PIPELINE

Definition - It is a technique of decomposing a sequential process into suboperation with each subprocess being executed in a dedicated segment that operates simultaneously or concurrently with all other segments.

→ In pipeline execution technique multiple instructions are overlapped in execution only if the resources, that are used in the same clock cycle for different stages are different then only overlapping is possible.

→ Advantage of pipeline is, it make the performance of CPU high. This is done by increasing throughput of the system by using pipeline. Throughput means no. of task completed per unit time so that the program can run faster even if no single instruction runs faster.

Principle of pipeline →

Pipeline is based on the principle that multiple no. of independent task are operating simultaneously or concurrently.

Speed of pipeline depends on no. of stages or segments present in pipeline.

Speed up = No. of pipeline stages or segments

General structure & consideration of pipeline →

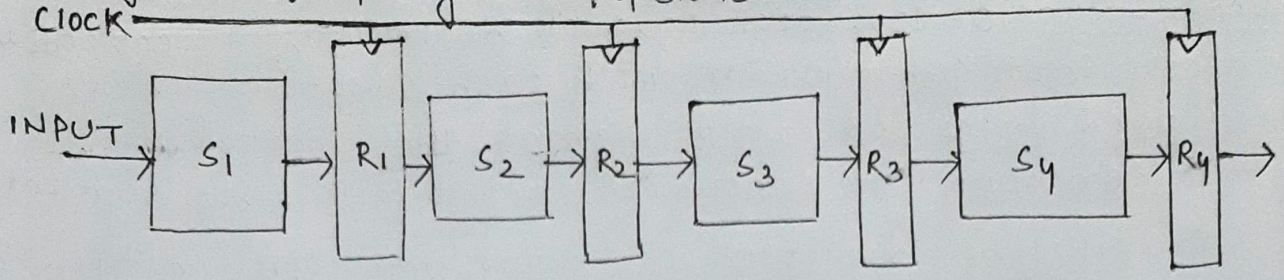
→ Any operation is divided into suboperation & each suboperation is performed in a segment (Si) which consist of combinational circuits.

→ The segments (Si) are separated by registers (Ri) where each registers store intermediate result of segments.

→ Information flows between adjacent segments under the control of common clock applied to all registers simultaneously.

→ A task is defined as total operations performed going through all segments.

Diagram of 4-segment pipeline-



→ The results moves in the segments one step down to the next segment. When no more data or inputs are available then the clock pulse go on continues till the last o/p (output) emerge out of the pipeline.

→ Behaviour of pipeline is explained by using space-time diagram where the x-axis contain clock cycle & y-axis contain segments.

→ Example -
Show the space-time diagram to process 4 task in a 3-segment pipeline.

CLK cycle →	1	2	3	4	5	6
Segments ↓ S_1	T_1	T_2	T_3	T_4		
S_2		T_1	T_2	T_3	T_4	
S_3			T_1	T_2	T_3	T_4

S_1, S_2 & S_3 - Segments
T_1, T_2, T_3 & T_4 - Tasks

Pipeline Speed up Calculation →

Let's consider 'n' is no. of task to be performed.
For conventional machine or non-pipelined machine -
t_n = Time to complete execution of single task.
T_1 = Total time to complete execution of 'n' no. of task
So, $T_1 = n \times t_n$
For pipelined machine -
t_p = Time to complete each suboperation.
T_K = Total time to complete execution of 'n' no. of task

Pipeline speed up calculation –

Let's consider 'n' no. of task to be processed.

Speed up = Total time required to process 'n' no. of task through non-pipelined system or conventional system / Total time required to process 'n' no. of task through pipeline system.

For non-pipelined system –

Let t_n = time to complete execution of a single instruction.

T_1 = time to complete execution of 'n' no. of task or instructions.

$$\boxed{T_1 = n \times t_n}$$

For pipeline system –

Let t_p = time to complete each sub-operation in segment or stage.

T_K = Total time to complete 'n' no. of task or instructions.

$$\boxed{T_K = \{K + (n-1)\}\, t_p}$$

Speed up of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio given above. i.e:

$$S_K \text{ (speed up)} = \frac{T_1}{T_K}$$

$$\boxed{S_K = \frac{n \times t_n}{\{K + (n-1)\}\, t_p}}$$

If the time taken to complete 'n' no. of task in non-pipeline system is same as the time required to complete 'n' no. of task in pipeline system then the speed up can be expressed as,

$$S_K = \frac{n \times t_n}{\{K + (n-1)\}\, t_p}$$

$$\Rightarrow S_K = \frac{n \times K t_p}{\{K + (n-1)\}\, t_p} \quad \left[\because t_n = K t_p\right]$$

$$\Rightarrow \boxed{S_K = \frac{nK}{K + (n-1)}}$$

If no. of task to be processed becomes much larger then $K+(n-1) = n$. So under this condition the speed up equation reduced to,

$$S_K = \frac{n\,t_n}{\{K+(n-1)\}+t_p}$$

$$\Rightarrow S_K = \frac{n\,t_n}{n+t_p} \quad \left[\because n \ggg ,\ K+(n-1)=n\right]$$

$$\Rightarrow \boxed{S_K = \frac{t_n}{t_p}}$$

($S_K = \frac{t_n}{t_p}$ is the actual speed up of pipeline)

If no. of task to be processed is more & also the time required to process 'n' no. of task in non-pipeline system is same as that of pipeline system then the speed up can be expressed as,

$$S_K = \frac{n\,t_n}{\{K+(n-1)\}+t_p}$$

$$\Rightarrow S_K = \frac{n \times K + t_p}{n+t_p} \quad \left[\because t_n = K+t_p \text{ \& } n \ggg ,\text{so } K+(n-1)=n\right]$$

