(11) ALU, Data Path & Control Unit

Parts of general purpose computer (GPC):-

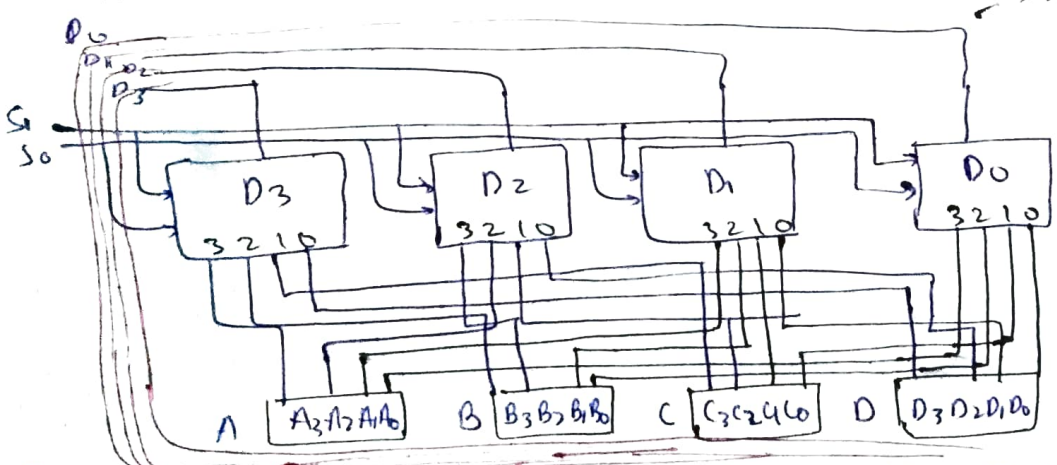
→ we have two types of machines that have I/P are :
① General purpose Computer ② Embedded Systems

Embedded Systems are stored program computer in which programs are included to perform specific task only.

Components of GPC are –
① Memory ② ALU → Add, S, M, D, Shift, EX-OR etc.
③ Registers [Collection of bits stored in sequence of flip flops]
④ Timing circuit to synchronize the Instruction.
⑤ Control Signals ⑥ flags [1 bit information as flip flop]
⑦ Common Bus or channel for connection among all components.

→ for timing signal we use sequence counters, such that using 'n' bit counter we can execute 2^n instructions.
→ Control signal can be generated using the logic gates
→ Bus can be maintained using the MUXs

Designing of bus



→ Look at the image in video at the time of recession

S_1	S_0
0	0
0	1
1	0
1	1

Generally we have lots of register of size 16b, 32b, 64b but here we are taking only 4 register of 4 bit each.

S_1	S_0	O/P
0	0	$D_3 D_2 D_1 D_0$
0	1	$C_3 C_2 C_1 C_0$
1	0	$B_3 B_2 B_1 B_0$
1	1	$A_3 A_2 A_1 A_0$

✓ No of MUX = Size of register
✓ Size of MUX = No. of register

Designing a Computer:

Assume the m/m size is 4096 X 16 means complete complete m/m size 4096 in which you can read a 16 bit of word at each clock.

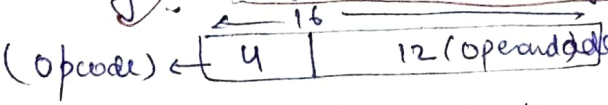
DR → Data register holds the operand fetched from m/m. Size of register should be 16 bit here.

AR → Address register stor the address. Size of register should be 12 bits because m/m size is 2^{12}.

Here DR can can hold 16 bit but AR can can send only 12 So, the most significant bit of AR will get filled with 0's and left 12 bits should be same.

AC → Accumulator is a register available with ALU. It stores generally output of ALU. (16 bit).

IR → Instruction register used to store and decode instruction. It is of size equals to size of max. instruction (16 bit).



Here $2^4 = 16$ different opn are possible.

We can also use operand addrs bits to increase no. of operations.

(PC) → Program Counter. (TR) → Temporary register.

IR/OR → Input / Output Register: Read each bit as input and send to m/m or ALU. (say 8 bit here).

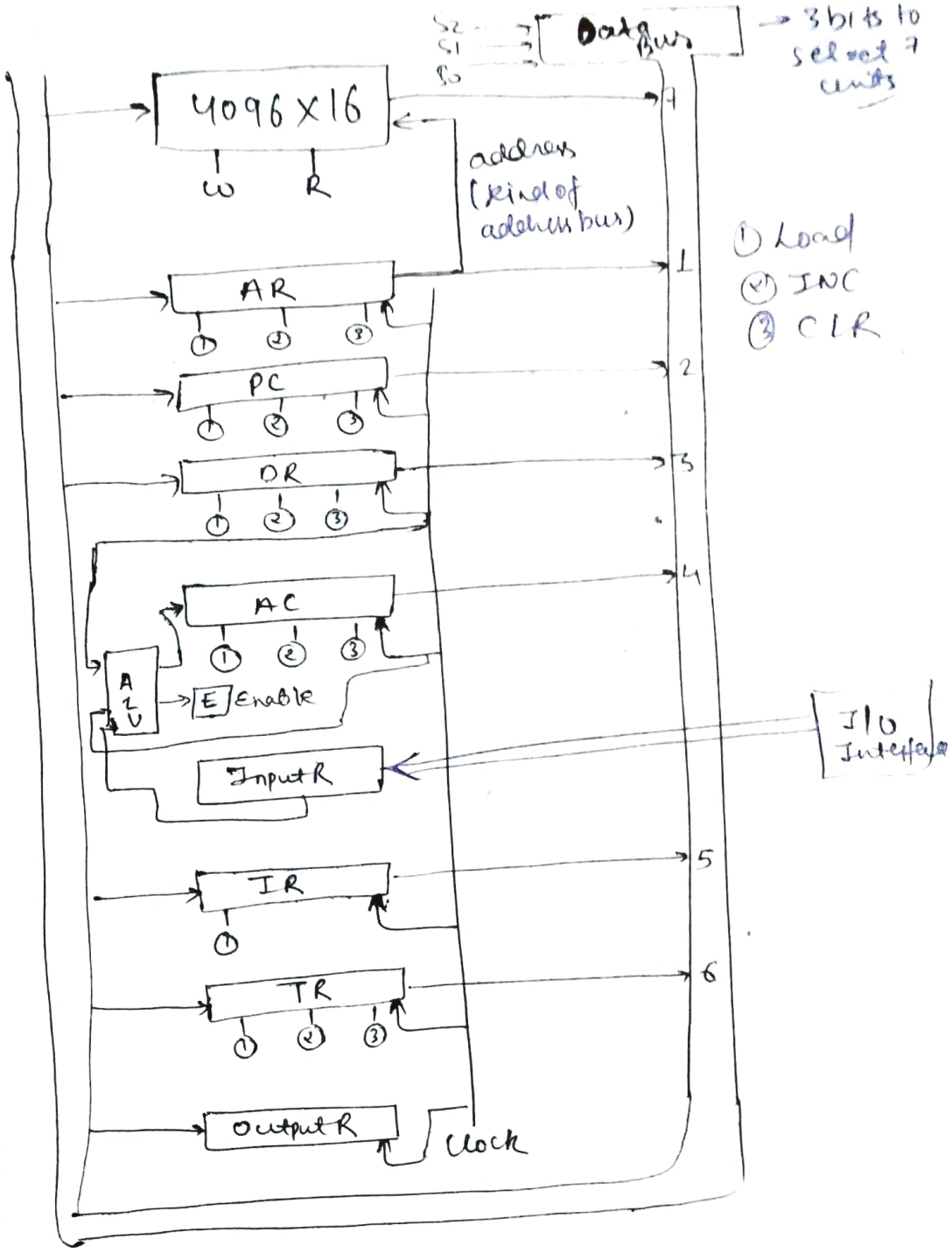
(ALU) → We are going to perform addition here.

Whatever the operation you want to select, select via $S_2 S_1 S_0$ (select lines).

