	7	200	CLA
0	7	400	CLE
0	7	200	CMA
0	7	100	CME
0	7	080	CIR — RAR
0	7	040	CIL — RAL (thro carry)
0	7	020	INC → thro carry
0	7	010	SPA — skip next inst. if AC>0
0	7	008	SNA — skip next inst. if AC<0
0	7	004	SZA — skip next inst. if AC=0
0	7	002	SZE — skip if carry bit 0.
0	7	001	HLT — Halt
1	7	800	INP → input to AC from data bus
1	7	400	OUT → put data from AC to data bus
1	7	200	SKI → skip next inst. if INP flag is 1
1	7	100	SKO → skip next Inst. if OUP flag is 1
1	7	080	ION → enable interrupt
1	7	040	IOF → disable interrupt.

Addressing modes

- Fetch → opcode fetched
- Decode → operands, operation determined
- Execute → execute.

→ implied mode ⇒ nothing is specified.
Eg: CMA (8085)
RAR / RAL / RRC / ..
HLT

Immediate Mode ⇒ The operand is present. Nothing else is being specified.

Ex. MVI 08H
ADI 2040H

Register mode ⇒ Registers are used

MOV A, B

- Direct: MOV A, B
- Indirect: MOV A, M / LDAX B

Autoincrement (use, ↑) **Autodecrement** (–, use) **Mode** ⇒ used in register indirect mode (not like INR & DCR)

- Modifies content of the registers.
- (use the content)

Effective ~~mode~~ Address ⇒ is defined to be the memory add. obtained from the computation dictated by the given addressing mode.

3000 ← ADD 2000

for direct ⇒ effective add.
for indirect ⇒ 4000 → effective add.

Relative Address mode ⇒

Here Offset is specified in the instruction.

Effective Add. = PC + (Part of add. specified in the instruction)
offset

- used in branch inst.

Advantage → Whenever branch inst. is used,
- memory space reqd. is less
- relocatability is achieved.

Index Addressing Mode ⇒

Index Rg. + Add. spec. in instruction = Effective add.

(if processor supports Index register)

Base Register Addressing Mode ⇒

Effective Add. = Base Add. + Add. specif. in inst.

Examples ⇒ PC = 200 (↑ Processor reg.)
R1 = 400
XR = 100 (↑ Index reg.)
AC

AC

PC = 200
RI = 400 → Reg.
XR = 100 → index reg.

As PC is 202

Add.	Memory	
200	LOAD to AC | Mode	2 byte
201	Add. = 500	
202	next inst.	
	⋮	
399	450	
400	700	
401	702	
	⋮	
500	800	
	⋮	
600	900	
	⋮	
700	300	
702	325	
800	300	

	EA	AC
Direct Add.	500	800
Indirect Add.	800	300
Immediate operand	~~201~~	500
Relative add.	(200+202) 700+2 =702	~~325~~ 325
Indexed Add.	600	900

(content of reg. is to be transferred.)

Register Add.	EA	AC
	~~400~~ -	~~700~~ 400
Reg. Indirect	400	~~700~~
Auto Increment	400 ~~702~~ (700/2)	700 ~~325~~ (399)
Auto Decrement	399	450

Assignment

⇒ Indexed Reg. Addressing mode → There is a index reg. in some processor that store the address of an address.

(IR + address specified in the instruction)

- looping
- jumping
- counter

II nd Phase

5/8/14

K – kb → kbits
KB – k bytes.

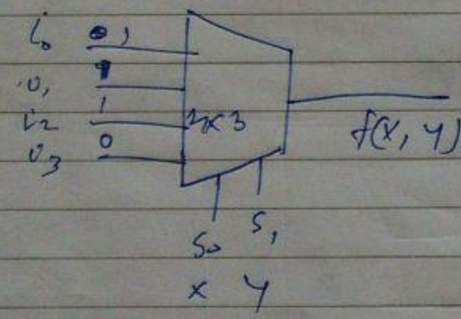
$\frac{24}{8} = 3$ bytes.

$\frac{600}{3} = 200$

$2^{32} \times 8$ bits $= 2^{32}$ bytes

$2^{32} \times 8 \times 2 = 2^{32} \times 2$ bytes

Chapter 5 Control Logic



S_0	S_1	f(x, y)
0	0	1
0	1	1
1	0	1
1	1	0

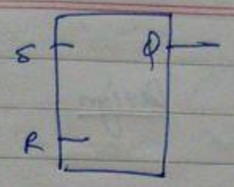
NAND gate

flip-flop – A bit of data.

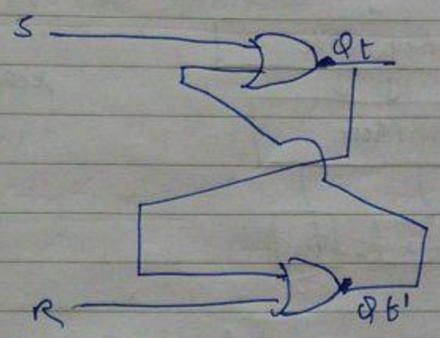
Read
write } → from/to register

↓ depends on type of triggering.

edge → both
level → one



Q_t	S	R	Q_{t+1}	
0	0	0	0	same
1	0	1	0	0
0	1	0	1	1
1	1	1	ND	

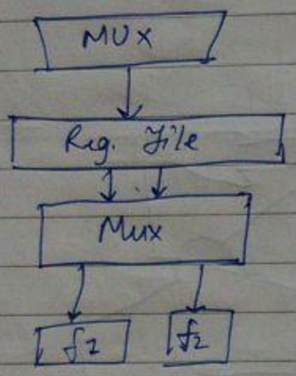


Ass. Can level triggered & edge triggered opn. have read/write in the same cycle.
→ used in master slave flip flop.

– edge triggered flip-flop can have read/write in the same cycle.

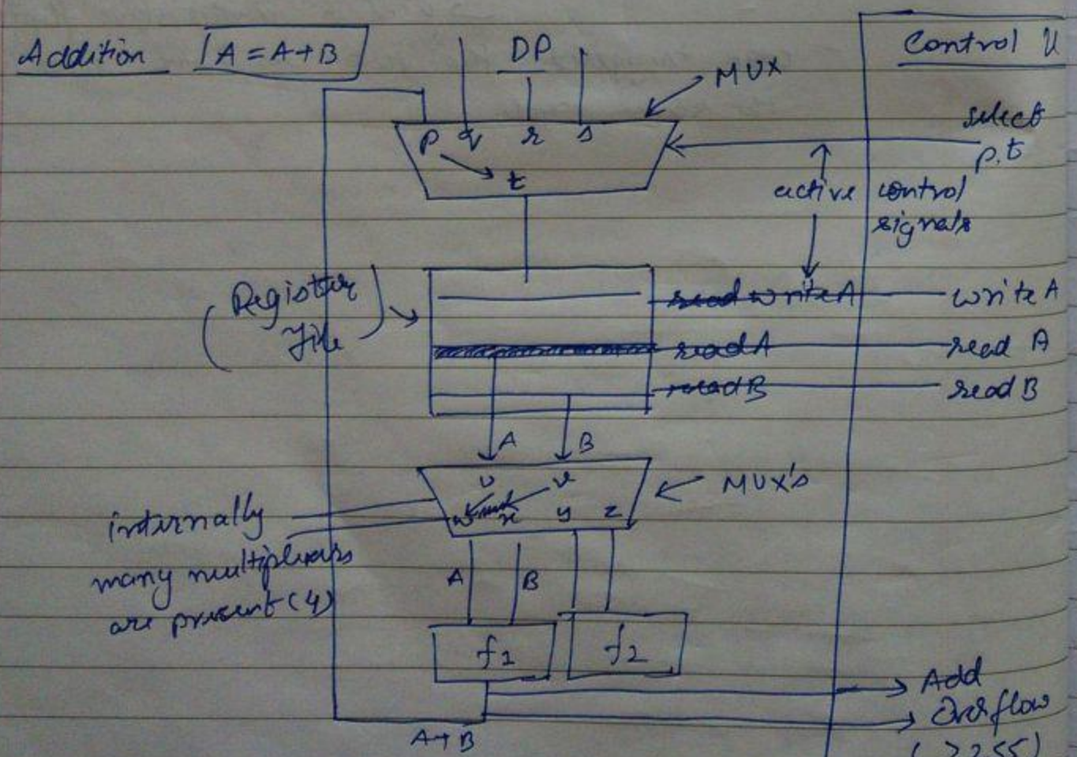
12/8/14

Chapter 5
Control Design



A = A + B
read A read B.

Control signals generated upon: →
① which function is to be used.
② which registers are to be used.



