*CPU (Central Processing Unit):-

→ brain of Computer

- CPU cycle – Time CPU takes to perform μ-opr^n

$$\text{CPU cycle Time} = \frac{1}{\text{Clock Rate}}$$

$$CR = 1.6\ GHz$$
$$= 1.6 \times 10^9\ \mu\text{-opr/sec}$$

⇒ CPI (Cycle per Instruction) –

Execution time –

- for 1 Instr^n = CPI_{avg} x cycle Time

$$ET = \frac{CPI_{avg}}{\text{Clock Rate}}$$

- for n Instr^n = n * CPI_{avg} * cycle Time

$$ET = \frac{n \times CPI_{avg}}{\text{Clock Rate}}$$

- Calculation of CPI_{avg} –

$$CPI_{avg} = \frac{\text{Total No. of Cycles}}{\text{Total No. of Instr}^n}$$

$$= \frac{\sum n_i \times CPI_i}{\sum n_i}$$

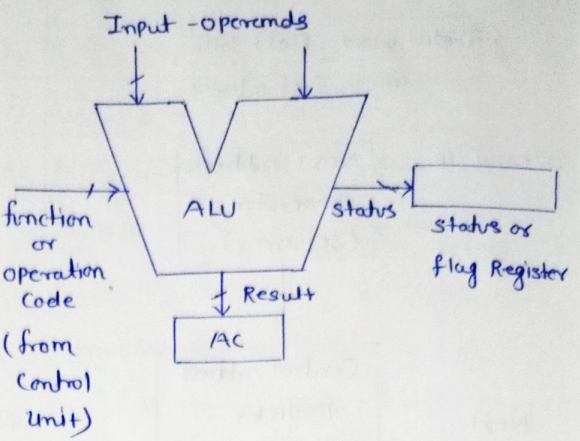
⇒ MIPS (Million Instruction per sec.)

↳ used to give CPU performance

$$MIPS = \frac{\text{No. of Instr}^n}{\text{Total time} \times 10^6}$$

$$MIPS = \frac{\text{Clock Rate}}{CPI_{avg} \times 10^6}$$

* ALU (Arithmetic Logic unit):-

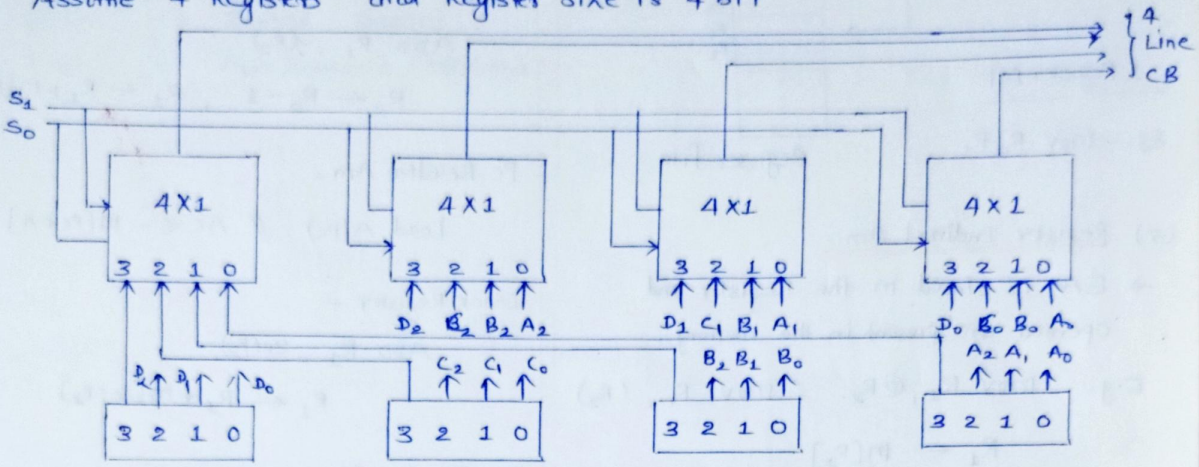


* How Data Transfer from R_A to R_B

$R_A \longrightarrow R_B$: R_{Aout} R_{Bin}

- Number of multiplexer = Size of Register (# bits in Register)
- Size of mux = Number of Registers

