# Chapter 6

# PIPELINING & SUPERSCALAR TECHNIQUES

## • LINEAR PIPELINE

- Cascade of processing stages to perform a fixed function over a stream of data flowing from stage 1 to stage n

- **Applications:** instruction execution
  arithmetic computation
  memory-access functions

Models → Asynchronous
Models → Synchronous

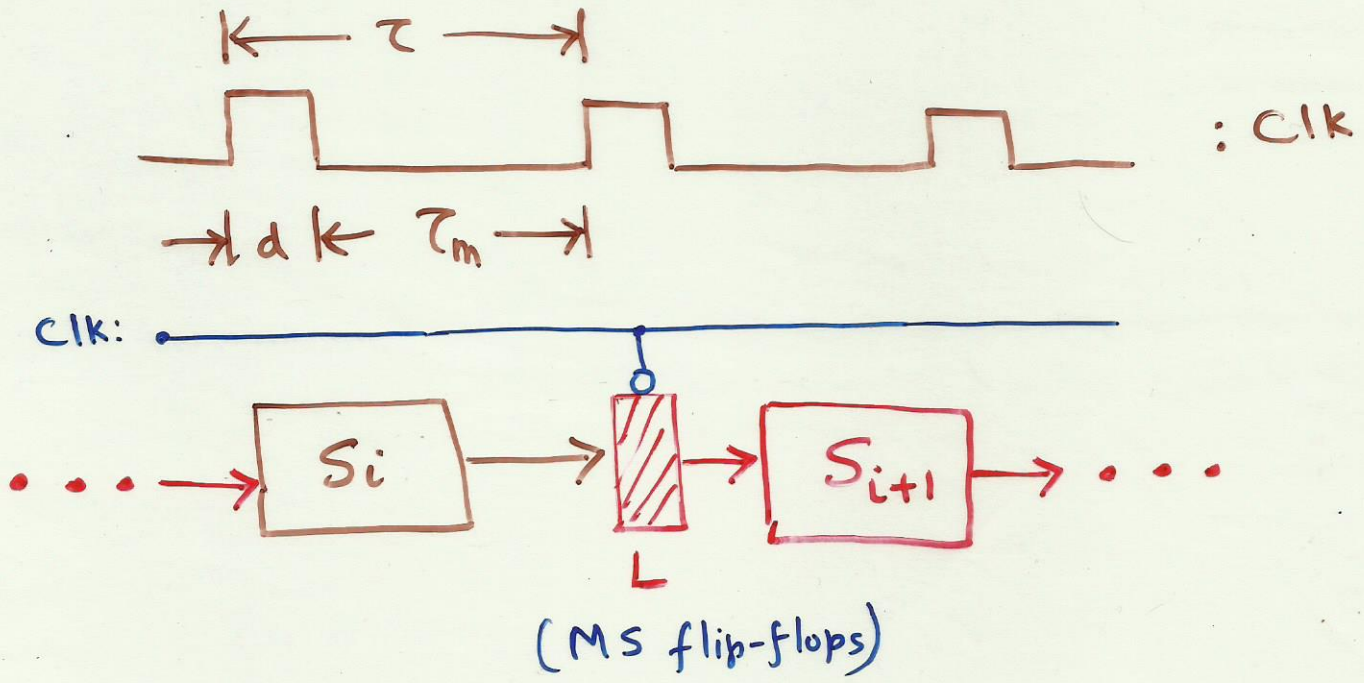
Models → depends on the flow control along the pipeline stages.

## Asynchronous Model

- Adjacent stages use handshaking protocol (Ready / ack signals)
- Wormhole routing ;
- Variable throughput is possible
- delay per stage is different

[fig. 6.1 (a), pg: 266]

## SYNCHRONOUS MODEL

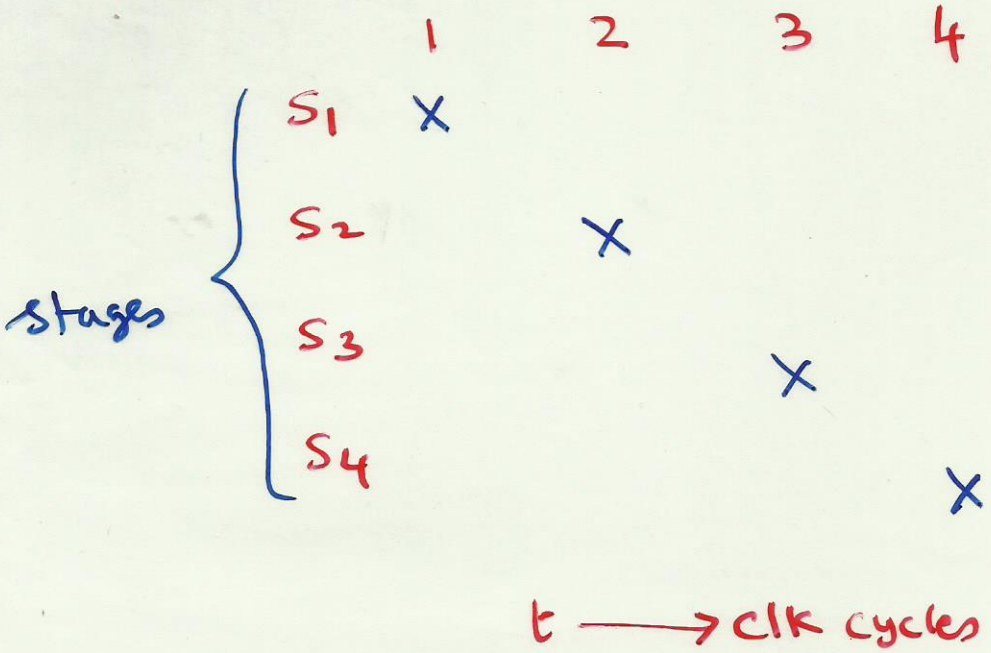



(MS flip-flops)

- Upon arrival of a clock pulse, all the latches transfer data to the next stage simultaneously

- All stages have more-or-less equal delays

Pipeline utilization is studied by using a reservation table.




- After 4 clock cycles an o/p appears from stage $S_4$.

$\tau_m$ : maximum stage delay : $\max_{1 \le i \le k} \{\tau_i\}$

$$\tau = \tau_m + d \quad \rightarrow (1)$$

In general, $\tau_m \gg d$.

## Pipeline frequency $(f) = (1/\tau) \rightarrow (2)$

Thus, $f$ represents the max. throughput. However, the actual throughput may be less than $f$, as it depends on the initiation of the successive tasks.

