Pipeline Architectecture: Instruction Pipeline.

In this a stream of instructions can be executed by overlapping fetch, decode and execute phase of a instruction cycle.

A instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipline. Thus we can execute multiple instructions simultaneously.

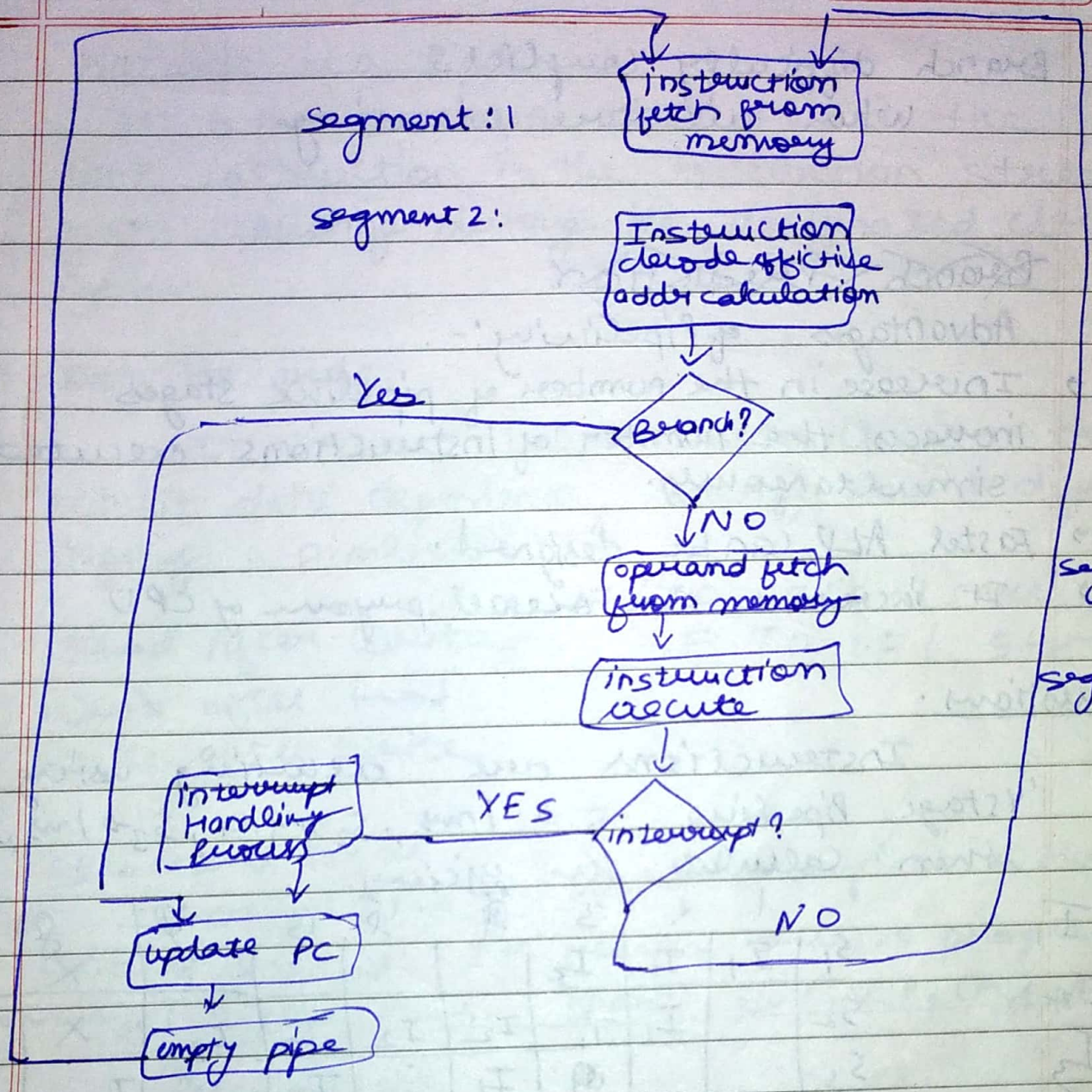
Steps	1	2	3	4	5	6	7	8	9	10
Instruction (Branch) ①	IF	ID	OF	EX						
②		IF	ID	OF	EX					
3			IF	ID	OF	EX				
4				IF	ID	OF	EX			
5					IF			IF	ID	OF
6									IF	ID
7										
8										

jump statement

Timing of instruction Pipeline

IF => Instruction Fetch
ID => Instruction Decode
OF => operand Fetch
EX => Execute

2



Problems in instructions Pipeline:

* Resource conflicts
when multiple segments are trying to access same resource, than it leads to resource conflicts.

* Data dependency conflicts
when a instruction depends on another instruction cycle, but that instruction is not calculated till now.

- Designing pipeling processor is complex.

Page :
Date : / /

- Branch difficulty conflicts when we are executing

~~Branch Prediction~~

Advantages of Pipelining:-

→ Increase in the number of pipeline stages increases the number of instructions executed simultaneously.

→ Faster ALU can be designed.

→ It increases the overall process of CPU

Questions.

Instructions are executing using 4stage Pipeline $S_1 \rightarrow 1ns$, $S_2 \rightarrow 2ns$, $S_3 \rightarrow 1ns$, $S_4 \rightarrow 3ns$ then calculate the efficiency.

	1	2	3	4	5	6	7	8	9
I_1 S_1	I_1	I_2	I_3					X	X
I_2 S_2		I_1	I_1	I_2	I_2	I_2	I_3	X	X
I_3 S_3				I_1		I_2		I_3	X
S_4					I_1		I_2	X	I_3

$$\text{efficiency} = \frac{\text{No of cycles utilized}}{\text{Total cycles}}$$

$$= \frac{15}{36} \% \text{ efficiency.}$$

Page :
Date : / /

Hazards in a Pipeline:

It is the scenario that prevent the next instruction in the instruction stream from executing during its designated clock cycles.

→ Data Hazards

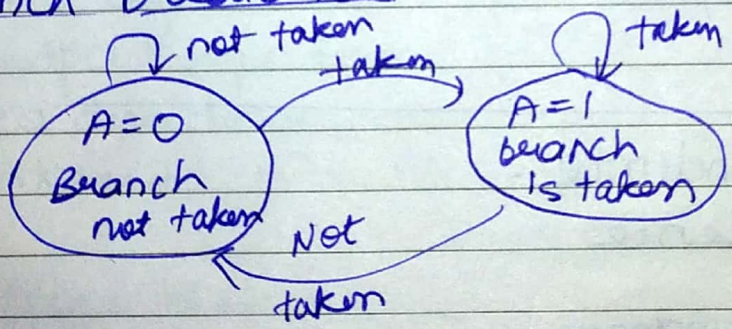
It occurs when instructions that exhibit data dependence, modify data in different stages of a pipeline.