Cycle

1 — Write A, Read A, Read B — Select pt, Select u.w, Select v.x — Add

For 16-bit addition

AX
BX

AH AL
+BH +BL
+AC

AH AL

Cycle

1 — Write A, Read A, Read B — Select pt, Select uw, Select vx — Add

2 — 4 — " — Add with carry

Control Unit can be of two types → Hardwired / Microprogrammed.

loop: if cond 1 = true then add
else sub

if cond 2 = true then output
else loop

output : ------

classmate
Date
Page

Current state	i/p	next state	function
Add 1	cond1 = true	add2	Add
	cond1 = false	add2	Sub
Add 2	cond2 = true	add4	——
	cond2 = false	add 1	——

i/p	o/p
gcd (x, y)	z
gcd (12, 18)	6

Pseudocode →
```
gcd (x,y)
{ if (y > x)
  { gcd(x,y)
    18 % 12          x = 18
  y = y % x          y = 12
  } gcd (x,y)
  else
    o/p 'y'
  }
```

13 15

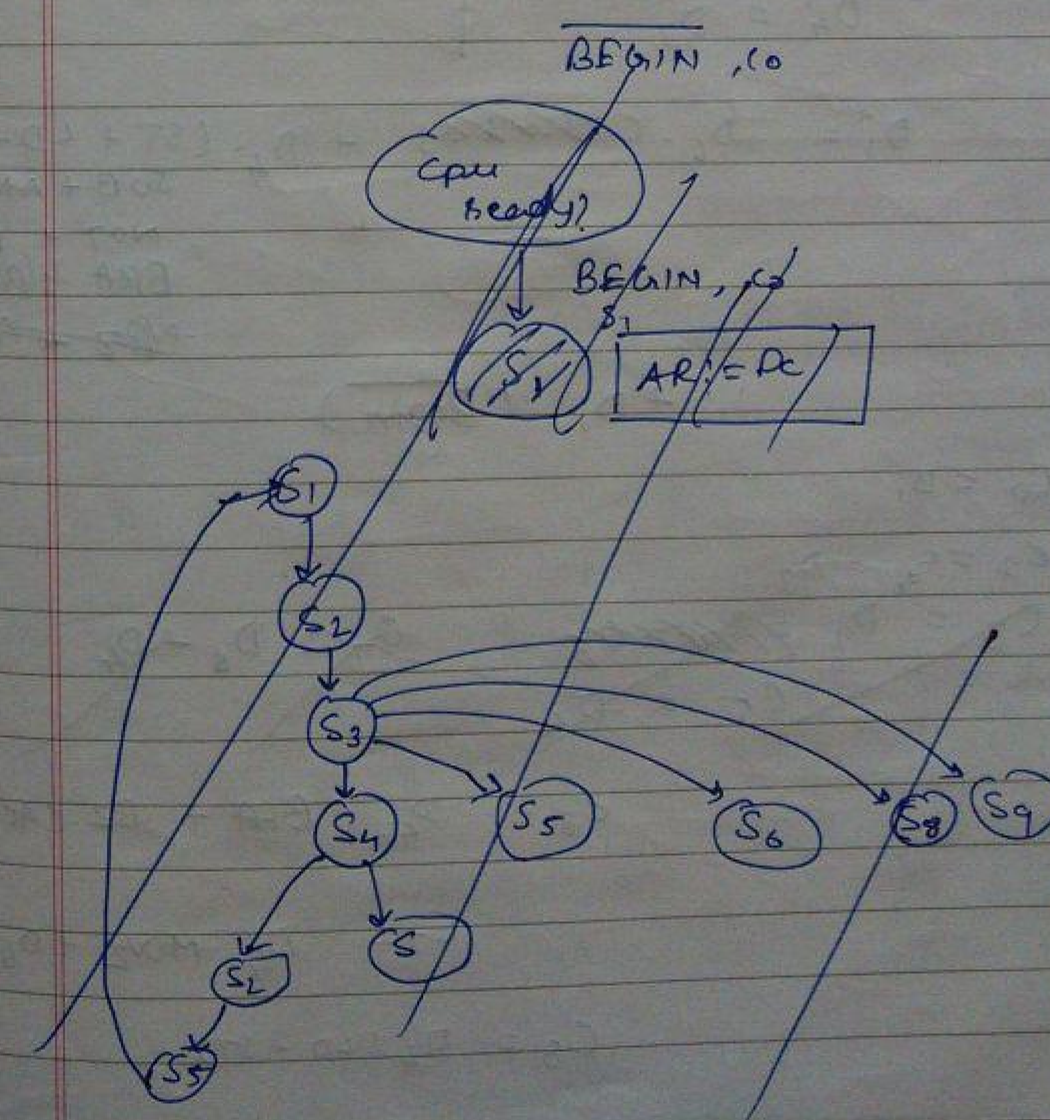
```
gcd(x, y)
{  while(x! = y)
   {if( y>x || x>y)
     {
       z = y % x
     } if(x==0 or
  }    exit  y==0)
```

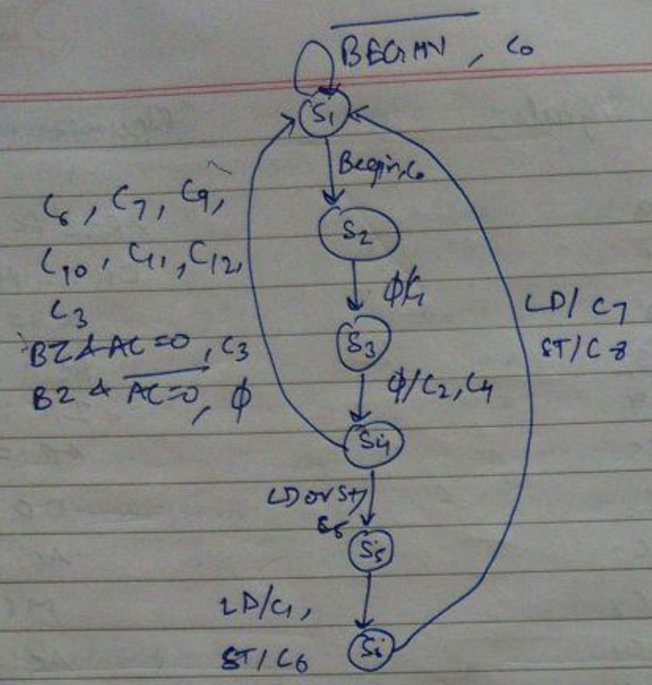
```
while (x!=0)
{
  if(x>y)
    x = x-y
  els
    y=y-x
}
o/p z = x
if (x==0 || y==0)
op '0'
```

13,15
15 % 13 = 2

classmate
Date
Page

Control Signals	Operation
C_0	AR := PC
C_1	DR := M(AR)
C_2	PC := PC + 1
C_3	PC := DR(ADR)
C_4	IR := DR(OP)
C_5	AR := DR(ADR)
C_6	DR := AC
C_7	AC := DR
C_8	M(AR) := DR
C_9	AC := AC + DR
C_{10}	AC := AC - DR
C_{11}	AC := AC and DR
C_{12}	AC := not AC





$D_2^+ = D_1 . Begin$

$D_3^+ = D_2$

$D_4^+ = D_3$

$D_5^+ = \overline{D_4} . (LD + ST)$

$D_6^+ = D_5 . (LD)$

$D_1^+ = D_6 . (ST + LD + ADD + SUB + AND + NOT + BRA + (BZ ~~& AC=0~~)) + D_4 . (ST + LD + ADD + SUB + AND + NOT + BRA + (BZ ~~& AC=0~~)) + D_1 . (\overline{Begin})$

$C_0 = D_1$

$C_2 = C_4 = D_3$

$C_3 = D_4$ ~~+ ADD + SUB + AND + NOT~~

$C_5 = D_4$

$C_7 = D_6 + D_4$

$C_3 = D_4 . (BRA + BZ . (AC=0))$

$C_7 = D_4 . MOV_2 + D_6 . LD$

$C_5 = D_4 (LD + ST)$

9/9/14

12/9/14

CMAR - Control Memory Address Register

Address field - which instn. is to be executed next.

3rd Sessional

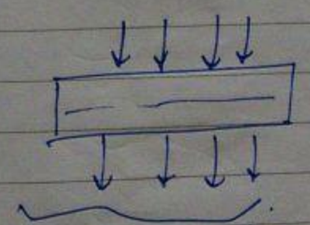
Classical — faster ↔ → slower (Why?) — One-hot

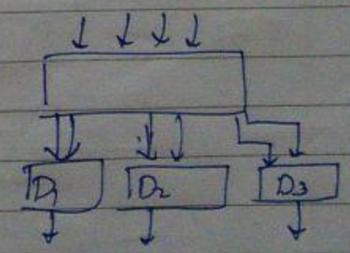
if(x % 2 == 0)	if(x and 1)
x ← x+1	x ← x-1
else	else
x ← x-1	x ← x+1

Microprogrammed Control Unit

- ROM
- if RAM then WCM → Writable Control Memory.

→ Microprograms contains microinstructions.
→ Microinstruction contains microoperation(s).