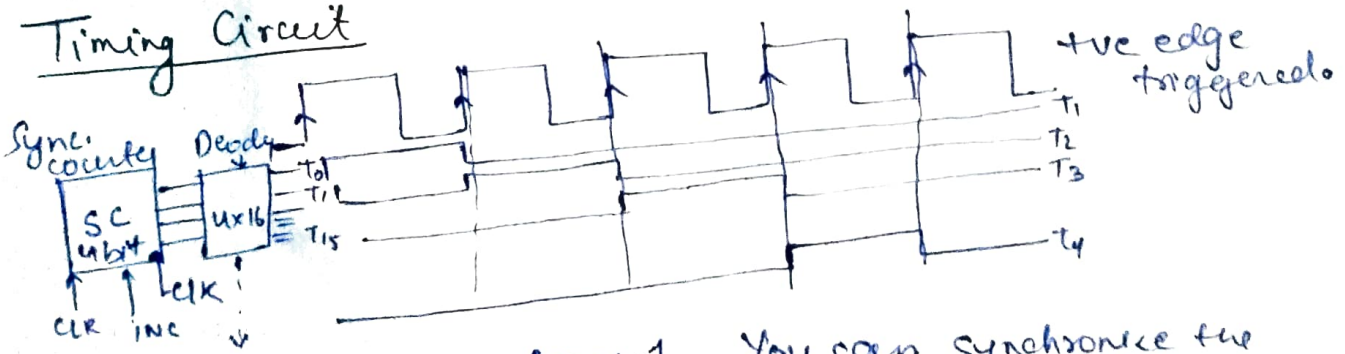
Bus → Data, Address, Control

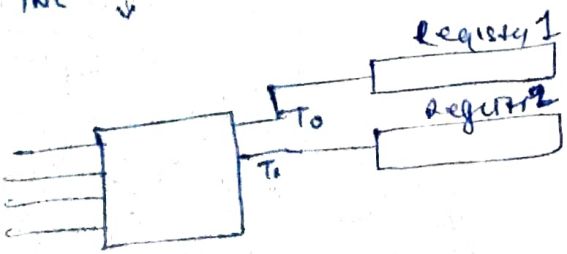
We are going to use data bus here.

Control bus contains control signals (LD, INC, CLR). These are not individual lines from control unit.



Timing Circuit

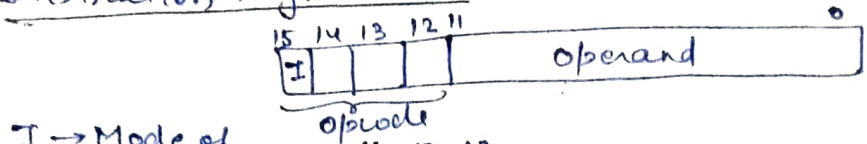




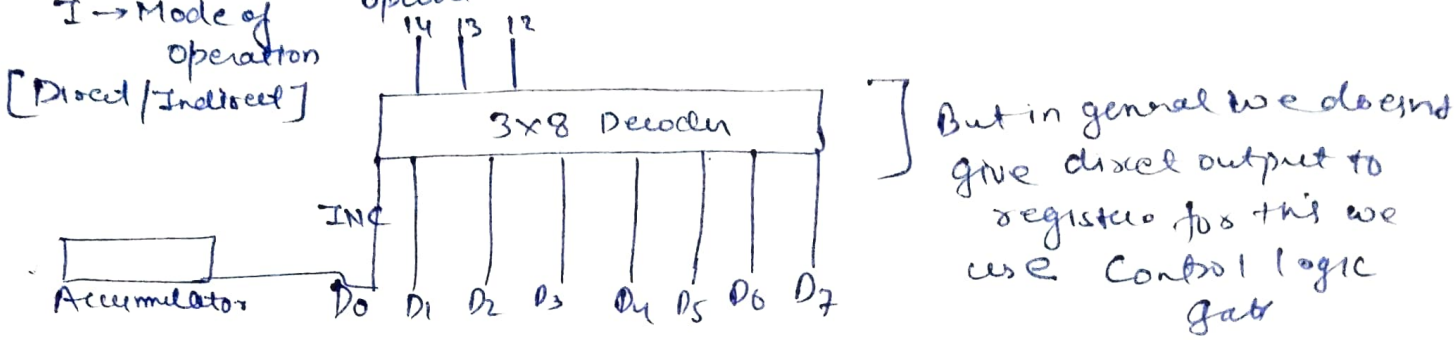
You can synchronize the counter the by connecting it with decoder and decoder will manage it using time pulses.

→ Each time only one register will be activated.
→ It is also known as hardwired program control Unit.

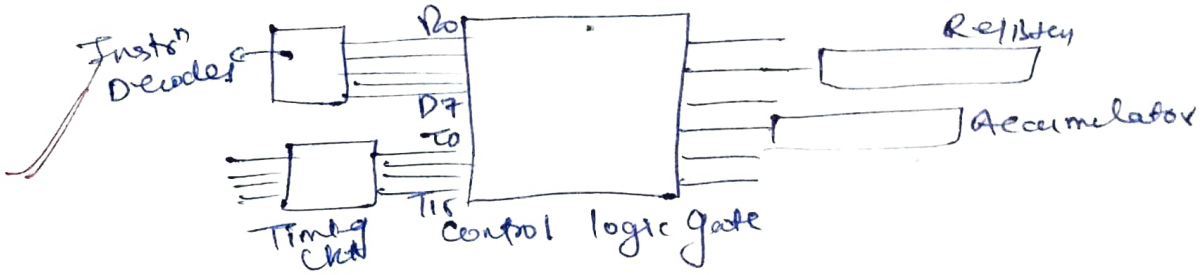
Instruction Register (IR)



I → Mode of Operation [Direct / Indirect]



But in general we don't give direct output to register for this we use Control logic gate



Instruction Cycle :- Phases of Instn cycle :-

① fetching ② Decoding ③ Compute effective address ④ Execute

- Direct (DO Nothing)
- Indirect (Access m/m)

fetching :

I_0 : AR ← PC

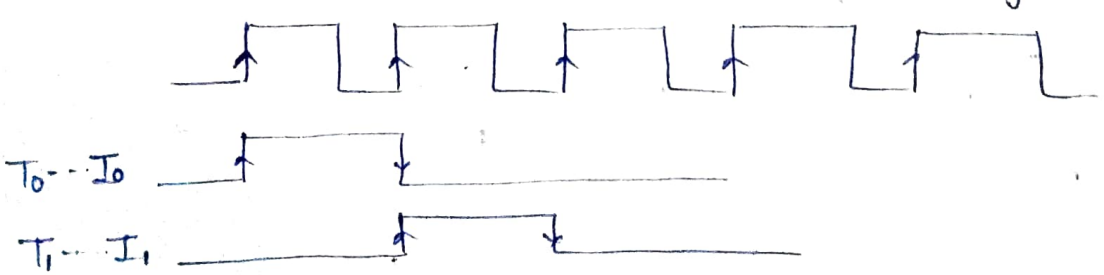
↓ Add. will load Register PC

I_1 : IR ← M[AR]

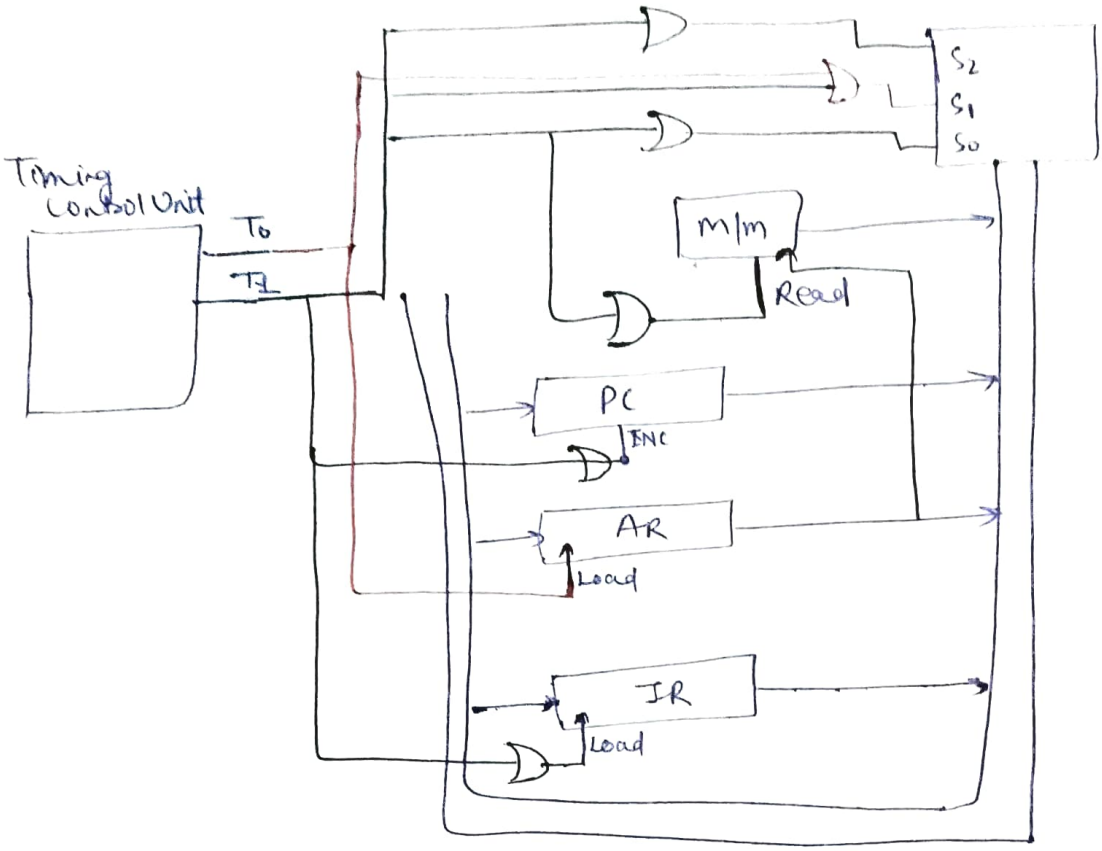
↓ Load content of AR to m/m [M] and load it to IR.