Assume 4 Registers and Register size is 4 bit

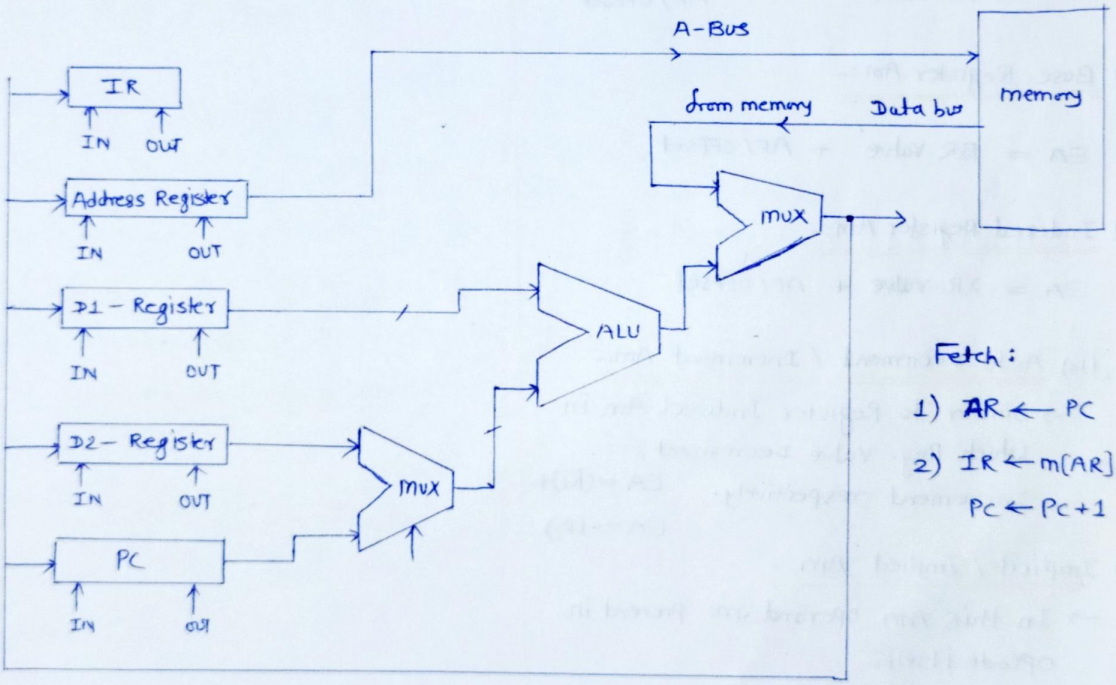


S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

for (0,0) ⇒ R_A Enable and data Come at Common bus

for (0,1) ⇒ R_B Enable and data transfer from Common bus to Register B

* Data Path :-



Fetch :

1) AR ← PC
2) IR ← m[AR]
   PC ← PC + 1

* Control Unit :-

→ Generates Control signals and sends to various Components of Computer

→ Name of Control signal → Control variable

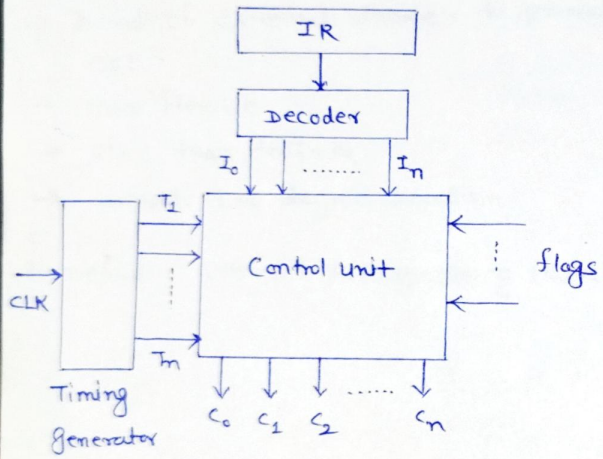
→ Control word – Collection of signals generated by CU.

1) Hard-wired Control unit :-

→ Control Logic is implemented with Gates, flip-flops and other digital Electronics circuit.

⊕ → Can be optimized to produce a faster mode of operation.

⊖ → Re-arranging the wires among various Components is difficult.



Instr / Time	I_1	I_2	I_3	I_4	I_5
T_1	C_0, C_6, C_9	:	:	:	:
T_2	C_1, C_6, C_9, C_{12}	:	:	:	:
T_3	..	:	:	:	:
T_4	:	:	:	:	:

→ Based on the table Boolean formula (eqn) is created.