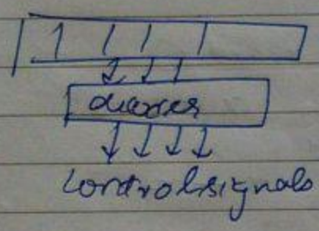


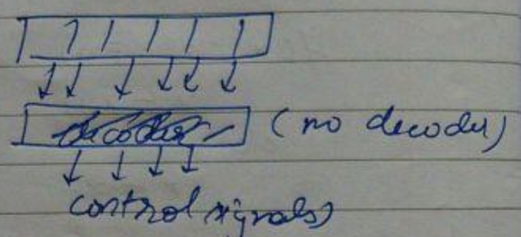


Format of microinstructions :→

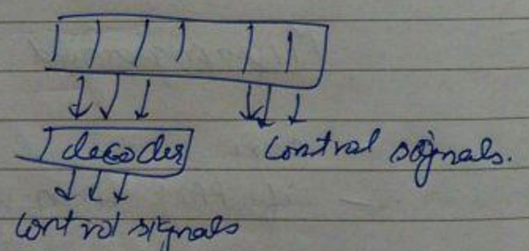
Vertical format
- shorter format
- less parallelism
- ~~less~~ more encoding

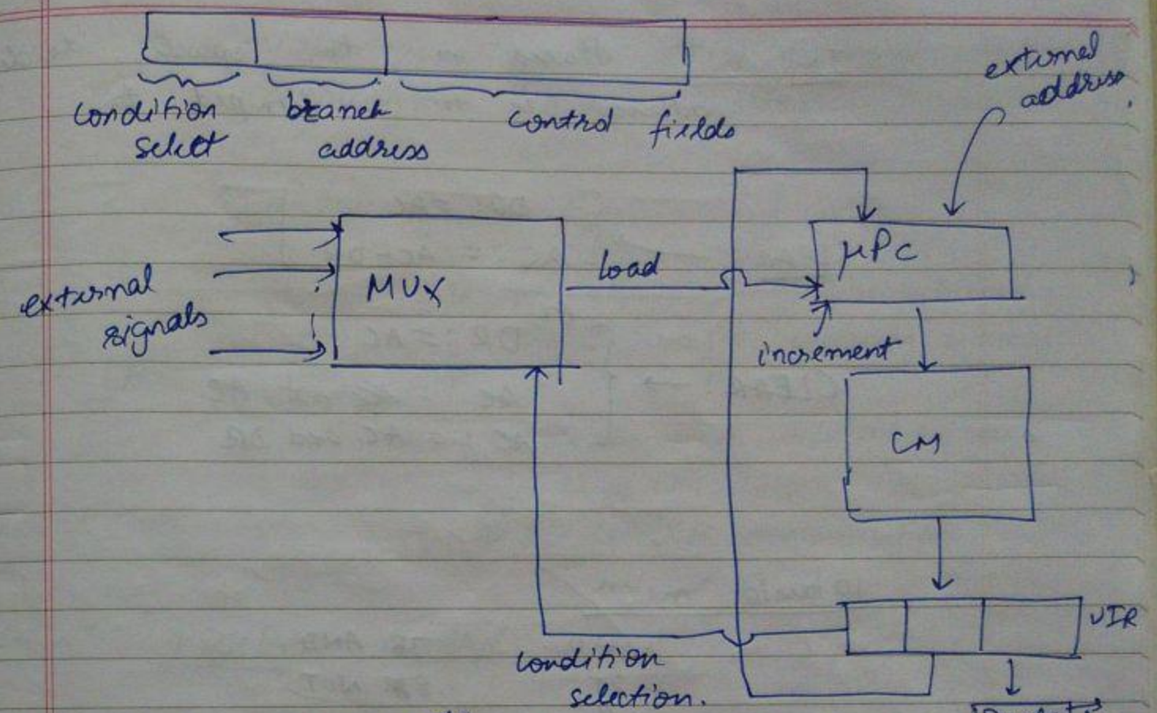
horizontal format
- longa format
- more parallelism
- less encoding





moderate format ⇒





μPc → micro program counter.
CM → control memory.
(microinstn)
μIR → microInstn Register.

Accumulator based CPU's Control Unit →

Basic Emulator :

FETCH: AR := PC
DR := in(AR)
PC := PC+1, IR := DR(OP)
goto IR

LD: AR := DR(Adr)
DR := M(AR)
AC := DR, goto FETCH

SUB: AC := AC - DR, goto FETCH

IR: Based on the opcode, decide which address is to be jumped to..

CLEAR: → { DR := AC
AC := AC - DR }

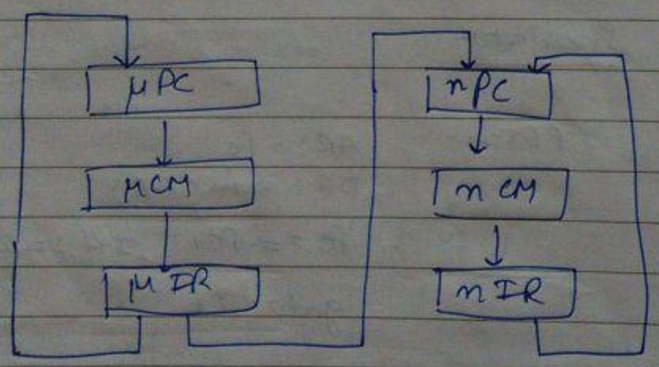
or

CLEAR: → { DR := AC
AC := ~~AC~~ not AC
AC := AC and DR

10 basic instn

1. LD
2. ST
3. Branch
4. BZ
5. SUB
6. ADD
7. AND
8. NOT
9. ~~AC~~ ← DR
10. DR ← AC

Nanoprogramming: first we go to a CM, from there, we again go to a CM.



* => microprogram is to actual program what nanoprogram is to microprogram.

Advantage of nanoprogramming: →

1) parallelism can be achieved.
2) size of control memory reduced.

Loosening of bonds between instn & hw.

Disadvantage: →

1> access time increases as 2 levels of CM.
2> complex organisation.

Pipeline Control:

1) Instruction pipeline:
2) Arithmetic pipelining.

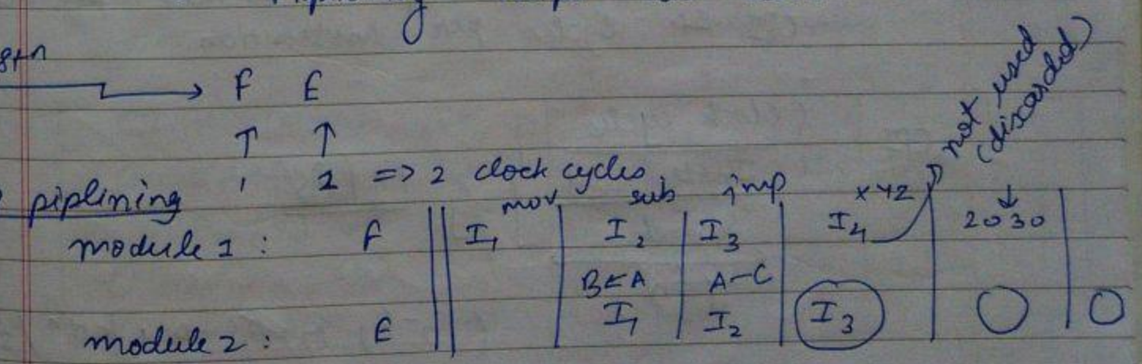
CPI: Clock ~~cycle~~ cycles per instruction.

The optimal CPI that can be achieved is 1.

→ Pipelining helps us achieve lower CPI.

Instn → F E
T T
1 1 => 2 clock cycles

using pipelining



8085
MOV B, A;
SUB C
JMP 20 30
} { I1 I2 I3

P I_1 — I_2 — —
E I_1 — I_2 — — } without pipelining

1000 instn → 2000 clock cycles.

1 I => 2 clock cycles.

P I_1 I_2 I_3 I_4 — —
E I_1 I_2 I_3 — —

2 I => 3 clock cycles
nI = (n+1) " "
1000 I = 1001 clock cycles.

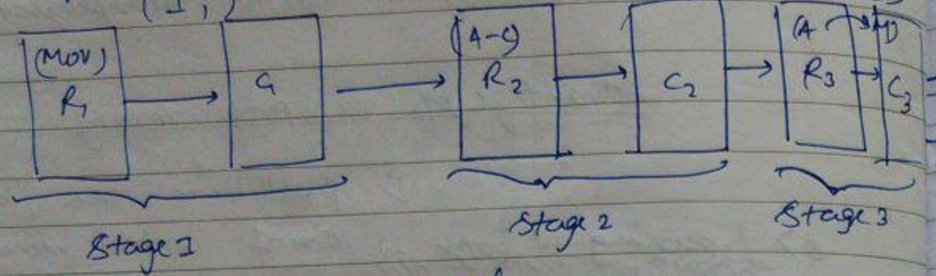
Total ~~time~~ clock cycles / sec

⇓

(CPI x MIPS x 10^{-6})

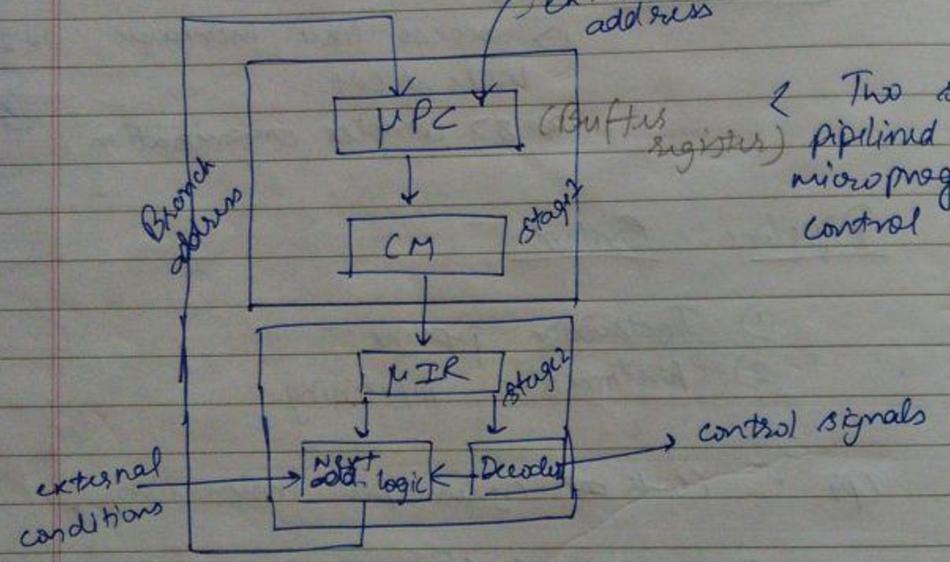
frequency of microprocessor = (MIPS x CPI)

Pipeline Control



4 stages for each instn.