Particular register will get select as

	S_2	S_1	S_0
PC :	0	1	0
Memory :	1	1	1



Instruction I_0 occur at T_0 and I_1 occur at T_1



We use the 'or' gate because it might be possible that more than 1 timing signal demanded for access.

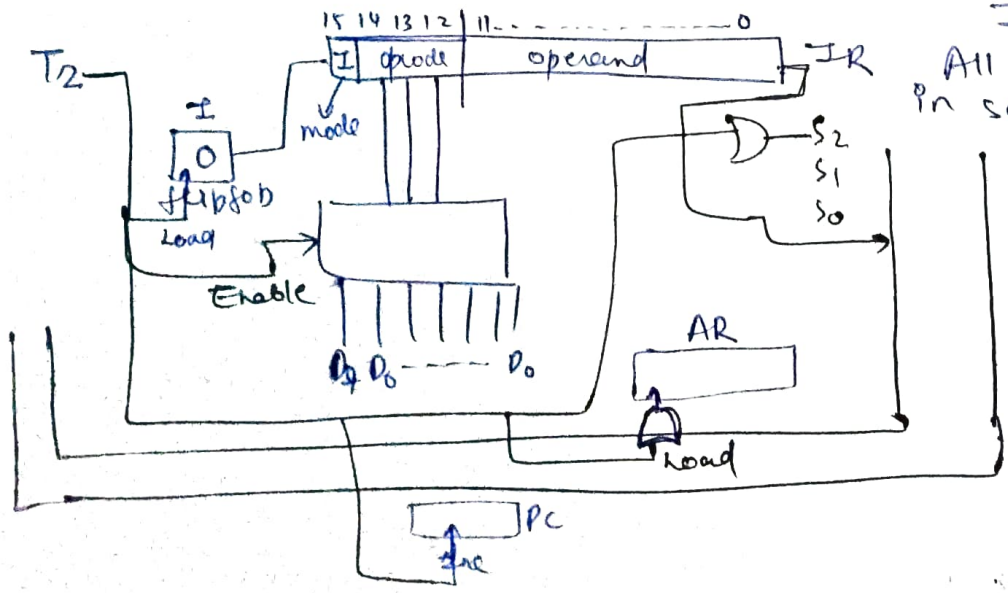
as I_1 get executed at the same same PC will also incremented as $PC \leftarrow PC + I_0$ so we give T_1 to INC in PC here too.

So, here instruction fetch is taking 2 cycles (T_0, T_1) but it might get vary from system to system.

$T_0, T_1 \rightarrow$ fetch

Decoding: T_2 : $D_7\ D_6 \cdots D_0 \leftarrow$ Decode IR(14,13,12); AR $\leftarrow$ IR(0,11)
$I \leftarrow IR(15)$

All 3 things will happen in same cycle



IR
S2 S1 S0
1 0 0

Memory size > Register size, So in case of register we can't hold 12 bits operand. So we can reduce it by devoting few bits for operation codes. So this way we can generate more than 8 opcodes

from decoder o/p : $D_0 \ldots D_6$ → Memory reference [I = 0/1]

$D_7 = 1$ → I = 0 → Register
I = 1 → I/O related

Memory reference address have both I = 0,1 variants

Effective add. given	I = 0	I = 1 → Needs to find effective address
AND	0xxx	8xxx
ADD	1xxx	9xxx
LDA	2xxx	Axxx
STA	3xxx	Bxxx
Branch Unconditional → BUN	4xxx	Cxxx
Branch & save address → BSA	5xxx	Dxxx
ISZ	6xxx	Exxx

Register related

CLA	7800	SZE	7002
CLE	7400	HLT	7001
CMA	7200		

I/O related

INP	F800
OUT	F400
SKI	F200
SKO	F100
ION	F080
IOF	F040

There can be more than these instructions

So in register and I/O instruction we are using the operand bits also.

$T_3 \rightarrow$ AR ← M[AR] : Used in case of calculating effective address So it is required for mem related I = 1 but not for I = 0.

If T_3 not required, we can start execution here

fetching Effective Address (T_3)

$D_0 \ldots D_6$ — Memory reference

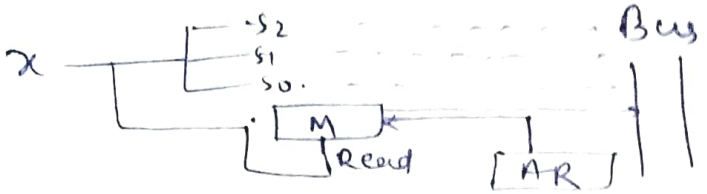
D_7 — I/O or Register related

Each instruction is no need to go through all the steps of instruction execution

If $D_7 = 0$, $I = 1$, and $T_3 = 1$ then we have to calculate the effective address.

$I = 1$ for indirect addressing mode

$AR \leftarrow M[x]$



→ wait for execution phase and implement it in next cycles

If $D_7 = 0$, $I = 0$ and $T_3 = 1$; No need to do anything so, T_3 is completely wasted or stall.

If $D_7 = 1$ and $I = 1, T_3 = 1$; I/O No stalling go to new phase that is execution.

$D_7 = 1$ $I = 0, T_3 = 1$; Go to execute phase / no waste of time.

So in m/m related we start execution from T_4 and in I/O and register related we start it from T_3.

Execution [fetch operands & execute]: (T_4)

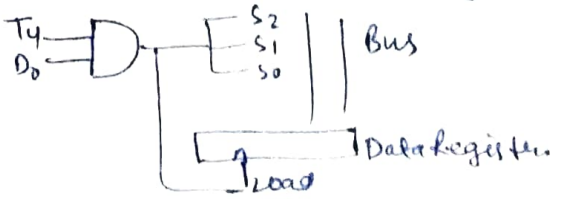
Say, in m/m reference instruction,

AND	D_0
ADD	D_1
LDA	D_2
STA	D_3
BUN	D_4
BSA	D_5
ISZ	D_6

In m/m reference instruction we perform fetching effective address at T_3.

AND: $T_4 = 1, D_0 = 1$ $\quad DR \leftarrow M[AR]$

Assume 1 operand is already in AC and second we fetch from m/m to data register.



$T_5 = 1, D_0 = 1$: $\quad AC \leftarrow DR$ AND AC

In generally fetching m/m is slow and takes more than 1 clock cycle. But here we are assuming ideally.

So, to execute it takes (T_0 to T_5) for AND Instr^n. [6 clocks]

(T_0, T_1)	(T_2)	(T_3)	(T_4)	(T_5)
Fetch Instn	Decode	EA (optional)	Fetch operand	Execution

After complete execution we need to clear the sequence counter to start from T_0 again.

So make

$T_5 : SC \leftarrow 0$

② ADD D_1 : $D_1 = 1$ $T_4 = 1$: $DR \leftarrow M[AR]$

[6 Clocks]

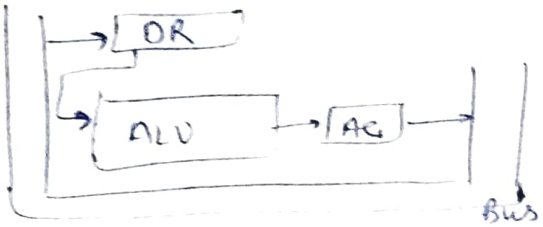
$D_1 = 1$ $T_5 = 1$

$AC \leftarrow DR + AC$

$E \leftarrow Cout$ [Carry]

$SC \leftarrow 0$

③ LDA D_2 : $D_2 = 1$ $T_4 = 1$: To load something into AC, pass the word to bus and then word will pass to AC through ALU. But ALU is connected with bus. So we load it in DR first then it will pass to AC.