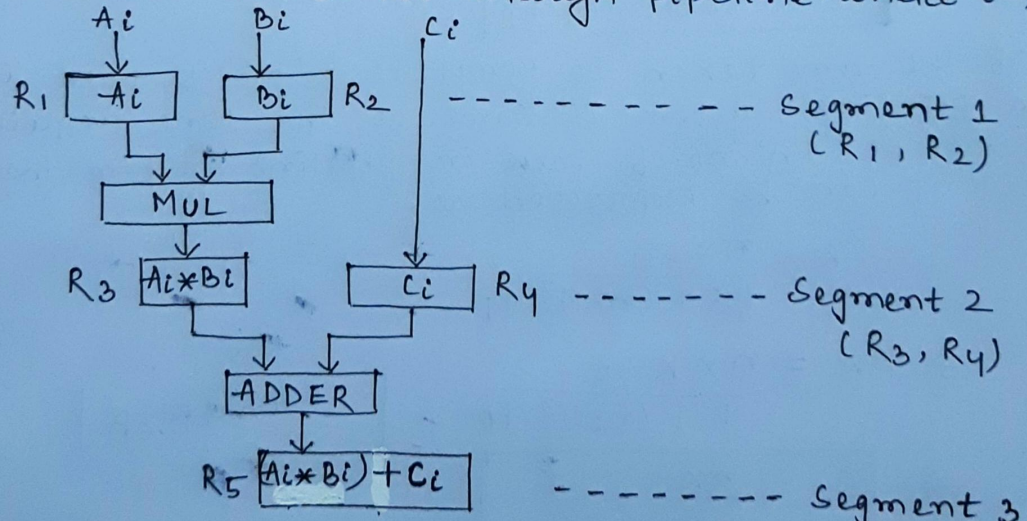
$$\Rightarrow \boxed{S_K = K}$$

This shows that the maximum speed up that a pipeline can provide is K. (It is hypothetical)

From the speed up formula it is clear that the total no. of clock cycles required to execute on process 'n' no. of task through pipeline is,

No. of clock cycle = $K+(n-1)$

e.g- Process $Ai * Bi + Ci$ through pipeline. where $i \to 1$ to 4



Segment →	S_1	S_2	S_3
Clock cycle ↓	(R_1, R_2)	(R_3, R_4)	(R_5)
t_1	A_1, B_1		
t_2	A_2, B_2	$(A_1 * B_1), C_1$	
t_3	A_3, B_3	$(A_2 * B_2), C_2$	$(A_1 * B_1) + C_1$
t_4	A_4, B_4	$(A_3 * B_3), C_3$	$(A_2 * B_2) + C_2$
t_5		$(A_4 * B_4), C_4$	$(A_3 * B_3) + C_3$
t_6			$(A_4 * B_4) + C_4$

Total no. of clock cycle required $= K + (n-1)$

$= 3 + (4-1)$

$= 6$ no. of

Q. In a 4-stage pipeline the time required to process a sub-operation within segment is 20 ns and no. of task is very large. The time required to process task in a non-pipeline system is 60 ns then calculate speed up.

Ans – $t_p = 20$ ns

$t_n = 60$ ns

$n \gg\gg$

$$S_K = \frac{t_n}{t_p} = \frac{60}{20} = 3$$

Q. In a 6-segment pipeline if the time required to complete execution of 'n' no. of task is same as that of non-pipeline system and $n \gg\gg$, calculate speed up (max).

Ans – $t_n = K t_p$ & $n \gg\gg$

So, Speed up $(S_K) = K = 6$

The maximum speed up = 6

Q. In a 4-segment pipeline time required to complete the sub-operation in segment or stage is 20 ns, no. of task to be processed is 100. (Assume that both pipeline & non-pipeline system requires same time to process total no. of task). calculate speed up.

Ans – $t_p = 20$ ns.

$n = 100$

$K = 4$

Speed up $(S_K) = \dfrac{n K t_p}{\{K + (n-1)\} t_p} = \dfrac{100 \times 4 \times 20}{\{4 + (100-1)\} 20} = \dfrac{8000}{2060} = 3.88$

$(t_n = K t_p)$

Types of pipeline →

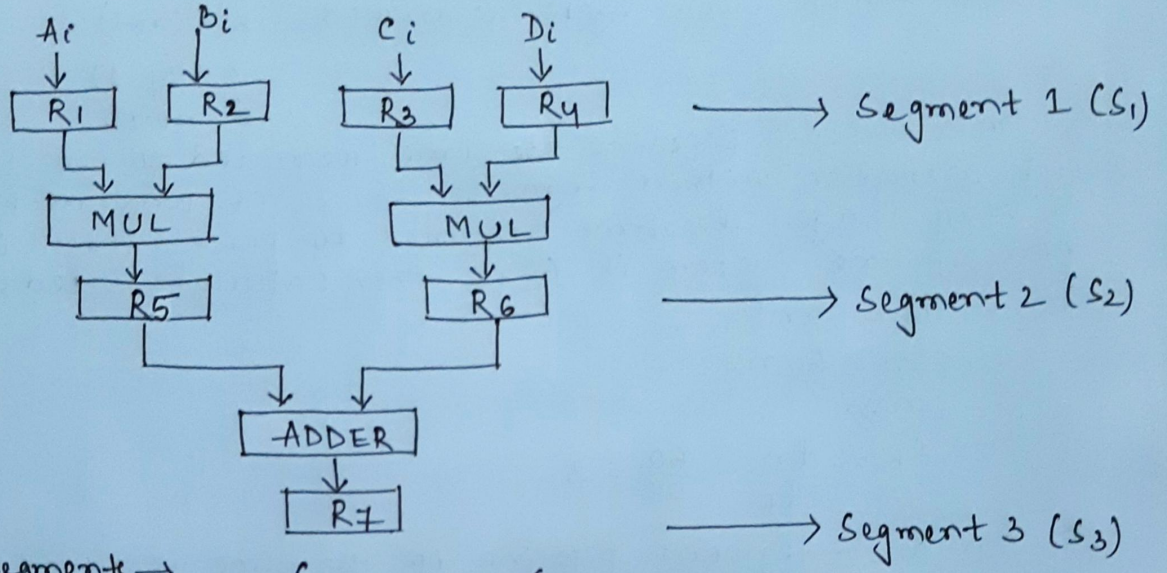
1. Arithmetic pipeline
2. Instruction pipeline
3. RISC pipeline.