(Instn fetch) IF — ~~I-cache~~ (I-cache)
(Operand Load) OL — (D-cache)
(Execution) Ex — memory access used.
(Operand store) OS — (D-cache)



< Two staged pipelined microprogrammed control unit >

CPU ⟷ cache → I-cache, D-cache.

D-cache → would be req'd. to read & write data simultaneously

without pipelining

IF	I_1	x	x		I_2	
OL	x	I_2	x		x	I_3
Ex	x	x	I_3	x		
OS	x	x	x	I_4		
	1	2	3	4	5	...

(4 clock cycles)

perform superscalar processing (CPI < 1). or greater than one instruction executed per clock cycle.

Superscalar m/c → The (fetching + execution) units are increased. Hence, multiple instructions can be executed at the same time.

Ex: with pipelining

		1	2	3	4	5	6	7	8
(Instn Fetch)	IF	I_1	I_2						
(Read from Reg)	RD	x	I_1	I_2					
(Exec^n)	Ex	x	x	I_1	D	I_2			
(Memory Access)	MA	x	x	x	I_1	D	I_2		
(Write Back)	WB	x	x	x	x	I_1	x	I_2	I_3

D → Stall clock cycle.

16/9/14

MIPS → Millions of instructions per second.
CPI → Cycles per instruction.

CPI { Clock cycles / Instructions } } } MIPS
{ Seconds / (fn.) }

(10^{-6}) MIPS → 1 Instn / sec.

(CPI)

→ Suppose an instn loads a data word x into into a register and is immediately followed by an instn that uses x in execution stage.

8085

①	I_1	LDA 3050	(5)	
	I_2	ADD B	(7)	
	I_3	MOV C, D	(8) / (6)	
			(using NOP)	(using I_3)

3050	1
A	2
B	3
C	4
D	5

possible solutions

② Use NOP (No operation) instead of ~~Dat~~ D.

1> stall clock cycle (1C) stall clock cycle.

2> use NOP 3> insert another instn whenever this case arises such that ~~no other~~ it doesn't depend on the present op.

load
ADD
SUB } (if operands used for these are not same)

S_1	I_1	I_2	x	x	x	x
S_2	x	I_1	x	I_2	x	x
S_3	x	x	I_1	x	I_2	x
S_4	x	x	x	I_1	x	I_2
	1	2	3	4	5	6

speed up factor = (8/6) = (4/3) = 1.33

Calculate the efficiency / utilization.

I → utilized (8)

x → non utilized. (24) − 8 = 16

efficiency

$(\frac{8}{30}) \times 100 = 26.67\%$ 0.33

$(\frac{8}{24}) \times 100 = 33.3\%$. in (1.)

* Efficiency = 1 can never be achieved.

Speed up = $\frac{S(m)}{}$ = $\frac{T(1)}{T(m)}$ → no pipelining
→ 'm' stages of pipelining
→ larges than (m)

(non-pipelined / pipelined)

Efficiency = $\frac{E(m)}{} = \frac{8}{24}$

If we are having 4 stages the

$S(4) = \frac{T(1)}{T(4)}$

Thus, $T(1) \leq m\,T(m)$.

relation betwn S(m) & E(m) ⇒

$\frac{8}{6} = 4 \times (\frac{8}{24})$

$E(m) \propto S(m)$

$S(m) = m.E(m)$?

PCR (Performance cost Ratio)

$PCR = \frac{f}{k}$ → (~~MIPS~~) clock frequency

↓ cost consumption for H/W

a → delay of non-pipelined ..

$\frac{a}{m}$ ⇒ delay of each stage.

b ⇒ delay due to buffer Reg.

pipeline clock period $T_c = \frac{a}{m} + b$

$k = cm + d$ → cost of data logic reg. processing
↳ cost of buffer reg / stage

$PCR^{-1} = T_c \cdot k$

$= \left(\frac{a}{m} + b\right)(cm + d)$

~~ac + a~~

$= \left(\frac{a + bm}{m}\right)(cm + d)$

$PCR^{-1} = \left(\frac{acm + bcm^2 + bdm + ad}{m}\right)$

$\Rightarrow PCR = \left(\frac{m}{\underbrace{acm + bcm^2 + bdm + ad}_{x}}\right)$

Differentiating PCR wrt m;

$PCR' = \frac{(1)(x) - m(ac + 2bcm + bd)}{(acm + bcm^2 + bdm + ad)^2}$

$= \left(\frac{x - m(ac + bd + 2bcm)}{x^2}\right)$

$= \frac{acm + bcm^2 + bdm + da - acm - bdm - 2bcm^2}{x^2}$

$$\frac{d}{dm}(PCR) = \left(\frac{ad - bcm^2}{x^2}\right)$$

reservation table → shows stage usage in every clock cycle while I is being executed.

For optimum value,

$PCR' = 0$

Initiation latency → delay occurring between 2 successive, collision free pipeline instn is called initiation latency.

$\Rightarrow ad - bcm^2 = 0$

$\Rightarrow ad = bcm^2 \Rightarrow m = \sqrt{\frac{ad}{bc}}$

Optimal value of m is given by

$$m_{opt} = \sqrt{\frac{ad}{bc}}$$

14/9/14 Collision

Reservation Table

[pictorial representation of which clock cycle and instn is currently being executed.]

tells when we are supposed to start execute next instn.

	1	2	3
S_1	x		
S_2		x	
S_3			x

Forbidden list of Reservation Table.

1 instn → 2nd instn.

	1	2	3	4	5	6	7	8	9	10	11
S_1	x	x	✓	✓		x		(+)			
S_2		x	x	✓	✓		x	(+)	(+)		
S_3			x	x	✓	✓		x		(+)	
S_4				x	x	✓	✓		x		(+)

→ Reservation Table for addition of 2 nos.

$S_1 \to 1, 1, 6$

$S_2 \to 2, 3, 7$

$S_3 \to 3, 4, 8$

$S_4 \to 4, 5, 9$

$I_1 \Rightarrow x$

$I_2 \Rightarrow$

→ {next instn cannot be started just in the next clock cycle} X

~~Forbidden list of~~ R ⇒ {1, 2, 5, 6}

We can start new instn from 3 4.

If we start from 5, then collision would occur.

Forbidden list ⇒ {1, 4, 5}

↳ refers to the clock cycle.

Upon generalizing we get;

(n, n+1, n+4, n+5) ⇒

Hence; (1, 4, 5).

Initiation Latency / Latency

→ If an instn is started at x+4 the next instn is at (x+n); Then the latency is 'n'.

→ To increase efficiency, we need to decrease the latency.

Schedule the pipeline:

If started at (n+2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S_1	I_1	I_1	I_2	I_3		I_1		I_2	I_3	I_3	I_4	I_4		I_3		I_4	I_5
S_2		I_1	I_1	I_2	I_2		I_1		I_2	I_3	I_3	I_4	I_4		I_3		I_5
S_3			I_1	I_2	I_2	I_1		I_2		I_3	I_3	I_4	I_4	I_3		I_5	
S_4				I_1	I_2	I_2	I_1		I_2	I_3	I_3	I_4	I_4	I_3	I_5		
		x			x	x		x	x		x	x					

I_1 I_2 I_3 I_4 I_5

If we start at (n+3) then.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S_1	I_1	I_1		I_2	I_2	I_1	I_3	I_3	I_2	I_4	I_4	I_3	I_5	I_5	I_4		
S_2		I_1	I_1		I_2	I_2	I_1	I_3	I_3	I_2	I_4	I_4	I_5	I_5	I_4		
S_3			I_1	I_1		I_2	I_2	I_1	I_3	I_3	I_2	I_4	I_4	I_5	I_5	I_4	
S_4				I_1	I_1		I_2	I_2	I_1	I_3	I_3	I_2	I_4	I_4	I_5	I_5	I_4
	①	2	3	④	5	6	⑦	8	9	10	11	12	13	14	15	16	17
		x			x	x		x	x		x	x					

↑ optimal scheduling method

steady state efficiency = 1.

Hazards → not achieving

- Structural
- Control
- Data

- (less resources)
- (2 instn trying to access the same resource)

FI → Fetch instn
DI → Decode Instn
CO → Compute Operand add.
FO → Fetch operand
EI → Execute instn
WO → Write operand.

MOV R_1, x ← from memory

	1	2	3	4	5	6	7	8
FI	I_1	I_2	I_3	X				
DI		I_1	I_4	I_3				
CO			I_1	I_2	I_3			
FO				I_1	I_2	I_3		
EI					I_1	I_2	I_3	
WO						I_1	I_2	I_3

in 4th clock cycle, I_1 is in 'FO' phase

We can't start I_4 in 4th clock cycle as its already in 'Fetch operand' from memory phase.