## Clock skewing

$$d + t_{max} + s \leq \tau \leq \tau_m + t_{min} - s$$

i.e., $t_{max}$: time delay of the longest logic path within a stage

$t_{min}$: shortest logic path delay

Thus to avoid any race, we need

$$\tau_m \geq t_{max} + s \quad [s: \text{offset time}]$$

& $d \leq t_{min} - s$

- if $s = 0$ (ideal case) $t_{max} = \tau_m$ & $t_{min} = d$

If we have k stages & n tasks

$n > k$; we need

k cycles to complete the first job

+ (n-1) cycles to complete the other jobs

(1 cycle per task is the assumption)

$$\Rightarrow T_k = [k + (n-1)]\tau \longrightarrow (3)$$

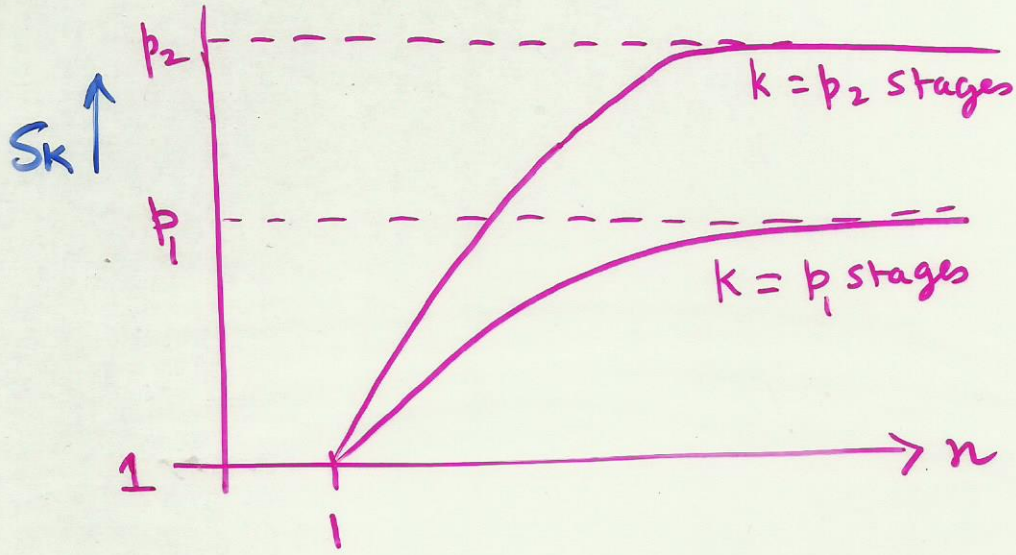
$$\text{Speed up } (S_k) = \frac{T(\text{non-pipelined system})}{T(\text{pipelined system})}$$

$$= \frac{n \cdot k \cdot \tau}{[k + (n-1)]\tau} \longrightarrow \{\text{equivalent function non-pipelined processor with a delay } (k\tau).\}$$

$$S_k = \frac{nk}{k + (n-1)} \longrightarrow (4)$$

$\longrightarrow$ [See fig 6.2(a), $S_k$ Vs n, for some fixed k]

Thus, as $n \to \infty$, $S_k \to k$. This can be seen from the fig.




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

Can we determine an optimal # of stages to be used?

Micropipelining ⟶ finest level of pipelining

$2 \leq k \leq 15$

Macropipelining ⟶ at the "processor level"

Optimal choice ⟶ maximize

$$\left(\frac{\text{Performance}}{\text{Cost}}\right) \quad [PCR]$$

$t$ : total time required for a non-pipelined sequential pgm

With $k$ stages, the required clock period is

$$p = \left(\frac{t}{k}\right) + d \quad , \quad d: \text{latch delay}$$

$\Rightarrow$ Throughput (max) $f = 1/p$

$$= \frac{1}{\left[t/k + d\right]}$$

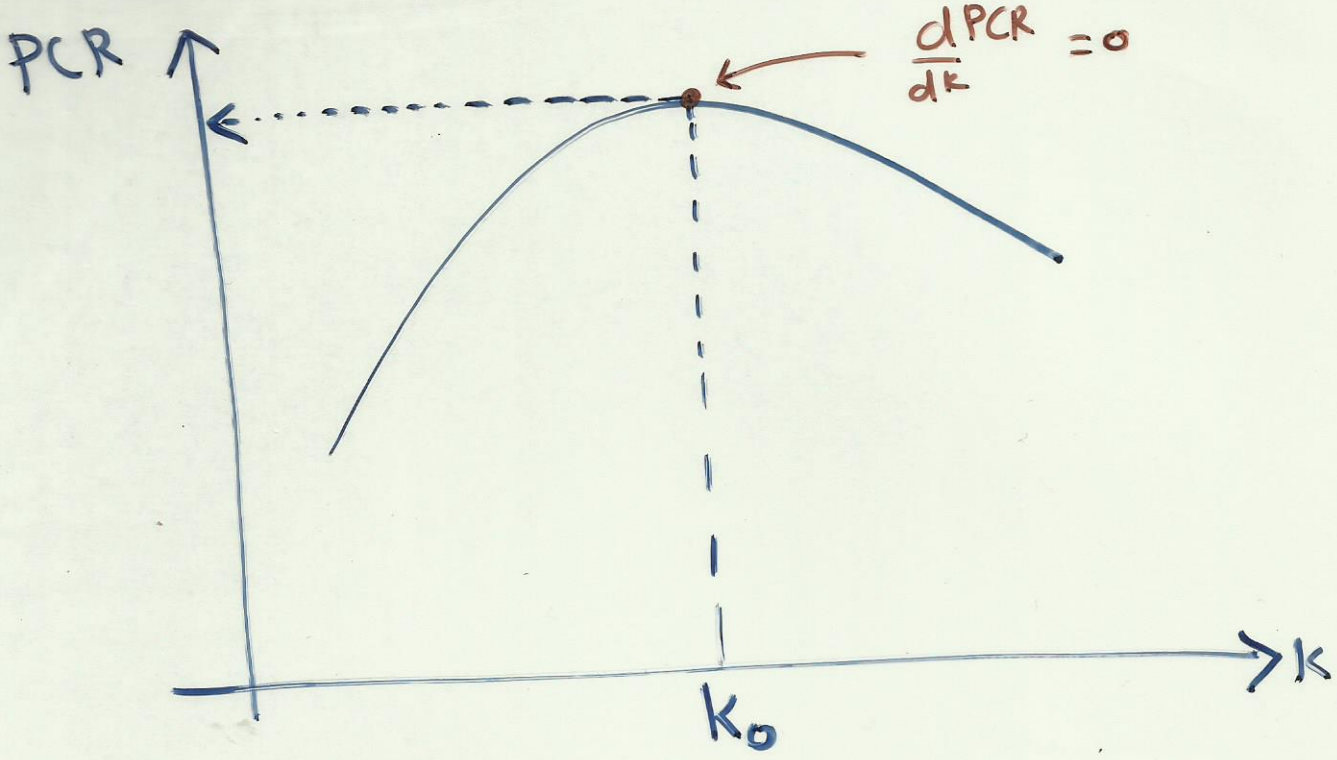
Total pipeline cost is estimated by: $(c + kh)$,

where, $c$ : Cost of all logic stages

$h$ : Cost of each latch

Thus, $PCR = \left(\frac{f}{c + kh}\right)$

$$PCR = \frac{1}{\left[\frac{t}{k} + d\right]\left[c + kh\right]} \longrightarrow (5)$$




$k_o$ (optimal # of stages)

$$k_o = \sqrt{\frac{t \cdot c}{d \cdot h}} \longrightarrow (6)$$

$$\text{Efficiency } (E_k) = \left(\frac{S_k}{k}\right) = \frac{n}{k+(n-1)}$$

[Speed-up per stage]

$\longrightarrow$ (7)

As, $n \to \infty$, $E_k \to 1$, as expected

Also, $E_k \geq 1/k$ [Sets a lower bound on $E_k$, when $n=1$]

Pipeline throughput

$H_k$ : # of tasks performed per unit time

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

$$H_k = \frac{n \cdot f}{k+(n-1)} \quad \rightarrow (8)$$

Thus, $H_k = E_k \cdot f = E_k/\tau = \frac{S_k}{k\tau}$ ←

---

Problems: 6.1, 6.4,

# Non-linear Pipeline Processors

- Allows feedforward & feedback connections in addition to the streamline connections.