1. Arithmetic Pipeline –

→ Arithmetic pipeline divides an arithmetic operation into sub-operations for execution in pipeline segments.

→ It is found in very high speed computers. It is basically used in solving floating-pt scientific operations.

→ e.g. $(Ai * Bi) + (Ci * Di)$ where $i \rightarrow 1$ to 4



Segments →	S_1				S_2		S_3
Clock cycle ↓	R_1	R_2	R_3	R_4	R_5	R_6	R_7
	Ai	Bi	Ci	Di	(Ai*Bi)	(Ci*Di)	(Ai*Bi)+(Ci*Di)
t_1	A_1	B_1	C_1	D_1			
t_2	A_2	B_2	C_2	D_2	$(A_1 * B_1)$	$(C_1 * D_1)$	
t_3	A_3	B_3	C_3	D_3	$(A_2 * B_2)$	$(C_2 * D_2)$	$(A_1 * B_1) + (C_1 * D_1)$
t_4	A_4	B_4	C_4	D_4	$(A_3 * B_3)$	$(C_3 * D_3)$	$(A_2 * B_2) + (C_2 * D_2)$
t_5					$(A_4 * B_4)$	$(C_4 * D_4)$	$(A_3 * B_3) + (C_3 * D_3)$
t_6							$(A_4 * B_4) + (C_4 * D_4)$

No. of clock cycle $= K + (n-1)$

$= 3 + (4-1)$

$= 6$

e.g. floating-pt addition & subtraction.

Let the two inputs to the floating-pt adder are,

$X = A \times 2^a$

$Y = B \times 2^b$.

where A & B = Mantissa

a & b = exponents

The sub-operations for floating-pt addition & subtractions are,

1. Compare the exponents.
2. Align the mantissa of smaller exponents.
3. ADD or SUB the mantissa.
4. Normalize the results.

Compairision is done by subtracting the exponents. Then the mantissa of smaller exponents is shifted to the right to align the mantissa. After aligning the ADD operation is to be done & after that the result is normalized to avoid overflow or underflow.

The comparator, shifter, Adder or subtractor, incrementor & decrementor in arithmetic pipeline are implemented using combinational circuit.

$X = 0.9504 \times 10^3$

$Y = 0.8200 \times 10^2$

Now, after aligning $X = 0.9504 \times 10^3$

$Y = 0.0820 \times 10^3$

After Addition result is, $Z = x + y = 1.0324 \times 10^3$

After normalizing the result is $Z = 0.10324 \times 10^4$

Q. calculate the value of tp for the above e.g. if the time delay of 5 segments are given as,
$t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns & $t_5 = 50$ ns and the interface register has a delay of $t_5 = 10$ ns. For the same calculate t_n (non-pipeline floating-pt delay) & speed up.

Ans - For non-pipeline,

$t_n = t_1 + t_2 + t_3 + t_4 + t_5 + t_r$

$= 60 + 70 + 100 + 80 + 50 + 10$

$= 370$ ns

For pipeline, t_p = maximum time delay among all segments + Register or intermediate register delay.

$= 100 + 10$

$= 110$ ns

Speed up $= \frac{t_n}{t_p} = \frac{370}{110} = 3.36$

Q. Determine no. of clock cycles that a pipeline take to Process 200 task in a 6-segment pipeline.

Ans: $K = 6, n = 200$

No. of clock cycles required $= K + (n-1) = 6 + (200-1)$

$= 6 + 199 = 205$

Q. A non-pipeline system takes 50 ns to process a task. The same task can be processed in a 6-segment pipeline with a clock cycle of 10 ns. Determine the speed up calculation for 100 task. what is the maximum speed up that can be achieved?

Ans: $t_n = 50$ ns ; $t_p = 10$ ns ; $K = 6$; $n = 100$

Speed up $(S_K) = \frac{nt_n}{\{K + (n-1)\} t_p} = \frac{100 \times 50}{\{6 + (100-1)\} \times 10} = \frac{5000}{1050}$

$= 4.76$

Maximum speed up in real or actual $= \frac{t_n}{t_p} = \frac{50}{10} = 5$

Q. Calculate the time required to process 50 task in 6-segment pipeline, where segment time delay is 20 ns.

Ans: $K = 6, n = 50, t_p = 20$ ns

T_1 = time required to process 'n' no. of task in Pipeline system.

$T_1 = \{K + (n-1)\} \times t_p$

$= \{6 + (50-1)\} \times 20$

$= 1100$ ns

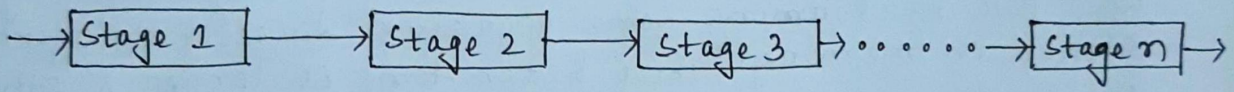
2. Instruction Pipeline -

→ Instruction pipeline is very complex because instruction execution is extremly complex & involves several operation which are executed successively. This implies a large amount of hardware works at a given time.

→ In instruction pipeline technique multiple no. of instructi-ons execution are overlapped. This is solved without using any additional hardware but only by letting different parts of the hardware work for different instructions at the same time.

→ The instruction pipeline organization of a processor is similar to an assembly line i.e: the work to be done in an instruction is broken down into smaller steps or pieces, each of which takes a fraction of time needed to complete the entire instruction.

