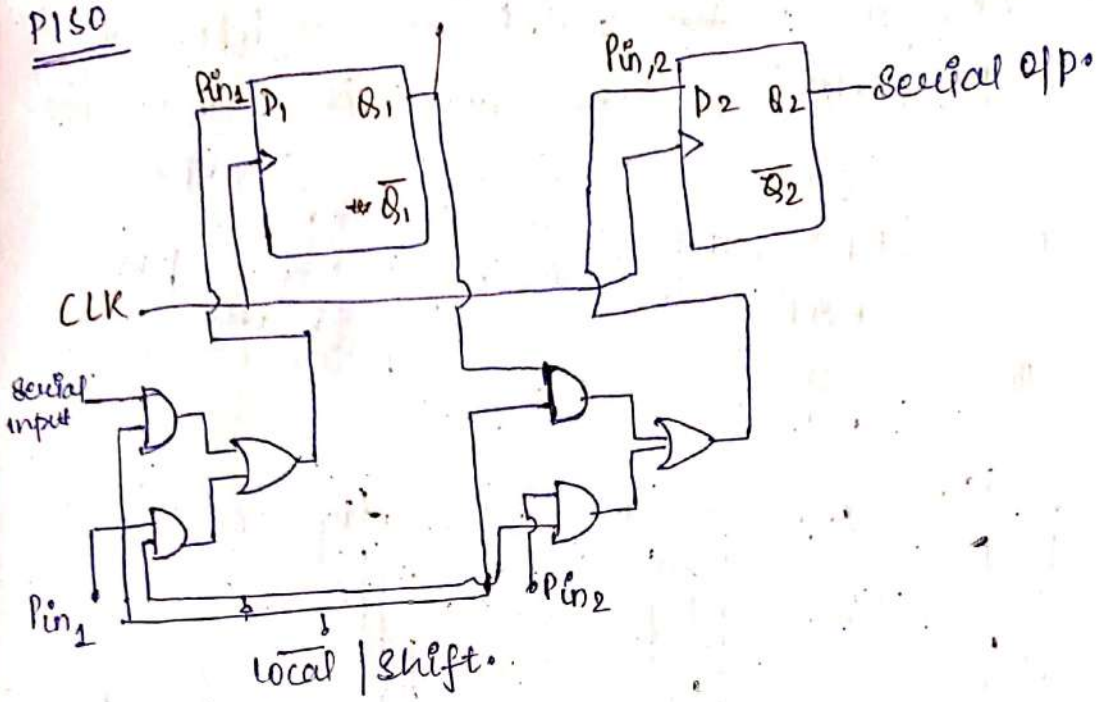
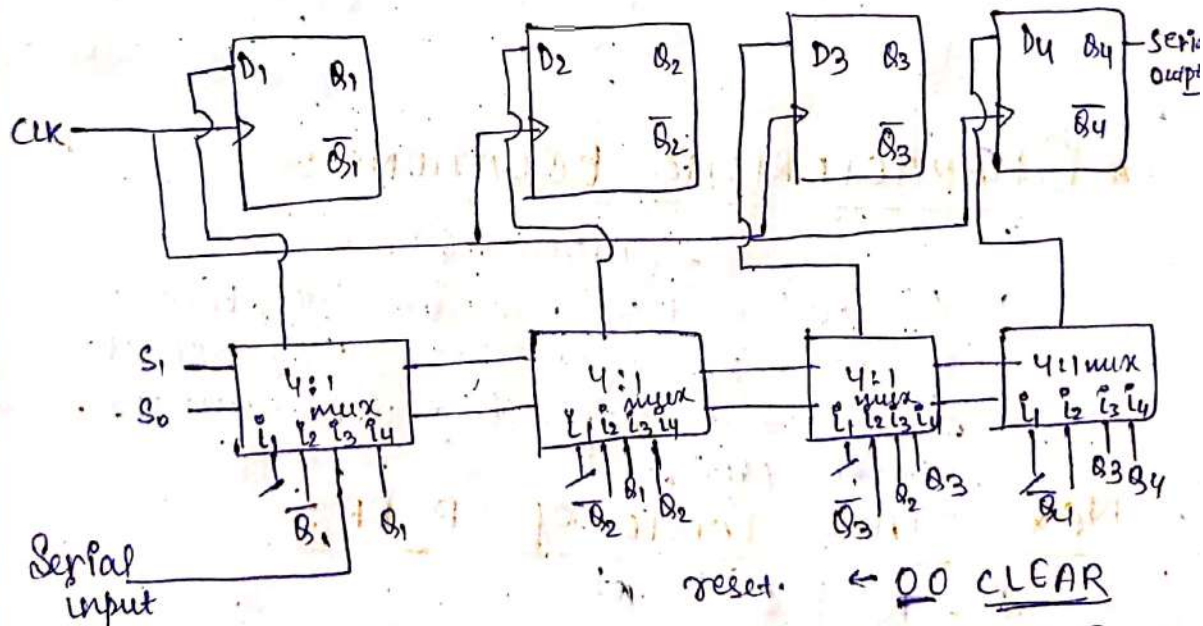
PISO



* UNIVERSAL REGISTER :- (Universal Shift Register)

↳ It is a register which such a design that multiple operations can be performed.



∴ reset. ← 00 CLEAR

change (opp.) ← 01 TOGGLE

10 SHIFT

11 HOLD.

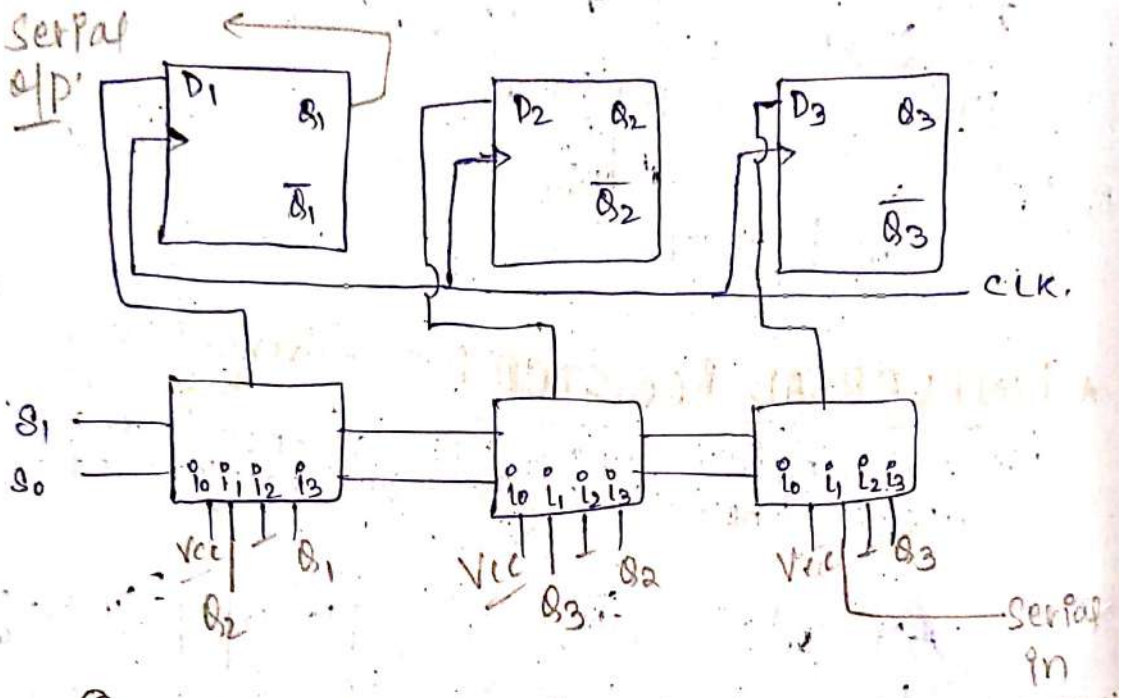
Right-shift

Ques Design a ~~ALU~~ universal register to perform 3-Bit operations as below:

00 - SET
01 - LEFT SHIFT
10 - CLEAR
11 - HOLD

Imp

// Show serial i/p & o/p lines correctly.
// Circular shift, o/p will never go out.
i/p will be circular.



★ CHARACTERISTIC EQUATIONS :-

$$Q^+ = f(\text{Inputs}; Q)$$

↳ Eqn which the describes the behaviour of each FF such that they describe next state depending upon current state (i/p).

NEXT STATE TABLE of SR FF :

S	R	Q	Q^+	
0	0	0	0	} retains
0	0	1	1	
0	1	0	0	} reset
0	1	1	0	
1	0	0	1	} set
1	0	1	1	
1	1	0	X	} don't care.
1	1	1	X	

K-map

S \ RQ	00	01	11	10
0	0	1	0	0
1	1	1	X	X

$Q^+ = S + \bar{R}Q$ → Characteristic eqn of S-R Flip Flop.