[see fig 6.3 (a), pg: 271]

With the feedforward & feedback connections, scheduling of tasks on a pipeline becomes a difficult problem.

→ basic question: how to maximize the throughput of the pipeline?

Equivalently,

With what minimum time gap, can initiate tasks?

TABLE 1.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $S_1$ | X | | | | | X | | X |
| $S_2$ | | X | | X | | | | |
| $S_3$ | | | X | | X | | X | |

$\longrightarrow t$

→ inital state will be given. In this fig, this is the <u>way</u> in which data flows through the pibeline.

<u>Reservation table</u> ⟶ timing diagram; also referred to as a time-space graph

Each function evaluation is specified by a reservation table.

- <u>Fig 6.3 (b)</u> (pg: 271) → we will consider this table for analysis throughout.

# Some Terminology !

- **Evaluation time**: # of columns in a RT

- A **checkmark** (x) in a cell (i,j) means stage i is used at the $j^{th}$ clk cycle.

- **Latency**: # of time units (clk cycles) between 2 initiations of a pipeline is the latency between them.

- Any attempt to use a stage by 2 initiations at the same time results in a **collision**

Collision → resource conflicts ; must be avoided

- Latencies that cause collision are called forbidden latencies. (FL)

Refer to Table 1.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | X | | X | | | X | | XX | | X |
| 2 | | X | | XX | | X | | | | |
| 3 | | | X | | XX | | XX | | X | |

first initiation $t=1$
Second initiation $t=3$
} latency of 2

Jhus, 2 is a FL

Similarly, 5 is also a FL

→ Verify: $FL = \{2, 4, 5, 7\}$

⇒ Permissible latencies

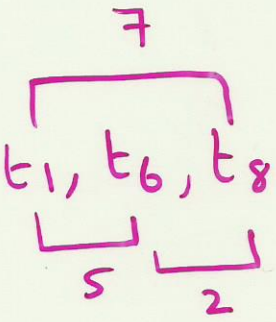
$P = \{1, 3, 6, 8\}$

- How to find/determine the forbidden latencies, given a RT ?

Check the distance between any two checkmarks in the same row of the RT.

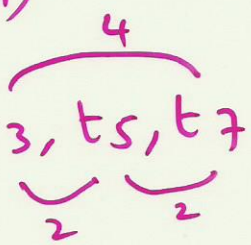
Refer to Table 1.

row 1 : FL → {2, 5, 7}




row 2 : FL → {2} ($t_2, t_4$)

row 3 : FL → {2, 4}




⇒ FL → {2, 4, 5, 7}

- Latency Sequence : sequence of permissible (non-forbidden) latencies between successive initiations

• **Latency cycle** : latency seqvence which repeats the same subseqvence indefinitely.

Refer to Table. 1.

LC (1,8) = 1, 8, 1, 8, .....

⇒ $t_1$ → job 1
$t_2$ → job 2
$t_{10}$ → job 3
$t_{11}$ → job 4
⋮

[see fig 6.5 (a)]

i.e., successive initiations of new tasks are separated by one cycle and 8 cycles alternately.

Similarly, LC(3) = 3, 3, 3, .....
LC(6) = 6, 6, 6, .....

• **Average Latency (ALC(...))** = $\frac{\sum_i l_i}{\#\text{ of latencies}}$

ALC(1,8) = $\frac{1+8}{2}$, ALC(3) = 3, ALC(6) = 6

- Given these, how do 9 effect a collision free scheduling ?

formal treatment

## Collision vectors

→ For a RT with n columns, the maximum FL (m) $\leq n-1$.

→ The permissible latency p should be as small as possible, i.e.,

$$1 \leq p \leq m-1$$

→ $C = (C_m, C_{m-1}, C_{m-2}, \cdots, C_2 C_1)$ is a m-tuple vector that captures the FL & P latencies

→ $\begin{cases} C_i = 1, & \text{if latency } i \text{ causes collision} \\ = 0, & \text{if latency } i \text{ is permissible} \end{cases}$

Note that $C_m = 1$ always, as it corresponds to the max. FL.