Each of the stage or steps is a pipe stage or pipe segments. Pipe stages or segments are connected to form a pipe.



→ The time required for moving an instruction from one stage to the next is called a machine cycle or one clock cycle. The execution of an instruction takes several clock cycles or machine cycles as it passes through pipeline.

→ Different stages of instruction pipeline are,

(a) Fetch instruction (FI) - During this stage read the next expected instruction into a buffer.

(b) Decode instruction (DI) - During this stage decode the opcode or operation code & operand specifier.

(c) Calculate operand (CO) - During this stage the address calculation or effective address of source operand is calculated.

(d) Fetch operand (FO) - During this stage each operand or data from memory is fetched. If the operand is present in processor register then it doesnot require FO stages.

(e) Execution instruction (EI) - During this stage the specified operation is performed & write or store the result in any processor register if any specified.

(6) Write operand (WO) – During this stage the result is written into memory.

Different stages of 6-segment pipeline are,
FI, DI, CO, FO, EI, WO

Different stages of 4-segment pipeline are,

FI — Fetch next instruction from memory.
DI, CO → DA — Decode instruction & calculate effective address of operand.
FO — Fetch operand by using effective address
EI, WO → EX — Execute instruction & write the result in memory.

Different stages of 2-segment instruction pipeline are,
FI - Fetch instruction.
EI - Execute instruction.

→ The completion time will be longer for fetch stage. So, execution will take long time for reading, storing & also for performing some operation directly on operand stored in memory. Thus fetch time is longer & other stages may have to wait for sometime before it can empty its buffer. This is to be done on waiting time is added to other stages than memory related stages because of making time delay & of each stages equal.

i.e: $\overset{\text{time}}{\langle}$Delay of FI stages = $\overset{\text{time}}{\langle}$Delay of DI stage = time delay of CO stage = time delay of FO stage = time delay of EI stage = time delay of WO stage.

$$t_p = \underset{\text{time delay}}{\text{Maximum}}(FI, DI, CO, FO, EI, WO) + t_r$$

where t_r = register or intermediate register delay.

→ Segments that require memory access are,
FI, FO, WO.

→ Not all instructions goes through all the 6-stages of pipeline. e.g – LOAD instruction donot need WO stages. But to simplify instruction pipeline it is

assumed that each instruction requires all the 6-segment or 6-stages.

Q. Process 4 instruction in 2-segment pipeline & calculate no.of clock cycle.

clock cycle →	1	2	3	4	5
instruction i	FI	EI			
i+1		FI	EI		
i+2			FI	EI	
i+3				FI	EI

Total no.of clock cycle = $k + (n-1)$
$= 2 + (4-1)$
$= 2 + 3$
$= 5$

or

instruction →	i	i+1	i+2	i+3
clock cycle				
t_1	FI			
t_2	EI	FI		
t_3		EI	FI	
t_4			EI	FI
t_5				EI

or

clock cycle →	t_1	t_2	t_3	t_4	t_5
Segments ↓					
FI	i	i+1	i+2	i+3	
EI		i	i+1	i+2	i+3

Q. Process 4 instructions in 4-segment instruction pipeline.

clock cycle →	1	2	3	4	5	6	7
instruction i	FI	DA	FO	EX			
i+1		FI	DA	FO	EX		
i+2			FI	DA	FO	EX	
i+3				FI	DA	FO	EX

NO.OF CLOCK cycle = $k + (n-1)$
$= 4 + (4-1)$
$= 7$

Q. Process 4 instructions in 6-segment pipeline.

Clock cycle	1	2	3	4	5	6	7	8	9
Instruction i	FI	DI	CO	FO	EI	WO			
i+1		FI	DI	CO	FO	EI	WO		
i+2			FI	DI	CO	FO	EI	WO	
i+3				FI	DI	CO	FO	EI	WO

Total clock cycle = $K + (n-1) = 6 + (4-1) = 9$

→ Factors that limit the performance enhancement in instruction pipeline are,

(a) Some waiting time in various stages.

(b) Conditional branch instruction.

1) On conditional branch until the instruction is executed there is no way of knowing which instruction will come next. Hence in this case the pipeline simply loads the next instruction in sequence & proceeds. At the end of execution stage (EI), it is cleared that whether branch is taken or not.