Read After Write — IF/ID/OF/EX/save IF ID OF/ str
Write after Read
Write after Write

→ Structural Hazard.
It occurs due to resource conflict in the pipeline

→ control Hazard: It occurs due to program control instructions. Branch, jump, SKIP, CALL, RETURN

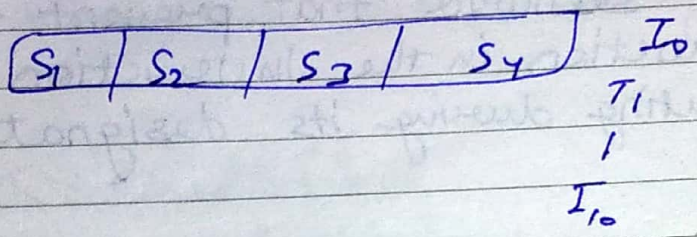
Branch Prediction



① Static Prediction → take my decision by looking
② Dynamic Prediction

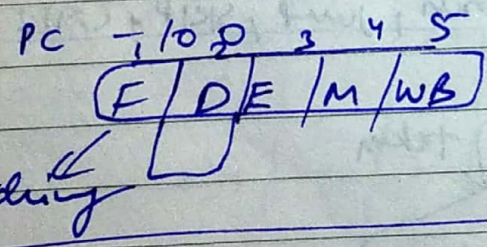
Page :
Date : / /

Delayed Branching



Branch history table

Branch address	Target address	prediction
80	100	1
129	400	1
34	300	0
75	600	1
~~75~~	700	1
100	800	0
161	900	0
518	1000	1
15		



Data Dependencies.

1. Data dependence
2. Control dependence
3. Resource Dependence.

Page :
Date : / /

Dependence graph

It is a directed graph which is used to describe the dependence relationship between the statements.

Nodes: correspond to statement (instructions)

Directed edges: shows the ordered relation among statements.

S2 will come after S1

Data Dependence: The ordering relationship between statements is indicated by data dependence

Data dependency: is a situation in which program statement refer to the data of a preceding statement.

5- types of data dependency:

1. flow dependency
2. Anti dependence
3. output dependence
4. I/O dependence
5. unknown "

(1) Flow dependce:

A statement S2 is flow dependent on statement S1 if an execution path exists from S1 to S2 and if at least one output of S1 feeds as an input in S2

It is denoted as:

$S1 \rightarrow S2$

Page :
Date : / /

example

```
S1: Load R1, A    // R1 ← Memory[A]
S2: ADD  R2, R1   // R2 ← R2 + R1
```

First S1 is executed then S1

(2.) Anti dependence

Statement S2 is anti dependent on the statement S1 if S2 follows S1 in the program order and if the output of S2 overlaps the input to S1.

It is denoted as

$S1 \overset{\text{anti}}{\rightarrow} S2$

example

```
S1: Add  R2, R1   // R2 ← R2 + R1
S2: Move R1, R3   // R1 ← R3
```

(3) Two statements are output dependent if they produce (write) the same output variable.

It is denoted as

$S1 \overset{o}{\rightarrow} S2$

example:

```
S1: Load R1, A    // R1 ← Memory[A]
S2: Move R1, R3   // R1 ← R3
```

(4.) I/O (Input + output) dependence.

Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file is referenced by both I/O statements.

Page :
Date : / /

Denoted as

$S1 \xrightarrow{I/O} S3$

example

S1: Read (4), A[1]

S3: Write (4), A[1]

⑤ Unknown dependence.

When dependence relation between two statements cannot be determined, it is known as unknown dependency.

Following situations:

→ The subscript of a variable is itself subscripted (indirect addressing). Eg: a[i[j]]

→ The subscript does not have loop index variable a[]

→ variable appears more than once with subscripts having different coefficient of loop variables. eg a[i] and a[j]

Example 1:

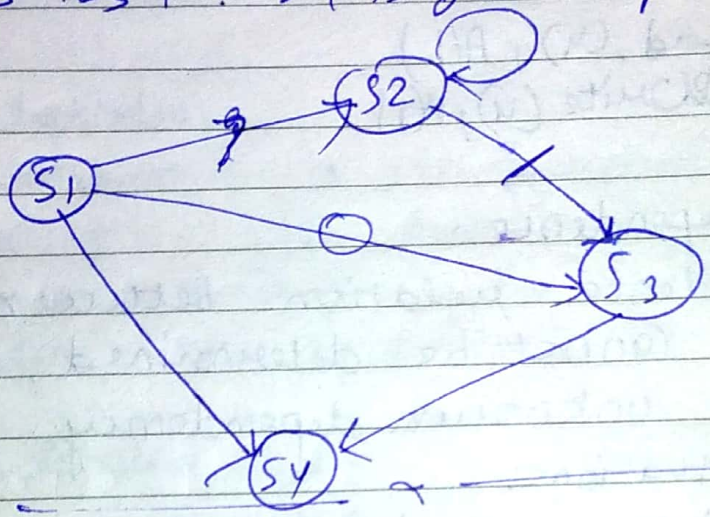
```
S1: Load  R1, A      // R1 ← Memory[A]
S2: Add   R2, R1     // R2 ← R2+R1
S3: Move  R1, R3     // R1 ← R3
S4: Store B, R1      // Memory[B] ← R1
```

- S1 to S2, S2 is flow dependent on S1
- S1 to S3, S3 is output dependent on S1
- S1 to S4, S4 is flow depend on S1

- S2 to S3, S3 is antidepend on S2
- S2 to S2, S2 is flow dependent on S2

S2 to S4 : No dependency

S3 to S4 : S4 is flow dependent on S3

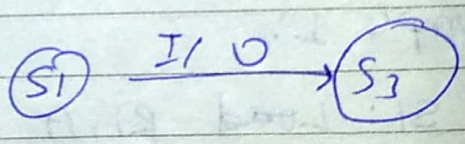


Flow dependency
S1 to S2
S3 to S4
S2 to S2
S1 to S4
Anti - dependency
S2 to S3
Output dependency
S1 to S3

Example 2: I/O Dependencies

S1: read(4), A(i) // Read array A from tape unit 4
S2: rewind(4) // Rewind tape unit 4
S3: write(4), B(i) // Write array B into tape unit 4
S4: rewind(4) // Rewind tape unit 4

I/O dependency
S1 to S3



No other dependency.

