

Pipelined Processor Design

P. K. Roy

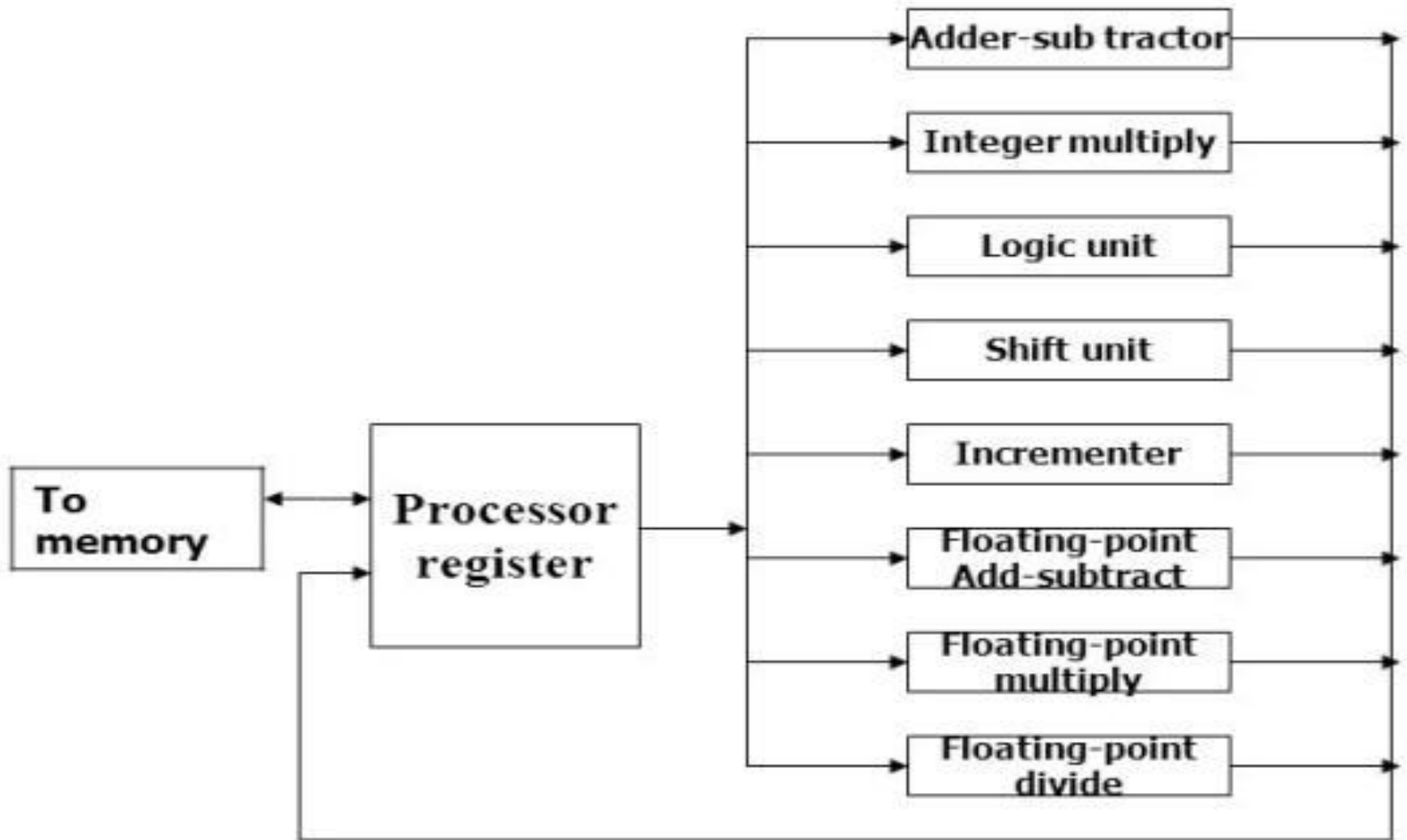
Asst. Professor

Siliguri Institute of Technology

Introduction

- Pipelining is one way of improving the overall processing performance of a processor
- A basic technique to improve performance - always applied in high performance systems.
- The operation of the processor are divided into a number of sequential actions.
- Each action is performed by a separate logic unit which are linked together in a “pipeline.”