★ J-K Flip Flop:

Next state Table

J	K	Q	Q^+	
0	0	0	0	retain
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	1	toggle Effect
1	1	1	0	

J \ KQ	00	01	11	10
0	0	1	0	0
1	1	1	0	1

$\therefore Q^+ = \bar{R}Q + S\bar{Q}$

$Q^+ = \bar{K}Q + J\bar{Q}$

★ D-Flip Flop:

Next State Table

D	Q	Q^+	
0	0	0	reset
0	1	0	
1	0	1	set
1	1	1	

D \ Q	0	1
0	0	0
1	1	1

$Q^+ = D$

★ T-Flip Flop:

Next State Table

T	Q	Q^+	
0	0	0	retains
0	1	1	
1	0	1	toggles
1	1	0	

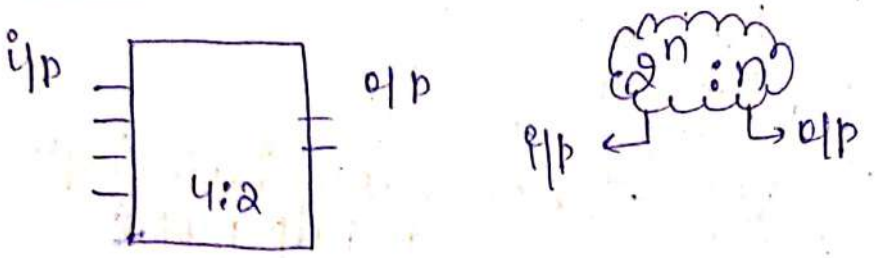
K-map

T \ Q	0	1
0	0	1
1	1	0

$Q^+ = \bar{T}Q + T\bar{Q}$

$= T \oplus Q$

★ Encoders :



i_0	i_1	i_2	i_3	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

$y_1 = i_2 + i_3$

$y_0 = i_1 + i_3$

If 1 0 1 0 → 1 0 → ? limitation

$\therefore y_1 = i_2 + i_3 = 1 + 0 = 1$

$y_0 = i_1 + i_3 = 0 + 0 = 0$

★ Priority Encoder gives ~~the~~ priority to the highest bit.

i_0	i_1	i_2	i_3	y_1	y_0
1	0	0	0	0	0
X	1	0	0	0	1
X	X	1	0	1	0
X	X	X	1	1	1

It will give priority only to 1 ← value

1 0 1 0 → 1 0 ?

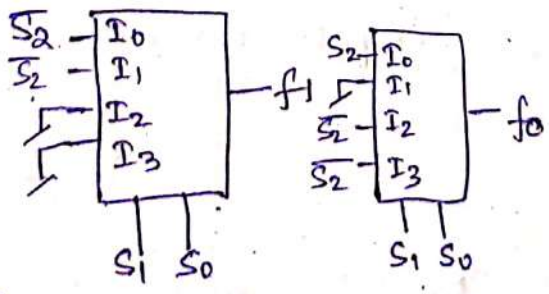
↳ gives priority to this highest bit.

LP ISA-1 (Sample)

(6). Assumption let shop No. be 1 – 8.

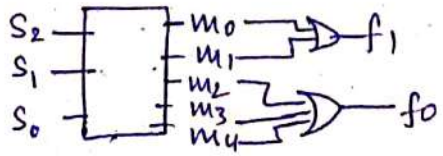
S_3	S_2	S_1	S_0	f_1	f_0
0	0	0	0	X	X
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	X	X
1	0	1	0	X	X
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X

If we solve using MUX:



S_2	S_1	S_0	f_1	f_0
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

Using DECODER:

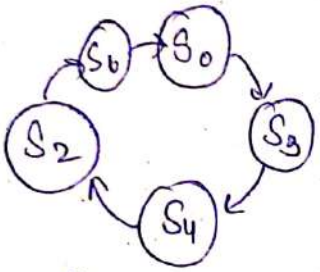


COUNTERS:

→ • No ext. ip or switch.
• It is a self-driven circuit.
• Once the clock pulse is given it will start generating sequence or pattern.

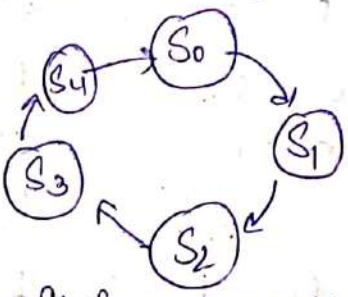
also called as → Sequence Generator
→ Pattern Generator

S_0	0	0	0
S_1	0	0	1
S_2	0	1	0
S_3	0	1	1



Random Counter.

State Diagram



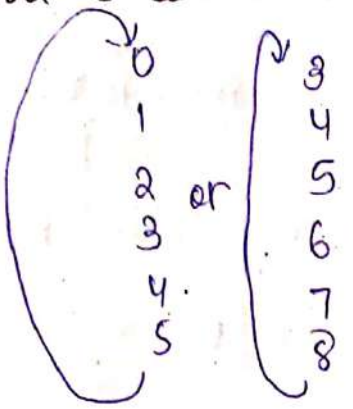
• It is a counter with 4 states.
• It changes its state in every clock pulse.
• 5 stable states.
• Since, the transistor is counting up.
∴ It is mod-5 up counter.

① Synchronous Counter:
• Design is feasible. (easier)
• One single master clock that is driving all the flip flops.
• Output is predictable.

② Asynchronous Counter:
• Each flip flop has different clock.
• Design is complex.

Mod-N counter: Counter with N stable states.

E.g. Mod-6 Counter (Up Counter)



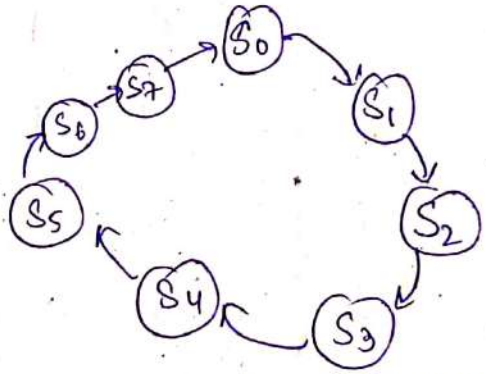
It should have 06 stable states.

Ques

Design a 3-bit Up Counter:

In a 3-bit Up counter, there can be 08 different states.

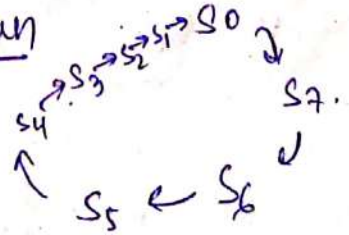
S_0	0	0	0
S_1	0	0	1
S_2	0	1	0
S_3	0	1	1
S_4	1	0	0
S_5	1	0	1
S_6	1	1	0
S_7	1	1	1



State diagram

- For designing, ∵ we are designing synchronous counter. ∴ There will be a master clock.
- Let we are designing it with JK FF.
- Counter does not have external i/p or switch. It is driven itself, & in such a way that the present state generates the next state.

For down

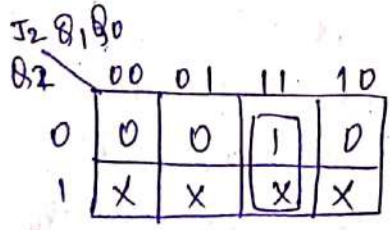


P.T.O

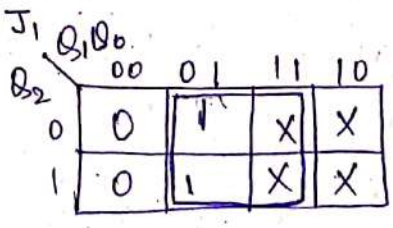
Excitation Table :

Present State			Next State			Excitation I/P					
Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	0	0	0	X	1	X	1	X	1

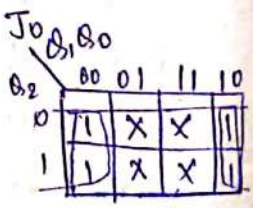
Now, for $J_i = f(Q_2, Q_1, Q_0)$
& $K_i = f(Q_2, Q_1, Q_0)$



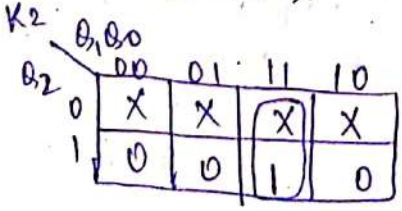
$J_2 = Q_1 Q_0$



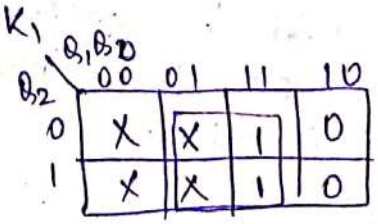
$J_1 = Q_0$



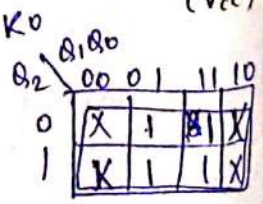
$J_0 = 1$ (V_{cc})