Q

	T_1	T_2	T_3	T_4	T_5
I_1	S_1, S_2, S_5	S_2, S_4, S_6	S_1, S_1	S_{10}	S_3, S_8
I_2	S_1, S_2, S_5	S_8, S_9, S_{10}	S_5, S_6, S_7	S_6	S_{10}
I_3	S_1, S_2, S_5	S_7, S_8, S_{10}	S_2, S_6, S_9	S_{10}	S_1, S_3
I_4	S_1, S_2, S_5	S_2, S_6, S_7	S_5, S_{10}	S_6, S_9	S_{10}

10 Control Signals – ($S_1 - S_{10}$)

find boolean exp for S_5 and S_{10}

Ans –

$$S_5 = I_1T_1 + I_2T_1 + I_3T_1 + I_4T_1 + I_2T_3 + I_4T_3$$

$$S_5 = T_1(I_1 + I_2 + I_3 + I_4) + T_3(I_2 + I_4)$$

$$S_5 = T_1 + T_3(I_2 + I_4)$$

$$S_{10} = I_2T_2 + I_3T_2 + I_4T_3 + I_3T_4 + I_2T_5 + I_4T_5$$

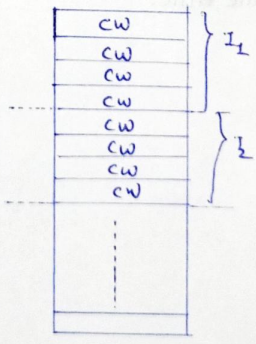
$$= (I_2 + I_3)T_2 + I_4T_3 + I_3T_4 + (I_2 + I_4)T_5$$

2) Micro-programmed Control unit :-

→ Control Logic is implemented with micro-programs.

⊕ → updating the Control Logic is Easy

⊖ → Slower than hardwired Control unit



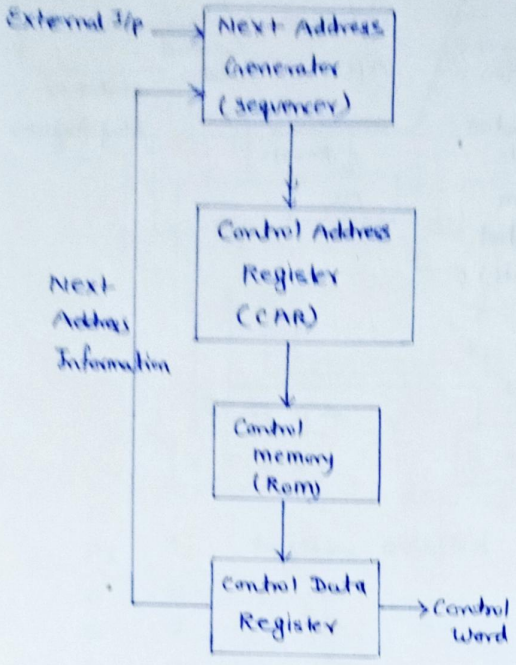
Control memory

Control Word sequencing :-

On an address in control memory

Control Word	Next Info

← Micro Instruction →



Standard micro-Instruction format-

Signals	mux select	Next Add.

(Control word)

Q. Consider a CPU

1) 120 Control signal bits
2) micro-programmed CU micro-Instr^n -

Signals	mux select	Next add

3) 16 input mux
4) 16 distinct Instr^n supported by CPU

64 μ-operation per Instr^n.

Sol Total No. of micro Instr^n = 16×64

$= 2^{10}$

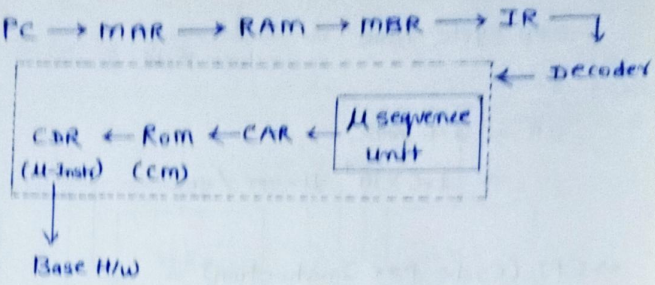
Address bit - 10

Signals	mux select	Next n
120	4	10

Control memory size = (134×1024)

= 134 K bits

μ-Programmed CU design-



μ - Instruction format

Branch (BC) Condition	Flag	Control Field	Control memory Address

- Branch (BC) Condition: one of a system supported Branch Cond^n → $[\log_2 \#BC's]$
  - E.g. - 4 BC
  - $\log_2(4) = 2$ bit
- Flag: $\log_2(\#flags)$
- Control memory Address: $\log_2(CM\ size)$

CM - Control memory

Control Signal

Decoded Format (Horizontal μ program)

→ 1 bit / CS

Eg - 3 bits → 3 CS

or

8 CS → 8 bit

Encoded Format (vertical μ program)

→ n bit — 2^n C.S.