Control dependence:

control dependency arise.
When the order of execution of statements cannot be determined before runtime.
for example: if condition, will not be resolved until runtime.

Page :
Date : / /

Control dependence often prohibits parallelism from being exploited.

Compilers are used to eliminate this control dependence and exploit the parallelism.

Control-independent example

```
for (i=0; i<n; i++){
    a[i] = c[i]
    if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example

```
for (i=1; i<n; i++){
    if (a[i-1] < 0) a[i] = 1;
}
```

→ dependent on previous value

Resource dependence.

Resource dependence is concerned with conflicts in using shared resources, such as integer and floating point units (ALU), registers and memory areas among parallel events etc.

→ ALU conflicts are called ALU dependence

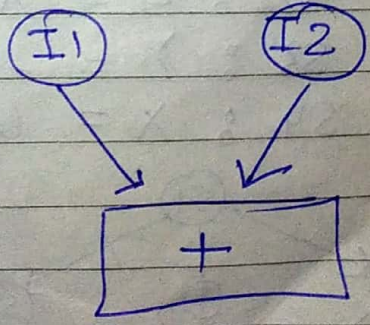
→ Memory (storage) conflicts are called storage dependence.

example:

I_1 and I_2 are accessing addition unit, so ALU dependent.

I1: A = B + C

I2: G = D + H



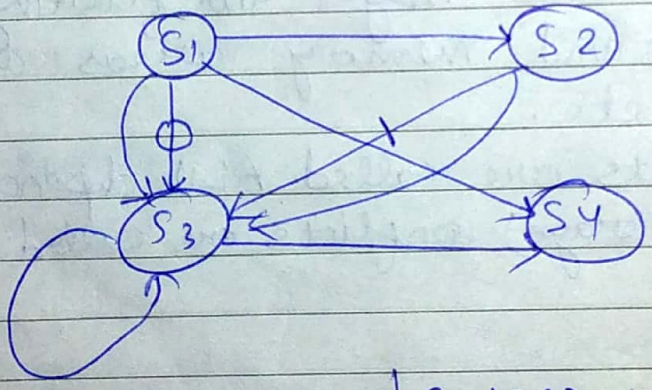
Numericals

Page :
Date : / /

a) S1 : A = B + D
S2 : C = A * 3
S3 : A = A + C
S4 : E = A / 2

S1 to S2 : S2 is flow dependent on S1
S1 to S3 : S3 is forward output dependent on S1
S1 to S4 : S4 is flow dependent on S1 and flow.
S2 to S3 : S3 is Anti depent on S2
S2 to S4 : No dependency.

S3 to S3 : flow dependency.
S3 to S4 : flow dependency

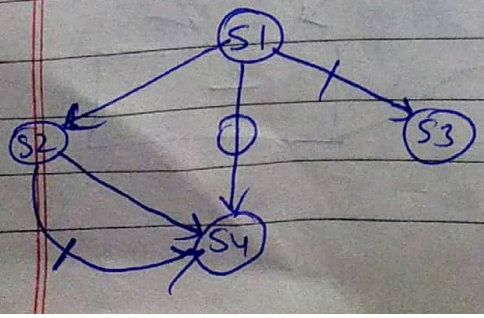


example 2 :
S1 : X = SIN(Y)
S2 : Z = X + W
S3 : Y = -2.5 * W
S4 : X = Cos(Z)

S1 to S2 : flow
S1 to S3 : anti
S1 to S4 : output

S2 to S3 : NO
S2 to S4 : flow, anti

S3 to S4 : NO d—



question 2:

Page :
Date : / /

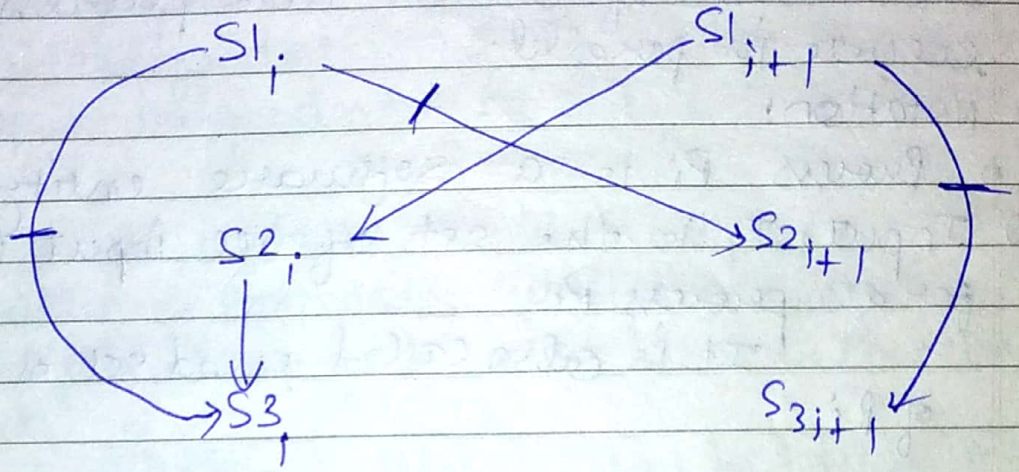
Do I = 1, N

S1: A(I+1) = B(I-1) + C(I)

B(I) = A(I) * k

C(I) = B(I) - 1

to continue



Q-3

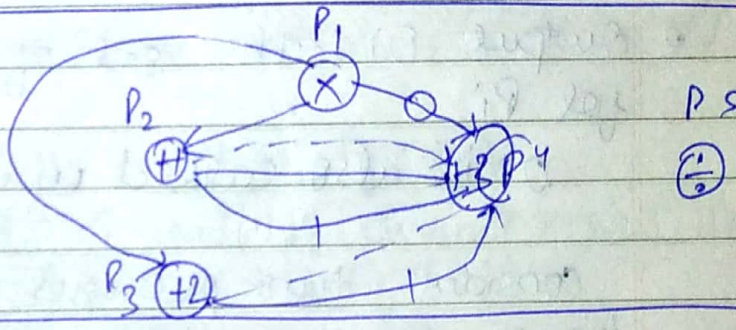
P1 : C = D * E

P2 : M = G + C

P3 : A = B + C

P4 : C = L + M

P5 : F = G ÷ E



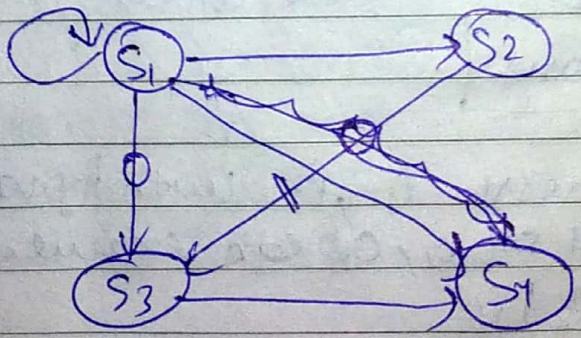
Q-4

S1 : Load R1, A // R1 ← Memory[A]

S2 : Add R2, R1 // R2 ← R2 + R1

S3 : Move R1, R3 // R1 ← R3

S4 : Store B, R1 // Memory[B] ← R1



Page :
Date : / /

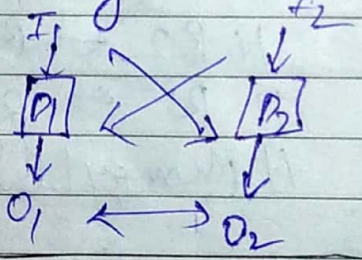
~~Question 2~~

Bernstein's conditions

In 1966, Bernstein gave a set of conditions on the basis of which two processes can execute in parallel.

- Notation:
  - Process P_i is a software entity.
  - Input I_i is the set of all input variables for a process P_i
    It is also called read set or domain of P_i
  - Output O_i is the set of all output variables for P_i
    O_i is also called write set or range of P_i.

Consider two processes P_1 and P_2, their inputs are I_1 and I_2 and outputs are O_1 and O_2 respectively



The two process can execute in parallel if they are independent and do not create longstiy results

Bernstein conditions

Two processes P_1, P_2 with input sets I_1, I_2 and output set O_1, O_2 can execute in parallel (denoted by $P_1 \| P_2$) if

$$\begin{cases} I_1 \cap O_2 = \phi \\ I_2 \cap O_1 = \phi \\ O_1 \cap O_2 = \phi \end{cases}$$

These 3 conditions are known as Bernstein's Conditions.

Page :
Date : / /

In term of data dependencies:

Bernstein conditions imply that two processes can execute in parallel if they are:

- flow-independent $I_2 \cap O_1 = \phi$
- anti-independent $I_1 \cap O_2 = \phi$
- output-independent $O_1 \cap O_2 = \phi$

In general $P_1, P_2, P_3 \ldots P_k$ can execute in parallel if Bernsteins conditions are satisfied on a pair-wise basis. This means that:

$P_1 \| P_2 \| P_3 \| \ldots \| P_k$ if and only if $P_i \| P_j$ for all $i \neq j$

Property of parallelism relation ($\|$) is:

Commutative ($P_i \| P_j$ implies $P_j \| P_i$)
Non transitive ($P_i \| P_j$ and $P_j \| P_k$ does not imply $P_i \| P_k$)
Associative $(P_i \| P_j) \| P_k = P_i \| (P_j \| P_k)$

Question 1