Processors with multiple functional units

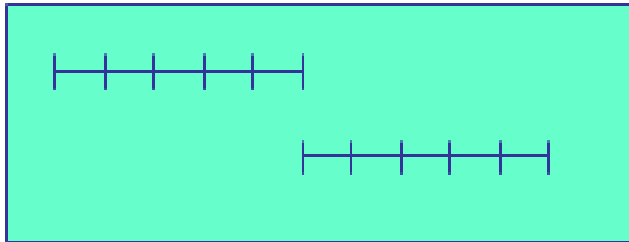


Types of Pipelined processors

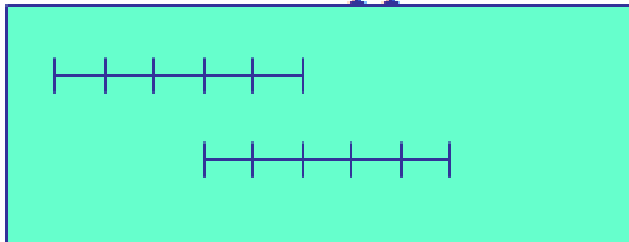
- Degree of overlap
 - Serial, Overlapped, Pipelined, Super-pipelined
- Depth
 - Shallow, Deep
- Structure
 - Linear, Non – linear
- Scheduling of operations
 - Static, Dynamic