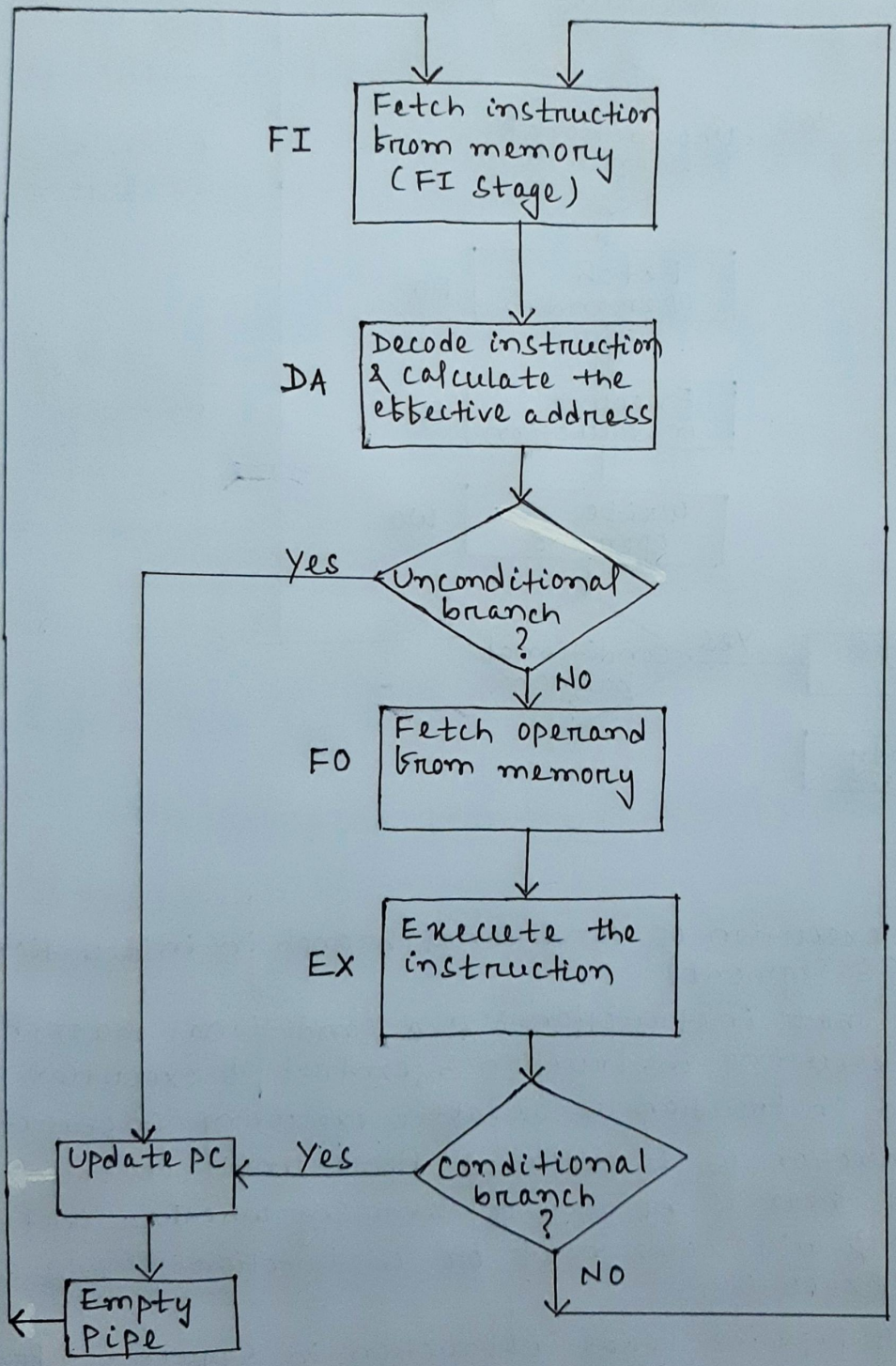
→ If branch is taken then at the end of execution or EI stage, the pipeline must be cleared or flushed all those instructions that are fetched after a branch instruction. This results in performance penalty which occurs because we couldnot anticipate the branch is taken or not correctly.

→ If branch is not taken then normal processing of pipeline goes ON.

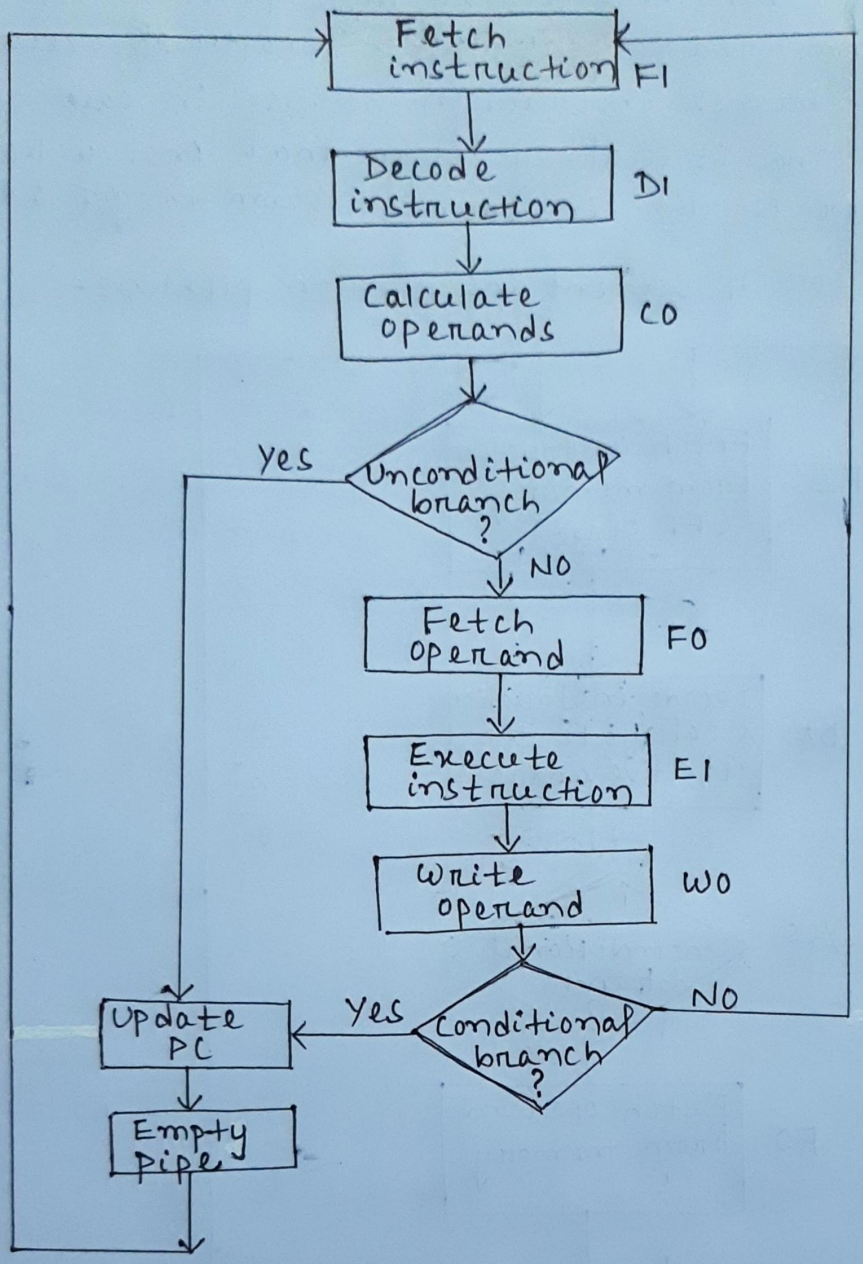
2) After DI stage it is cleared that whether there is a conditional branch or unconditional branch. If there is unconditional branch then the branch instruction address is retrieved in FO stage & before that if the pipeline loads any instruction & proceeded them then they are cleared or flused that results in performance penalty. But after DI stage if it is cleared that it is a conditional branch then step 1 is carried out.