Control Hazards → In (because of) Branch instn;

BR ADD

FI	I_1	X						
DI		I_1	X					
CO			I_1	X				
FO				I_1	X			
EI					I_1	X	ADD	
WO						I_1	X	

unless the whole 'ADD' procedure has been completely executed, we can't start new instn.

Data Hazards → Because of data dependency.

ADD B => A = A+B
SUB B => A = A-B

↓ executed in serial order

	1	2	3	4	5	6	7	8	9
FI	I_1	I_2							
DI		I_1	I_2						
CO			I_1	I_1					
FO				I_1	X		I_2		
EI					I_1	X		I_2	
WO						I_1	X		I_2

Can be wntd. from 7th clock cycle because ~~operand~~ operand is updated ~~from~~ in 6th one.

	f	D	E	W
I_1	1	2	2	1
I_2	2	3	3	2
I_3	3	1	1	1
I_4	1	1	1	1

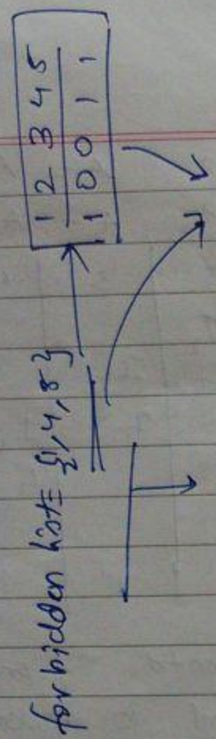
→ refers to the clock cycle no.

I_1 → 1 times fetched.
→ 2 " decoded.
→ 2 times executed.
→ 1 time written.

1 2 3 4 5 6 7

f D E W

Su in mobile gallery



Collision vector

CV0 = 10011

Collision register

CR = 00000
(initially 0)

Whenever a new clock cycle is being used, then we need to shift CR left by one bit
(whenever beginning a new task)

23/9/24

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F	I_1	I_2	I_3	I_3	I_3	I_4														
D		I_1		I_2		I_2														
E			I_1		I_1		I_1													
W																				

Non optimal scheduling (SI)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	2	2	□	1	□	2	3	3	4	4	□	3	□	4	

Optimal scheduling (SI)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	□	2	2	1	3	3	2	4	4	3	5	5	4	6	

Task initiation diagram X

→ - Whenever initializing a new task in a clock cycle, we (OR) CR with CV0 along with a left shift

CR = 00000 CV0 = 10011

← left shift + OR

CR = 10011

— If no new task is initialized then only left shift needs to be performed.

CR = ~~00000~~ 00110

— In next clock cycle, we are starting a new task

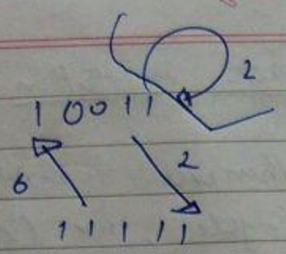
∴ CR ← left shift & OR with ~~CR~~ CV0.

and so on...

10011 ↔ 11111 These 2 states are 6 clock cycles ~~and~~ apart.

	11111	③
<<1	11110	④
<<1	11100	⑤
<<1	11000	⑥
<<1	10000	⑦
<<1	00000	⑧
<<1	00000	
+ OR	10011	
	10011	⑨

```
1<< 00110   (10)
1<< 01100
 OR 10011
    11111   (11)
1<< 11110   (12)
1<< 11100   (13)
1<< 11000   (14)
1<< 10000   (15)
1<< 00000   (16)
1<< 00000
OR  10011   (17)
    10011
```



starting from 4th clock cycle above state diagram is obtained.

For optimal scheduling mechanism:

CVo = 10011
CR = 00000

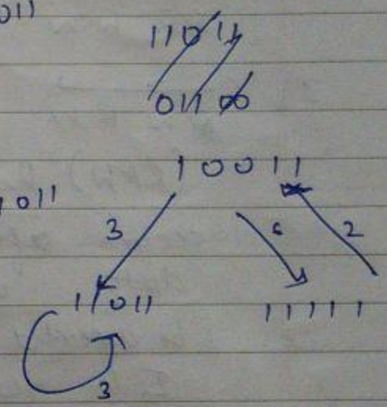
```
<<1   00000
OR    10011
      10011   (1)
$<<1  00110   (2)
<<1   01100   (3)
<<1   11000
OR    10011
      11011   (4)
<<1   10110   (5)
<<1   01100   (6)
<<1   11000
OR    10011
      11011   (7)
<<1   10110   (8)
```

11011
11011
11011

```
<<1   01100   (9)
<<1   11000
OR    10011
      11011   (10)
<<1   10110   (11)
<<1   01100   (12)
<<1   11000
OR    10011
      11011   (13)
<<1   10110   (14)
<<1   01100   (15)
<<1   11000
OR    10011
      11011   (16)
<<1   10110   (17)
```

11011
11011

Thus after 3 clock cycles each, states repeat themselves.



Task initiation diagram.

→ Optimal one is where latency is minimized.

p → probability of encountering branch instn.
q → probability of taking branch instn (actually).
n → instn stream. (for each instn)
m → no. of stages simultaneously being executed

$$\frac{\text{clock cycle}}{\text{instn}} = \text{if taking a branch} + \text{if not taking a branch.}$$

$$CPI = pqmn + (1-pq)n$$

$$\frac{\text{Clock cycles}}{n} = 1 + pq(m+1)$$

Ex: $m = 6 \quad p = 0.2 \quad q = 0.4$

$$CPI = 1 + (0.2)(0.4)(5) = 1 + 0.08 \times 5 = (1.4)$$

→ In execution of branch instⁿ using pipelining, if delaying the jump would increase the efficiency then its called delayed jumps.

RAW (Read After Write)

I_1 RR WW — R, R } no problem created.

I_2 RW RW

(RAW) Read After Write Hazard → If reading occurs after 'write' instⁿ, then incorrect data would be read. This hazard would be averted if I_2 could be stalled when I_1 is reading. [True dependency]

(WAR) Write after Read Hazard → Here I_2 would get incorrect data after I_1 has executed 'write' instruction. [anti dependency] (Why?)

(WAW) Write After Write Hazard [output dependency]

forwarding path → output of intermediate stage would be given to the 'next' stage(s).

Problem Defn.: 34% branch instⁿ, 3 clock cycles per instⁿ are wasted. Speed up factor is 8 and each clock cycle is of 10μs. How many stages are present in the pipeline system?

Ans → Let no. of stages = n

$8 = \frac{n \times 10\mu s}{(1+3\times0.34)\times 10\mu s}$ (time wasted)

$= \left(\frac{n}{1+1.02}\right) \Rightarrow n = 8 \times 2.02$

≈16

0101 (-5) → 1019 417 → 0001 0001 (4) (7) (4)

Non-Restoring Division Algo

0000
-1101
0011

0001
-0011
1110

AC — K bits
MQ — K bits

MQ_0 → LSB of MQ gives the quotient saved after a subⁿ or addⁿ.

Total no. of addⁿ / subⁿ are 'k' & total no. of shifts = k+1

(14th Nov)

4 bits for divisor & 7 bits dividend.

Sign bits S → shows the sign.

---

dividend of 2 words & divisor of 1 word.

① Check sign of divisor.
② check sign of dividend
if (1) then take 2's complement

$\phi_{in} = S_{div} \odot S_{ALU}$

101
①
1⊙0
0

After left shift { MSB dividend ⊕ S_{div} → 0 Sub, 1 Add.

classmate

Date
Page

95

A3

0·4 = (-4)+1 = (-3)

(1+10) % 5 = (1) (R)

95433

43395

(C4)

5 4 3 3 9

12345

10123 (2)

ptr = 2

0 1 2 3 4 5 6 7 8 9

(2)

ptr (3)

1% R5 R4

(R10)

R10 R9

10

9%5 = 4 (4)

10%5 = 0

4%5 = 4

(19)

954339

(5)

0 – 9

(20)

54339

95433

+2345

(8)%5

0 – 20

(19) % 5 = (4)

19

39543

01234

(19 % 5 = 4)

33954

+21

2 + 4 = 6 % 2

(20)

43395

24%5 = (4)

2 + 22

%5 = (2)

(2 – 5) = (-3) + 1

54339

95433 (1)

39543 (2)

33954 (3)

43395 (4)

0 2345

54339

01234

01234

–4 +5

1 2 4 3 5 7 9 6 8 10 12 14 16 19 20 11, 13 ...

classmate

Date
Page

1111 1110
+ 00000001

00000000 00000100 0

00000000 00000010 0

0000 0000 00000001 0

11111101 00000001 0

11111110 10000000 1

0

(-7) (8)
+0111 01000
1001

54339

12345
54339

R1 => A[1] = 5

(1) 95433

C4 ~~33954~~ 43395

R5

A[5] = 5

(2) 39543

(3) 33954

43395 (1)

(4) 43395

33954 (2)

39543 (3)

95433 (4)

54339

54339
43395 (1)
33954 (2)
39543 (3)
95433 (3)
39543 (2)
33954 (2)
43395 (1)

54339

54339

a[100]

```
for(i=0; i<10; i++)
{ for(j=0; j<10; j++)
  {
    if(a[i] > a[j])
      t
  }
}
```

```
for (i=0; i<strlen(str); i++)
{ if( str[i]==' ')
  { for(j=i+1; j<strlen(str); j++)
    temp= str[j]-
    str[j]= str[i]
    str[i]= temp
  }
}
```

C100 → 002F
C101 → 001A
C102 → 0025
C103 → 0020
C104 → 001B
C105 → 0016
C106 → 0011
C107 → 000C
C108 → 0007
C109 → 0002

LAB 2

If the no. is larger than 50, 1 in C100,
less than 50, 2
= 50, 3

Sorting of memory loc C100, C109

Sort Ascending order

C100 — C109

109 100

11 | 9 | 10 | 15 | 7 | 8 | 4 | 12 | 11 | 34

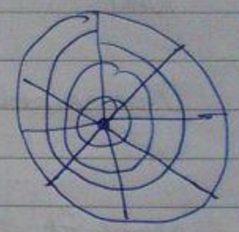
CM → Cache Memory

13/9/14

Ch-6 Memory Organization

SSS

Access Type:
1> Serial Access
2> Random Access
3> Semirandom Access.