Degree of overlap

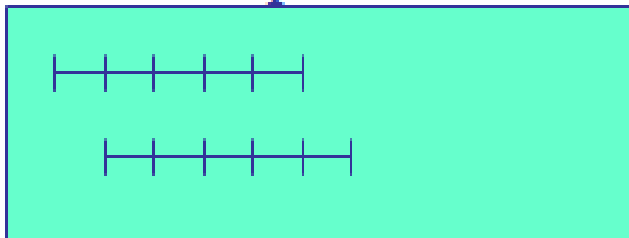
Serial



Overlapped

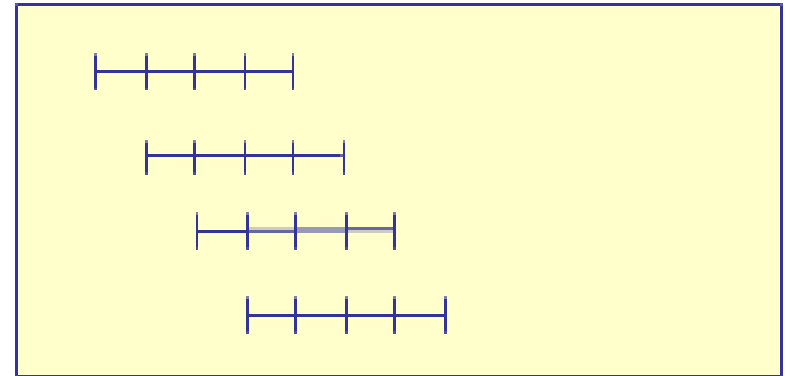


Pipelined

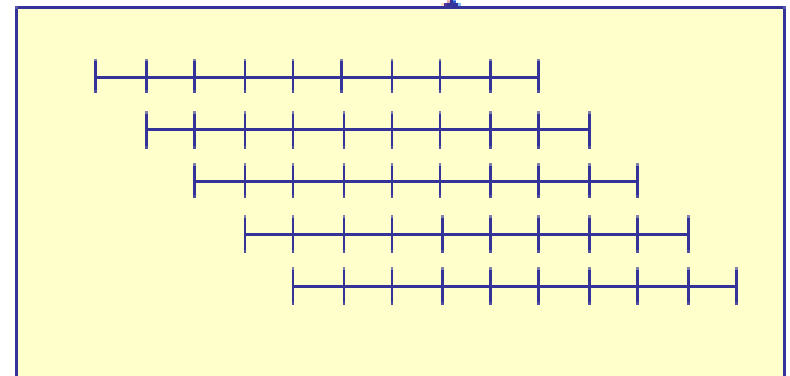


Depth

Shallow

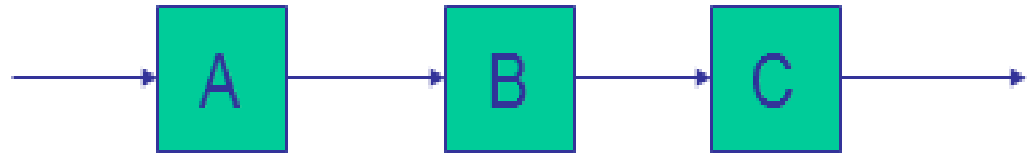


Deep

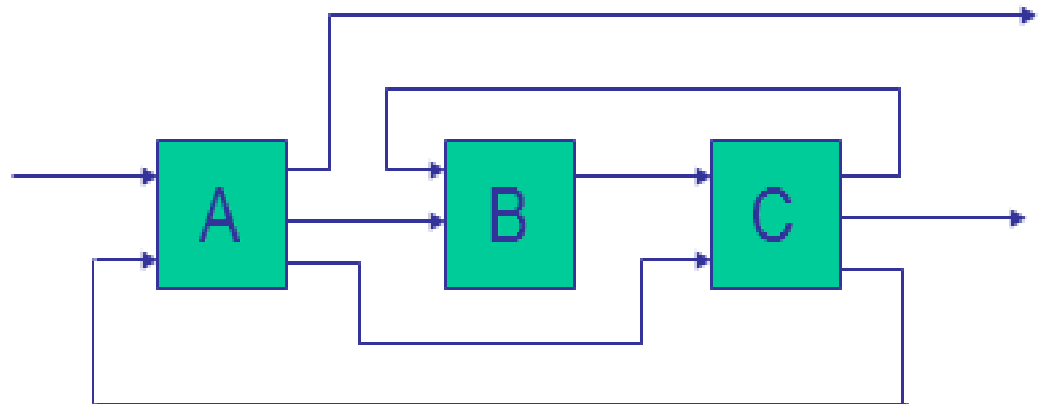


Pipeline Structure

Linear
Pipeline



Non-linear
Pipeline



Sequence: A, B, C, B, C, A, C, A

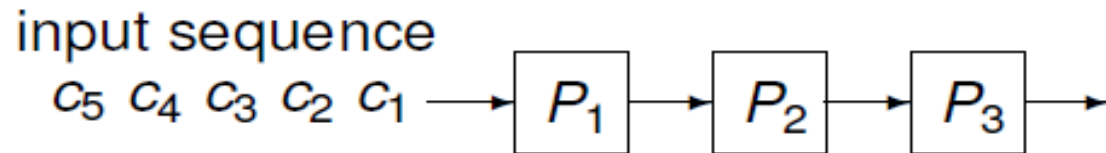
Scheduling/timing alternatives

- **Static**
 - same sequence of stages for all instructions
 - all actions in order
 - if one instruction stalls, all subsequent instructions are delayed
- **Dynamic**
 - above conditions are relaxed
 - higher throughput is achieved

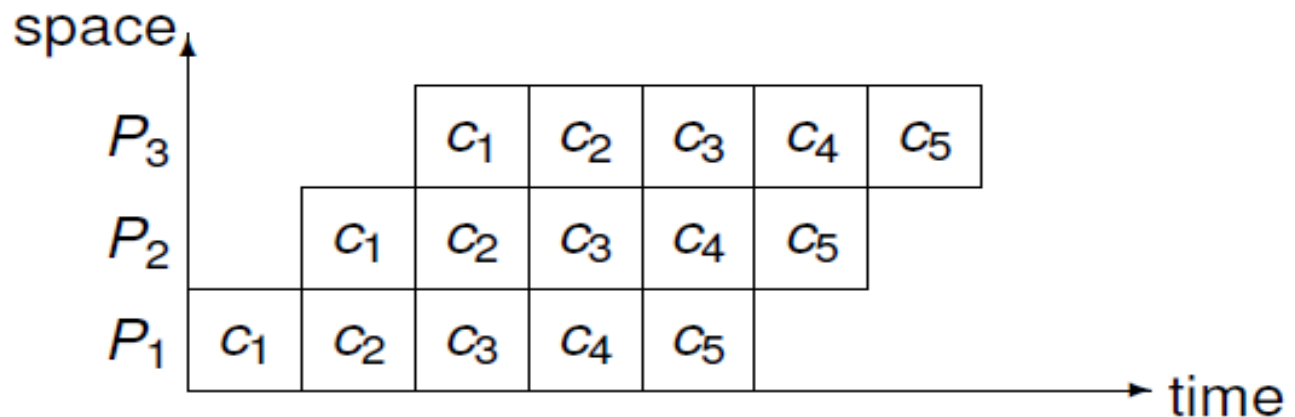
Space-Time Diagram

To represent the overlapped operations of a pipelined processor

Consider a simplified car manufacturing process in three stages:
(1) assemble exterior, (2) fix interior, and (3) paint and finish:



The corresponding space-time diagram is below:



After 3 time units, one car per time unit is completed.