3) In case of conditional branch if the anticipation is not correct then there will be more performance Penalty as compared to performance penalty of executing unconditional branch instruction because in case of unconditional one it is always clear that instruction executed next is the instruction from target address.

4) Flowchart for 4-Segment instruction pipeline-



5) Flowchart for 6-Segment instruction pipeline -



6) e.g. of execution of conditional branch in instruction pipeline. (6-Segment)

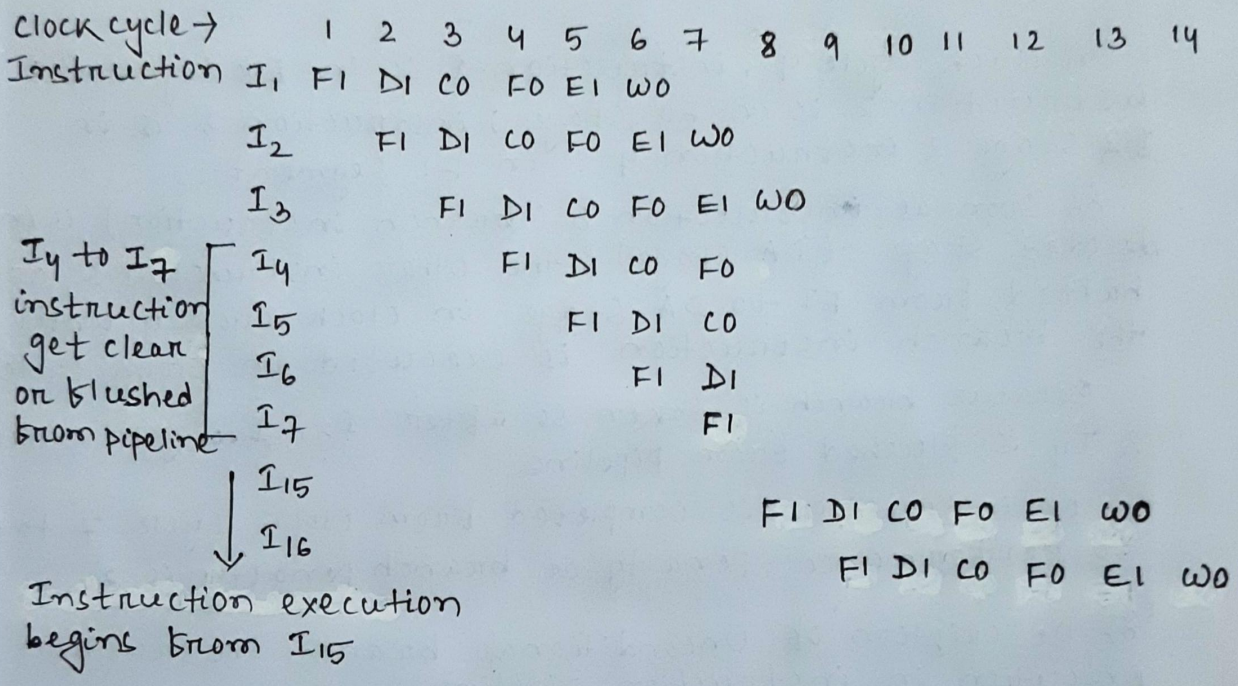
Assume that instruction 3 is a conditional branch & after executing instruction 3, control of execution shifted to instruction 15 or after executing instruction 3 next instruction 15 is executed. Hence until instruction 3 is executed there is no way of knowing whether ~~instruct~~ instruction 4 will come next or instruction 15 will come next for execution.

Initially pipeline loads instruction in sequence & proceeds

Assume that branch is taken for this example but the branch is taken is determined after E1 stage of instruction 3 at clock cycle 7. So, at clock cycle 8 it is clear that branch is taken for I_3 (instruction 3). Therefore at clock cycle 8 flush or clear the pipeline by removing I_4 to I_7 from pipeline

At clock cycle 8, instruction 15 is entered into the pipeline into the FI stage.

No instruction get completed during clock cycle 9 to 12. Hence performance penalty is 4.



Performance penalty or branch penalty = 4

7) Description of 4-segment instruction pipeline to handle conditional branch.

Assume that instruction 3 is a branch instruction & branch is taken here. Before branch instruction, all the instructions like I_1, I_2 are executed normally but the instructions following I_3 are affected.

If branch is taken then after I_3 the instruction from target address are executed.

But if branch is not taken then after I_3, instruction I_4 get executed next.

clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
Instruction I_1	FI	DA	FO	EX								
I_2		FI	DA	FO	EX							
I_3			FI	DA	FO	EX						
Flushed from the Pipeline ← ~~I_4~~ I_{10}				~~FI~~	—	—	FI	DA	FO	EX		
I_{11}								FI	DA	FO	EX	
I_{12}									FI	DA	FO	EX

Performance penalty or branch penalty = 3

On clock cycle 4, instruction 1 is in EX segment; instruction 2 is in FO stage; instruction 3 is in DA stage & instruction 4 is in FI segment.

As soon as instruction 3 (branch instruction) is in decoded stage (DA stage), the other instructions are halted from FI to DA stage in clock cycle 4 until the branch instruction is executed in clock cycle 6.

Because branch is taken so after I_3, I_{10} get executed & I_4 is flushed from pipeline

No instruction is completed from clock cycle 7 to 9, so performance penalty or branch penalty is 3.