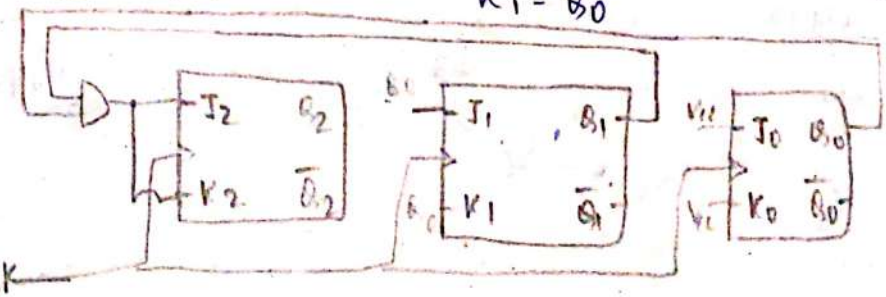
$K_2 = Q_1 Q_0$



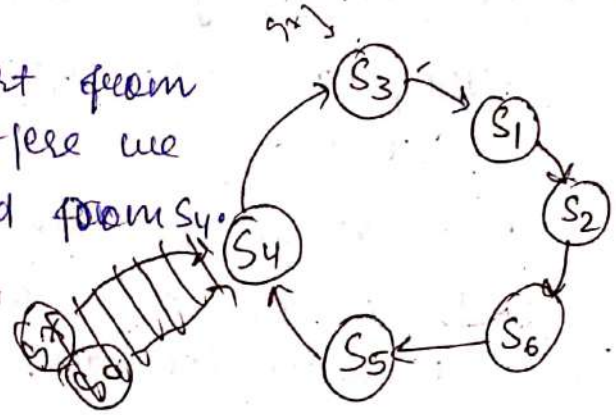
$K_1 = Q_0$



$K_0 = 1$ (V_{cc})



Ques Draw Excitation table for

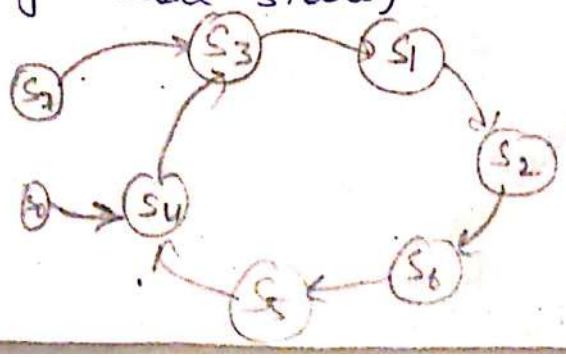


We can start from any state. Here we have started from S_4.

Present State			Next State			EXCITATION INPUTS					
Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	J_2	K_2	J_1	K_1	J_0	K_0
1	0	0	0	1	1	X	1	1	X	1	X
0	1	1	0	0	1	0	X	X	1	X	0
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	1	0	1	X	X	0	0	X
1	1	0	1	0	1	X	0	X	1	1	X
1	0	1	1	0	0	X	0	0	X	X	1
undefined state 1	1	1	0	1	1	X	1	X	0	X	0
undefined state 0	0	0	1	0	0	1	X	0	X	0	X

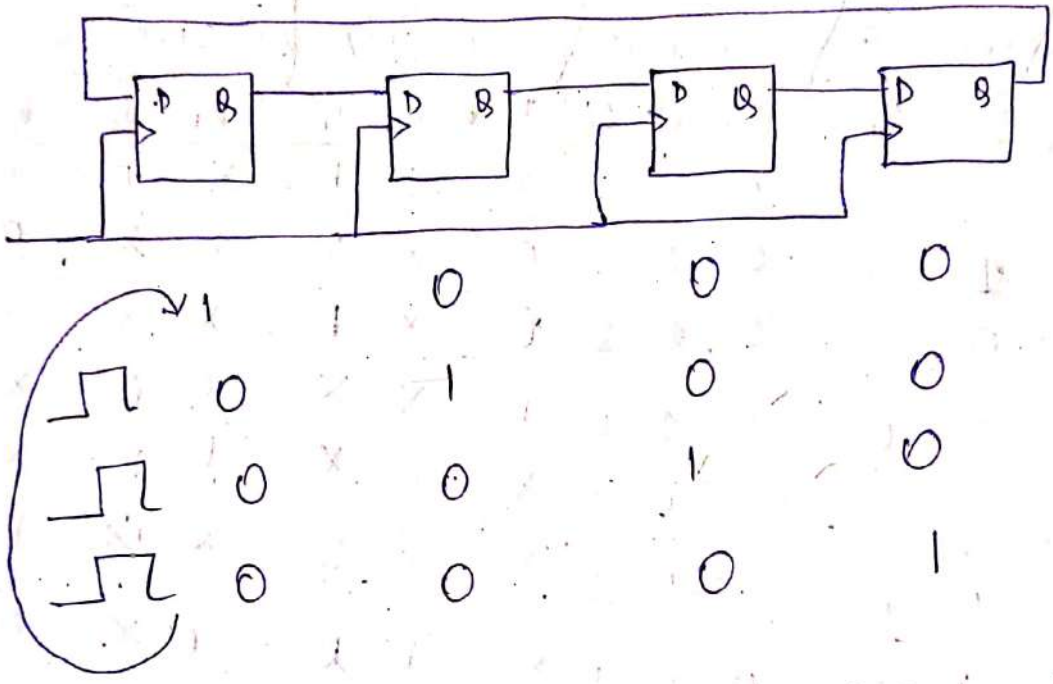
If by chance, counter enters into undefined state i.e. S_0 and S_7 in this case; then we can modify the design of the counter in such a way that it becomes self-correcting counter i.e. whenever it reach to S_0 & S_7 it will move to next state (any stable state).

∴

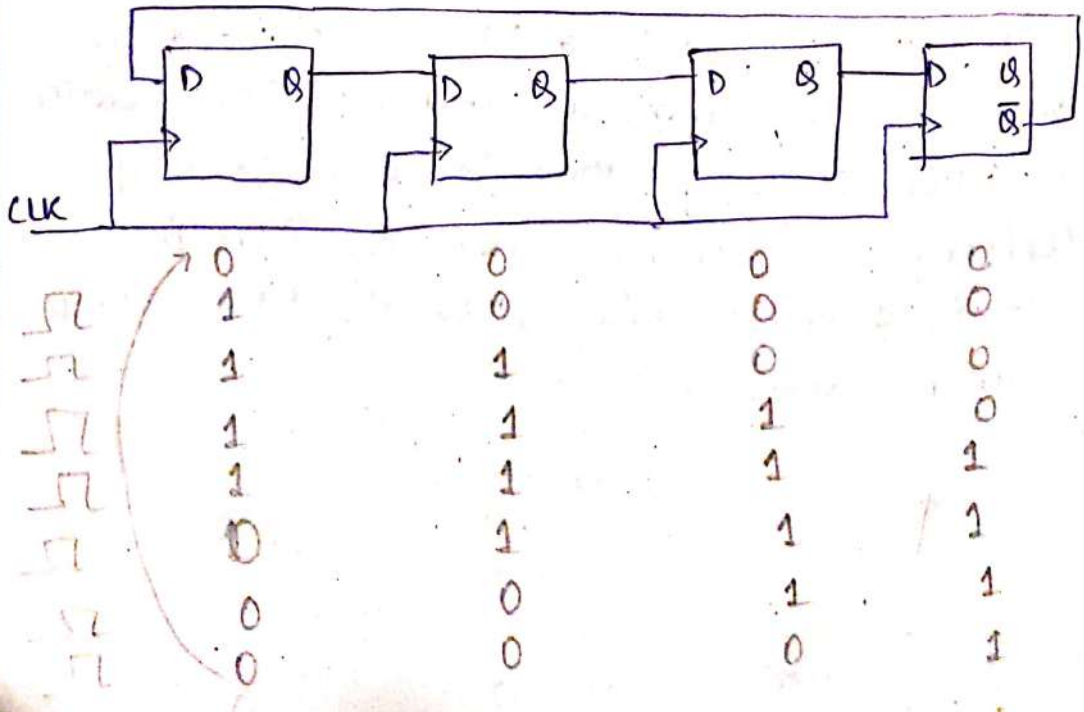


✱ Mod-4 Ring Counter :-

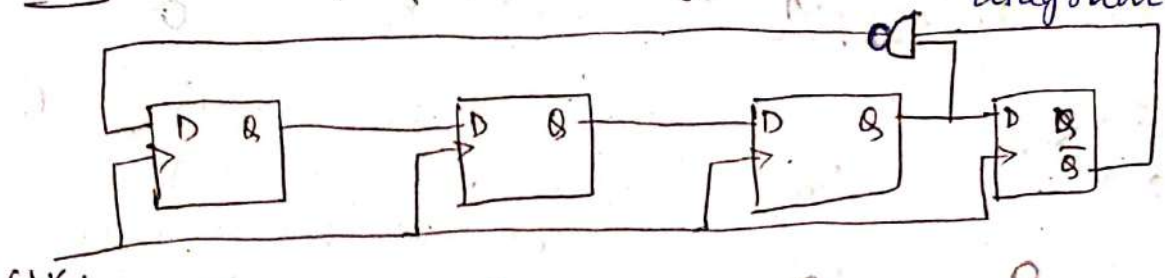
- They are other form synchronous counter.
- It's design is similar to shift register
- There are 04 states & there is a ring operation (counter).