```
a = x + y;
b = x + z;
```

Solution:

$I_1 = \{x, y\}$ $O_1 = \{a\}$
$I_2 = \{x, z\}$ $O_2 = \{b\}$

Here $I_1 \cap O_2 = \phi$
$I_2 \cap O_1 = \phi$
$I_1 \cap O_2 = \phi$

So, Bernstein Condition are satisfied.

Question 2:

$a = x + y$

$b = a + b$;

Page :
Date : / /

Solution:

Here: $I_1 = \{x, y\}$ $O_1 = \{a\}$

$I_2 = \{a, b\}$ $O_2 = \{b\}$

Here $I_1 \cap O_2 = \phi$

$I_2 \cap O_1 \neq \phi$

$O_1 \cap O_2 = \phi$

So, Bernstein conditions are not satisfied. Hence, these two statements cannot be executed in parallel.

Q.3

P1 : C = D x E

P2 : M = G + C

P3 : A = B + C

P4 : C = L + M

P5 : F = G ÷ E

Input sets:	Output sets:
$I1 = \{D, E\}$	$O1 = \{C\}$
$I2 = \{G, C\}$	$O2 = \{M\}$
$I3 = \{B, C\}$	$O3 = \{A\}$
$I4 = \{L, M\}$	$O4 = \{C\}$
$I5 = \{G, E\}$	$O5 = \{F\}$

① Using Bernstein condition we find that:

For P1 & P2 $I_1 \cap O_2 = \phi$, $I_2 \cap O_1 = \{C\}$, $O_1 \cap O_2 = \phi$

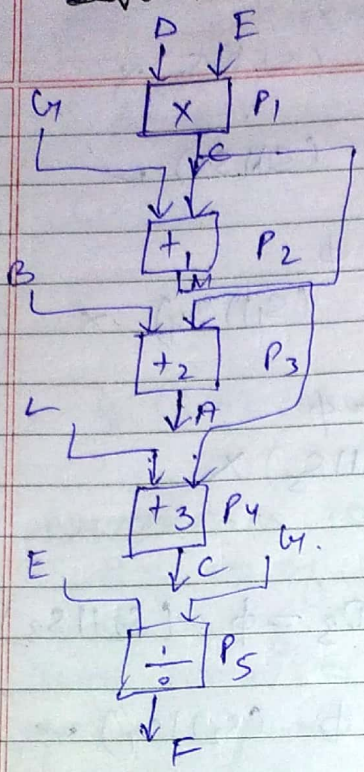
Bernstein condition does not satisfied

So P_1 and P_2 can not be executed in Parallel.

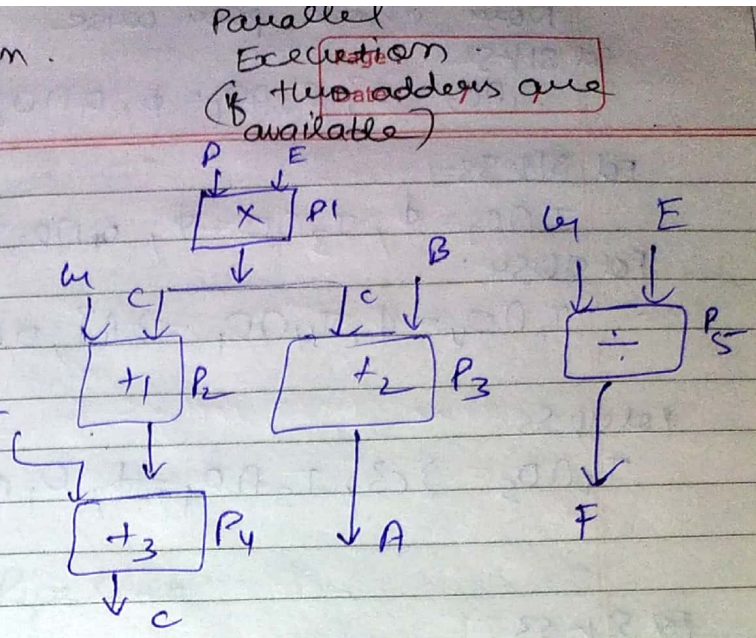
For P1 & P3 calculation are same.

Not satisfied.

Sequential execution.



Parallel Execution (if two adders are available)



Question 2:

S1: A = B + C

S2: C = B * D

S3: S = 0

S4: DO I = A, 100

S = S + X(1)

END DO

S5: IF (S.GT. 1000) C = C * 2

Input sets	Output sets
I1 = {B, C}	O1 = {A}
I2 = {B, D}	O2 = {C}
I3 = {b}	O3 = {S}
I4 = {A, S, I, X}	O4 = {S}
I5 = {S, C}	O5 = {C}

Now check pair wise

For S1 & S2

$I_1 \cap O_2 = \{c\}$, $I_2 \cap O_1 = \phi$, $O_1 \cap O_2 = \phi$ $(S_1 \not\| S_2)$ ✗

For S1 & S3 =

$I_1 \cap O_3 = \phi$, $I_3 \cap O_1 = \phi$, $O_1 \cap O_3 = \phi$ $(S_1 \| S_3)$ ✓

For S1 & S4 :

$I_1 \cap O_4 = \phi$, $I_4 \cap O_1 = \{A\}$, $O_1 \cap O_4 = \phi$

$(S_1 \not\| S_4)$ ✗

For S1 & S5 :

$I_1 \cap O_5 = \{c\}$, $I_5 \cap O_1 = \phi$, $O_1 \cap O_5 = \phi$

$(S_1 \not\| S_5)$ ✗

For S2 & S3

$I_2 \cap O_3 = \phi$, $I_3 \cap O_2 = \{c\}$, $O_2 \cap O_3 = \phi$ $(S_2 \| S_3)$ ✓

For S2 & S4

$I_2 \cap O_4 = \phi$, $I_4 \cap O_2 = \phi$, $O_2 \cap O_4 = \phi$ $(S_2 \| S_4)$ ✓

For S2 & S5 :

$I_2 \cap O_5 = \phi$, $I_5 \cap O_2 = \{c\}$, $O_2 \cap O_5 = \phi$

$(S_1 \not\| S_5)$ ✗

For S3 & S4 :

$I_3 \cap O_4 = \phi$, $I_4 \cap O_3 = \{s\}$, $O_3 \cap O_4 = \{s\}$, $(S_3 \not\| S_4)$ ✗

For S3 & S5 :

$I_3 \cap O_5 = \phi$, $I_5 \cap O_3 = \{s\}$, $O_3 \cap O_5 = \phi$

$(S_3 \not\| S_5)$ ✗

For S4 & S5

$I_4 \cap O_5 = \phi$, $I_5 \cap O_4 = \{s\}$, $O_4 \cap O_5 = \phi$

$(S_4 \not\| S_5)$ ✗

So 3 pairs can execute in parallel

$(S_1 \| S_3)$, $(S_2 \| S_3)$, $(S_2 \| S_4)$

Q- Now Reconstruct the given program to maximize the parallelism.

Because 3 pairs can execute in parallel
(S1, I1, S3), (S2, I1, S4)

Page :
Date : / /

P1: A = B + C S = 0

P2: C = B * D DO I = A, 100
S = S + X(I)
END DO

P3: if (S.GT.1000) Q = C * Z

Numericals on Pipeline:

K = no of stages
n = no of tasks

Total time to process 'n tasks' in 'K- stage pipelined processor':

$T_k = [K + (n-1)]$ clock cycles

K cycles for first task and
n-1 cycles for remaining n-1 tasks

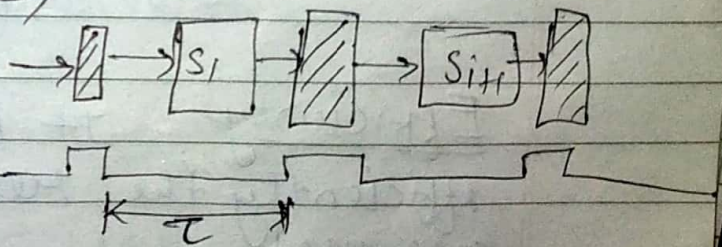
when pipelin not used

$T_1 = nK$

Some common terms related to pipeline structure:

(1.) Pipeline clock period (τ)

Let
τ_i = Time delay of stage S_i
d = time delay of latch L
τ_m = maximum stage delay



Page :
Date : / /

Clock period: τ

$$\tau = \max\{\tau_i\}_{i=1}^{k} + d$$

$$\tau = \tau_m + d$$

Pipeline frequency: f

$$f = \frac{1}{\tau}$$

Ideally, k-stage pipeline process n tasks in $k+(n-1)$ clock cycles.

k cycles for first task

n-1 cycles for remaining n-1 tasks

Total time to process n tasks in pipelined processor:

$$T_k = [k+(n-1)]\tau \qquad \tau \rightarrow \text{clock period.}$$

Total time for non pipelined processor

$$T_1 = nk\tau$$

$$\text{Speed up factor} = \frac{\text{Time for non pipelined}}{\text{Time for pipelined}}$$