→ In serial access, we have only one r/w head per plate

→ In random access, we have r/w head per track.

Random Access + Serial Access = Semi Random Access.

Different types of erasable memory:
- DRO (Destructive read out memory).
- Dynamic
- Volatile

Non-erasable memory
- NDRO
- Static
- Non volatile

In Dynamic RAM is composed of capacitors: If they are Charged; represents 1; else represents 0.

- Refreshing in dynamic RAM is hence time consuming.

- Static RAM consists of ~~cache memory~~ flip-flops. eg; Cache memory.

- Static RAM is costlier and less denser than dynamic RAM.

Non-volatile → contents of RAM don't get erased if power failure occurs.

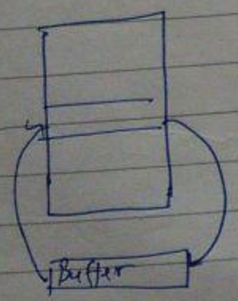
volatile → contents of RAM would get erased if power failure occurs.

Flash memory: EEPROM or E²PROM

Access time: time interval between the request & the 'request' getting served is called access time.

Cycle time: Time elapsed between 2 consecutive memory read operations.

In DRO,



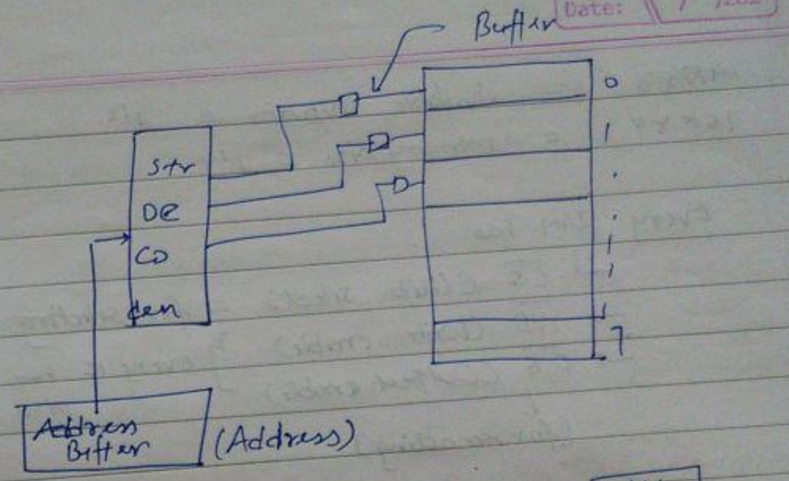
2nd request gets served after restoring the 1st request. Hence, restoring time is included in cycle time.

$t_{Access} < t_{cycle}$ ( )

$t_{Access} = t_{cycle}$ (?)

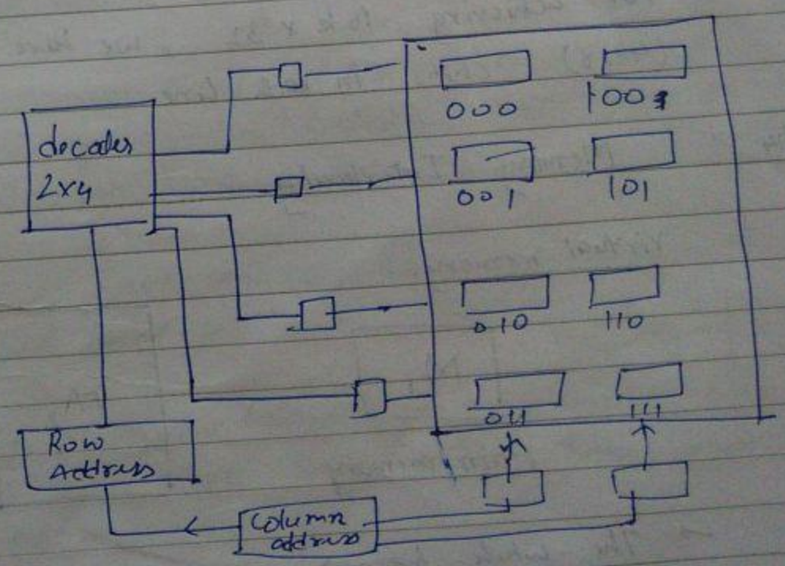
data transfer rate = no. of b/s transferred per unit of time.

$$b_M = \left(\frac{N}{T}\right)$$



(8 data buffers)

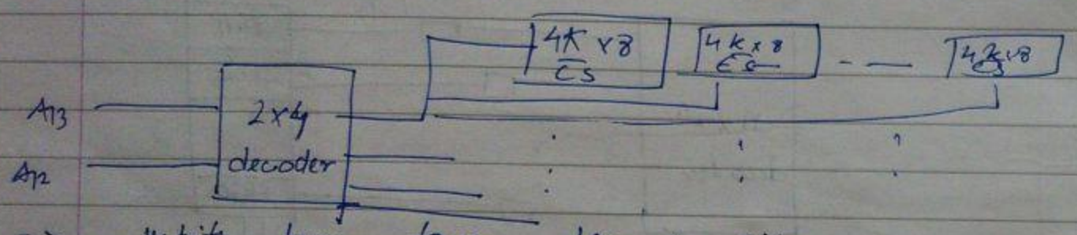
6 data buffers



Page:
Date: / /201

* 4K x 8 = Availble chip size = 2^{12}
16K x 8 = memory size = 2^{14}

Every RAM has
→ $\overline{CS}$ (Chip select) = for selecting IC
→ $\overline{WE}$ (Write enable) } every IC has these 2 pins
→ $\overline{OE}$ (Output enable)
↓
(for reading)



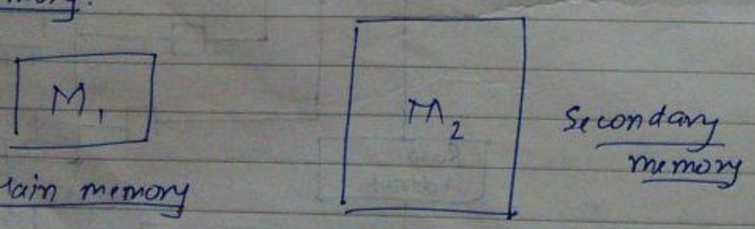
=> 14 bits have from bits to bit13.

=> A_{13} & A_{12} can be used for selecting $\overline{CS}$. (R)

=> For achieving 16K x 32, we have to attach 4 (4K x 8) chips in each line.

5/9/14 Memory Interleaving:

Virtual memory:



→ The whole program may not be stored in the main memory, only part of the program is stored.

Page:
Date: / /201

→ To user, it seems as if the whole program is stored in the main memory. (called virtual memory).

→ Functions of memory management unit:
1) transfer blocks of data from S.M. to M.M.
2) deallocate & reallocate blocks.
3) free the blocks allocated previously.

* Any program stores all the instructions at continuous memory locations so, processor knows that the next instn would probably be at the next memory location. This is called 'Locality of Reference'.

Same is for C.M. processor. It would transfer whole block instead of one byte from C.M. to M.M.

Performance of memory system:

1) Size of block
2) Size of memory itself (size of C.M. or M.M)
3) Replacement of Algorithm (decides which block should be removed if enough space is not available)

Spatial locality: next consecutive locations are referenced by the processor and placed into main memory.

Temporal locality: Segment in which loop is present, processor has to process the whole loop repeatedly hence, whole loop is placed inside the main memory.

Average costs of memory : $C = \left(\frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}\right)$

C_1 = costs of main memory.
S_1 = storage capacity of main memory.
C_2 = cost of secondary memory.
S_2 = storage capacity of secondary memory.

If (cache memory) is also present;

$$\left(\text{Cache memory}\right) C = \left(\frac{C_1 S_1 + C_2 S_2 + C_3 S_3}{S_1 + S_2 + S_3}\right)$$

If particular word refirenced by processor is present in main memory; then it is called 'Hit' and if not then 'Miss'.

'Hit' refers to the probability of particular word refirenced by processor, present in Main memory.

$$H = \left(\frac{N_1}{N_1 + N_2}\right)$$

N_1 = no. of address refirences made to M_1 memory by the prausor.

N_2 = " " " " " " M_2 " ..

Miss = $1 - H$.

Average Access Time => $t_A = H \cdot t_{A_1} + (1-H)\, t_{A_2}$.

t_{A_1} = Access time for main memory.
$t_{A_2} = t_B + t_{A_1}$ = Access time of S.M. → (Secondary memory)
t_B = block tranfer time from SM to M.M.