✱ TWISTED RING COUNTER :
(Switch Tail or Johnson)

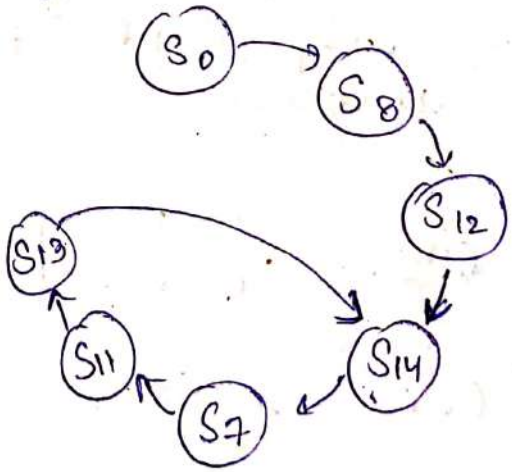


Ques Write the state for this: & draw state diagram.



0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

∴ State diagram:-

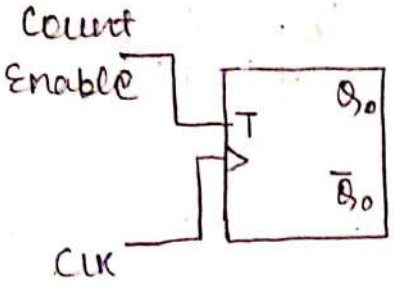


There should not be bidirectional transitions.

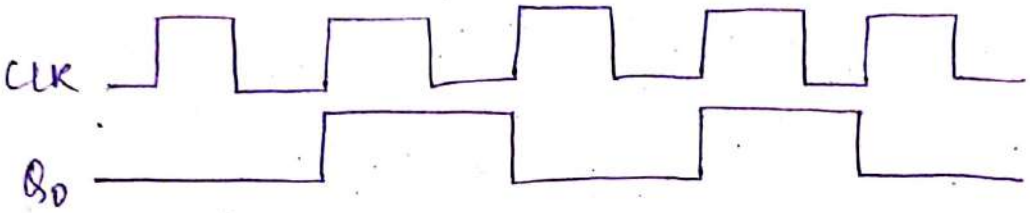
Rules:

- Transition should not be bidirectional.
- Each state has only one transition.
  But a transition can have more than one input.

* Asynchronous Counter:



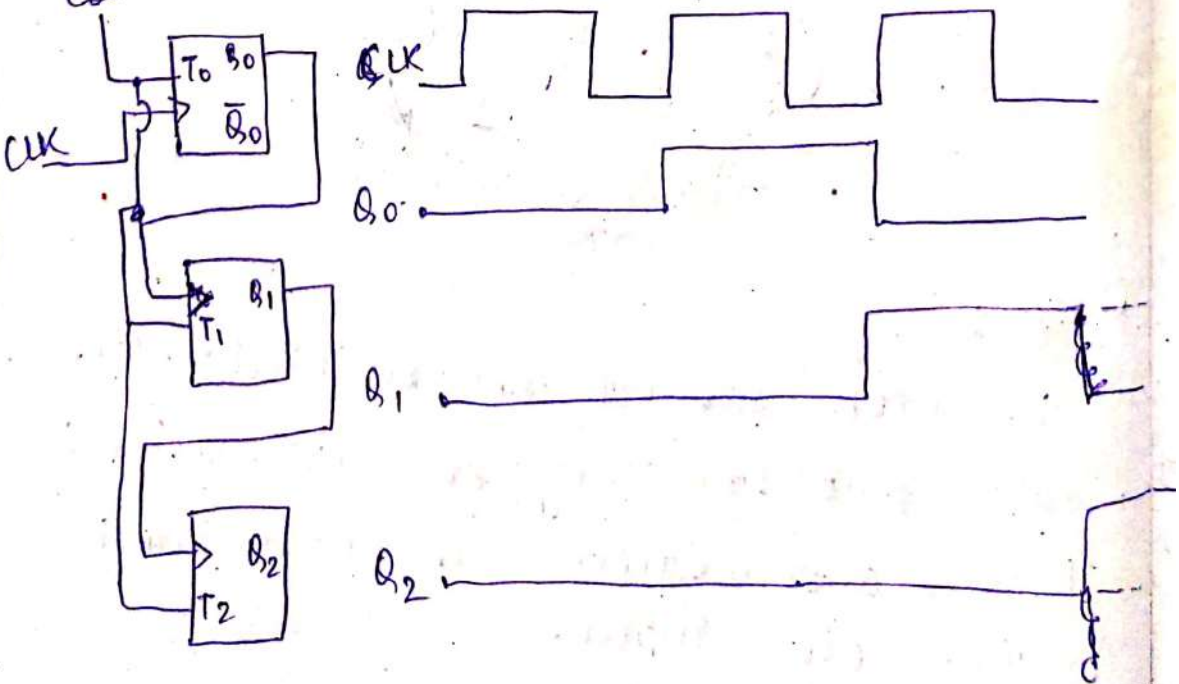
- When count enable is 0, the state of T flip flop will be retained.
- When count enable is 1, it will toggle with every clock pulse.



In Asynchronous Counter, the state of the first FF will be the clock pulse for second FF & so on.

Ques Design a mod 4 Asynchronous counter using T FF that counts even numbers.

Sol^n: **Later**



Soln: Synchronous Counter that counts even no. (mod-4).

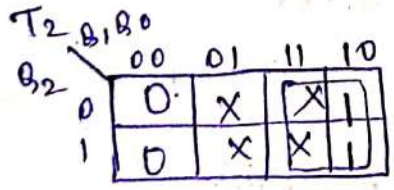
State diagram:

∵ 3 bits are used to represent the highest no. 6.

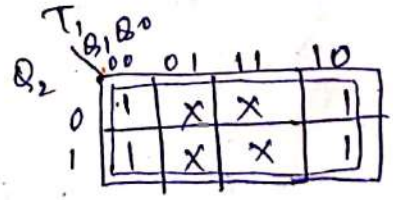
∴ 3 Flip Flops (T-FF) are used

Excitation Table

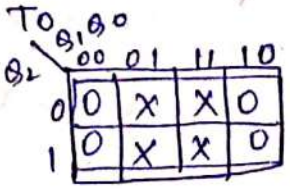
Present			Next					
Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	T_2	T_1	T_0
0	0	0	0	1	0	0	1	0
0	1	0	1	0	0	1	1	0
1	0	0	1	1	0	0	1	0
~~1~~	~~1~~	0	0	0	0	~~1~~	~~1~~	0
Self-correcting: 0	0	1	x	x	x	x	x	x
0	1	1	x	x	x	x	x	x
1	0	1	x	x	x	x	x	x
1	1	1	x	x	x	x	x	x



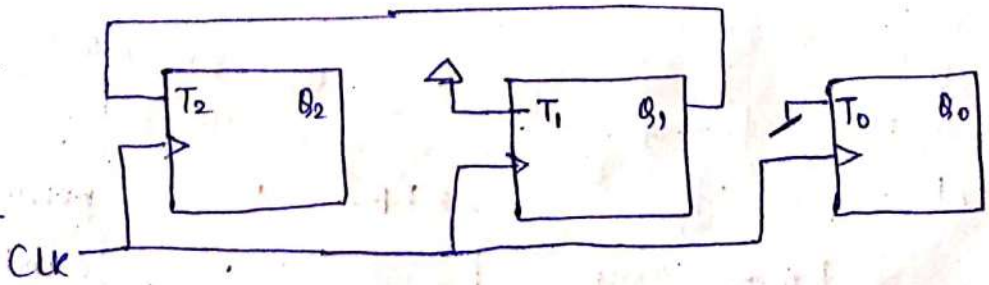
$T_2 = Q_1$



$T_1 = 1$ (V_{cc})



$T_0 = 0$ (Gnd)