$$= \frac{nk\tau}{(k+(n-1))\tau} = \frac{nk}{k+n-1}$$

Efficiency: It is an indicator how efficiently the resource of the pipeline are used.

$$E_k = \frac{\text{Speed up}}{k} = \frac{nk}{(k+n-1)\times k}$$

$$\boxed{E_k = \frac{n}{k+n-1}}$$

Page:
Date: / /

Through put:

The number of tasks that can be completed per unit time is called pipeline throughput.

$$H_k = \frac{E_k}{\tau} = \frac{n}{(k+n-1)\tau} = \frac{n}{n+(k-1)}$$

Numericals:

1- Determine the number of clock cycles that it takes to process 200 tasks in a six segment pipeline.

$k=6$

$n=200$

No of clock cycles $= k+n-1$

$= 6+200-1$

$= 205$ clock cycles.

Q.2 Non pipeline takes 50ns to process a task same task can be processed in a six-segment pipeline with a clock cycle of 10ns.

Find (a) speed up ratio of 100 tasks

$t_n = 50ns$

$t_p = 10ns$, $k=6$

$n=100$

$$\text{speed up} = \frac{n t_n}{(n+k-1)\tau_p}$$

$$= \frac{100 \times 50}{(100+6-1)\times 10}$$

$$= \frac{100 \times 5}{105} = 4.76$$

$$\text{Speed up} = \frac{n T_A}{(n+k-1)T_p} = \frac{T_n}{\left(1+\frac{k}{n}-\frac{1}{n}\right)T_p}$$

now $n \to \infty$

$$\text{Speed up} = \frac{T_n}{T_p}$$

The maximum speed up (S) can be achieved, when no of task increases is, $S = K$ as $n \to \infty$

So Maximum speed up will be 6

i.e $(S = 6)$

Q-2 Time delays of 4 stages

$\tau_1 = 60\ ns$

$\tau_2 = 70\ ns$

$\tau_3 = 90\ ns$

$\tau_4 = 80\ ns$

and interface latch has a delay of 10ns

(a) Clock Period

(b) Frequency

(c) speed up

$$\text{clock period} = \tau_{max} + d$$
$$= 90 + 10$$
$$= 100\ ns$$

$$\text{frequ} = \frac{1}{100\ ns}$$
$$= \frac{10^9}{100} = 10^7\ Hz$$

$$\text{Speed up} = \frac{\text{Non-Pipeline cycle}}{\text{Pipeline Cycle Time}}$$
$$= \frac{\tau_1 + \tau_2 + \tau_3 + \tau_4}{\tau}$$
$$= \frac{60 + 70 + 90 + 80}{100} = 2.8$$

Q-3 Consider 4-segment adder pipeline.
$\tau_1 = 50ns$, $\tau_2 = 30ns$, $\tau_3 = 95ns$, $\tau_4 = 45ns$
interface latch = 3ns

(a) How long would it take add 100 pair of numbers in the pipeline?

Clock period = 95 + 5 = 100 ns

Time for 100 pair = $(n+k-1)\tau$
$= (100+4-1)\,100$
$= 103 \times 100$ ns
$= 10.3$ ms

(b) How can we reduce the total time to about to one half of the time calculated in (a).

To reduce total time period, divide 3 stage to 2 stages

$\tau_3 = 95$ ns → $\tau_{3_1} = 50ns$, $\tau_{3_2} = 45ns$

Clock period = $\tau_m + d$
$= 50ns + 5$
$= 55ns$

Total time = $(k+n-1)\tau$
$= (5+100-1) \times 55$ ns
$T_k = 5720$ ns

Q-4
$n = 15000$
$f = 25$ MHz
$k = 5$

Speed up factor ?
efficiency ?
throughput

Speed up = $\frac{nk\tau}{[k+(n-1)]\tau}$

Page :
Date : / /

$= \frac{nk}{k+(n-1)} = \frac{5000 \times 5}{5+5000-1}$

$= 4.999$

efficiency = $\frac{\text{Speed up}}{k}$

$= \frac{4.999}{k} = \frac{4.999}{5} = 0.999$

Throughput(H_k) : $\frac{E}{\tau}$

$= \frac{nf}{k+n-1} = 24.99$

Question 5:

upgraded

non-pipelined	Pipelined
clock rate = 15 MHz	k = 5
average cycles per instruction = 4	clock speed = 20 MHz

Speed up = $\frac{\text{non-pipeline}}{\text{Pipe lined}}$

$= \frac{\frac{4}{15} \times 100}{(k+n-1)\times\tau} \cdot \frac{\frac{4}{15} \times 100}{(k+n-1)\frac{1}{f}}$

$= \frac{T_1}{T_2} = \frac{16}{5.2} = 3.02$

Throughput = $\frac{\text{efficiency}}{\text{clock period}} = \frac{nf}{k+n-1}$

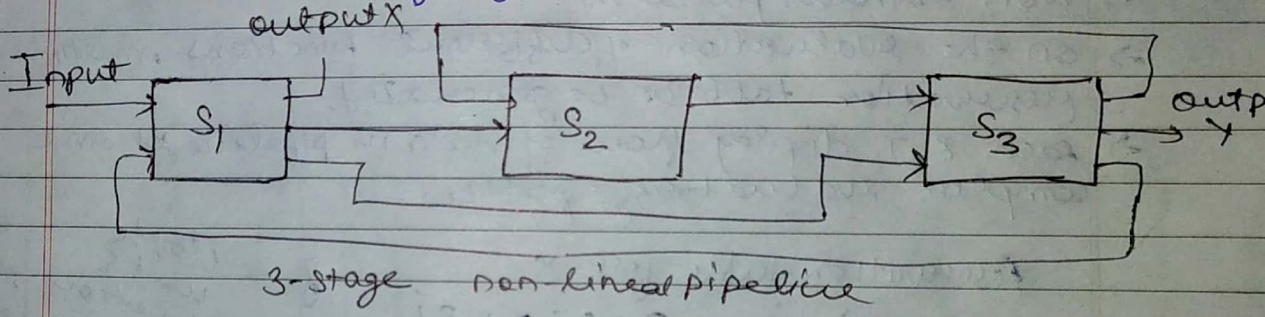
$= 19.23$

Throughput for non pipelining
$= \frac{15}{4} \times 10^6$ instr/sec

$= 6.75$ MIPS

Linear Pipelines: They are used to perform fixed function
Non-Linear Pipeline: are dynamic pipelines. It allows feed back and feed forward connections in addition to streamline connections.

- It can be reconfigured to perform different functions at different time i.e Multifunctional
- Gives more than one output, output of the pipeline is not necessarily from the last stage.



3-stage non-linear pipeline

Two function X and Y not necessarily from last stage