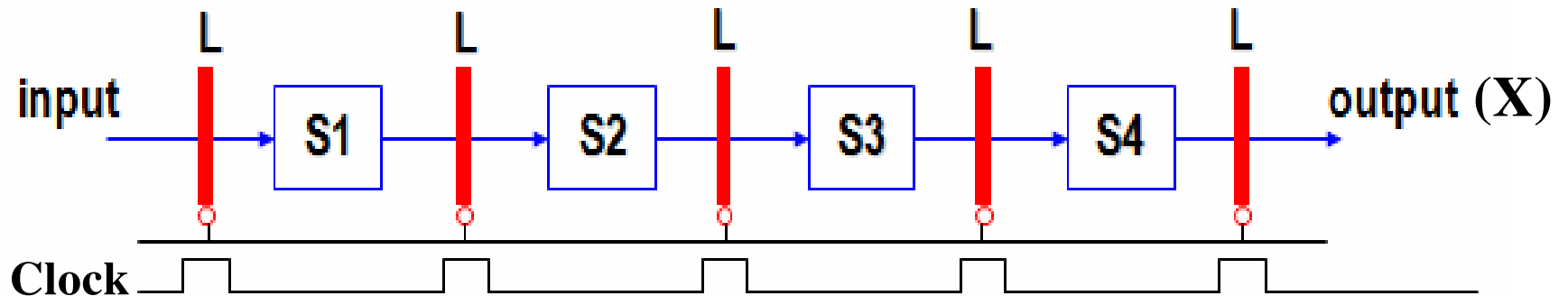
p-stage pipelines

- A pipeline with p processors is a p -stage pipeline.
 - Suppose every process takes one time unit to complete.
 - How long till a p -stage pipeline completes n inputs?
 - A p -stage pipeline on n inputs:
 - After p time units the first input is done.
 - Then, for the remaining $n - 1$ items, the pipeline completes at a rate of one item per time unit.
- $\Rightarrow p + n - 1$ time units for the p -stage pipeline to complete n inputs.
- A time unit is called a **pipeline cycle**.
 - The time taken by the first $p - 1$ cycles is the **pipeline latency**.

Reservation Table

- A 2D chart to show how successive parallel stages are utilized for a specific function (o/p) evaluation in successive cycles.
- Represents the flow of data through the pipeline for one complete evaluation of a given function.
- The total no. of clock units in the reservation table is the **evaluation time** for the given function.

Example (Linear/Static Pipeline)