$DR \leftarrow M[AR]$

$D_2 T_5$: $AC \leftarrow DR$, $SC \leftarrow 0$ [6 Clocks]

④ STA D_3 : It can be done in 1 cycle only.

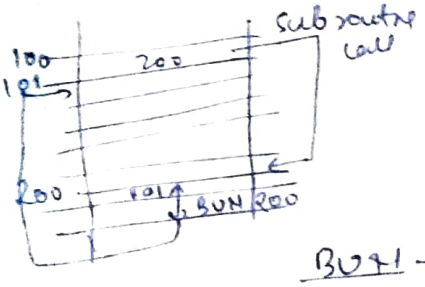
$D_3 T_4$: $M[AR] \leftarrow AC$, $SC \leftarrow 0$ [5 Clocks]

⑤ BUN D_4 : $D_4 T_4$: $PC \leftarrow AR$, $SC \leftarrow 0$

[5 Clocks]

It will set Program counter to Instruction execute next to

⑥ BSA D_5 : You come back to 101 after executing subroutine. for this we save 101 at 200 and next Instruction of subroutine is BUN 200 means you have to go 200 and find the next executing address there.



BUI → Branch Unconditional.

$D_5 T_4$: $M[AR] \leftarrow PC$

$AR \leftarrow AR + 1$

$D_5 T_5$: $PC \leftarrow AR$; $SC \leftarrow 0$

Method we are using here doesn't support recursion.

[6 Clocks]

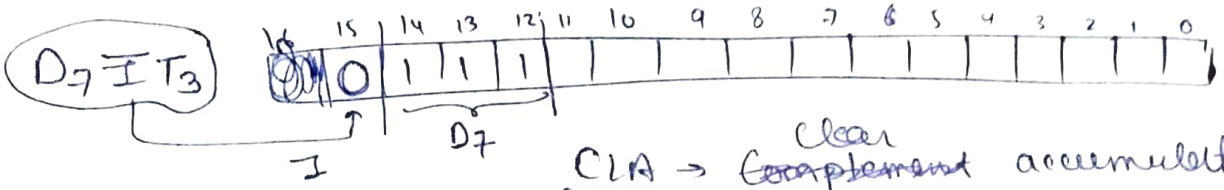
ISZ D_6 : Increment and skip if zero. [7 clocks]

$D_6 T_4$: $DR \leftarrow M[AR]$

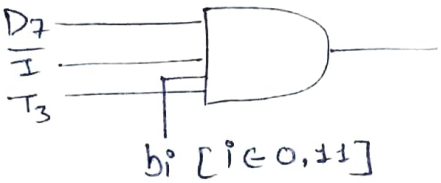
$D_6 T_5$: $DR \leftarrow DR + 1$

$D_6 T_6$: $M[AR] \leftarrow DR$; if $(DR == 0)$ $PC \leftarrow PC + 1$
$SC \leftarrow 0$

Register reference instruction → We start executing from T_3 and sometimes we can even finish at T_3.



CLA → Clear accumulator
$b_{11} = 1$



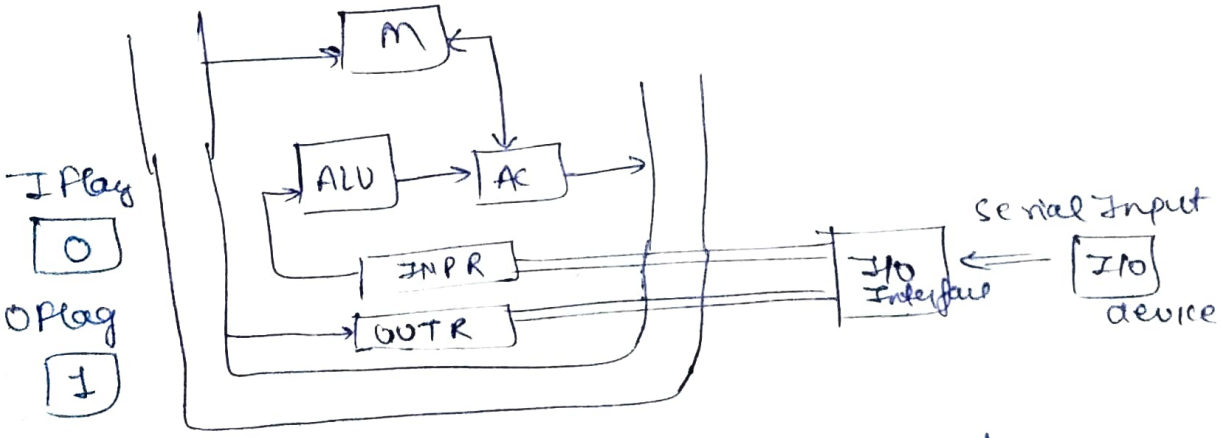
CLE : $b_{10} = 1$

CMA : $b_9 = 1$ Complement accumulator

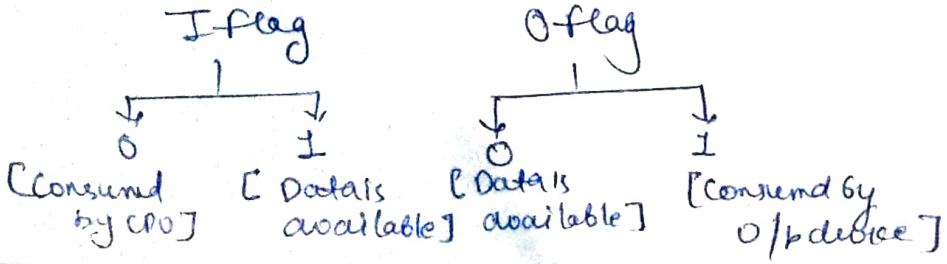
Similarly we can implement total 12 instructions in this size of register by changing value of b_i.

→ All the register instruction requires only 4 clocks.

I/O related instruction : [Total 4 clocks required]



CPU works faster than I/O device so to make synchronization we need I/O flags



15	14	13	12	11	...	0
1	1	1	1			
I	D7					

$D_7 = 1 \Rightarrow I = T_3$:

$T_3 = 1$

D_7	I	Remark
0	0	[stall]
0	1	M [Effective Hold]
1	0	Register
1	1	I/O

① INP → $b_{11} = 1$
② OUT → $b_{10} = 1$
③ SKI → $b_9 = 1$
④ SKO → $b_8 \rightarrow 1$
⑤ ION → $b_7 \rightarrow 1$
⑥ IOF → $b_6 \rightarrow 1$

bp [i∈ 6,11]
↓
We have different AND gate for each value of here.

INP : Read INPUT : AC ← INPR [If IFlag = 1] ; FGI ← 0 (IFlag)
OUT : Move o/p : OUTR ← AC ; FGO ← 0
SKI : If (FGI = 1) ; PC ← PC + 1 : skip Input
SKO : if (FGO = 1) ; PC ← PC + 1 : skip output
ION : Interrupt ON : INT ← 1
IOF : Interrup OFF : INT ← 0

SC ← 0 in all

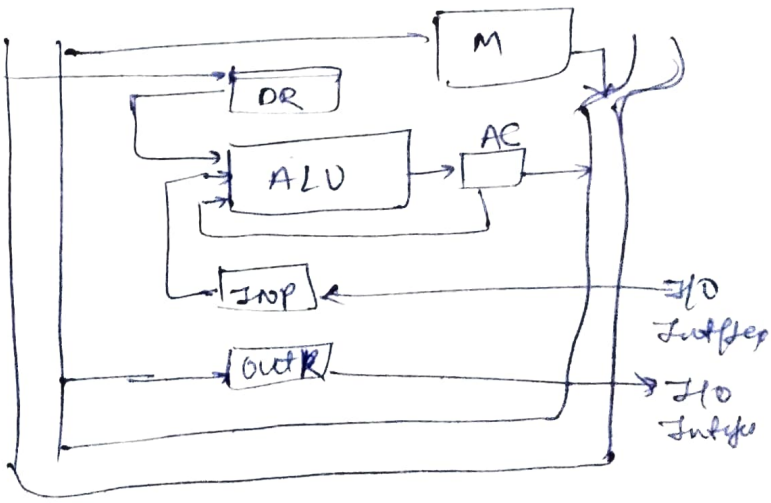
Interrupts are generally hold by OS only.