Refer to Table 1. $n = 8$, $m \leq 7$

$FL = \{2, 4, 5, 7\}$

$P = \{1, 6, 3, 8\}$

$m = 7 \Rightarrow$ 7 bit collision vector

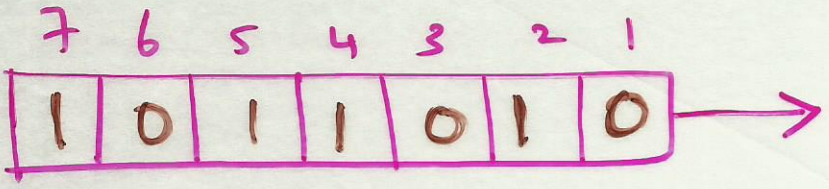
$C = C_7\ C_6\ C_5\ C_4\ C_3\ C_2\ C_1$

$1\ 0\ 1\ 1\ 0\ 1\ 0$

Thus, the collision vector (to evaluate a function X) for a given RT is

$$C_X = 1011010$$

O/p function $C_Y = \cdots\cdots$




m-bit shift register

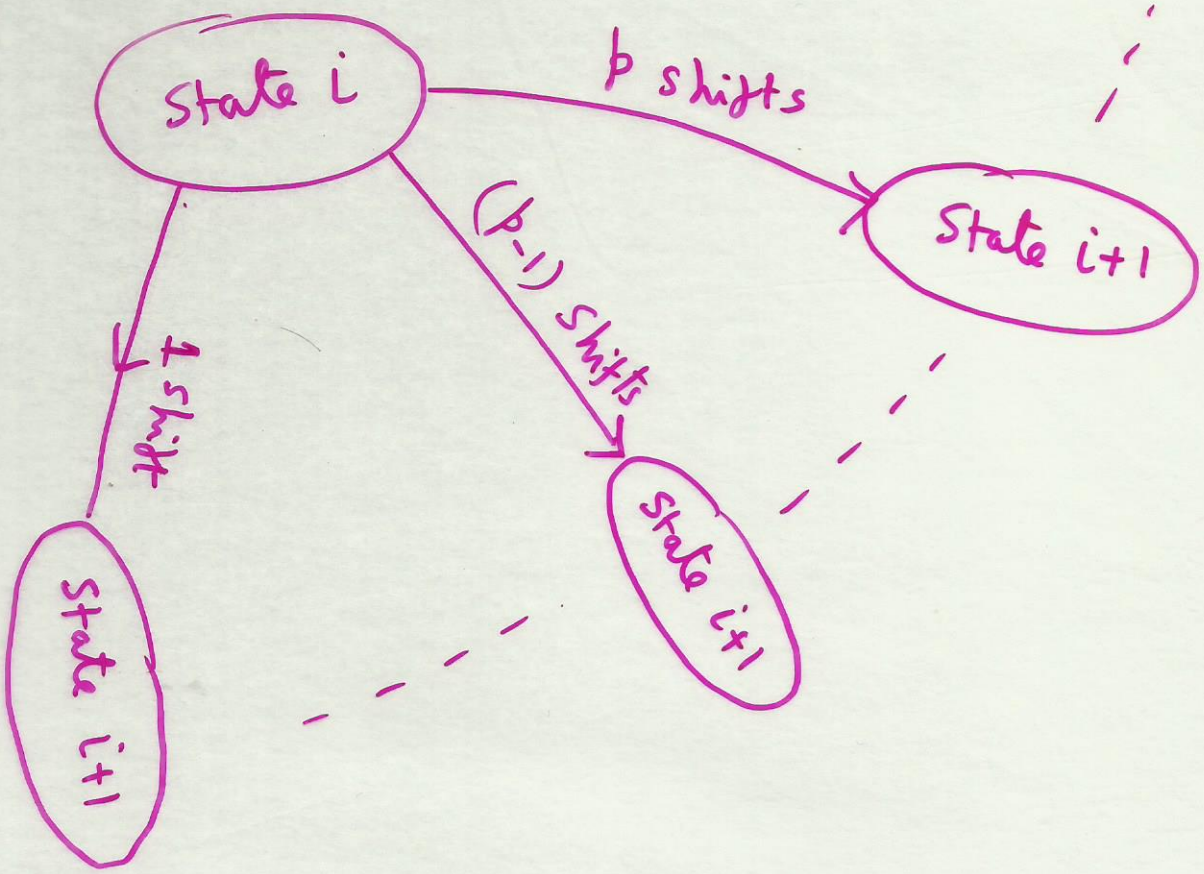
At time 1, we have an initial collision vector

Each 1-bit shift corresponds to an increase in the latency by 1.

Thus, starting from time 1, when a 0-bit emerges from the right end after $p$ shifts, this means $p$ is a permissible latency

Verify: $t_1, t_3, t_6$ & {1,3,6} are Per. lat.

- As we shift to the right, a logical '0' enters from the left end.




- next state after p shifts is obtained by [Cx OR (shifted contents in SR)]

  ↖ bitwise OR operation

Jhus,

$C_x = (1011010)$ $\xrightarrow{\text{rt s}}$ 0101101 / 1011010

after 1 right shift gives

→ 0101101 (shifted contents)

1011010 ($C_x$)

1111111 ← next state

$C_x = (1011010)$

1 right shift : 0101101

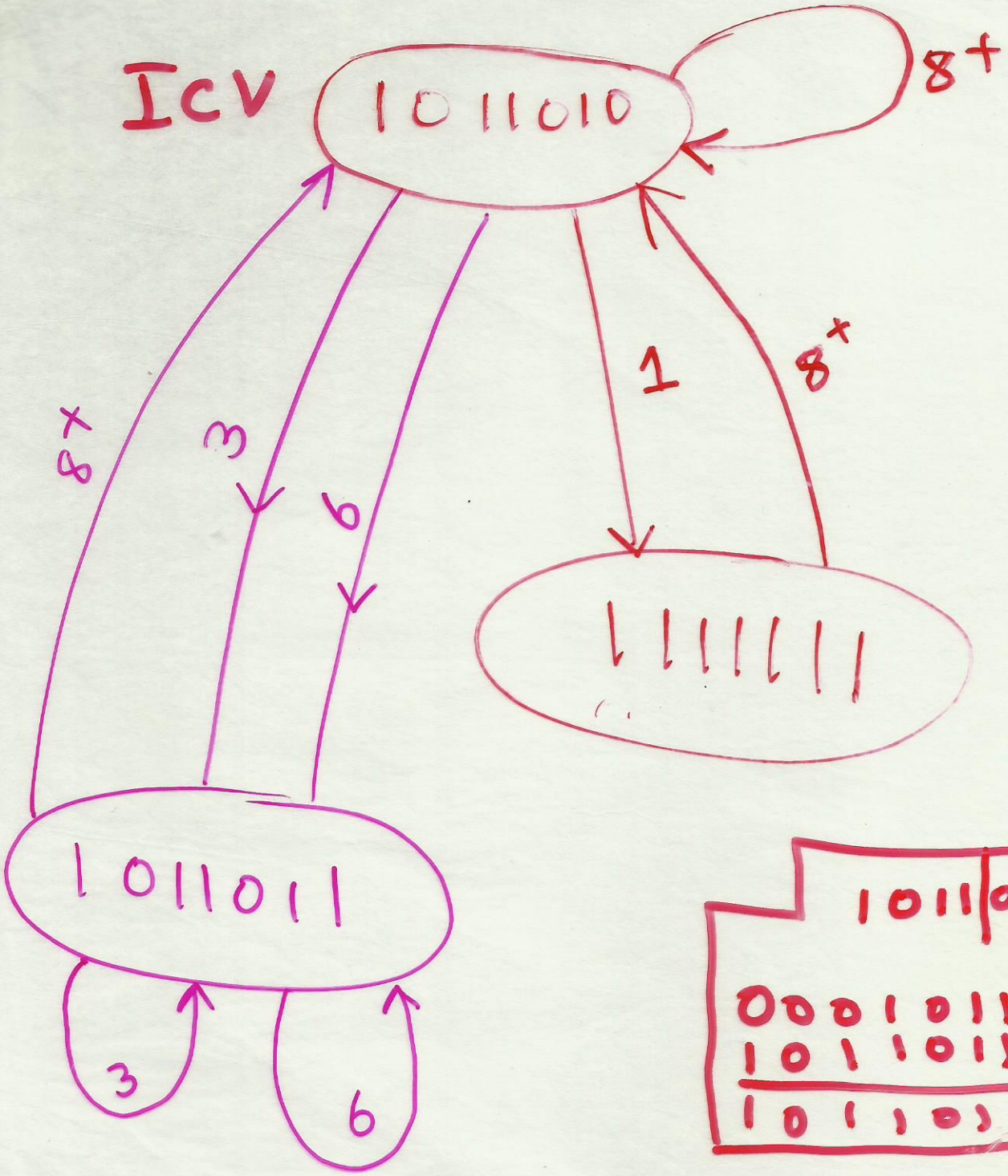
2nd r.sh : 0010110 ✔

3rd r.sh : 0001011 ←

next state after 3 shifts

```
  0 0 0 1 0 1 1
(OR) 1 0 1 1 0 1 0  ← Cx
-----------------
  1 0 1 1 0 1 1  ✔
-----------------
```

- With $C_x$, if you shift 8 times, you will reach $C_x$
- In general, with (m+1) shifts or greater all transitions are redirected to the initial state (denoted as $m^+$)







State diagram for table 1 (See fig 6.6(b)

- Once the initial collision vector is known state diagram can be uniquely determined.

Thus,

→ 0's & 1's in the current state, say at time t, indicate the P & FL latencies, at time t.

→ bitwise ORing of the shifted contents of the present state with the initial collision vector is meant to prevent collisions from future initiations starting at time t+1 onwards.