E.g. - 3 bit ⇒ $2^3 = 8$ CS

• Horizontal μ-programming :-

→ Control signal express in Decoded format.

1 bit ⇒ 1 CS

→ It support Longer Control word.

200 CS ⇒ 200 bit

→ No Needed of External decoder
→ flexible Compared to Hardwired.
→ Fast than vertical
→ Supports high degree Parallism.

• Vertical - μ - Programming :- (None/one)

→ Control signal represented in Encoded format.

n bit ⇒ 2^n CS

→ It supports Shorter Control word.

200 CS ⇒ 8 bit

→ Need of External decoder to generate CS.
→ more flexible
→ slow than Horizonta
→ supports Low degree parallism

Default - Vertical μ programming (CISC)

* RISC :-

→ (Reduced Instruction set Computer)
→ Less Number of Instructions
→ Fixed Length Instruction
→ Simple Instructions
→ Limited Addressing modes
→ Easy to implement using hardwired Control unit.
→ one cycle per Instruction
→ Register to Register Arithmetic operation only
→ more Number of Registers

* CISC :-

→ Complex Instruction set Computer
→ more Number of Instructions
→ Variable Length Instructions
→ Complex Instructions
→ more & Complex Addressing modes
→ Difficult to Implement using hardwired Control unit.
→ multiple cycles per Instn.
→ Register to memory or memory to Register Arithmetic operations possible
→ Less Number of Registers

RISC is preferrable for pipeline CPU.

*

* Micro-operations :-

- We have small-small operation in fetch cycle & Execute cycle these small operation is called 'micro-operations'.
- These are Elementary opr^n in the Hardware.
- micro-operation Consume 1 cycle to Complete the execution.
- Control signals are required to execute the micro operation.

Machine Instr^n : MOV R_0, R_1

RTL (Register Transfer Lang): $R_0 \leftarrow R_1$

micro-operation :

T_1 : R_1 out R_0 in

1) Fetch Cycle:-

T_1 : PC $\rightarrow$ MAR PC_{out} MAR_{in}

T_2 : MBR $\leftarrow$ M[MAR] MAR_{out} MBR_{in}] SB

PC $\leftarrow$ PC+1 PC_{out} PC_{in}] LB

T_3 : IR $\leftarrow$ MBR MBR_{out} IR_{in}

This is called μ-programme.

→ Rules for μ-operation Grouping:-

- Proper sequence must be followed
- Conflicts must be avoided
- must not read and write same register at same time.

* Interrupts:-

→ unusual events that disrupts the normal execution of the program.

→ When CPU receive Interrupt -

1) It Completes execution of current Instruction
2) Saves the ~~states~~ status (PC, PSW etc) of current process onto the stack.
3) Branches to service the interrupt.
4) Resumes the previous process by taking out the values from stack

- **ISR (Interrupt service Routine):-**
  A Routine or function, execution of which services the Interrupt
- **Vector Address:**
  Address or Reference of ISR
- **IVT (Interrupt vector Table):-**
  → This is a Part of memory, where the ISR are present.
  → Each ISR must end with IRET (Interrupt Return Statement)

* Hardware Interrupt:- (Raised by H/w)

→ External Interrupt (Asynchronous)
- Comes from I/o devices or Timing devices
- from Circuit, Power Supply etc.
- Exa- I/o Interrupt
  Power failure
  Hardware Error
  Timeout

→ Internal Interrupt (synchronous)
- Arises from Illegal or Erroneous use of Instruction or data.
- Also known as Traps.
- Exa- Illegal opCode
  Register overflow
  Page fault

* Software Interrupt:- (Raised by S/w)

↳ These are special call Instruction that behave like an Interrupt rather than Subroutine Calls.

Eg. Supervisor Call Instrⁿ (SVC)

* Maskable Interrupt:-

→ The H/w Interrupt which Can be delayed ~~##~~ When much high priority Interrupt has occured at the Same time.

* Non-maskable Interrupt:-

→ The H/w Interrupt which Can not be delayed.