8) Description of unconditional branch instruction execution in instruction pipeline →

Consider that- instruction no. 3 or I_3 is an unconditional branch instruction in a program & when instruction no. 3 or I_3 get executed then the control of execution moves to or jumps to instruction no. 15 or I_{15} unconditionally.

CIK cyc →	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction I_1	FI	DI	CO	FO	EI	WO								
I_2		FI	DI	CO	FO	EI	WO							
I_3			FI	DI	CO	FO	EI	WO						
I_{15} ~~I_4~~				~~FI~~	—	—	FI	DI	CO	FO	EI	WO		
I_{16}								FI	DI	CO	FO	EI	WO	
I_{17}									FI	DI	CO	FO	EI	WO

Performance Penalty = 3

→ After FO stage or FO segment of unconditional branch instruction the next instruction that is going to be executed is known. And for the e.g. given, it is instruction 15 or I_{15} that is executed next after unconditional branch instruction I_3.

→ Instruction next to I_3 that is I_4 get fetched in normal process of pipeline. This is because before DI stage of I_3 instruction it is not known that it is unconditional branch. But after DI stage of I_3, it is confirmed that I_3 is a unconditional branch & after I_3 the next instruction that get executed is instruction 15 or I_{15} from target address given in I_3 instruction. Therefore I_4 is removed or flushed from instruction pipeline.

→ Performance penalty or branch penalty for a unconditional branch instruction in a 6-segment instruction pipeline = 3.

Pipeline Hazards –

Pipeline hazards are the situations that prevent the next instruction in the instruction stream executing during its designated clock cycle. The instruction is said to be stalled. When an instruction is stalled, all instruction following or later the stalled one are also stalled in the pipeline. But the instructions present before the stalled one are not affected or continue in normal.

Types of pipeline Hazard –

1. Structural Hazard
2. Data Hazard
3. Control Hazard.

1. Structural Hazard –

Structural hazard occurs if the hardware resources required by the simultaneous overlapped instruction's execution are same.

i.e: Resources like memory or any functional unit are requested by more than one instruction at the same time or at the same clock pulse.

e.g-

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11
Instruction I_1	FI	DI	CO	FO	EI	WO					
I_2		FI	DI	CO	FO	EI	WO				
I_3			FI	DI	CO	—	FO	EI	WO		
I_4				FI	DI	—	CO	FO	EI	WO	
I_5					FI	—	DI	CO	FO	EI	WO

In the above e.g. I_1 & I_3 instructions access from same memory location. i.e: I_1 instruction wants to write result on memory with the same address where I_3 instruction wants to read the operand or data and the write operation of I_1 (WO) & read operation of I_3 (FO) performed in same clock cycle, CIK 6 for same memory location. Hence there exist resource dependency between I_1 & I_3.

Therefore I_3 instruction has a stall at CIK 6 to avoid resource dependency. This implies instructions following I_3, which are I_4 & I_5 also get stalled during the same clock cycle 6. Hence performance penalty for this example is 1.

e.g-

Clock cycle →	1	2	3	4	5	6		8	9	10
ADD R1, R2	FI	DI	CO	FO	EI	WO				
SUB R1, R3		FI	DI	CO	—	—	FO	EI	WO	
INR R5			FI	DI	—	—	CO	FO	EI	WO

ADD R1, R2 ; $R_1 \leftarrow R_1 + R_2$

SUB R1, R3 ; $R_1 \leftarrow R_1 - R_3$

INR R5 ; $R_5 \leftarrow R_5 + 1$

In this e.g. for SUB operation the operand or data required are R_1, R_3. R_3 is available but R_1 is not available for EI operation of SUB because $R_1 = R_1 + R_2$ which means that after R_1 write operation it is

available by ADD operation. Therefore EI operation of SUB get stalled till WO operation of ADD is not complete

So, performance penalty = 2.

Remedy –

The remedy of structural hazard is,

→ By using duplicate resources.

→ By providing separate data & instruction cache.

2. Data Hazard –

It occurs due to data dependency. It occurs when the execution of an instruction depends on the results of a previous instruction.

e.g- ADD R_1, R_2, R_3 ; $R_1 \leftarrow R_2 + R_3$
SUB R_4, R_1, R_5 ; $R_4 \leftarrow R_1 - R_5$
MOV R_6, R_7

Clock cycle →	1	2	3	4	5	6	7	8	9	10
ADD R_1, R_2, R_3	FI	DI	CO	FO	EI	WO				
SUB R_4, R_1, R_5		FI	DI	CO	—	—	FO	EI	WO	
MOV R_6, R_7			FI	DI	—	—	CO	FO	EI	WO

Performance penalty = 2

In the above e.g. the SUB instruction can not move to FO stage because the SUB instruction depends on data of R_1 which get available after WO stage. Hence there are two stalls until the ADD instruction has written the result into R_1. Therefore, the performance penalty for this e.g = 2.

Remedy →

The remedy for data hazards can be,

→ Hardware technique

(a) Interlock

(b) Forwarding

(a) Interlock – The hardware detects data dependency & delays scheduling of the dependent instruction by stalling enough clock cycle.

(b) Forwarding – It also known as bypassing or short circuiting. In forwarding method it is the hardware

detects the value needed for the current operation is the one produced by the previous operation but which hasnot yet been written back, it selects the forwarded result as ALU o/p instead of the value read from register or memory.

e.g. of forwarding,

CIK cycle →	1	2	3	4	5	6	7	8
MUL R_2, R_3	FI	DI	CO	FO	EI	WO		
ADD R_1, R_2		FI	DI	CO	—	FO	EI	WO

Here after EI stage of MUL instruction the result is available by forwarding. so performance penalty is reduced to 1 stall.