→ <u>Collision free scheduling</u> ⇒ choose that permissible latency that has minimum average latency (MAL)

## Greedy cycles

There are infinitely many latency cycles one can trace from the SD.

(1,8)

(1,8,6,8)

(3)

(6)

(3,6,3)

⋮

we look for "Simple Cycles"

↓

Latency cycle in which each state appears only once (including constant cycles)

(1,8), (3,8), (6,8), (3), (6), (8)

→ (1,8,6,8) is not simple

Since state (1011010) is traversed twice

## Some simple cycles are greedy.

- Greedy cycle is one whose edges are all made with minimum latencies from their respective starting states.

→ refer to the SD (slide # 21) [Fig 6.6(b)]

greedy cycles : (1,8) & 3

- Greedy cycles must first be simple cycles
- Avg. latency of a greedy cycle must be lower than those of simple cycles

→ (1,8) → $\frac{1+8}{2} = 4.5$

$4.5 < ALC(6,8)$

↖ simple cycle

The greedy cycle (3) has a constant latency & is equal to the minimal average latency (MAL), in this example.

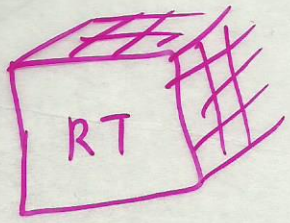
* At least one of the greedy cycles will lead to MAL

Thus,

Collision free scheduling ⇒ find all the greedy cycles from the set of simple cycles.

→ greedy cycle yielding the MAL is the final solution!

So, the procedure is :

given : 


→ find (FL) & (P) sets

→ form the (CV)

→ draw (SD)

→ identify simple cycles

→ identify greedy cycles

→ extract MAL !

Now, can we improve the throughput ?

If so, how ?

→ The purpose is to achieve an optimal latency cycle, which is absolutely the shortest.

# Some Observations

[1972 by Shar]

Max # of Checkmarks in any row $\leq$ MAL $\leq$ average latency of any greedy cycle

and avg. latency of any greedy cycle $\leq$ # of 1's in the initial cv + 1

facts : (after going through the proof's)

- Optimal latency cycle $\in$ {greedy cycles}
- However, a greedy cycle is not sufficient to guarantee the optimality of MAL. It is the lower bound that guarantees the optimality.

[our e.g: MAL = 3 meets the lower bound]

From our Table 1 (6.6 b)
& SD ←

→ Upper bound on the MAL = 4 + 1 = 5

→ To optimize the MAL, we need to hit the lower bound by modifying the RT.

---

Approach: reduce the max. # of checkmarks in any row, but RT must preserve the function!

⇝ Idea is to introduce non-compute delays

⇒ modified SD! → results in greedy cycles that meet the low. bound.

# Example 6.3 (page 277-279)

Consider

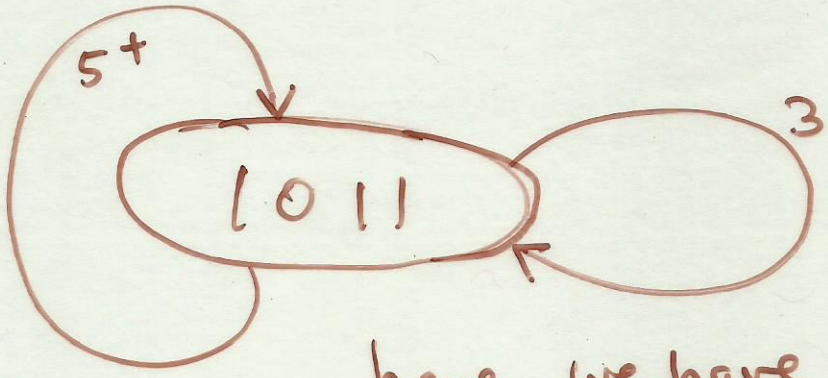
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $S_1$ | X | | | | X |
| $S_2$ | | X | | X | |
| $S_3$ | | | X | X | |

$FL = \{4, 2, 1\}$ $\quad m = 4$

$C = C_4\, C_3\, C_2\, C_1$

$\therefore C = (1\ 0\ 1\ 1)$

∴




3 shifts

0001

1011 (OR)

1011

here we have a greedy cycle of latency 3 = MAL

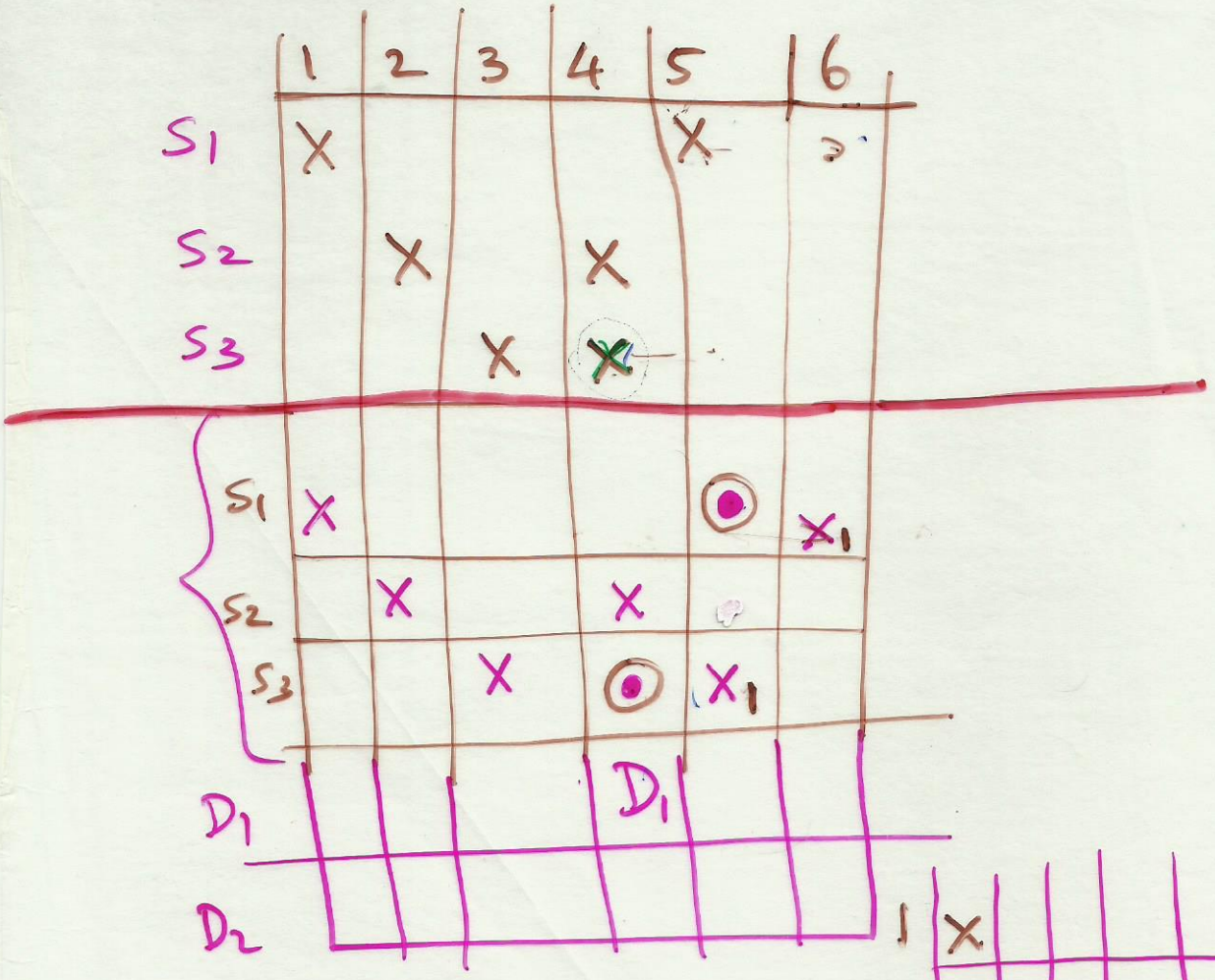
But in row 2, we have 2 checkmarks.