* Vectored Interrupt:-

→ Device sends Interrupt and vector.

* Non-vectored Interrupt:-

→ Device sends only Interrupt
→ CPU executes a default service Routine.
→ CPU obtain location of actual ISR.

* Time Required in Interrupt Io:-

= Interrupt overhead time + service time

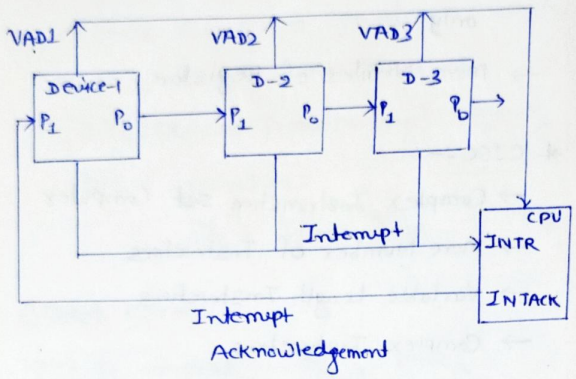
* Simultaneous Interrupt:-

If 2 devices generate the Interrupt Simultaneously, then highest priority deviced Interrupt services first

* Priority Based Interrupt Handling:-

- Software Soln
  - Run a Program & Resolve the priority
- Hardware Soln
  - Serial (daisy chaining)
  - Parallel

* Daisy chaining:-



VAD – Vector Address

INTR – Interrupt Request

• 8085 Interrupt structure:-

		Priority
(H/w Interrupt)	RST 7.5	3
	RST 6.5	4
	RST 5.5	5
	RST 4.5	2
(Non-vectored)	Trap	1
	INTR	6

(S/w Interrupt) → RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6, RST 7

(All vectored)

* Flynn's classification of Computer:-

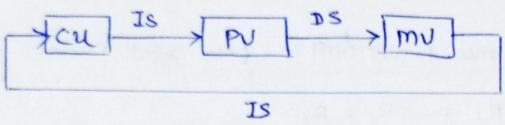
1) SISD (Single Instruction Single Data Stream)

2) SIMD (Single Instruction multiple Data Stream)
   ↳ pipeline

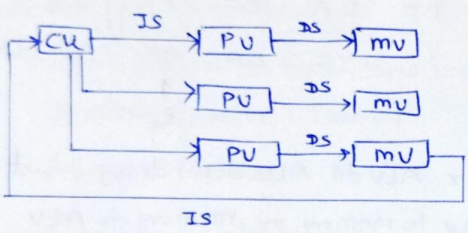
3) MISD (multiple Instruction Single Data stream)
   ↳ Theoretical

4) MIMD (multiple Instruction multiple Data Stream)
   ↳ multiple pipeline)

SISD-

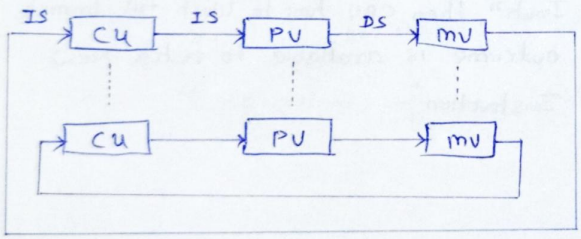


SIMD-



- DS - Data stream
- PU - Processing unit

MIMD-



* Pipelining :-

→ This is useful, when Same processing is applied over multiple Inputs.

→ Technique to decompose a sequential Process into sub-operations, Sub-operations are performed in segments

→ Task: one opr^n performed in all segments.

→ All segments performed respective Sub-operations parallely.

Exa- $A_i * B_i + C_i$, for $i = 1,2,3,4$

Sub-operations:

$R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$

$R_3 \leftarrow R_1 * R_2$, $R_4 \leftarrow C_i$

$R_5 \leftarrow R_3 + R_4$

Clock Pulse	segment1		seg-2		seg-3
	R_1	R_2	R_3	R_4	R_5
C_1	A_1	B_1	~~$A_1 * B_1$~~	C_1	
C_2	A_2	B_2	$A_1 * B_1$	C_1	
C_3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
C_4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
C_5			$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
C_6					$A_4 * B_4 + C_4$

Non-Pipeline is a sequential process

	Non pipeline	Pipeline
$n = 5$	15 cycles	7 cycles
$n = 6$	18 cycles	8 cycles

* Pipeline cycle time:-

Min time in which all segments can perform their respective sub-operations.