Streamline connections:	Feed forward	Feed back
S1 to S2	S1 to S3	From S3 to S2
S2 to S3		S3 to S1

Reservation Table

It specifies the utilization pattern of successive stages in a pipeline

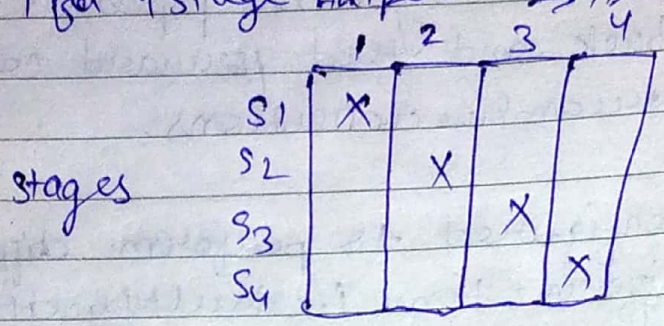
- Rows represents stages and columns represent clock time units.

For Linear Pipeline

Page :
Date : / /

* Reservation table for Linear Pipeline Processor follows diagonal streamline pattern.

R-T for 4 stage Pipeline → Time (clock cycles)



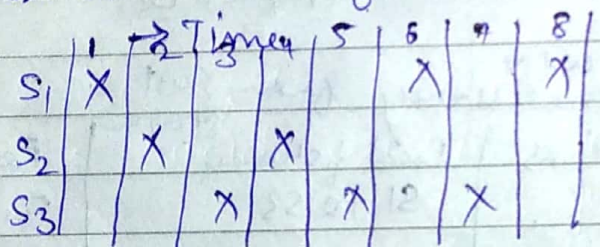
For Non-Linear Pipeline.

→ Non-Linear pattern

→ On the evaluation of different functions multiple reservation table can be generated.

→ Each R-T display flow of data in pipeline for one complete evaluation of fn.

Reservation table for X



1 = clock units.
= evaluation time.

Processing sequence for function X

S1 → S2 → S3 → S2 → S3 → S1 → S3 → S1

→ Multiple check marks in a row means repeated usage of same stage in different cycle.

→ Contiguous check marks - extended usage of stage over more than one cycle.

→ Multiple check marks in simultaneous usage of multiple stages same cycle.

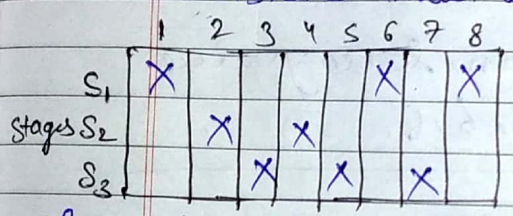
→ Total no of clock units = evaluation time.

static pipeline → linear pipeline.
dynamic → non-linear pipeline

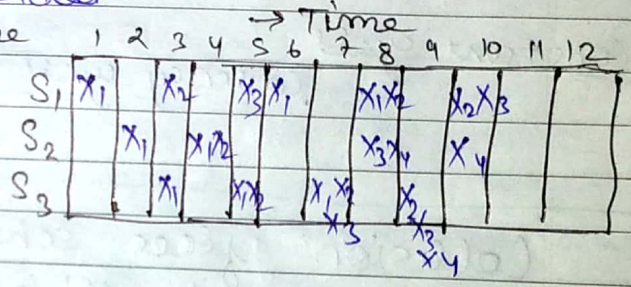
Page :
Date : / /

Latency :
- An initiation refers to the start of single function evaluation.
- No of clock cycles between two initiation of pipeline is the latency.
- Latency values must be positive integers.

Collisions :
- An attempt by two or more initiation to use the same pipeline stage at the same time is called collision.
- Collision implies resource conflicts between two initiations.
- Collisions must be avoided.



Reservation table of function



Forbidden Latencies

Permissible latencies

- Latencies that does not cause collisions are permissible latencies.
- Permissible latencies = {1, 3, 6, 8} for last problem

Latency sequence:
Sequence of permissible latencies
{1, 3, 6, 8}.

Latency cycle:
A latency sequence that repeats the same subsequence (cycle) indefinitely.
For ex: repeat after 1, 3, 6, or 8.

Average latency
the sum of all latencies divided by the number of latencies along the cycle.

$$\frac{1+8}{2} = 4.5$$

Constant cycle:
latency cycle one or more latency value
6, 6, 6, 6.

Collision free scheduling

To avoid collisions and to achieve high throughput, all the tasks awaiting for initiation should be properly scheduled.

while scheduling main objective is
To obtain shortest average latency between initiations without causing collisions.

Steps:

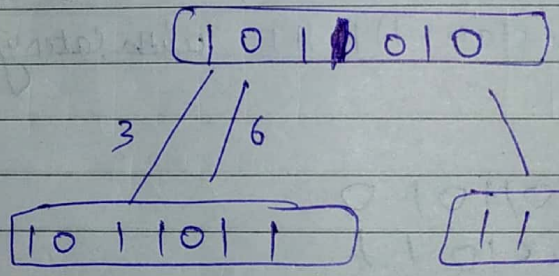
Collision vector:
It is m-bit binary vector $C = (C_m C_{m-1} \ldots C_2 C_1)$ that can be obtained from combined from combine set of permissible and forbidden latency:

$m <= n-1$, n = column.

$C_i = 1$ Forbidden.
$C_i = 0$ Permissible
eg $C_x = 1011010$

In the previous example
forbidden latencies = {2,4,5,7}
Maxim F.L = 7
So m = 7
collision vector $C_x = \{C_7 C_6 C_5 C_4 C_3 C_2 C_1\}$
Permissible latency = {1,3,6}
Now for Forbidden latencies = {2,4,5,7}
collision vector
$= \{C_7 C_6 C_5 C_4 C_3 C_2 C_1\} = (1011010)$
1 0 1 1 0 1 0

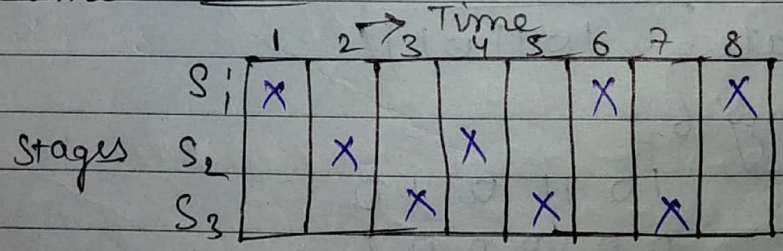
State diagram
Constructed from collision vector



It specifies the permissible state transitions among successive initiations.

Numericals

1.



S1: {(6-1), (8-6), (8-1)} i.e {5,2,7}
S2: {(4-2)} i.e {2}
S3: {(5-3), (7-5), (7-3)} i.e {2,2,4}
forbidden latencie = {2,4,5,7}
It cause collision.

Permissible latencies = {1, 3, 6, 8+}
It does not cause collision

Page:
Date: all are permissible.

Maximum forbidden latency (m) = 7
so the collision vector is of 7 bits.

Collision vector = $\{C_7 C_6 C_5 C_4 C_3 C_2 C_1\}$
1 = forbidden.
0 - Permissible.
Collision vector = (1011010)

Permissible latency = {1, 3, 6, 8+}
Initial collision vector
1011010

with latency 1