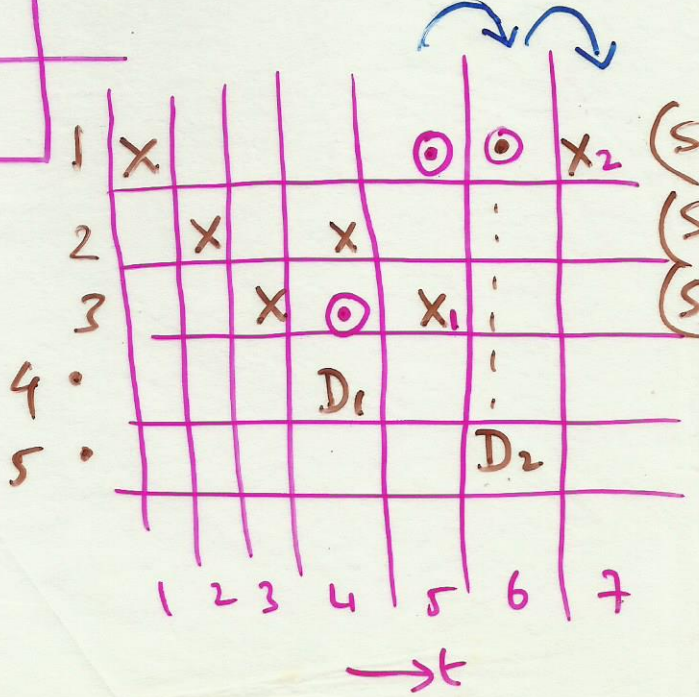
⇒ lower bound is 2

⇒ MAL = 3 is not optimal !

So, we go about inserting some non-computing delays.

→ a delay ($D_1$) after $S_3$ will delay both $X_1$ & $X_2$ operations one cycle beyond time 4



→ Delay elements are inserted as extra stages.

→ **Note:** All that we need to take care is that $S_1$ starts after $S_2$ & $S_3$ finish their computation. This fact must be preserved! (see the original RT & modified RT now!)

→ Modified table has $3+2=5$ rows
$5+2=7$ columns

Now determine the CV for this table.

100010

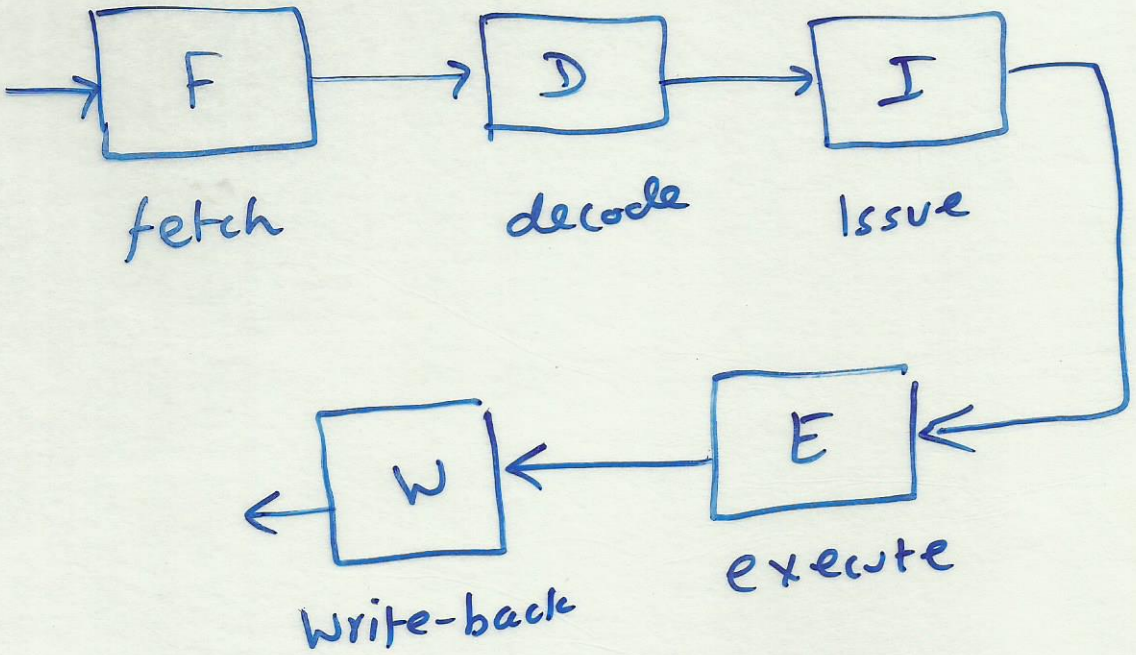
See the SD (pg: 278) ←

Verify the MAL given by greedy cycle (1,3) ⇒ $\frac{1+3}{2} = 2$

↑ lower bound for the MAL.

# Instruction Pipeline Design

- We will see some typical pipelines used in CISC & RISC scalar processors.




F: fetches instructions from cache, usually one per cycle

D: reveals the instruction function to be performed & identifies the resources needed

I: Reserves resources (GPRs, buses, etc) + Operands are read from the registers.

E : Execution stage

W : Writing back the results into the registers

* Memory loading/storing operations are usually treated as a part of E stage.

in-order execution: If an instruction is blocked from issuing due to any data/resource dependence/conflicts, all the instructions that follow are blocked [see fig 6.9 (b)]

Reordered issuing:

Issue all the required load operations in the beginning.

→ results in an improved time performance.

# Example of an out-of-order execution

(Delay due to data dependency).

I1) ADD R1, R2, R3 — C(R3) ← C(R1) + C(R2)

I2) MUL R3, R4, R5 — C(R5) ← C(R3) * C(R4)

I3) SUB R7, R2, R6 — C(R6) ← C(R7) − C(R2)

I4) INC R3 — C(R3) ← C(R3) + 1.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| I1 | FI | DE | EX | MEM | SR | | | | |
| I2 | | FI | DE | X | X | EX | MEM | SR | |
| I3 | | | FI | DE | EX | MEM | SR | | |
| I4 | | | | FI | DE | X | ~~MEM~~ | MEM | SR |

. MUL (I2) should not be executed in cycle 4 as the value of R3 it needs will not be stored in the reg. file by I1 & hvs not available.

→ data dependcy, called <u>data hazard</u>.

- But, I3 does not need R3 & hence, it can proceed without waiting (!!)

- I4 also needs R3. It can (EX) at clock cycle 6 as the result is available for use.