(12)

If there is any input coming from Input Interface the sequence is

Input Intefce → INPR
BUS ← Ac ← ALU
↳ memory.

M/m → DR → ALU → AC
OUTR ← BUS



Programmed I/O : Suppose, our computer is programmed to execute 1MIPS, then,

Say rate of input generate is 1o character per sec.

Say Program for I/O devices is

200	SKI
201	BUN 200
202	INPR

we can get input by just using last instruction but due to input flag check condition we are executing 1st instruction regularly, called busy waiting.

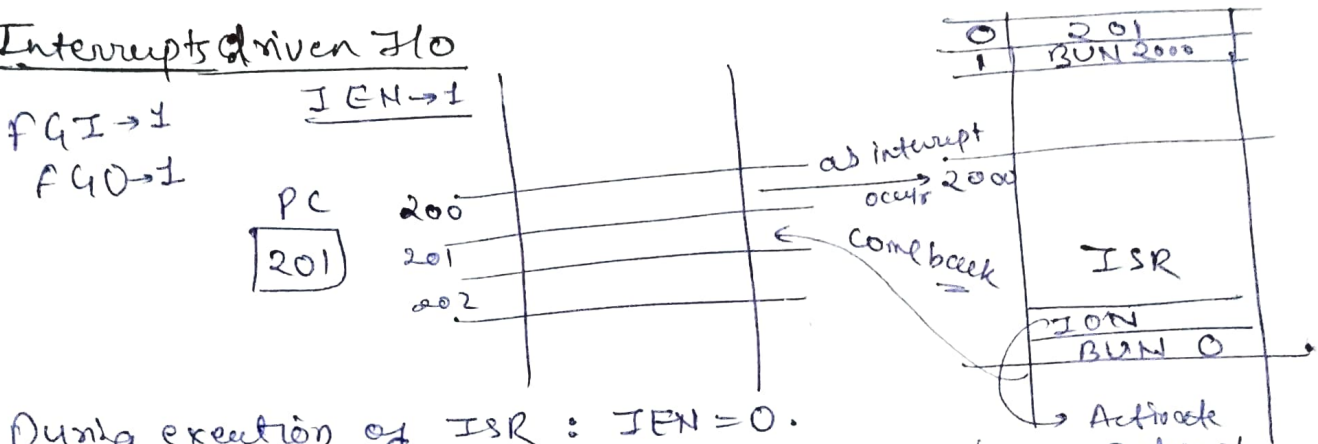
So 1 character = 10^5 usec means in that time we can execute 10^5 instruction but we are executing only 1 instruction, so its waste of time.

Interrupt driven; IEN = 1 [Program can be Interrupt]

[Interrupt Enable]

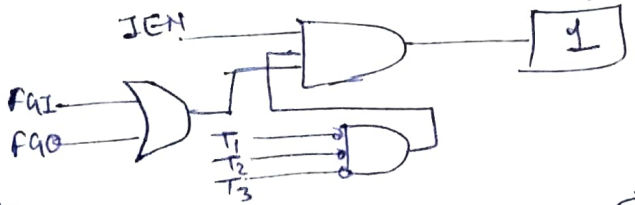
We can't give control of IEN flag to user. It will always dealt in OS mode.

Interrupts driven IO



During execution of ISR : IEN = 0.

Condition for writing an interrupt



R = 1 means program wants to be interrupt

Interrupt Cycle

[we check R at every T0 clock]

RT_0 : AR ← 0 ;

RT_1 : M[AR] ← PC ;

RT_2 : PC ← 0 ;

RT_3 : PC ← PC + 1 ;
IEN ← 0 ;
R ← 0 ; SC ← 0

Temp. register (TR)

(or)

RT_0 : AR ← 0 ;
TR ← PC

RT_1 : M[AR] ← TR ;
PC ← 0

RT_2 : PC ← PC + 1
IEN ← 0 ;
R ← 0 ; SC ← 0

Two ways

Instr Cycle

$\overline{R}T_0$: fetch
$\overline{R}T_1$: fetch
$\overline{R}T_2$: Decode
$\overline{T_3}$ / T_4 } Execute

If no interrupt (R) then it follows normal execution

⇒ You can find interrupt upto T_2 only. else no interrupt

If $\overline{T_0}\,\overline{T_1}\,\overline{T_2}$ IEN and (FGI or FGO) is 1 then R = 1

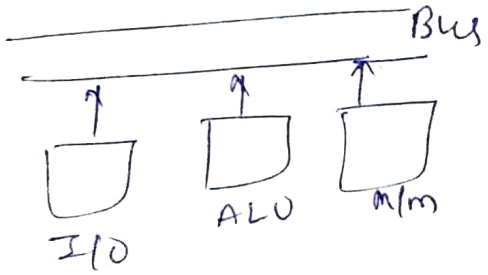
If R ≠ 1 in T_0 then we will try to keep R ≠ 1 until T_1 & T_2 gets done.

So, to handle Interrupt we are wasting $T_1\,T_2\,T_3$. This is an overhead.

So handle overheads we have DMA [Direct m/m access]

Till now we have

but in DMA



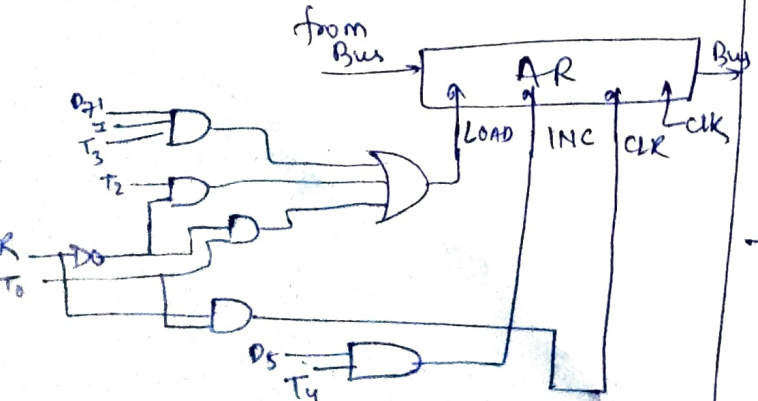
(24)

Example of Control Unit

LOAD AR : $R'T_0 + R'T_2 + D_7' I T_3$

CLR AR : RT_0

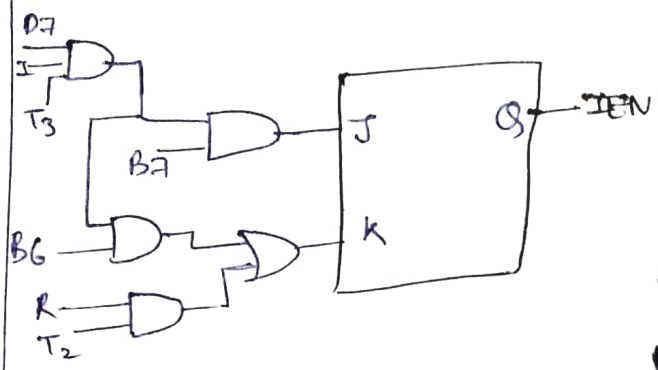
INR AR : $D_5 T_4$



→ Given example is Control Unit for Address Register.

Similarly we combine all the control units to form a computer.

Interrup Enable flag



$D_7 = 1$, $I = 1$

$D_7 I T_3 B_7$: IEN ← 1

$D_7 I T_3 B_6$: IEN ← 0

$R T_2$: IEN ← 0

ION, IOF will work if $B_7 = 1$

Hardware CU is faster, but it is not flexible. You can't change the operation of CU after complete manufacturing.

You can't add or delete more instruction.

So we need "Programm control Unit.

→ Instn set is fixed in hardware.

To perform an operation we need to perform several steps called 'u' operations

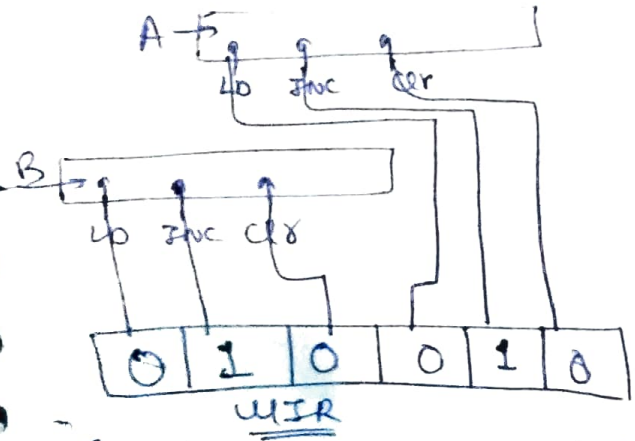
Ex: AC ← AC + R1

LDA 21
LDA R1, 9
ADD R1
] u operations

Each 'u' operations are performed using Control Unit.