$\therefore t_A = H t_{A_1} + (1-H)(t_B + t_{A_1})$
$= t_B - H t_B + t_{A_1}$ => $\boxed{t_A = t_{A_1} + (1-H) t_B}$

(e) Access efficiency $= \left(\frac{t_{A_1}}{t_A}\right)$

(r) Access ration $= \left(\frac{t_{A_2}}{t_{A_1}}\right)$

$\left(\frac{t_A}{t_{A_1}}\right) = \left(\frac{H t_{A_1} + (1-H) t_{A_2}}{t_{A_1}}\right)$

$\therefore \left(\frac{1}{e}\right) = H + (1-H)\, r$

$\therefore e = \left(\frac{1}{H + (1-H) r}\right)$

$$\boxed{e = \left(\frac{1}{(1-r) H + r}\right)}$$

Space Utilization :→

$u = \left(\frac{su}{s}\right)$ (Space Utilization)

su = space frequently used by processor.
s = total space.

1> If the block transferred from S.M. to M.M. is not used then it will be unutilized.

2> If the transferred block is smaller than the space present in M.M, so the remaining space is wasted. [called 'empty region'].

→ To solve the problem of 'empty region', we use the technique of 'paging'.

— Pages are blocks of equal size.

– 'Page Frame' is a structure where we are fitting our page.

19/9/14 Address Translations :

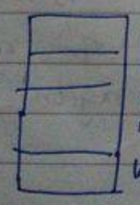
- Addresses should be mapped. Virtual address to physical address.

- can be done by programmer.
- can be done by compiler.
- can be done by Loader.
  ↓
  object code that decides when to load.

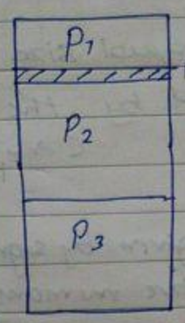
- can be done by Memory Management Unit (MMU). at runtime.

Dynamic Memory Allocation → Stack, Queue.
Static " " → ?

Actual address = Base Add. + Displacement (offset)



= Base address : offset. D



Limit address,
While executing a procedure P_1, the processor may require more memory, hence it may overlap the memory allocated to program P_2. For preventing this, a 'limit address' is used to avoid overlapping.

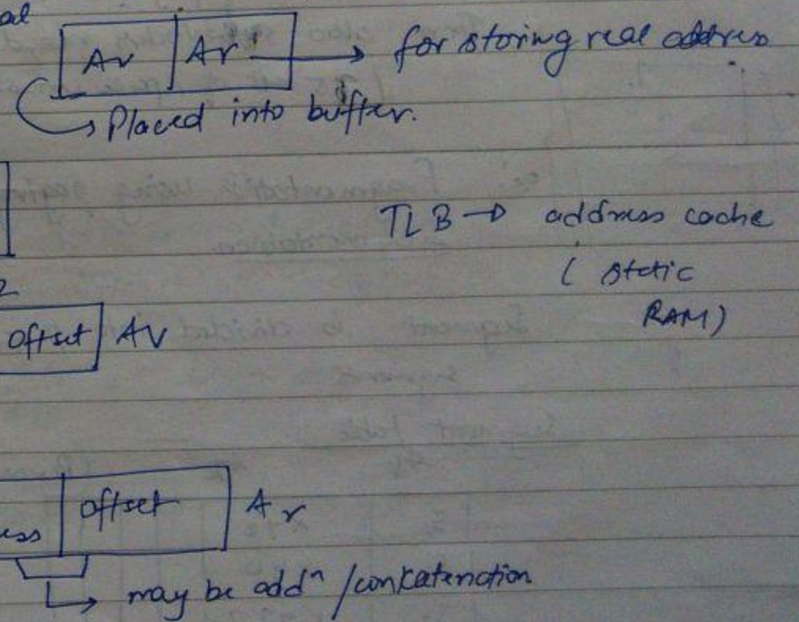
Advantage of using Base Address : 1) size of memory reduced

BA = 00010H
D = 00003H

- relocation of blocks is performed for 'defragmentation'.

Translation Look Aside Buffer :

- special buffer used for mapping virtual address to base address of real address.



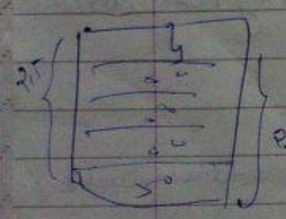
Segments and Pages →

1) Pages are the blocks of equal size.
Segments' boundary is decided by the program. (variable length).

2) Permission (read/write) can be given by segments.
While pages are not at consecutive memory loc^n.
It's difficult but possible.

3) We may remove one segment, if another segment is req^d, its size must be ≤ removed seg.
This doesn't create a problem in case of pages as they are of equal sizes.

4) Internal fragmentation is present in segments.
215 KB data
50 KB of page size.
Then also 5 blocks req^d.
(35 KB of space unused.)

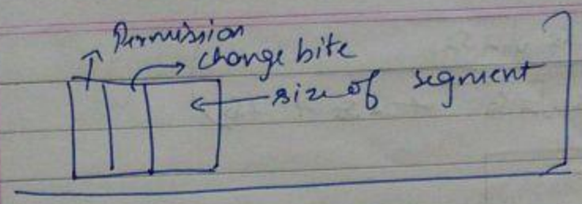


e.g. Fragmentation using paging is called external fragmentation.

Segment is divided into pages, called paged segments.

Segment table:

A_V	A_R (Permissions)				
S_1	x72				
S_2	xyu				
S_3	x9ZZ				



If change bit is 1 then write operation is done from MM to SM.

→ Smaller the size of page, smaller the wastage.

→ However, the no. of pages would be increased.

Optimum Page size:

$$S = \frac{Sp}{2} + \frac{Ss}{Sp} \quad —①$$

Sp = no. of entries.
Ss = segment's size.
$\frac{Ss}{Sp}$ = no. of entries per segment.

$\frac{Sp}{2}$ = space which is not utilized.

S → space overhead.

$$u = \left(\frac{Ss}{Ss+S}\right) = \left(\frac{Ss}{Ss + \left[\frac{Sp}{2} + \frac{Ss}{Sp}\right]}\right)$$

$$= \left(\frac{2Sp \cdot Ss}{2Sp \cdot Ss + Sp^2 + 2Ss}\right)$$

To maximize, differentiate ① equate to 0.

$$\Rightarrow \frac{dSs}{dSp} = \frac{1}{2} - \frac{Ss}{Sp^2} = 0$$

$$\Rightarrow \boxed{2Ss = Sp^2} \Rightarrow \boxed{Sp = \sqrt{2Ss}}$$

$$\Rightarrow U_{opt} = \left(\frac{2Ss\sqrt{2Ss}}{2Ss\sqrt{2Ss} + 2Ss + 2Ss}\right)$$

$$\boxed{U_{opt} = \frac{\sqrt{2Ss}}{\sqrt{2Ss} + 2}}$$

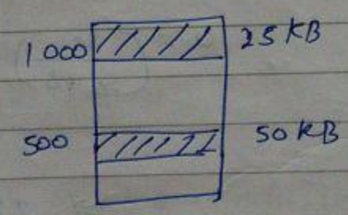
$$\Rightarrow U_{opt} = \boxed{\frac{1}{1 + \frac{\sqrt{2}}{\sqrt{Ss}}}}$$

Functions of Memory Management Unit:

1) Allocate & deallocate the block.

* Available list: Contains the - address
  - free space available (at that address)

1000	25
500	50



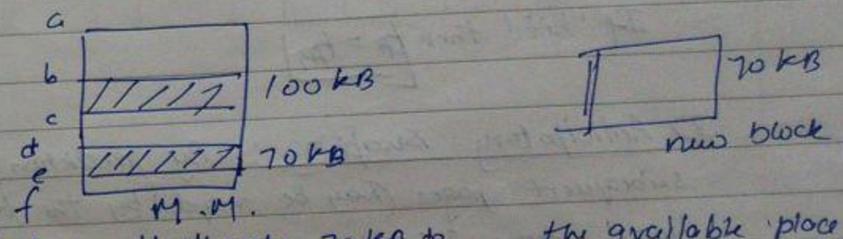
* Occupied list: Contains the address at which the space is occupied (includes size).

* Directory list: Directory means programs and segments related to it. It also contains the address of blocks if not present in M.M.

Preemptive Memory Allocation: Existing blocks are removed and replaced with new ones.

Non-preemptive memory allocation: Existing blocks are not disturbed but ~~fitt~~ fit into free space.

1) First fit: In M.M, M.M.U will scan the available list and the free space.



- M.M.U would allocate 70 KB to the available place, whichever it finds first
- fragmentation is much more.
- 'processor time' is less.

2) Best fit: 70 KB is placed in the 75 KB block. only 5 KB wastage. Hence, less fragmentation and higher processing time.

Preemptive Memory Allocation:

1) Compaction Allocation method →

- Here smaller blocks are combined to make it into a bigger block.

2> Demand Swapping: Processor demands for certain blocks. If not present, then these blocks are brought in from SM to MM. 'Miss' occurs.

$t_A = t_B + t_{A_1}$;

t_A = total access time.
t_B = block transfer time from SM to MM.
t_{A_1} = Access time from main memory.