- But, ALU is being used in clock cycle 6 ⇒ it has to wait till cycle 7 to increment R3.

* Out-of-order execution. Sometimes Unacceptable!

Soln: lock the pipeline. Thus,

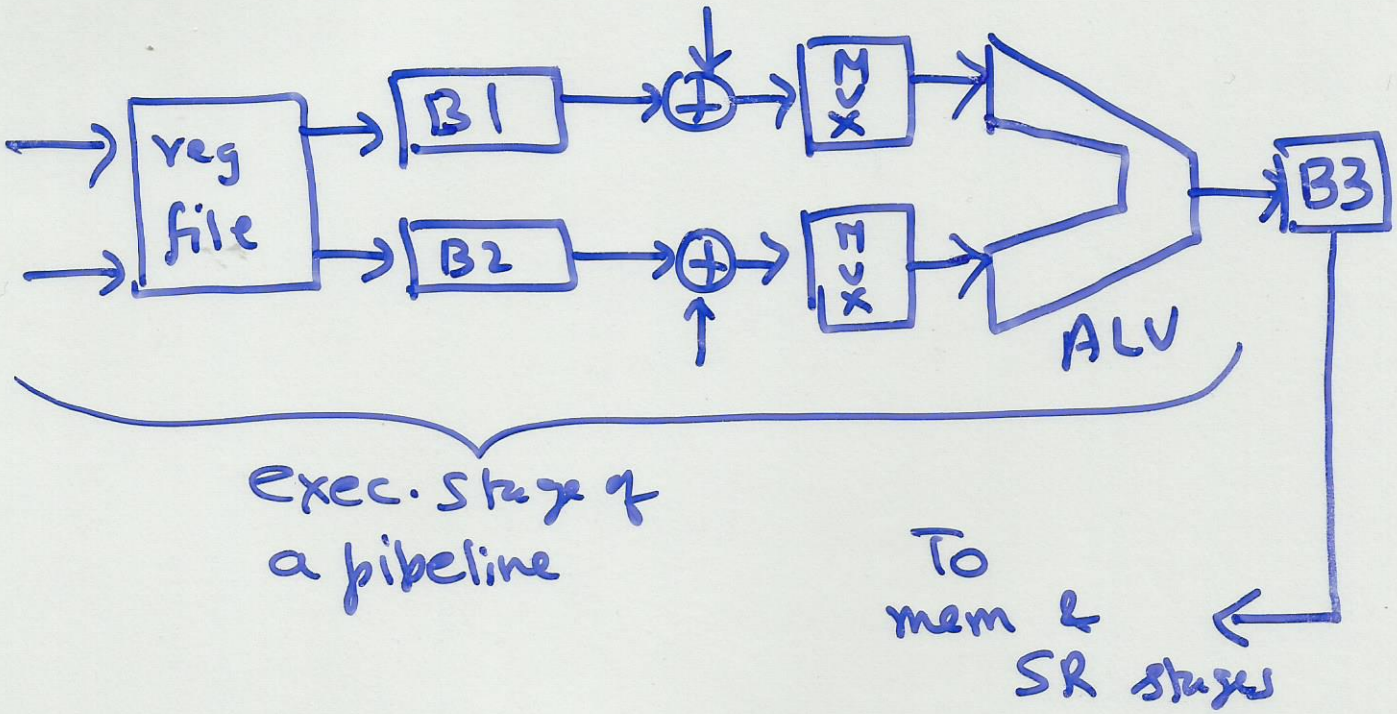
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | FI | DE | EX | MEM | SR | | | | | | |
| I2 | | FI | DE | X | X | EX | MEM | SR | | | |
| I3 | | | FI | (X) | (X) | DE | EX | MEM | SR | | |
| I4 | | | | (X) | (X) | FI | DE | EX | MEM | SR | |

finish time: 10 units

Here, the order (in-order) is preserved, but takes 1 cycle more!

# How to avoid pipeline delays due to data dependency?

H/w solution — register forwarding




- Key idea - instead of waiting till the SR stage, a special path is provided from B3 to ALU so that the result of the current inst. is made available to the next instr. This is referred to as reg. forwarding
  - Std feature in all pipelined processors

→ Calculation of timing is between the start of the first execution to the start of the last execution (in your book - ex. fig 6.9)
This is to eliminate the "start-up" delays in the pipeline.

Typically:

- load/store (E stage) : 4 clk cycles
- floating pt +, × : 3 clk cycles

These are typical CISC values!

→ Pay attention to fig 6.9.

→ Reading assignment → Example 6.4: [MIPS R4000 IP]

# Delays in pipeline due to resource Constraints

Suppose if one common memory is used for both data & instructions, then:

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| → | i | F | D | E | (M) | R/W | | | |
| | i+1 | | F | D | E | M | R/W | | |
| | i+2 | | | F | D | E | M | R/W | |
| → | i+3 | | | | (F) | D | E | M | R/W |

Either i or i+3 can be executed ⇒ "pipeline stalling"

## Loss of Speed-up?

Let f : fraction of the total # of instructions that are floating pt instructions that take (n+k) cycles.

m: # of instructions to be executed

n : # of stages in the pipeline.

Time to execute $m$ instructions with no pipelining $= m(1-f)\cdot n + m\cdot f(n+k)$

Time taken <u>with</u> pipelining:

$n + (m-1)\{(1-f) + kf\}$ ← first instruction

$\Rightarrow$ Speed-up =

$$\frac{m(1-f)n + m\cdot f(n+k)}{n + (m-1)\{(1-f)+kf\}}$$

$$= \frac{(n+kf)}{\left(\frac{n}{m}\right) + \left(\frac{m-1}{m}\right)\left((1-f)+kf\right)} \leftarrow$$

$$\simeq \frac{n}{1-f+kf} \qquad m \gg n, \quad n \gg kf.$$

So, if 10% of instructions are flo. pt. instructions, with each fl. point instr. takes 2 extra clk cycles,

$k=2,\ f=0.1$ Speed-up $= \frac{n}{0.9+0.2} = \underline{\underline{0.909\,n}}$

# Superscalar Performance

Compare the relative performance of a superscalar processor with that of a scalar base machine. [see pg: 159 for detn.]