Total Execution time = (No. of cycles) x Cycle time

- Consider K segment pipeline with clock cycle time t_p to perform n tasks

→ Time required to perform 1st task = $K * t_p$

→ Time required to perform remaining tasks - $(n-1) * t_p$

Total time = $(K+n-1) * t_p$

- Consider a non-pipeline t_n is time to perform one task -

→ Time Required to perform n tasks - $= n * t_n$

performance of pipeline is given by speed up (S)

$$S = \frac{\text{Non-Pipeline time}}{\text{pipeline time}} = \frac{ET_{NP}}{ET_P}$$

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

When $n >> K$

$$S = \frac{n * t_n}{n * t_p}$$

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

→ When time required to perform 1 task is Equal in Pipeline or Non-Pipeline both -

$t_n = K * t_p$

$S = K$

* Synchronous Pipeline :-

* Types of Pipelining :-

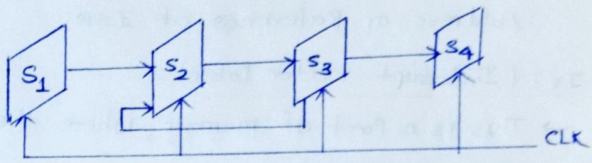
1) Linear Pipeline :- A Linear Pipeline Consists of a sequence of processing stages where Each stage performs a specific task.

(Uni direction)

2) Non-Linear pipeline :-

→ It may have multiple paths

→ It Consists Reservation table.



3) Synchronous Pipeline :-

→ All stages operate under a common clock signal and data is transffered at regular clock Intervals. (It Consists Buffers to store data)

4) Asynchronous pipeline :-

→ Each stage operate Independently without a global CLK.

→ Data is passed using handshake signal.

5) Uniform delay pipeline :-

→ All pipeline stages have equal Processing delay / time.

6) Non uniform delay Pipeline :-

→ stages have different processing delays.

Case I – When uniform delay : t_p = Stage delay

Case II – When uniform delay pipeline with stage buffer delay : t_p = Stage delay + buffer delay

Case III – When uniform delay with Non-uniform buffer:

$$t_p = \text{Stage delay} + \max(\text{buffer delay})$$

Case IV – Uniform Stage delay pipeline with skew time overhead –

t_p = Stage delay + skew time overhead

Case-V – Non uniform pipeline with buffer delay

$t_p = \max(\text{Stage delay}) + \text{buffer delay}$

* Latency and Throughput:-

↳ After how much time, new input is given to system : Latency

Pipeline – Latency = t_p

Non pipeline – Latency = t_n

↳ No. of ~~opr~~ tasks per unit time : Throughput

Pipeline – Throughput = $\frac{n}{(K+n-1) * t_p}$

if $n >> K$

Throughput = $\frac{1}{t_p}$

* Instruction Pipeline:-

Assume 5 segment Instruction Pipeline :-

IF – Instruction fetch

DA – Decode and Address Calculation

OF – Operand Fetch

EX – Execution

WB – Write Back

clock cycle

	1	2	3	4	5	6	7	8	9	10
I_1	IF	DA	OF	EX	WB					
I_2		IF	DA	OF	EX	WB				
I_3			IF	DA	OF	EX	WB			
I_4				IF	DA	OF	EX	WB		
I_5					IF	DA	OF	EX	WB	

If There is decoded a branch Instⁿ then Next Instrⁿ will remove and pipeline Will Wait for result of branch Instⁿ Condition is available (Result will be available after Execution phase)

Remove Instrⁿ are also fetch again.

→ In this, Pipeline takes some Extra cycles, that are called stall cycles.

Total No. of cycles = (K+n-1) + Total stall cycles

If branch Condition is Evaluated in i^{th} place –

then No. of stall cycles = $i - 1$

* Pipeline Hazard :-

→ Situations that prevent the next Instrⁿ from being executing during its designated clock cycle.

→ It causes problem in smooth execution of Instruction in pipeline.

Types :-

1) Structural Hazard / Resource Conflict
2) Data Hazard / Data Dependency
3) Control Hazard / Branch difficulty

1) Structural Hazard :-

→ 2 different segments try to use same resource at same time.