09/10/19 Unit 02

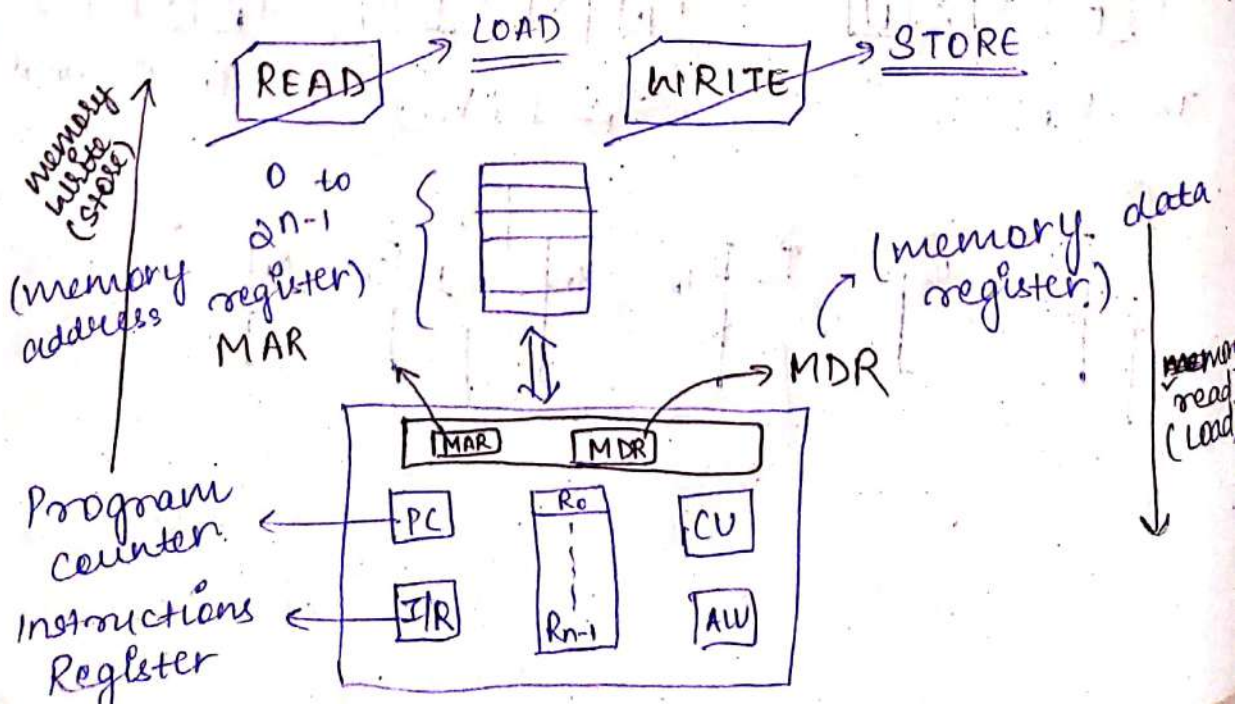
Chapter 03: Basic Structure & Processing Unit

* C = a+b ; → read 'a' → (reading from memory)
  ↓ read 'b'
  2 read operations & 1 memory write operation. Select '+' for ALU write to 'C'.

* scanf ("%d", &a); → data has to be read from I/O & then written to memory

LOad R2, LOC

- Reads the contents from memory location whose address is symbolically given by LOC & loads them into processor register R2.
- Contents of LOC is preserved.
- Contents of R2 is overwritten.



→ 64 - Bit machine (WORD LENGTH)

:- It can read 64 bits to a variable A.

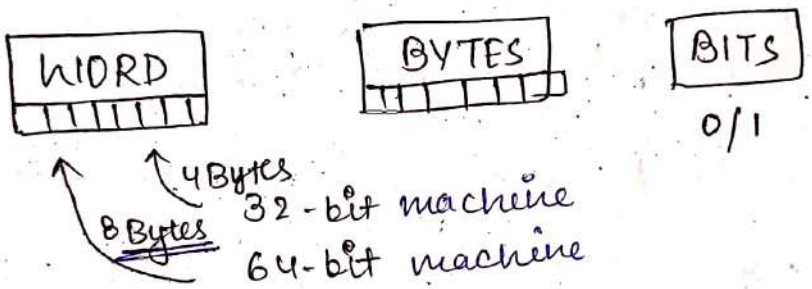
:- It can write 64 bits.

:- It can perform 64 ALU operations at a time.

:- It can even read more than 64 bit.

→ Every memory location stores a WORD.

- For 64 Bit machine we have 8 BYTES in one WORD.
- For 32 bit machine, one WORD has 04 BYTES.



- Every memory location has its own unique address.
- Processor- memory Interface has two special registers: MAR & MDR.

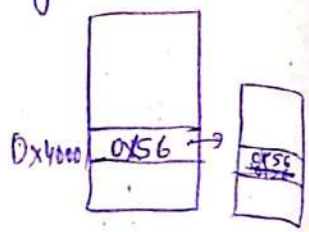
MAR: (memory address Register)
It holds the address of operand.

MDR: (memory data Register)
It holds the data. (the states of all 64 bits/ 32 bits).

Data can vary whereas addresses are unique & doesnot change.

* Load R2, LOC → 0x4000 (Self).

- It means there is a memory whose address is 0x4000.
- Its content is 0X56 (let).
- There is a register R2.



- The content of 0x4000 is stored to R2.
- It is a memory read operation.
- So, the control unit will load the address into MAR. ∴ MAR 0x4000.
- It will generate control signal READ.
- After completion of this READ signal, the address' content will be loaded into MDR. ∴ MDR 0X56.
- This MDR data is written into R2.

* WRITE:

For WRITE operation, Control unit will generate WRITE signal & the contents present in MDR will be written to the memory.

* Program Counter (PC): It holds (points) the address of the instruction that is supposed to be executed next.

- It does not always point to the sequential next instruction. There may be a func^n call.

* IR (Instruction Register): It holds the instructions that is being executed.

* General Purpose Register

$R_0, - - - R_{n-1}$ are general purpose registors.
- They are fastest accessed in the ~~memory~~ processor.

- RISC : Reduced Instruction Set Computer
- CISC : Complex Instruction Set computer

→ Every Instruction is only one word long.
(" " " will be stored in 1 memory locat.

* LOad R2, LOC : (3Ir)
- Instruction fetch
- Decode
- Execute

} Phases

1) Instruction fetch : Instruction from the main memory is loaded on Instruction resistor i.e. IR (via memory - processor (proc - mem) ✓ (interface) interface).

2). Decode : Control signals are generated by control unit to load the address into MAR & content into MDR.

3). Execute : 'load' internally generates memory Read signal to read from LOC & ~~mar~~ Register write signal to store (write) it to the content of register R_2.

★ Store R4, LOC :

- It will copy the content of R4 to a memory location pointed by LOC.
- The state of R4 will be retained whereas content of LOC will be overwritten.

1). Instruction fetch : Instruction from Instruct register (R4) is copied to main-memory location pointed by LOC via processor-memory interface.

2). ~~Decode~~ Execute : The address of R4 will be stored in MAR & its content in MDR. Then control unit will generate write signal to write the content present in MDR to LOC which is present in main-memory.

3). Decode : Control unit generates Read signal to read from R4 & then generate write signal to write it to the memory location pointed by LOC.

★ ADD R4, R2, R3 :

R4 = R2 + R3.

Execute :-

- R2 & R3 will be fetched to ALU.
- Addn operation will take place. & data will be stored back to R4.
- No memory & processor interaction.

* Memory locations & Addresses:

① Data line : It includes the data which is supposed to be read or write. (Bidirectional)

Processor ⟷ Memory.

② Address line : It includes the address. ~~It holds~~ the no. of memory location is same as its bits.

Farom 'n' address line, we can access 2^n memory locations (address from 0 to $2^n - 1$).

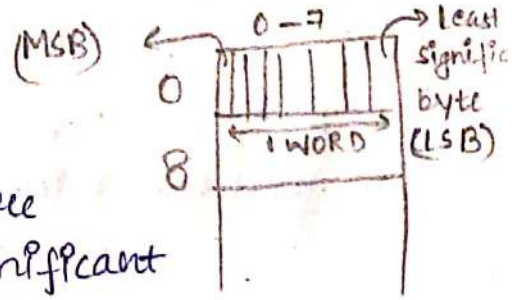
n (address line) ⟶ 2^n memory locations $(0 - 2^n - 1)$ $\equiv n=3$ $2^3 = 8$ mem-loc

possible 8 combinations 000

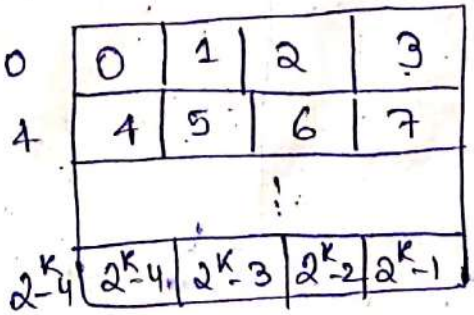
Each location is of WORD length.