N: # independent instructions ~~for work pipeline~~

for base scalar machine

$$T(1,1) = k + (N-1) \quad \text{(base cycles)}$$

$$T(m,1) = k + \frac{N-m}{m}$$

(m: # of pipelines)

k: time required to execute the first m instructions

$\left(\frac{N-m}{m}\right)$ ← remaining # of instructions, m instructions per cycle with m pipelines

$$\therefore S(m,1) = \frac{T(1,1)}{T(m,1)}$$

(Speed-up)

$$= \frac{N+k-1}{\frac{N}{m}+k-1}$$

$$= \frac{m(N+k-1)}{N+m(k-1)} \leftarrow$$

$\Rightarrow$ as $N \to \infty$, $S(m,1) \to m$, as expected

---

## Superpipelined Design:

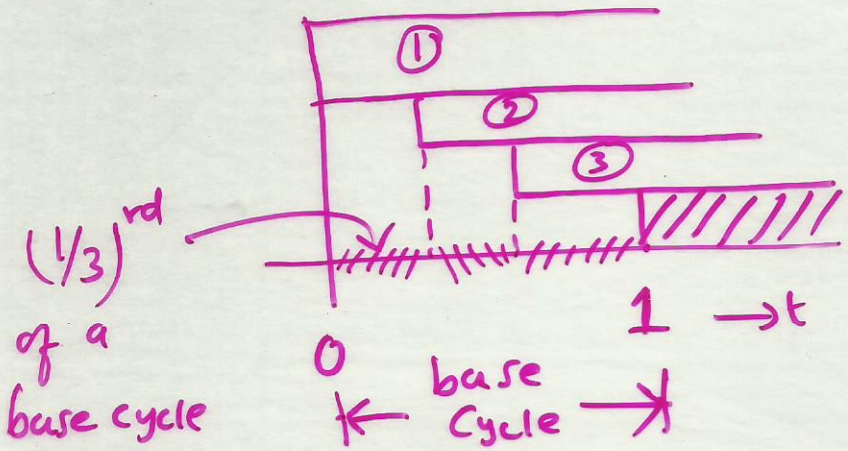
for a superpipelined machine of degree $\underline{n}$ with k stages in the pipeline

$$T(1,\underline{n}) = k + \frac{1}{n}(N-1) \quad \text{(base cycles)}$$

degree n: pipeline cycle time = $\left(\frac{1}{n}\right)$ of the base cycle.

Thus, $n = 3$

→ an instruction is issued per $(1/3)^{rd}$ of a base cycle




Speed-up $S(1,n) = \dfrac{T(1,1)}{T(1,n)}$

$$S(1,n) = \frac{k+N-1}{k+\frac{1}{n}(N-1)} = \frac{n(k+N-1)}{nk+N-1}$$

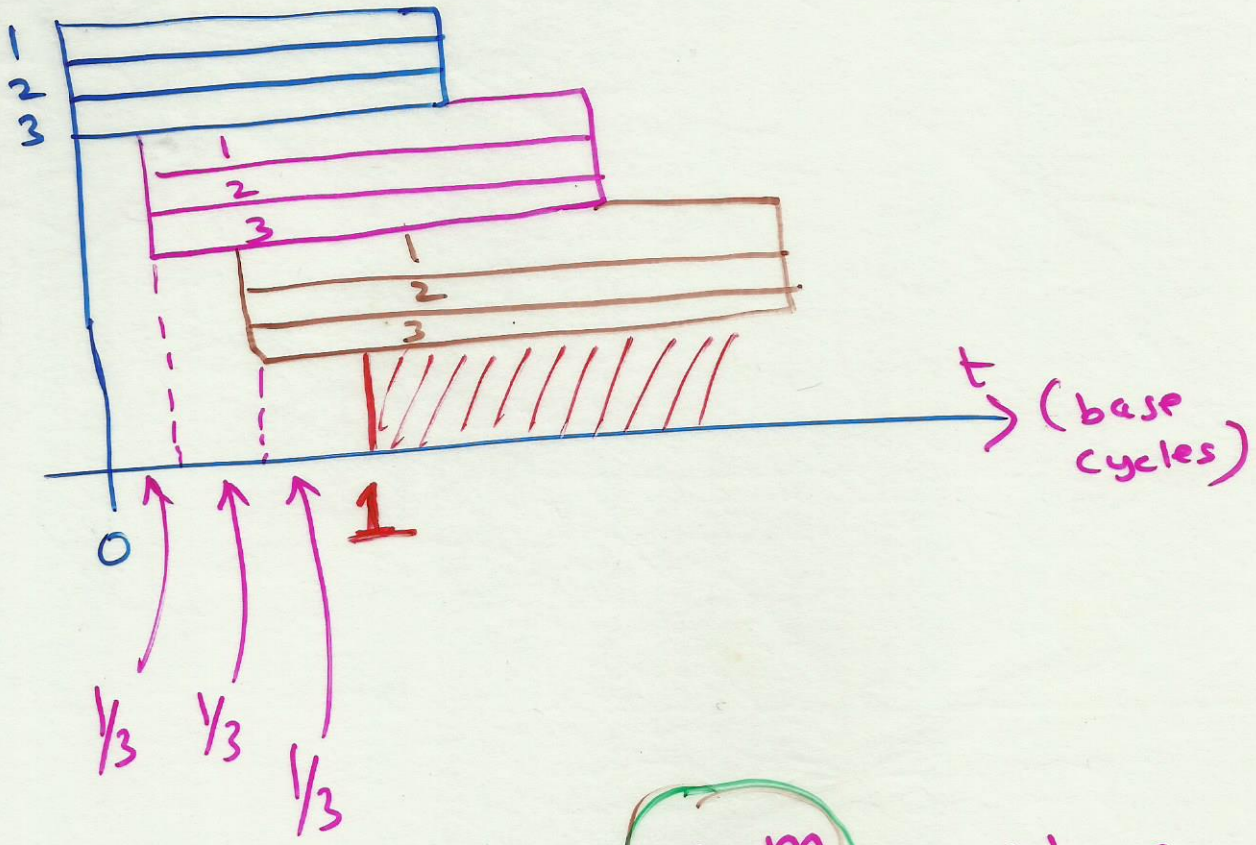
$\Rightarrow S(1,n) \to n$, as $N \to \infty$

# Superpipelined Superscalar Design

→ degree $(m, n)$

$m$ - instructions per cycle

pipeline cycle $(1/n)$ of the base cycle.




$$T(m,n) = k + \frac{N-m}{m \cdot n} \quad \text{(base cycles)}$$

$$S(m,n) = \frac{T(1,1)}{T(m,n)} = \frac{m \cdot n\,(k+N-1)}{m \cdot n \cdot k + N - m}$$

$\Rightarrow S(m,n) \to m \cdot n$ as $N \to \infty$, as expected!

# Performance Metrics

## Other flavors of "speed-up"

I : problem instance ; P : # of processors

Q : parallel program ; n : size of I

### (1) Relative Speed-up

$$\frac{\text{Time to solve I using program Q and 1 processor}}{\text{Time to solve I using program Q and P processors}}$$

Comments : Purely depends on the characteristics of the instance I being solved as well as the size of P

### (2) Real Speed-up

$$\frac{\text{Time to solve I using best serial program and 1 proc.}}{\text{Time to solve I using program Q and P processors}}$$

Comments : The fastest algorithm might not be known and no single algorithm might be fastest in all instances for some applications

## (3) Absolute Speed up

Time to solve I using best serial program and 1 fastest processor

---

Time to solve I using program Q and P processors

Comments: we can also use the best known sequential algorithm most often used in practice

## (4) Asymptotic Speed up

Asymptotic complexity of the best serial program

---

Asymptotic complexity of Q using as many processors as needed.

Comments: for problems such as sorting, where the asymptotic complexity is not uniquely characterized by the instance size $n$, we use the worst-case complexity

## (5) Asymptotic relative speed-up

### Asymptotic complexity of Q using 1 processor

### Asymptotic complexity of Q using as many processors as needed

Comments: - none -

Ref: IEEE Parallel & Distributed Technology "Performance Metrics: keeping the focus on Runtime" pp. 43-56, Spring 1996, by Saltaj Sahni & Venkat Jhanvantri