Exa- Mul – 2 cycle
ADD – 1 cycle

```
MUL - IF  DA  OF  EX  EX  WB
ADD -     IF  DA  OF  -   EX  WB
```

Solution – 1) Increase stall cycle
2) Increase No. of resource

2) Data Hazard :-

→ Result of an Instruction is used as input in next.

i : $R_1 \leftarrow R_2 + R_3$
i+1 : $R_5 \leftarrow R_1 + R_4$

```
i   -  IF  DA  OF  EX  WB
i+1 -      IF  DA  OF
```

→ A General Instrⁿ pipeline Cannot detect data dependency.

→ Solution of data dependency is provided by Compiler.

[Solution – delayed Load
↳ This is a Software solution

H/W Solution (When pipeline detect and solve)

H/w Interlock ⇓ having stalls in case of data dependency

Operand forwarding or Bypassing

```
i  : IF  DA  OF  EX  WB
i+1:     IF  DA  -   -   OF  EX  WB
```

* Operand Forwarding :- (No stall cycles)

i : $R_1 \leftarrow R_2 + R_3$
i+1 : $R_5 \leftarrow R_1 + R_4$

```
i  : IF  DA  OF  EX  R  WB
                     (R1)
i+1:     IF  DA  OF  EX  WB
                 ↑
                 R4
```

→ Use for ALU to ALU dependency, but for ALU to memory or memory to ALU stalls are required.

3) Control Hazard :-

→ It occurs due to branch Instrⁿ

→ When CPU decodes an Instrⁿ on branch Instrⁿ, then CPU has to wait till branch outcome is available to fetch New Instruction.

⇒ A General Pipeline Can not detect the branch problem.

Then Compiler provides solution ⇒ Delayed Branch (S/W solution)

i : branch
j : ....

i : branch → IF DA OF EX WB
No IF DA OF EX WB
No IF DA OF EX
No IF DA OF
j : ..... IF DA

No operation or Independent Instrn

- H/W solution – "branch prediction"

→ It predicts Wheather the current branch Instrn will take jump or not.
- If prediction is correct ⇒ No problem
- If prediction is wrong ⇒ Roll back to branch Instrn & take correct path.

* Classification of Data Hazard :-

→ 1) RAW (Read After write)
2) WAW (write After write)
3) WAR (write After Read)

Let ~~There~~ There are Two instrn i, j and and i should execute before j.

1) RAW :- (True dependency)

If j reads a source before it is written by i, hence j incorrectly gets old value.

i : $R_1 \leftarrow R_2 + R_3$
j : $R_5 \leftarrow R_1 + R_4$

⇒ Soln –
↳ S/W :- delayed Load
→ H/W :- H/W Interlock, Operand forwarding

2) WAW :- (False dependency)

j writes in a destination before i writes it.

i : $R_1 \leftarrow R_2 + R_3$
j : $R_1 \leftarrow R_4 * R_5$

3) WAR :- (Anti-dependency, false depen)

j writes in a destination before i reads it, hence i incorrectly reads new value.

Soln – Register Renaming for both WAW and WAR.

- In Ideal Case–

Pipeline Efficiency (η) = $\frac{t_n}{t_p}$

bcz of Hazards achieved speed up – S

$\eta = \frac{S}{S'} \times 1$

* Performance Evaluation of Pipeline with Stalls :-

$$S = \frac{ET_{NP}}{ET_P}$$

- Avg. ET = CPI_{avg} * Clk cycle Time
- CPI_{avg} = 1 + Avg No. of stalls / Instr

$$S = \frac{K}{1 + \text{Nu. of stalls / Instr}}$$

* No. of stalls from branch =

Branch penalty * Branch freq.

* Branch penalty = I - 1

⇓

The stage at which target is available.

* RISC Pipeline :-

5-stages :-

1) IF - Instruction fetch + PC

2) ID - Instruction decode + Registers Read

3) + Target Calculation for branch Instrn
   + Condition check and pc updation

3) Ex - ALU operation

4) mem - memory Access ⇒ Load / Store

5) WB - write back Result in Register

Exa -

$C = A + B$

C, A, B are memory operands

I_1 : Load R_1 , [A] $R_1 \leftarrow [A]$

I_2 : Load R_2 , [B] $R_2 \leftarrow [B]$

I_3 : ADD R_1 , R_2 $R_1 \leftarrow R_1 + R_2$

I_4 : Store [C], R_1 $[C] \leftarrow R_1$

IF ~ ID EX ~ mem ~ WB

I_1 : IF ID mem WB

I_2 : IF ID mem WB

I_3 : IF ID Ex WB

I_4 : IF ID mem