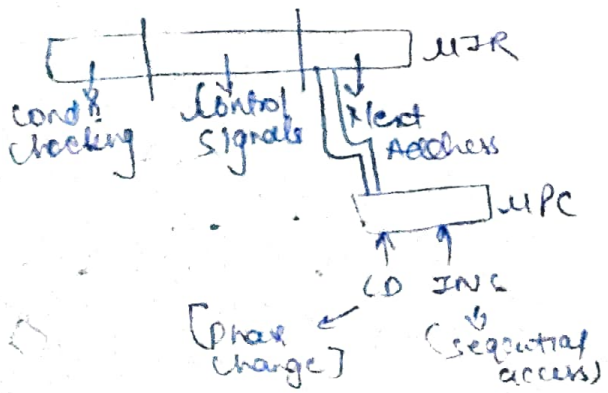
Hardwired → we use hardware as control unit, but we can also do it using software called micro programming

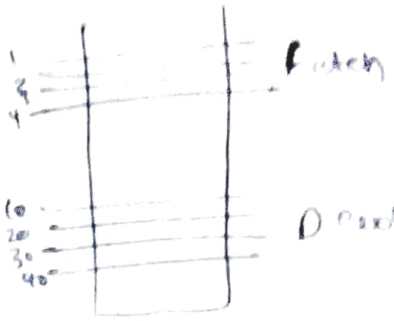
We have micro instructions as-



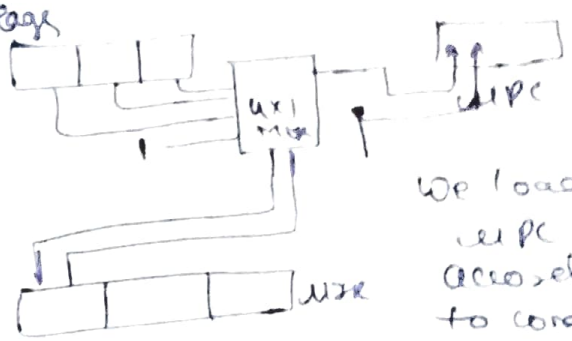
Above MIR, increments the content of A and B called as control word.

Control word stored in control memory. [ROM]





Inside fetch MPC just gets increment but in switch of from fetch → Decode we have to load new address to flags



We load to MPC according to condition block in MIR otherwise INC occur automatically.

Condition → This part depends upon no of flags we use for n flags we need $\lceil \log_2 n \rceil$ k1 bits

NA → Size depends on size of control mem. If Control mem have m words then $\lceil \log_2 m \rceil$ k2 bits.

Control Signal → Depends on no of instructions required

MPrograming can be done in 2 ways

① Horizontal: We can increase parallelism and its fast

② Vertical: It is slow because of extra use of decoders

For horizontal partition for N instruction we needs N bit in control signal, in vertical partition we need $\lceil \log N \rceil$ bits

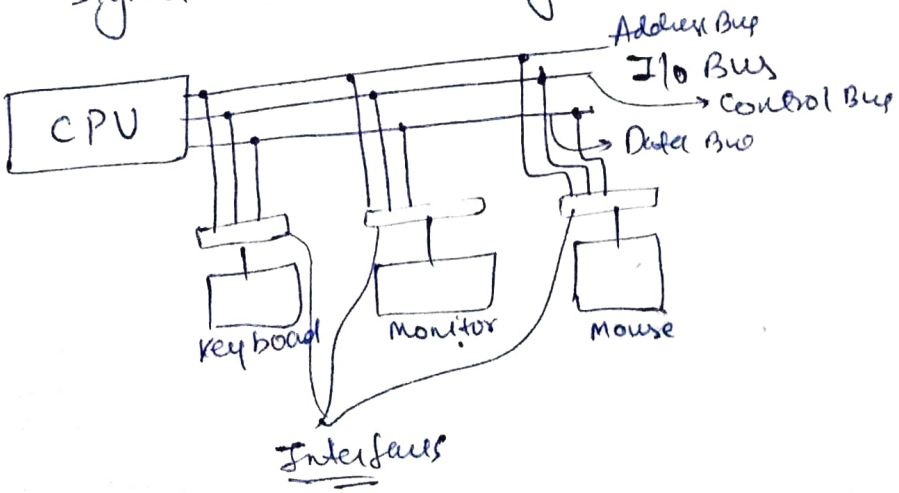
H.c.P	V.c.P
1) Control word is large	1) Control word is small
2) High parallelism	2) No parallelism
3) Faster	3) Slower

→ We can also develop hybrid of both.

Hardwired Control Unit	Microprogrammed Control Unit
(1) fixed Instruction	(1) Flexible Instructions
(2) High speed	(2) Slower as compare
(3) Expensive	(3) cheap
(4) Complex	(4) simple
Ex → RISC	Ex → CISC

I/O Interface

Need: (1) Peripherals are electromechanical and electromagnetic, but CPU & memory are electronic devices. The signal conversion may be required.



(2) I/O devices have slower data transfer rates.

(3) Data codes and formats ~~have~~ in I/O are different from CPU.

(4) All devices are different from each other.

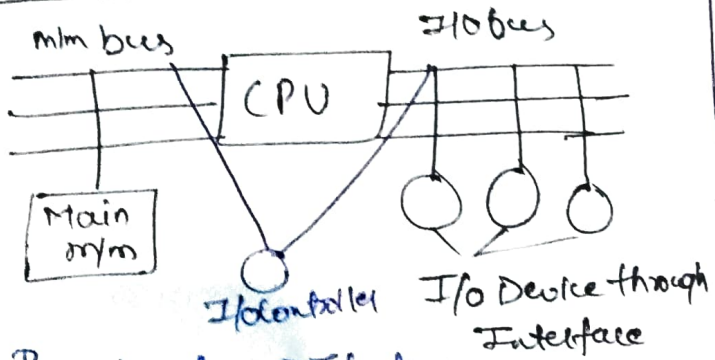
Interface gets activated according to the value kept by CPU on address bus.

Once Interface gets activated CPU generates commands to control bus and then access the data.

Types of commands -

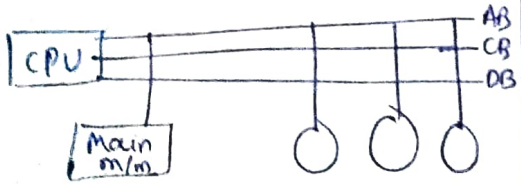
1. Control Commands
2. Status Commands
   ↳ To get the present status of devices
3. Input Command → To read
4. Output Command → To Write.

Isolated vs M/m mapped I/O



Peripherals → I/O device

2nd Model



Suppose CPU generate address then there is a conflict that the address is for main m/m or the peripherals. To resolve this we have control bus.

CB have 4 different lines along with its functionality

1. I/O Read
2. I/O Write
3. Memory Read
4. Memory Write

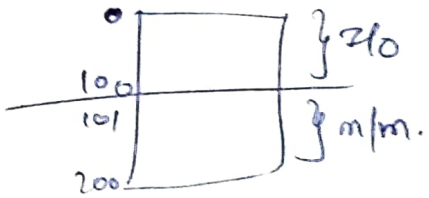
①, ② = 1 means I/O op^n else m/m operation.

This is called as Isolated I/O concept. It says confusion b/w m/m and I/O are connected on same side then only it is Isolated. But we have to remember the same

I/O : Input / Output
M/m : Load / Store

In m/m mapped I/O we divide main m/m into 2 parts

1. for m/m purpose
2. for I/O purpose



Adv.: We can use same Ins^n for both m/m and I/O.

Address of I/O means address of register inside interface.

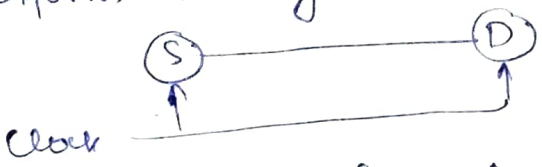
Serial vs Parallel Transmission

Serial Mode Ex → Keyboard

Parallel Mode

Serial	Parallel
① Shift registers required	① No Shift registers
② Burst Errors	② Bit error
③ Cheap	③ Costly
④ Less reliable	④ More reliable
⑤ Large distance	⑤ Less distance
⑥ Ex→ keyboard	⑥ Bus

for large distance b/w source and destination we can't perform task of both with just one clock as-



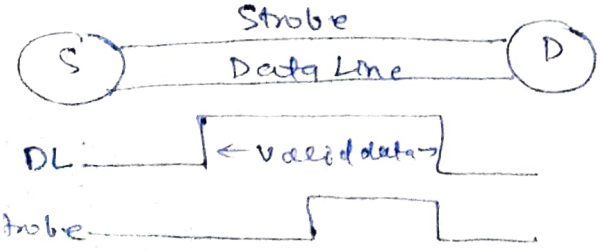
→ It creates clock skew problem.