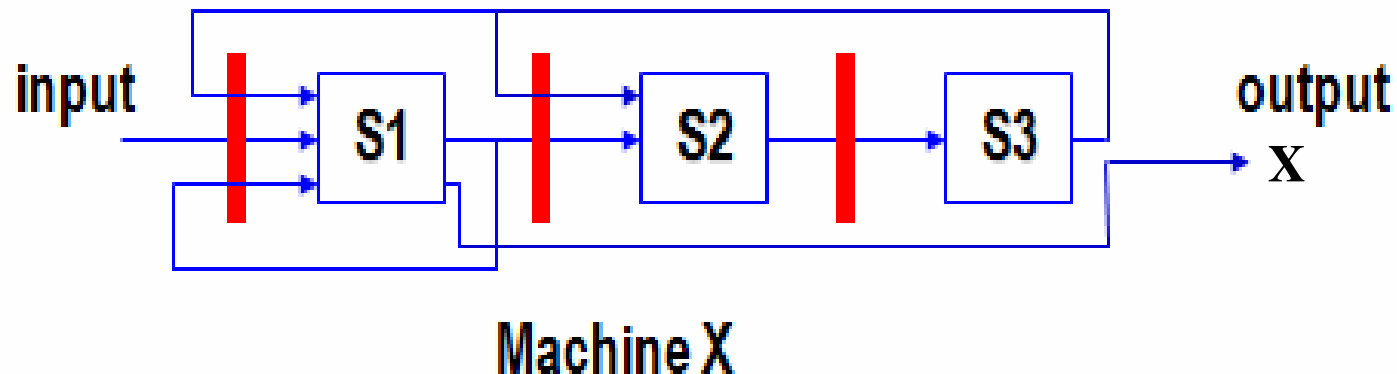


Time →

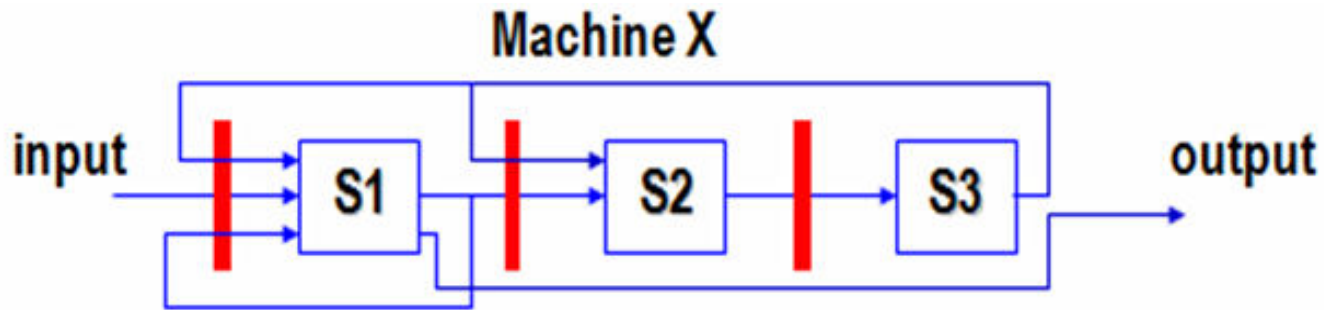
	1	2	3	4
Stage ↓	S1	X		
	S2		X	
	S3			X
	S4			

Non-linear (or Dynamic) Pipeline

- Stages might have different latencies
- Multiple paths out of one stage to other stages
- Multi-functional
- Multiple processors (k -stages) as linear pipeline
- Variable functions of individual processors
- Functions may be dynamically assigned
- Feed-forward and feedback connections