→ Software Technique –

Software technique is a way of rescheduling the instruction in a program so that stall can be avoided.

e.g. of software technique,

```
        LOAD  Rb, b
        LOAD  Rc, C
Data    ADD   Ra, Rb, Rc   } Data
Depen-  STORE R5, Ra       } Depen-
-cy     LOAD  Re, e          -cy
        LOAD  Rf, f
```

(unscheduled code)

```
LOAD  Rb, b
LOAD  Rc, C
LOAD  Re, e
ADD   Ra, Rb, Rc
LOAD  Rf, f
STORE R5, Ra
```

(Scheduled code)

3. Control Hazard –

control hazard occurs because of branch instructi-ons, mostly for conditional branch instructions. Hence due to branch instructions there are performance penalties. To reduce performance penalties or to avoid the chance of performance penalties there are several remedies.

Remedy →

a. Prefetch Target Instruction.
b. Branch Target Buffer.
c. Loop Buffer
d. Branch prediction.
e. Delayed Branch.

1. Multiple stream →

→ In multiple stream replicates the initial portion of pipeline & allow pipeline to fetch both instruction stream that is branch taken & branch not taken. And both are saved until branch is executed. Then select the right instruction stream & discard the wrong or not required instruction stream.

→ A problem associated with multiple stream that is, by using multiple pipelines there arise contention delays for access to the register & memory.

2. Prefetch Branch Target →

→ In this technique when a conditional branch is recognized then the target of branch is prefetched, in addition to the instruction following the branch.

→ Here if the branch is taken then the target is already Prefetched.

3. Loop Buffer →

→ It is a small high speed memory maintained by the instruction fetch stage of the pipeline & containing the 'n' most recently fetched instruction in sequence.

→ If the branch is to be taken then the hardware first checks whether the branch target is within the buffer or not. If it is present then the next instruction fetched from buffer.

→ Therefore loop buffer is like a cache memory which acts as a storage of entire loop that allows to execute a loop by accessing 'n' most recently fetched instructions of loop from Loop buffer without accessing memory.

4. Delayed Branch →

→ In this technique pipeline performance improved by automatically rearranging instructions of a program. So, the delay due to branch instruction is handled by the compiler. The compiler analyze the instructions of program & rearrange the instructions of program to avoid delay due to branch.

e.g. of delayed branch-

```
LOAD  R1, 4000
INR   R5
Branch to X
ADD   R2, R3
SUB   R5, R4
MOV   R6, R7
MOV   R8, R9
X:
```

Rearrange of Instructions in Program to avoid delayed to Branch.

```
LOAD  R1, 4000
INR   R5
Branch to X
MOV   R6, R7
MOV   R8, R9
# NOP
ADD   R2, R3
SUB   R5, R4
X:
```

5. Branch prediction →

→ On branch prediction various techniques are used to predict whether a branch will be taken or not. These techniques are,

(a) Predict never taken
(b) Predict always taken.
(c) Predict by opcode.
(d) Taken or not taken switch.
(e) Branch history table.

(a), (b), (c) → Static branch stratagies
(d), (e) → Dynamic branch stratagies.

→ Static branch stratagies means they donot depend on execution history of conditional branch instruction. And dynamic branch strategies means they depends on execution history of conditional branch instruction.

a) Predict never taken-

Assume that branch will not be taken & continue to fetch instructions in sequence from next instruction.

b) Predict always taken-

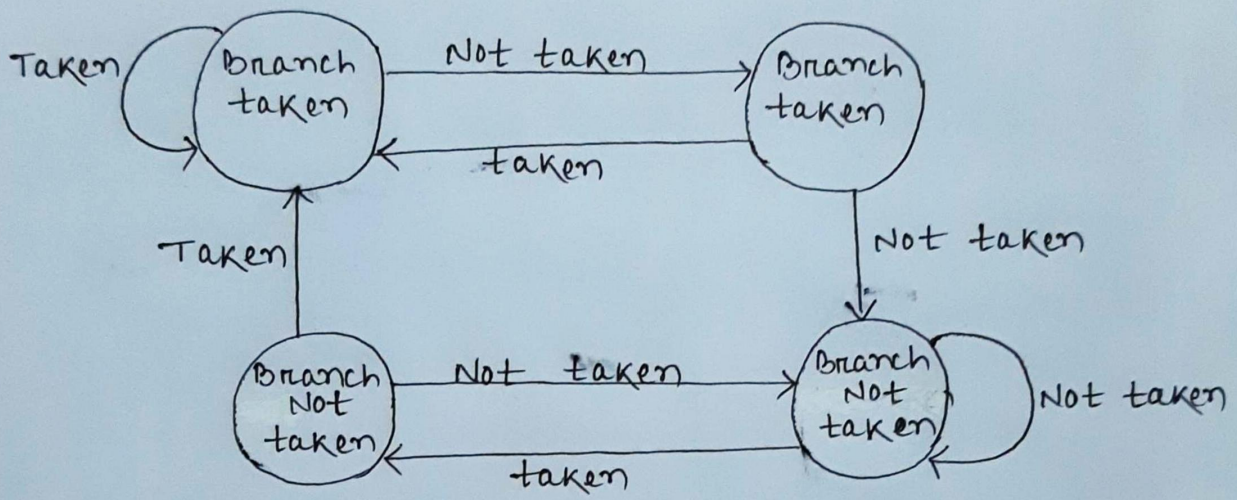
Assume that branch is always taken & fetch the instructions from target address.

c) Predict by opcode-

Here the dicision made by the processor is based on opcode of branch instruction. The processor assumes that branch will be taken for certain branch opcodes not for others.

d) Taken or Not taken switch →

On this technique one or more bits are associated with each conditional branch instruction that reflects the recent history of the instruction (branch). These bits are referred to as a taken / not taken switch that directs the processor to make a particular decision for the next time the branch instruction is encountered.



(Branch prediction state diagram)

The branch prediction state diagram start at the upper left corner. This algorithm or diagram requires two consecutive wrong prediction to change the prediction decision.