So we can't use clock in this condition.

With clock (one) → Synchronous without clock (no/more than 1) → Asynchronous.

① Strobe Signal.
② Handshaking Method.

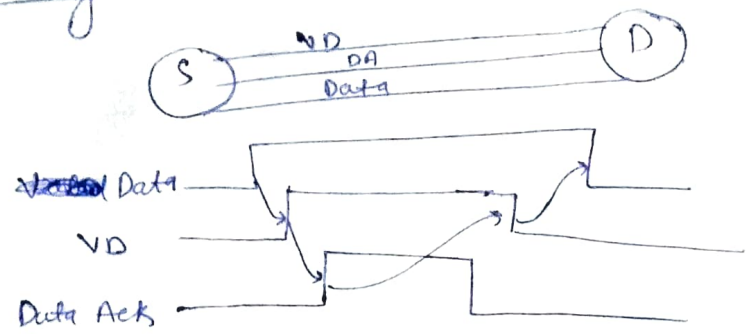
① Strobe



Here destination can't say, I have received the data. So it is a problem.

Handshaking

VD → Valid Data
DA → Data Accepted

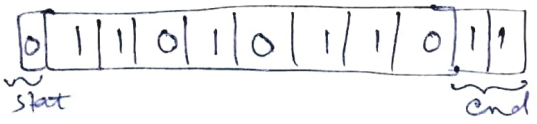


→ Till now whatever we discuss is data transmit by the source but it can be possible that data should ask by the destination.

→ Here we use single data line (serial) but we can use multiple data lines too (parallel) and rest is same.

Asynchronous Serial: 1 char = 8 bits

Data is : 11010110



we need to add some bits to separate or show the start & end.

Say to send 8 bits (1 char) we need to send 11 bits.

Suppose keyboard is giving I/p as 10 char persec

So total 11 x 10 = 110 bits needs to send.

→ In Async we have more than 1 clock and we need to again synchronize them.

Modes of I/O Transfer:

1. Programmed I/O
2. Interrupt driven I/O
3. DMA

(1) Each programmed I/O have a program like –

{
① Read the status flag
② If data is not available (status = 0) goto step 1
③ Move the data
}

Disadvantage → Busy waiting at step ②. So (μP) time gets wasted.

(2) Whenever device is ready with data, it will interrupt by ISR.

Disdv: Less overhead than Programmed I/O but exists.

→ Sensor Interrupts always takes highest priority.

Also, Interrupts can be generate by timers, traps, I/O.

Traps means you stuck in infinite loop (Busy waiting, using extra space, devide by 0).

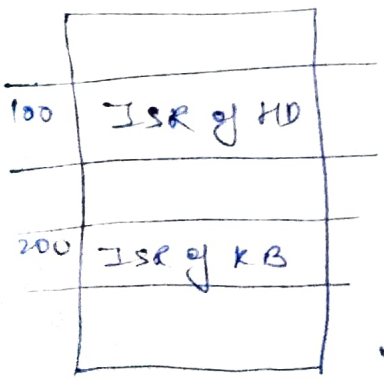
Priority of I/O devices

HD → Mouse → Printer → Keyboard
(1) (2) (3) (4)

Priority based Interrupts

HD	Mouse	Printer	KB	
1	1	1	1	HD
0	1	1	1	Mouse
0	0	1	1	Printer
0	0	0	1	KB

In general we have different ISR for each devices



Suppose KB interrupt gets interrupted in between by HD ISR then we have to go for ISR of HD in between.

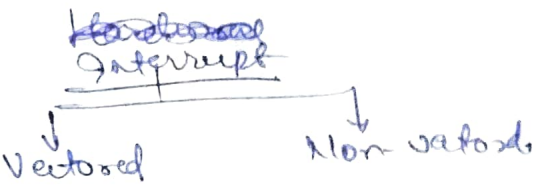
So, here is a problem that how can we get to know about the address of ISR of HD.

(1) Software Based
(2) Hardware Based

Software means pure software but hardware means mixup of both software and hardware.

In software approach we do polling, means we go to each device and ask him that, he is the one who perform ISR according to priority (Non-Vectored)

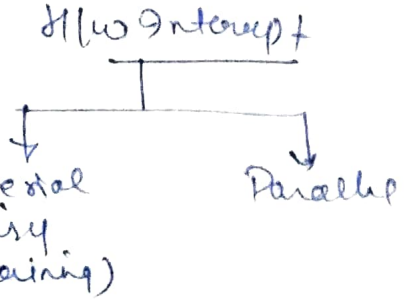
Disdv: Time consuming.



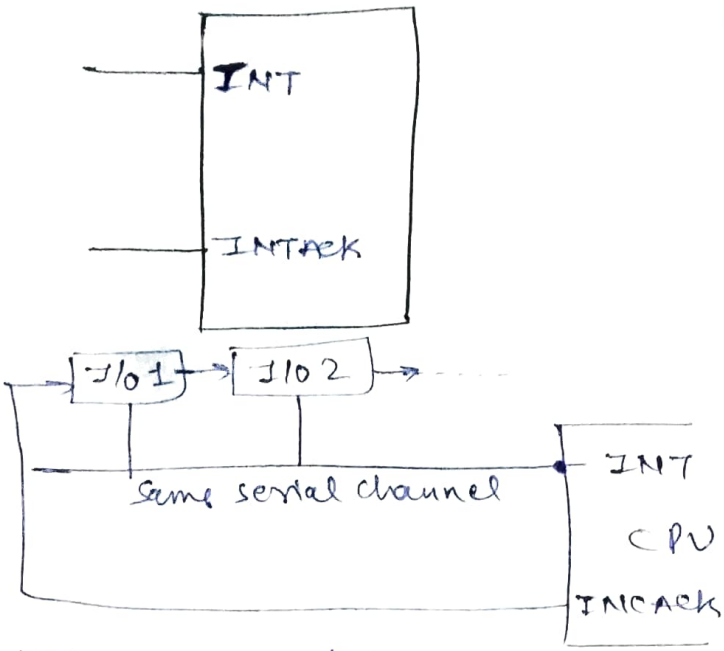
→ In vectored ISR generated or given with base address

→ In non-vectored base add. doesn't given

So need of polling is not required, faster, and Vectored.



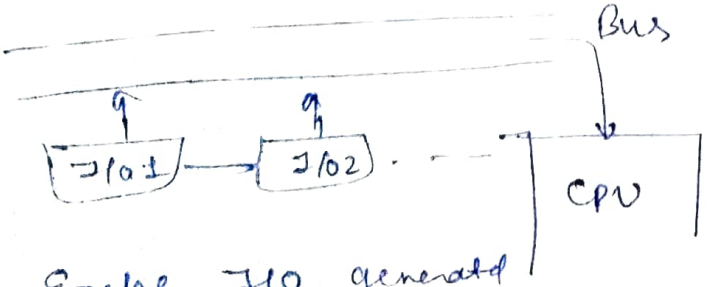
Daisy Chaining



INTACK used to accept Interrup by higher priority devices

Devices are connected in order of increasing priority if ACK signal reaches to device generate the Interrupt that device will not allow INTACK signal to go forward

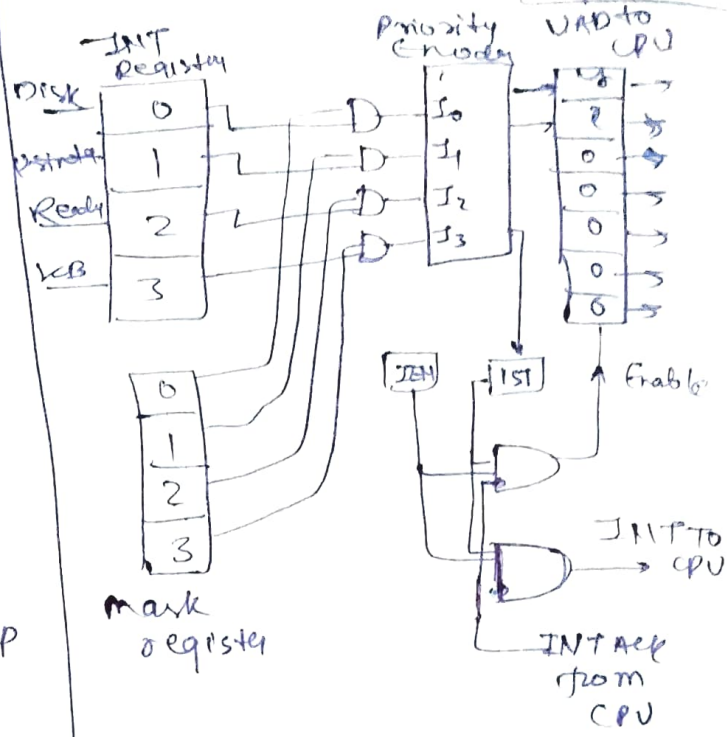
I/o device provide Interrupt signal and address of ISR



Each I/O generate vector address for ISR and load on Bus to support CPU.