```
  1011010
+ 0101101
---------
  1111111   Next State
```

So, the new state 1111111 with latency 1

with latency 3

```
  1011010
+ 0001011
---------
  1011011
```

next state = 1011011

with latency 6

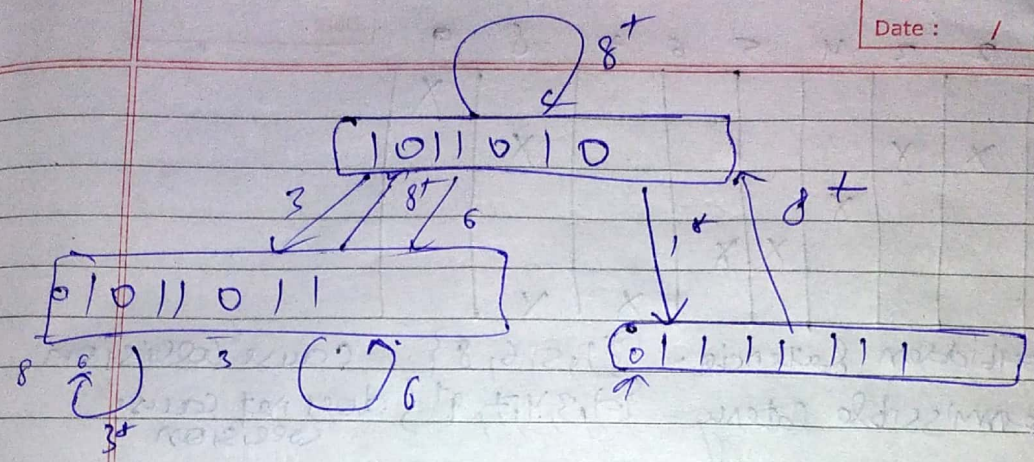
```
  1011010
  0000001
---------
  1011011
```

next state = 1011011

with latency 8