③ Control lines : It is responsible for generating control signals (Read (R) | Write ($\bar{w}$)) (Bidirectional)

BIG ENDIAN:

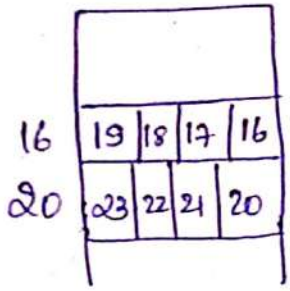


- lower byte addresses are used for the most significant byte of the word.
- Higher order byte stored at lowest address.



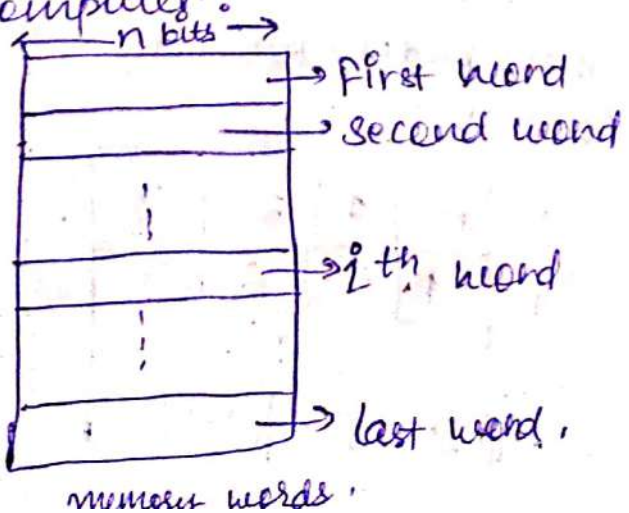
LITTLE ENDIAN:

- It is scheme where the lowest address is used for the least significant byte of the word.



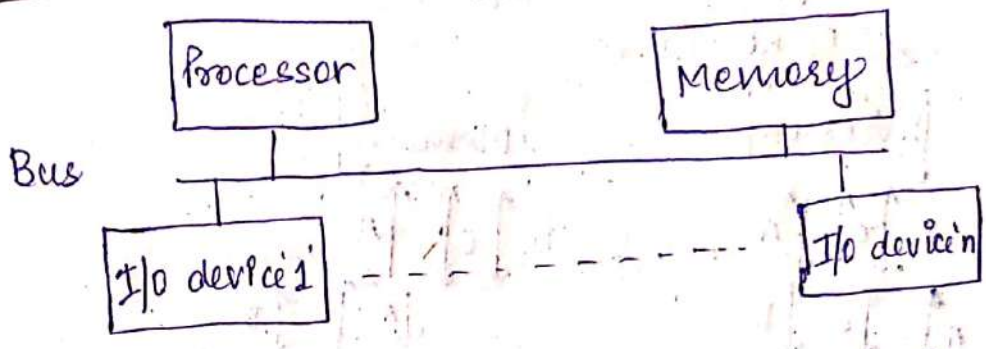
* MEMORY LOCATIONS AND ADDRESSES:

- Each storage cell in memory can store a bit of information (0 or 1).
- Group of 'n' bits (word) can be stored or retrieved in a single basic operation.
- 'n' is the word length.
- Memory is schematically represented as a collection of words as shown.
- Each memory location has a distinct name or address.
- Addresses of successive locations in the memory are typically numbered from 0 to 2^k-1.
- 2^k addresses constitute the address space of the computer.



Word address must be an even number
Imp that is words must be aligned on an even boundary.

★ BUS STRUCTURE:



Bus: It is a collection of control line, data line & address line.

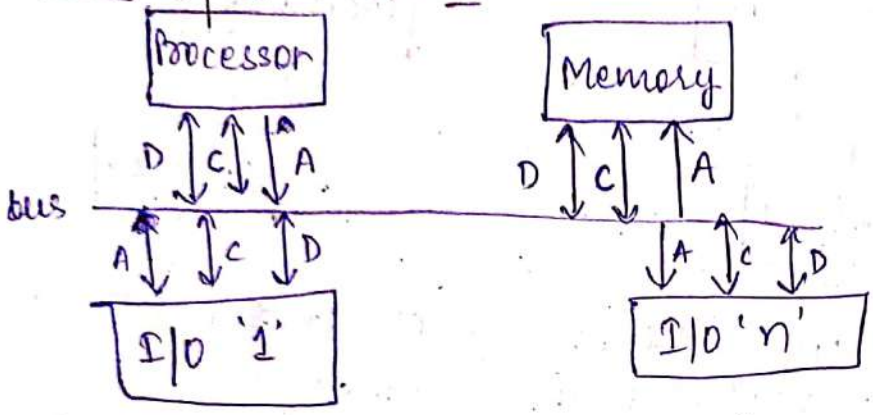
- Multiple I/O devices may be connected to the processor & the memory via a bus.
- Bus consists of three sets of lines to carry address, data & control signals.
- Each I/O device is assigned an unique address.
- To access an I/O device, the processor places the address on the address lines.
- The device recognizes the address, & responds to the control signals.

★ BUS OPERATION:

- A bus requires a set of rules, often called a bus protocol.
- The bus protocol determines when a device may place information on the bus, when it may load the data on the bus into one of its registers, & so on.
- These rules are implemented by control signals that indicate what & when actions are to be taken.

• Every peripheral will have its own unique address. Corresponding to the address & data that is loaded on the bus, only that I/O device will respond.

Master (as of now), here.



✱ Notations :

① RTN
(Register transfer notation)

- Include symbols:
  ( ) Index
  [ ] contens of
  ← Flow of data
- E.g R2 ← [LOC]
  destination ← Source
- E.g: Add R2, R3, R4
  R2 ← [R3] + [R4] : (RTN)

② ALN
(Assembly language notation)

- Include Mnemonics which are 02 or 03 letter words used for instruction.
- E.g: Load R2, LOC
  Mnemonics
  ↓ Add, Store, etc
- E.g Add R2, R3, R4
  (ALN)

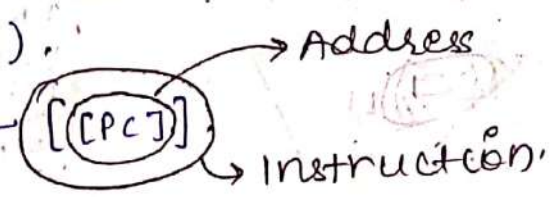
Ques Explain the process for executing an instruction. 6-M

First explain RTN, ALN & then explain INST.

Executing an Instruction:

6-M
Don't write only this much. Only 2 Marks will be given.

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into IR (Fetch Phase).



- Assuming that the memory is byte addressable, increment the contents of PC by 4. (Fetch phase).

$$PC \leftarrow [PC] + 4.$$

↳ If word length is 4-bytes.

If word-length is 8-bytes, then

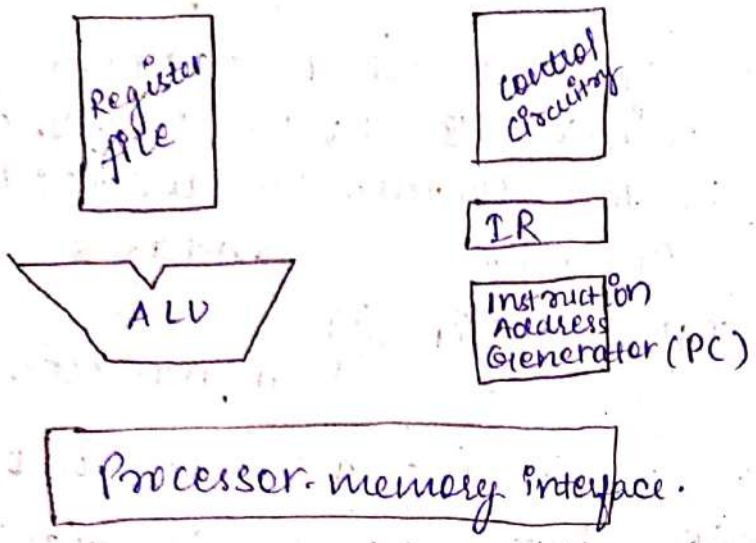
$$PC \leftarrow [PC] + 8.$$

- Carry out the actions specified by the instruction in the IR (execution phase).

PC is a pointer which always holds address & never stores data.

↳ If the content of PC is loaded into IR, then it is a data.

↳ Or, it can also be Instructions to be executed next.



* Stages of Operation :

Stage 1 : Fetch.
2 : Source Registers. (Reading Register)
3 : ALU.
4 : Memory Access (Read/Write)
5 : Destination Registers.
(write to register)

① Load R5, X(R7)

X = base address.
X + [R7] = effective address.

(i) Fetch the Instruction
Increment PC.

(ii) Read the contents of the register R7.
Decode the Instruction.