If 'hit' then $\boxed{t_A = t_{A_1}}$

3> Anticipatory Swapping: By prediction, some blocks/ subsequent pages (may be req.) by the processor. These are placed from SM to MM. As size of MM is limited; the unutilized blocks are removed from MM.

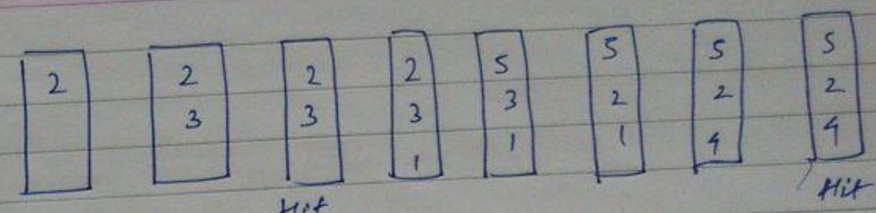
4> Replacement policies:

FIFO (First In First Out): The oldest block is removed.

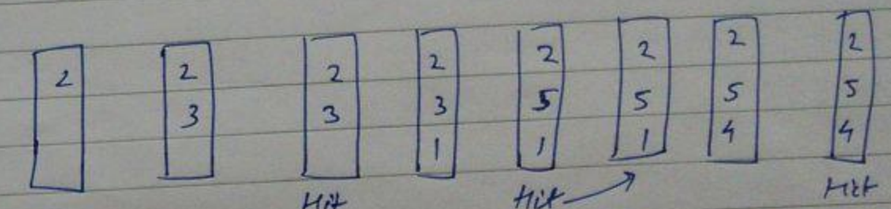
LRU: (Least Recently Used): Some of the other blocks may be useful to the processor. ~~to~~ So the least used blocks are removed first.

Time:	1	2	3	4	5	6	7	8
Address space:	2	3	2	1	5	2	4	5

FIFO



LRU



LRU → 3 hits ; FIFO → 2 hits.

Hits ratio = $\frac{3}{8}$; = $\frac{2}{8}$ = (0.25)

= 0.375

→ Implementation of LRU is more complex than FIFO.

Page:
Date: / /201

Page:
Date: 20/9 /2019

BMG

Chapter 1

- I/O devices
- Parallel processing
- System Bus

- Flynn's Classification of Computer Architecture.
  ↳ uses stream concept for describing machine structure
  - SISD, SIMD, MISD, MIMD [important]
    ↓ (common)
    ↓
    classical von Neumann architecture.

MIMD → Multiple instn streams, Multiple data streams.

Communication Methods →
- IntraSystem Communication. [within processor]
- InterSystem communication. (internet)
  - FM
  - Carriers.

Computer Networks
- Store & forward (entire msg sent at a time if resources are available)
- Packet Switching.

- LAN
- WAN

OSI (Open System Interconnection)
↓
guidelines for connection & tranfer data

CSMA → (Carrier Sense Multiple Access)
CD → (Collision Detection)
(in)
→ (Ethernet)

Page:
Date: / /201

Ethernet → is a standard over which various computers can be interconnected. (follows 'Bus' topology)

- Topologies → 1> Ring 2> Mesh 3> Star
4> Fully connected 5> Line 6> Tree 7> ~~Star~~ Bus

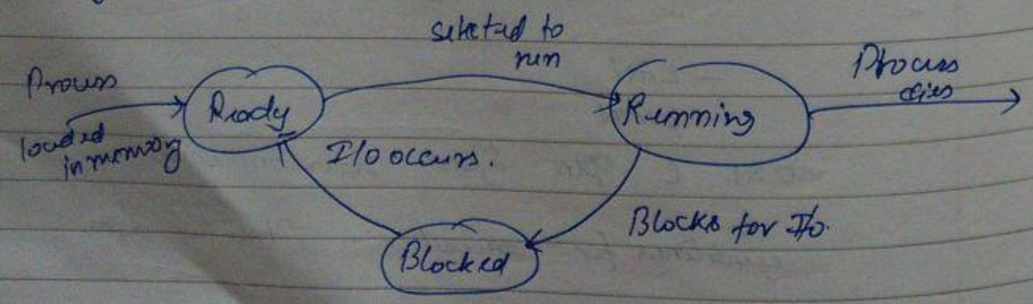
If multiple computers connected in 'Bus' topology are sending data simultaneously then collision occurs.

Internet → www.... [has TCP/IP]
internet → no TCP/IP ; (not www...)

- IO and System Control
1> IO-mapped IO
2> Memory-mapped IO

DMA (Direct Memory Access)
- increase speed of IO transfers by reducing the role of CPU.

Operating System.



FIFO
LIFO

Page:
Date: / /201

Fault Tolerance
- Hardware redundancy (no articulation point)
- Software redundancy
- Information redundancy → make multiple copies of the same data. (backups of the server maintained)
- Time redundancy
- Reliability.
  - H/W : minimum no. of times the H/W is failing.
  - S/W / network : whatever data is being sent should be recieved in the exact same state.

Data/
- Path forwarding → whenever data is available in the intermediary stage, the next instn should run only after all the previous instruction have run (refer to pipelining).
  1> Read A, B
  2> ADD A, B
  3> Read Accumulator
) step3 should be run only after step2 has been executed.

- Register renaming → Renaming of registers is done in such a way that the efficiency is improved by eliminating the redundancy.
  Ex: I 1> B := B-C
  2> C := 50
  3> D := D-C
  > data dependency between ① & ② (which hazard)

II 1> B:= B-C
2> A:= A+B

- Here path forwarding would be used.
- Content can be fetched from Accumulator (where B-C would be calculated)
- Thus renaming of C to 'A or B' would be helpful.

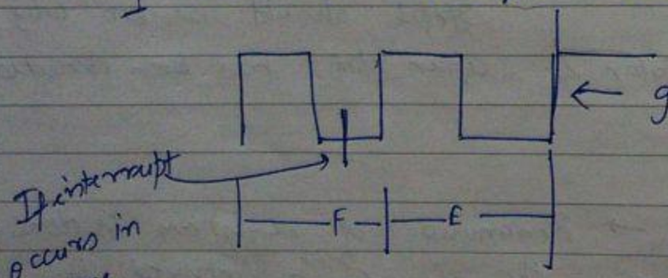
III
B:=B-C
C:=S/Sa
D:=D-C
A:= 1501
A:= A-C

Q Can the number of registers used be minimized?
→ In here if the results are not to be lost then the number of registers can't be reduced.

30/1/14

Interrupts → Temporarily suspend the mainline program to serve another subroutine. comes back to the main program after completing the subroutine.

Q When do interrupts occur?



Q When do we switch over to the Interrupt subroutine?
≡
→ After every instruction cycle, we check the presence of interrupts.

* In DMA, we check for interrupts after every machine cycle.

| M_1 | M_2 | M_3 |

* In pipelining, if we have interrupts, then; no instrⁿ would be over until the program is over.

	1	2	3	4	5	6
F	I_1	I_2				
D		I_1	I_2	I_2		
E			I_1	I_1	I_2	
W					I_1	I_2

⇒ Whenever we check for interrupts;
- we store all the registers in stack.
- we store PC
- then we switch to ISR.

If we have nested ISR's then again the whole procedure is followed for the 2nd interrupt.

If there is some problem in the flow of program to ISR and vice versa, then they are called imprecise Interrupt.

If the flow is proper then its called precise interrupt.

Eg: Consider Direct Mapped Cache of size 32 kbyte with block size of 32 bytes. The CPU generates 32 bit address. Find out no. of bits needed for cache indexing and no. of bits for (tag bits).

=> Address → 32 bits => (2^5) 5 bits for address.

$\frac{32\ \text{Kbytes}}{32\ \text{bytes}}$ = 1 K blocks

$= 2^{10}$ Blocks

=> 10 bits reqd.

32
−10
− 5
17 → bits reqd. for indexing.

1 X: CMP R_1, 0
2 JZ // jump on zero to 'Z'
3 MOV R_2, R_1
SHR R_1 // right shift R_1
SLR R_1 // Left shift R_1
CMP R_2, R_1
JZ Y // jump on zero to Y
INC R_3
Y: SHR R_1 // right shift R_1

JMP X
Z:

Initially
$R_1 \to 5$
$R_2 \to 0$
$R_3 \to 0$

Find the value of R_1 & R_3 whenever the control of program comes to Z for the first time.

=>

$R_1 \leftarrow 5$	$R_3 \leftarrow 1$
$R_2 \leftarrow 5$	$R_2 \leftarrow 2$
$R_1 \leftarrow 2$	JMPX
$R_3 \leftarrow 4$	$R_2 \leftarrow 2$
	$R_1 \leftarrow 1$

~~101~~ ~~110~~ 10

$R_1 \leftarrow 2$
Cmp R_1, $R_1 \leftarrow 0$

$R_2 \leftarrow 2$	$R_1 = R_2$	$R_1 \leftarrow 1$
$R_1 \leftarrow 1$	(JZY) JMPX	$R_2 \leftarrow 1$
$R_1 \leftarrow 2$		$R_1 \leftarrow 0$
Infinite loop		$R_1 \leftarrow 0$
		$R_3 \leftarrow 2$
		$R_1 \leftarrow 0$

* Answer ($R_1 \leftarrow 0$ & $R_3 \leftarrow 2$) when loop reaches label Z.

Generalization → (for values of R_1 & R_3)