```
  1011010
  0000000
---------
  1011010
```

Next state = 1011011



Simple Cycle: It is a latency cycle in which each state appears only once.

Greedy cycle: A greedy cycle is a simple cycle whose edges are all made with minimum latency or minimum avg

Latency cycle	Types of cycle	Average latency
(3)	Simple cycle const-	3 (MAL) greedy
(6)	Simple cycl "	6
(8)	Simple cycle "	8
(1, 8)	"	4.5 greedy
(3, 8)	"	
(6, 8)	"	
(3, 6, 3)	--	5
(3, 3, 6)	. — -	7
(1, 8, 3, 8)	— —	

minimum edge

generally there are 2 greedy cycles

MAL = 3

Question-2

	1	2	3	4	5	6	7	8	9
S1	X								X
S2		X	X					X	
S3				X					
S4					X	X			
S5							X	X	

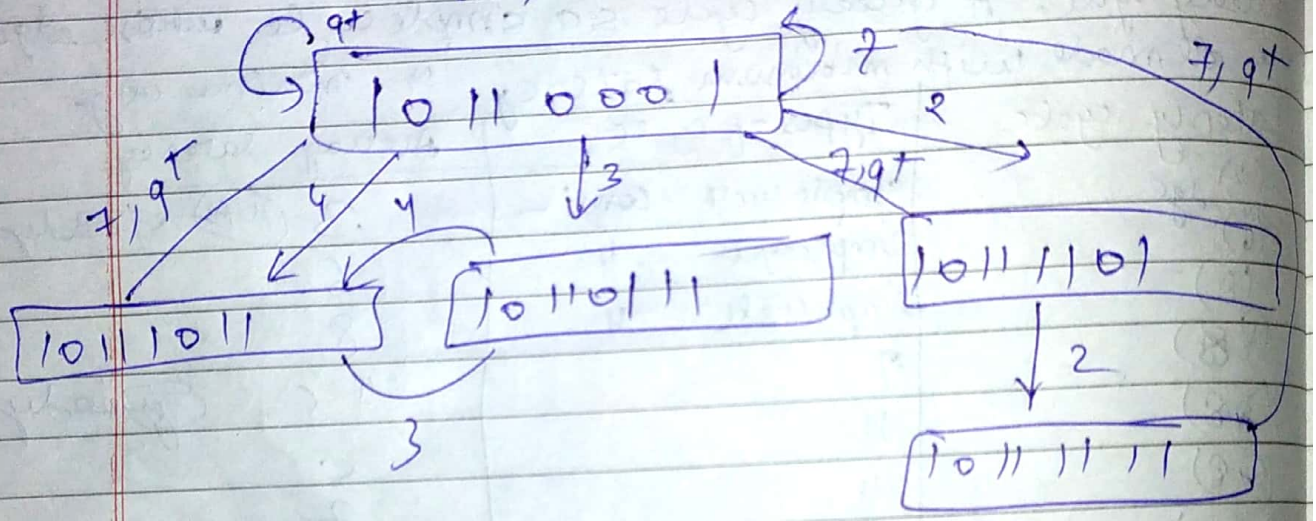
stages

forbidden latencies = {1, 5, 6, 8} cause collision

Permissible latency = {2, 3, 4, 7, 9+} does not cause collision

collision vector

: $\{C_8\ C_7\ C_6\ C_5\ C_4\ C_3\ C_2\ C_1\} = \{1\ 0\ 1\ 1\ 0\ 0\ 0\ 1\}$



Throughput = $\frac{f}{mal} = \frac{1}{MAL * \tau}$

Bound.

Lower Bound on MAL

The maximum no of check marks in any row of the reservation table.

~~Lower~~ if Lower bound ≠ MAL then optimal latency not achieved

upper bound of MAL

Then no of 1's in ICV + 1