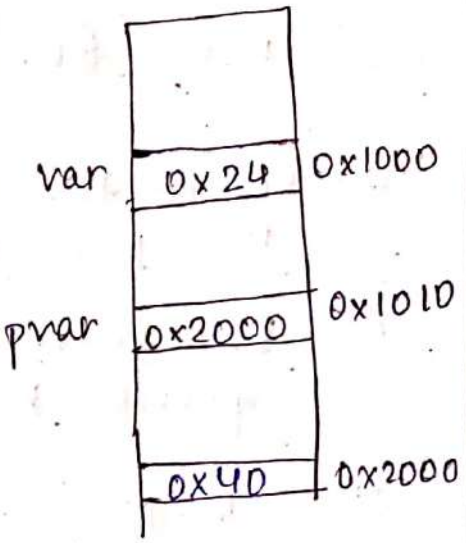
(iii) Perform ALU operations to get the effective address. X + [R7].

(iv) Read from the source memory location.

(v) Store (load) the contents to the destination register, R5.

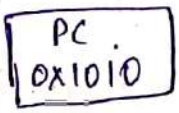
Eg. int var;
int *pvar;
pvar = 0x2000;

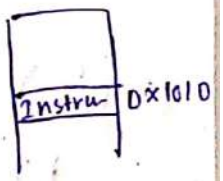
→ &var - 0x1000
→ var - 0x24
→ pvar - 0x2000
→ &pvar - 0x1010
→ *pvar - 0x40



(Imp.)

- R2 ← PC
  ↳ in RTN, it has no meaning.





- $R_2 \leftarrow [PC]$: Valid. 0x1010 will be loaded into R2. & R2 will consider it as data.
- IR ← [PC] : 0x1010 will be loaded into IR & IR will interpret 0x1010 as instruction & accordingly it will be decoded.
- IR ← [[PC]] = *pc. : The instruction stored at [PC] i.e. 0x1010 will be interpreted by IR.

(Q) Add R3, R4, R5.

(i) Fetch the instruction & increment PC.
(ii) Read the contents of R4, R5 & decode the instructions.
(iii) Compute [R4] + [R5]
(iv) NO action
(v) ~~Store~~ load the result into destination register R3.

(3) Store R6, X[R8].

(i) Fetch the instruction & also increment the program counter (PC).

(ii) Decode the instruction & read contents from Register R6 & R8.

(iii) Compute the effective address i.e. X + [R8].

(iv) Write the contents to X+[R8] of R6.

(v) No action. {Memory Access}

anything (any operation) related to memory will be written.

∴ here destination is not a register,
∴ No action.

- 32 registers in Register file.
  ↓
  Each register has 32 bits.
  ↓
  Every instruction is of 1 word long (32-bit).

* RISC : → It directly does not support memory access.

→ For a RISC processor, every instruction is of 1 word in length.

→ The operations of RISC processor include:
- Load/store instructions to access the operands.
- ALU operations.

→ Assumptions :

- There are 32 Registers each of 32 bit length.
- The word length is 32 bits.

→ The execution of Instruction is done in 05 stages.

Stages :

1). Fetch an Instruction & Increment the program counter.

{Fetching the Instruction from memory & Incrementing the PC to point to the next Instruction to be executed in the memory.}

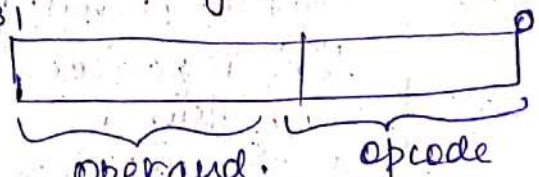
Load R3, N

- Every Instruction has 02 parts :

  Opcode : load

  Operands : R3 & N.

- Instruction Register (IR) is 32 bit long.



2). Decode the Instruction & read registers from the register file.

{Decoding the Instruction refers to generating appropriate control signals by CU to perform the Instructions in Instruction Register. (IR)

First part (some part) of IR is opcode & some part of IR is operand.}

3). Perform an ALU operation.

Eg 1 Load R5, X(R7).

ALU op: address computation i.e. X + [R7]

Eg 2 Store R2, X(R3)

ALU: X + [R3]

Eg 3 Add R2, R3, R4

ALU operation = [R3] + [R4].

4). Read or write memory data if the instruction involves a memory operand.

Load R3, X(R7)	Store R2, X(R3)	Add R2, R3, R4
Memory Operation → memory read	memory write	No memory operation involved.

5). Write the result into destination register, if needed.

Load R3, X(R7)	Store R2, X(R3)	Add R2, R3, R4
Destination Register → Write to R3	NO destination register (Involved) ∵ R2 is source register & destination is a memory location not a register	Write result to R2.

Ques Write all OS stages for instruction

Clear R3

→ Write operation to R3 so that every FF is reset.

R3: 1 1 0 1 1 → 0 0 0 0 0

1). Fetch the instruction into IR & Increment PC.

2). Decode the instruction. (~~& read content of R3~~ (no need to read R3))

3). Perform ALU operation ~~involved~~ to clear.

4). No memory operation (NO action)

5). ~~No dest. reg. involved.~~ (Write 0 to R3).

ALU operations: Clear R3

- $R3 := R3 \& R3'$:=0
- or $R3 := R3 \& 0$ =0
- or $R3 = 0$

} so that all values (each FF) of R3 is reset or becomes 0.

↓

If we are resetting R3 to 0, then no need to read R3 in step 2.

* HARDWARE COMPONENTS:

1). Register File
2). ALU
3). Datapath
4). Instruction Fetch Section

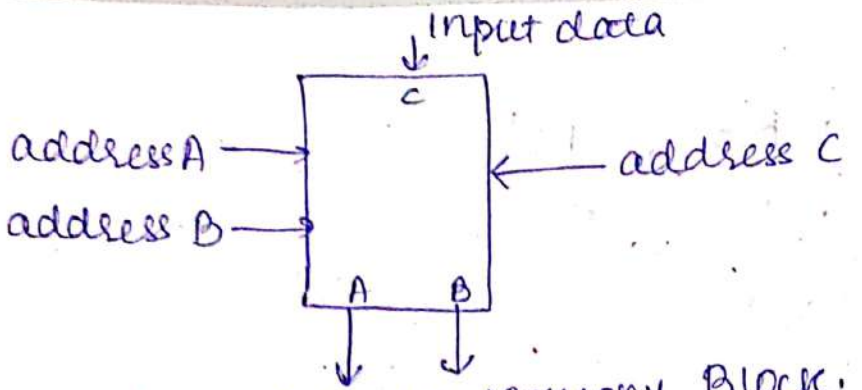
→ can read 02 registers at a time but can write only 01 register at a time.

Register File:

- Register file allows to read two registers from the file at one instant & the content of registers are available at both A & B.

The address of the registers to be read (source registers) are provided to the register file interface circuitry (RFC) through lines address A & address B.

- The register file allows to write to a single register at a time and the content to be written to the register (destination register) is made avl on code 'C' and the address of destination register should be provided at add. line numbered as address C here. (Fig. behind)



(a) Single Memory BLOCK.

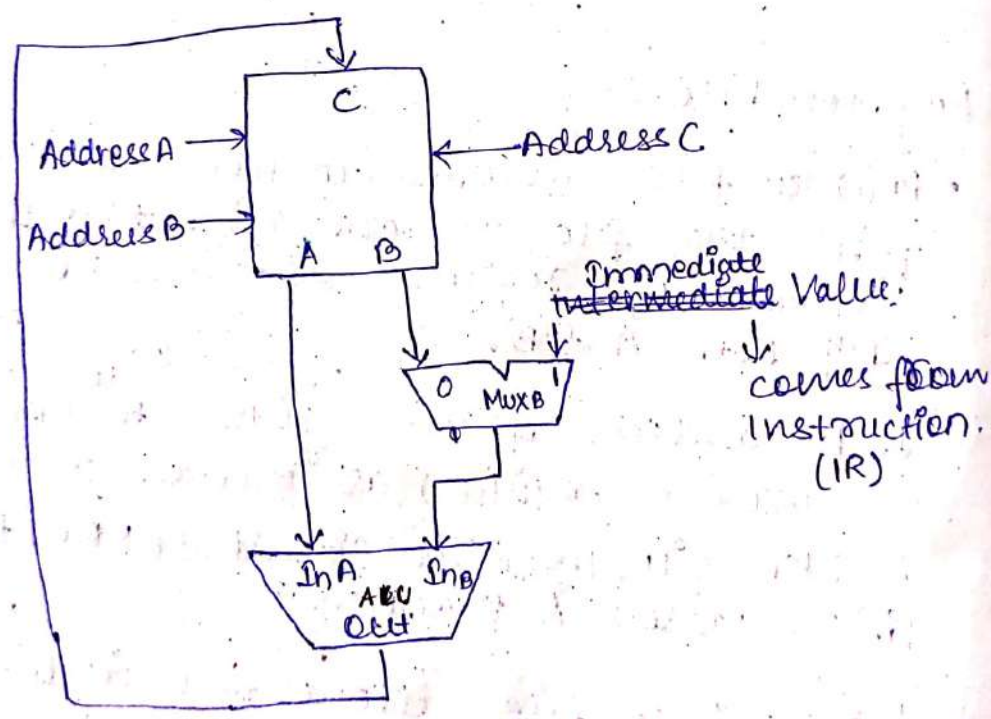
ALV:

Imp

*** must write for ALV operation

In RISC operation, both operands must be Register. RISC does not allow direct memory access. The contents need to be stored in register file first & then ALU operation is performed.

while CISC allows memory access as operands.