H/w implementation is faster than S/w implementation.

Parallel Connection



I_0	I_1	I_2	I_3	y	x
1	x	x	x	0	0
0	1	x	x	0	1
0	0	1	x	1	0
0	0	0	1	1	1

for Disk Interrupt, Vector Address → 00 000000

for Printer Interrupt, VAD is 01 000000

ISI = 1 means atleast one device generate interrupt

If IEN = 1 means there is an Interrupt arises and we are allowing it.

If IEN = 0 then it means CPU is busy and we can't handle this Interrupt

As Enable becomes 1, ISR gets loaded. firstly ISR update the mask register according to priority.

Interrupt Overhead.

In order to perform operations like read from Input R (INP) and write (OUT), we do perform lots of other instruction in support of these Instruction. All these are counted as overhead here.

Interrupt Cycle:

SP ← SP - 1 (Stack Pointer)

M[SP] ← PC

INTACK ← 1

PC ← VAD

IEN ← 0

→ Stack pointer grow upwards [1000 → 0]

Initial operations of ISR

There are overheads -

1. Clear lower level masks bit
2. clear IST.
3. Save content of processor registers.
4. Set IEN flag.
5. Proceed with service routine

final Operations of ISR

There are also overheads -

1. Clear IEN
2. Save content of processor register
3. Clear the bit in the interrupt register belongs to source that has been serviced.
4. Set lower level priority bits in mask register
5. Restore return address into PC and set IEN.

→ Absolute & direct addressing are same thing.
↳ load direct from given add.

→ In relative addressing size of addres. get reduced and useful in case of branching. [PC + Address.]

→ In operand forwarding if data is fetch from memory then we we can do read register along with m/m write and execution just after m/m write [2006 : 2.33]

→ In relative add. reduces instruction size, allows indexing of array with same Instruction, enable relocation.

→ Relative add. are calculated from absolute add. so they can't be greater than ~~for~~ faster than absolute add.

→ RFE (Return from exception is a priviliged Instruction performed by OS to switch between kernel → User after the end of handling an exception. So it is a trap Instr^n.

→ If one trap instruction is executing, no other trap can execute at the same time.

→ Control Hazards penalties can be partially not completely removed by dynamic branch predictions.

→ In delayed branching we generally keep Load/Store Instruction in delayed slots.

→ In pipeline, if WAW hazards also available in Instr^n along with RAW then do. (OF) after complete the execution (ALU) phases of previous Instruction.

RISC

→ Reduced Instr^n set
→ Less no. of Instr^n
→ fixed length Instr^n
→ more registers.
→ few no of addressing modes
→ Less cost and cheap
→ Single cycle Instructions
→ Only in registers. (Reg → Reg.)
→ Hardwired ~~Microprogrammed~~ control Unit.
→ CPI ≈ 1 using pipeling

CISC

→ Complex Instr^n set
→ Large no. of Instr^n
→ Variable length Instr^n format
→ Large no. of addressing mode
→ High cost and powerful
→ several cycle Instruction
→ manipulate direct in m/m
→ Hardwired control unit and microprogrammed
→ No pipeline used.

→ In RISC.

Window size = Local Reg. + Global Reg. + (IN + out)
= L + G + 2C

Register file size = (No. of register window) * (L+C) + G

→ Total no. of bits required to represent control signal in vertical programmed IO is, K (say).

No Parallelism & Slowdel

$K = \lceil \log_2(\text{Group1}) \rceil + \lceil \log_2(\text{Group2}) \rceil + \cdots$

In horizontal programmed IO: $k = (G1 + G2 + \cdots)$] Parallelism and faster

→ Memory mapped IIO uses the same set of read, write control lines for both memory & IIO transfers.

→ Isolated IIO has distinct input & output instructions.

→ M/m mapped IIO is faster, less complex and consumes less power as compare to IIO mapped.

① Implied : No address CLC.

② Immediate : MOVE R1, 35H → direct value is given

③ Register mode : Load from one register to another also called direct register mode : MOV R1, R2

④ Register Indirect Mode : MOV R1, [R2]

⑤ Direct / Absolute : Only one m/m reference ADD R1, [0301]

⑥ Indirect Mode : Symbol @, ().

⑦ Base register : Effective add. calculate by adding base register value to address feild value
PC = Base + Relative Value

⑧ Relative : Register contains offset and we get effective address by adding it to PC.
PC ← PC + Relative

→ In question of RAW dependency is like.

$R_0 \leftarrow M[Loc1]$ → I_1

$R_1 \leftarrow R_0 + R_2$ → I_2

then, I_2 will do execution phase after complet of m/m access phase of R_0 (m/m acess occur after exec

→ Execution Add R_1, R_2 → 3 Cycles → Direct

Add $R_1, @R_2$ → 4 Cycle → Indirect

→ Relative add. mode used to reduce instruction size, all indexing of array, easy relocation but doesn't enable fast address calculation than absoulute add

→ A CPO has 64 bit (3 B) instruction and starting addres in 300 then all consecutive addres are 303, 306, 30

→ LOAD Rd, 1000 (R_s) means $Rd \leftarrow [1000+1]$ (m/m location)

→ In even integer binary representation LSB should be '0'

→ In operand forwarding if case is —

$R_1 \leftarrow R_1 + R_2$]

$R_2 \leftarrow R_1 + R_3$] ~~RAR~~

then EX of I_2 will occur after EX of I_1

→ In delayed branching irrespective of condition, first instruction always executed

→ If ID and OF given as separate stages then in case of WAW dependency do perform OF stage after completion of first instruction.

→ Buffer between two stage come under consideration during pipeline otherwise we use only stage delay in non-pipeline mode

→ Interrup driven I/O is faster than programmed I/O
→ CALL and JUMP Instructions just checks the value of flag doesn't change its

→ In daisy chaining we gives non-uniform priority to all devices.

→ formatting floppy means writing identification information on all tracks & sectors.

→ Swap space is used to save process data.
→ performance gain = $\frac{\text{E programed I/O}}{\text{E Interrup I/O}}$

→ Burst block transfer (DMA) and vectored Interrupt needs high BIA.

⇒ for a magnetic disk with concentric circular tracks, the latency is not linearly proportional to seek distance due to higher capacity of tracks on the periphery of the platters.

→ Based and relative addressing modes are suitable for program relocation at run time not absolute & indirect

→ In absolute addressing mode address is inside the instruction

→ Normally Add execution takes 3 cycles if all operands follows direct addressing mode but if any one follows indirect then it takes 4 cycles and if both follows indirect.

→ Notation ADD Rd, 1000 : Rd ← Rd + 1000

ADD Rd, 1000(Rs) : Rd ← Rd + M[1000 + Rs]

→ In operand forwarding we provide o/p to ID phase along with Ex phase in register or in case of m/m just after m/m phase.

→ Speed Up = $\frac{\text{Pipe depth}}{(1 + \text{No. of stalls})} = \frac{\text{no. of stages}}{(1 + \text{No of stall Instr})}$

→ Time taken by Ins^n = (CPI) * (time taken by 1 Instr

→ Any implied register explicitely mentioned as an operand in operation is not possible.

→ Throughput : No. of Instructions per second.

→ In cache avg. m/m access time if 'h' is given : $t_{avg} = h * t_m$

→ If in question given that ignore the search time in cache the used

⇒ $H_1 t_1 + (1 - H_1) H_2 t_2 + \dots$

→ In direct mapping we use decoder MUX and in k-way set associative we use decoder MUX as well as (k x 1) way selection MUX.

→ In accessing array using col^n major order we generally posses all the element as miss.

→ In asynchronous serial data transfer we keep 0 is the begining and 2 - 1's at the end to provide synchronization.