- MUXB is used to select between register operands & Immediate value.

Datapath :

- In datapath, there is a need of temporary register so that the value can be stored and used at next stage.

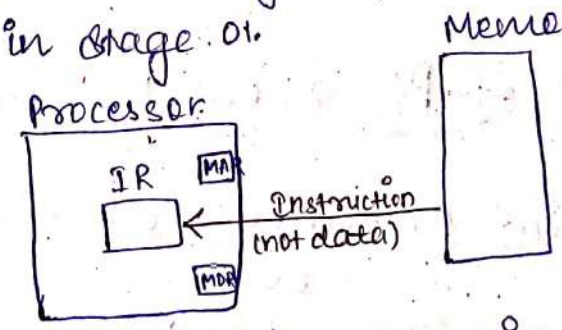
- 'Memory address line' is connected to processor memory interface in MAR.
- 'Memory data line' is connected to MDR. in stage 04, in processor memory interface. ↳ of datapath diagram.

- From Register to ALV, Rz stores the effective address of data and sends to MAR.

- In Stage 4, Return Address comes from program counter which stores the address of instruction to be executed next.

Instruction Fetch Section :

- It's objective is to fetch the instruction from the memory location to the instruction register in stage 01.



Once the address of instruction is known, it is stored in MAR. Now, the instruction ~~falls into~~ from that address falls into MDR.

Now, from MDR it comes to IR.

∴ We only need to know the add. of inst. which we can get from Instruction Address Generator.

- Once the instruction is loaded into IR, it is decoded by control circuitry & appropriate control signals are generated.
- The immediate values are extracted from IR and given to MUX B through the Immediate block.

e.g. → source reg. → destination reg.

- Store R2, X(R3)
  ↳ immediate value
- Load R2, X(R3)
- Call LOOP
- MOV R2, #90
- SUB R2, R3, R4 (all three are reg. 2 are source & 1 dest)

- Immediate block will extend the no. of ~~blocks~~ bits to 32 bits. (Appropriate zeros or signs are added). 0000 0 ← 01110
  1111111 ← 1001

Updation of PC {How PC is modified}

1). Sequential Execution: current value will be incremented by 4.
PC ← PC + 4.

2). Branching Instructions: current value will be incremented by offset.
PC ← P.C + offset

Jump offset is difference bet^{n} current value & target value of PC.

3). Subroutine (Function) Calls: PC will be updated with a new address.
- This new address is coming from the instruction, after the instruction is decoded in IR. Here the immediate value is extracted.

4). Subroutine Return: New address will be stored in PC. This new add. is return address coming from Register file.

* PC-temp is a temporary register where the processor stores the ~~return address~~ content of PC before updating the PC in case of func^{n} call type of instructions.
- This the address stored in PC-temp is backed up as return address in link register of Register file via MUX Y (i/p 2) and register RY.

Read Section 2.7 Pg 56, 57, 58, 6th Edition for Subroutine Call Instruction. Also Pg 61

Ques Write sequence of actions (5 stages) to fetch & execute an instruction with reference to RISC datapath.

ADD R3, R4, R5.

1). Fetch the instruction & ~~read the content of source registers R4 & R5.~~ increment the PC.
2). Decode the instruction & read the content of source registers R4 & R5.
3). Store these content into temporary registers.
3). ALU → [R4] + [R5]
4). Mem X
5). Write to R3.

1). Fetch the Instruction & Increment PC.

Memory Address Line :- ← [PC]
↳ Addres of the Instruction.

- Read Memory : reads the memory location whose address is in memory address line.
- [IR] ← Memory data,
- PC ← [PC] + 4. such that it starts pointing to the next inst.

2). DECODE

- Decode the instruction
- Address A should be R4 & B should be R5 after decoding.

RA ← [R4] RB ← [R5] {Read register}

3). ALU : ALU result is stored in Rz.

Rz ← [RA] + [RB]

4). Memory Access : Since, there is no memory operation, memory read or memory write signal will not be generated. ∴ RY will be loaded with the content of Rz.

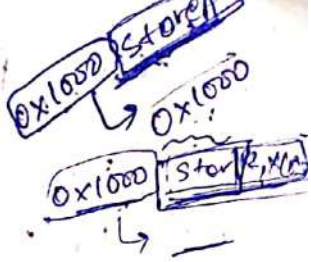
RY ← [Rz]

5). Destination Register : write to destination register. R3 ← [RY].

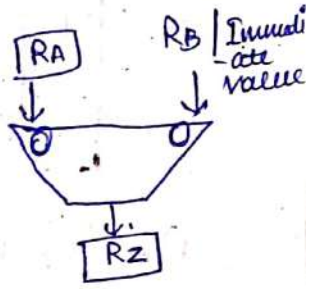
:RTN	R3 ← [R4] + [R5]
ALN	Add R3, R4, R5

★ Load R5, X(R7).

1). • Memory Add. line ← [PC].
• Read Memory.
• IR ← Memory data line.
• PC ← [PC] + 4.



2). • Decode.
• RA ← [R7]



3). • Calculate Effective Address:
Rz ← [RA] + Immediate value
// Rz is add. of memo. we want to read.

4). RY ← [Rz]. Memory Read.
= RY ← [R7]

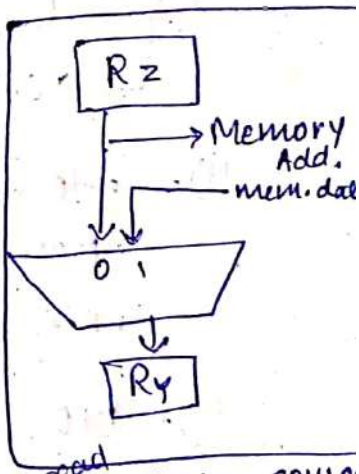
Memory Add. ← [Rz],
memory read,
RY ← Memory data

5). R5 ← [RY]

★ STORE R2, X(R3)

1). • Memory Add. line ← [PC]
• Read Memory
• IR ← Memory data line
• PC ← [PC] + 4.

P.T.O

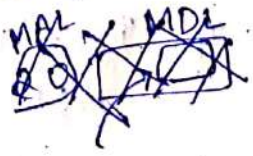


• Load dest, source
• Store source, dest.

2). • Decode
• RA ← [R2]

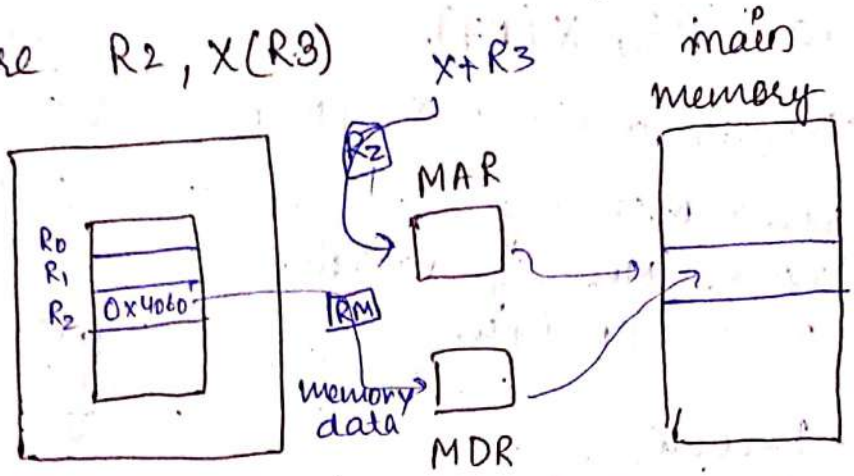
3). Rz ← [R3] + Immediate value.

4). • Memory Add. ← [Rz]
• Memory Write
• RY → Memory data.



P.T.O

* Store R2, X(R3)



1). _ same

2). Decode
- $R_B \leftarrow [R_2]$
- $R_A \leftarrow [R_3]$ // To compute effective memory address

3).
- $R_M \leftarrow [R_B]$ // RM is used for memory write
- $R_Z \leftarrow [R_A] + X$
  - address where we want to write
  - content we want to write

4).
- Memory Add. line (MAR) $\leftarrow [R_Z]$
- Memory Data line (MDR) $\leftarrow [R_M]$
- Memory write signal.

5). No action.

// If Instruction involves memory, ALU will calculate Effective memory Add

// If Instruction involves +, -, *, &...
: ALU will do the desired operation.

// Else, ALU will perform nothing.
(Control signals will be generated in such a way that ALU is disabled).