Reservation Table

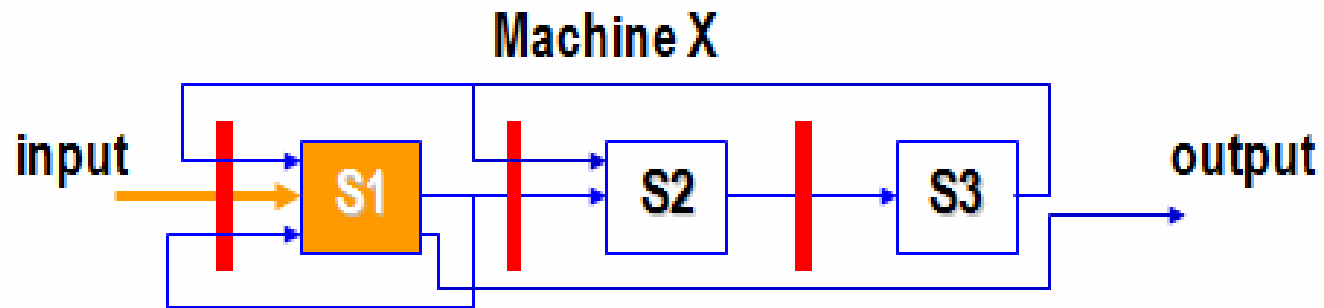


Reservation Table

Time →

Stage →		0	1	2	3	4	5	6	7
	S1								
	S2								
	S3								

Reservation Table

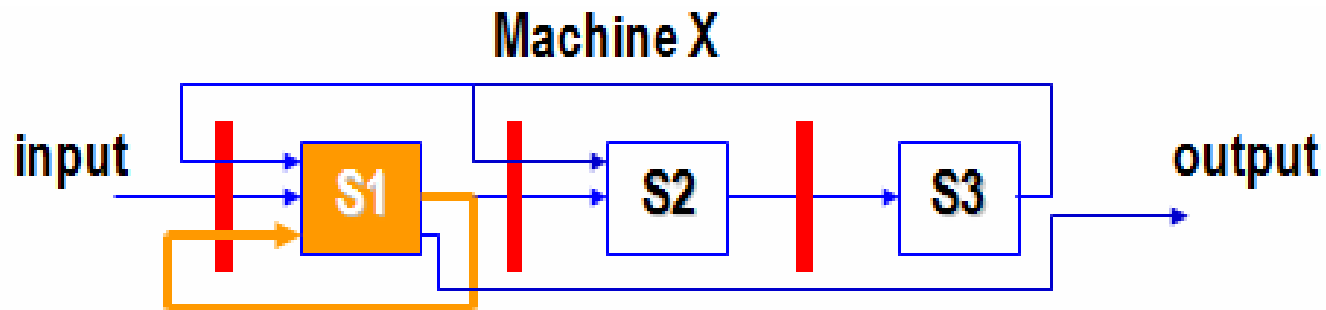


Reservation Table

Time →

	0	1	2	3	4	5	6	7
Stage →								
S1	X							
S2								
S3								

Reservation Table

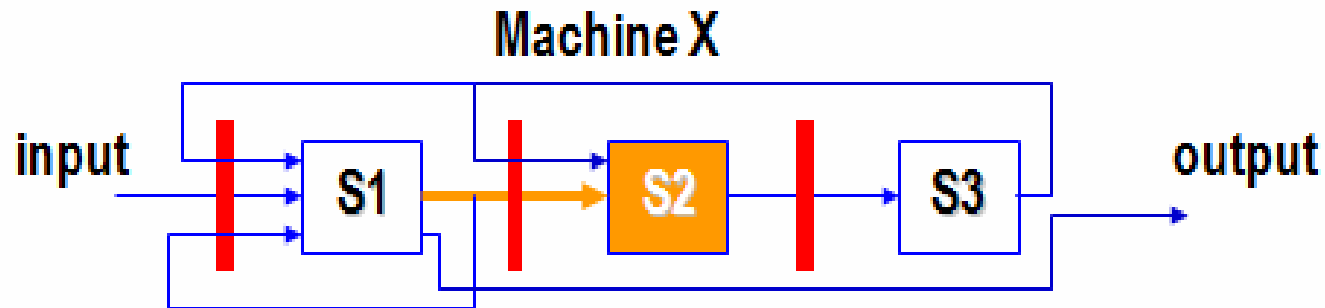


Reservation Table

Time →

Stage →		0	1	2	3	4	5	6	7
	S1	X	X						
	S2								
	S3								

Reservation Table

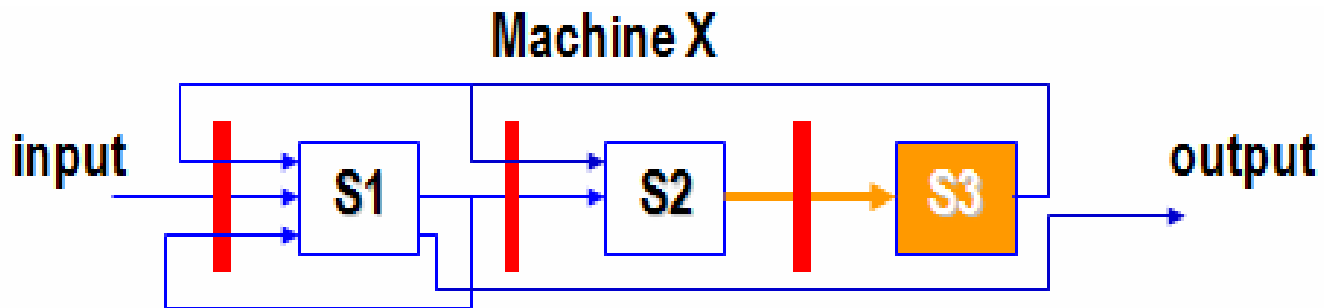


Reservation Table

Time →

Stage →		0	1	2	3	4	5	6	7
	S1	X	X						
	S2			X					
	S3								

Reservation Table

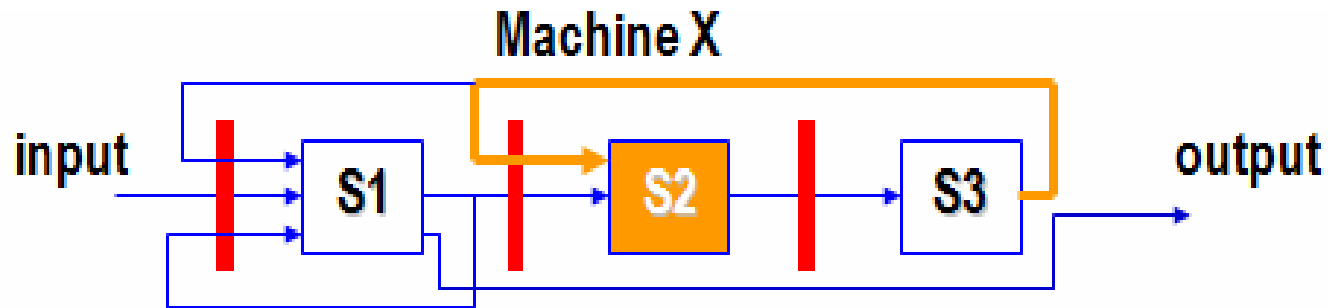


Reservation Table

Time →

	0	1	2	3	4	5	6	7
S1	X	X						
S2			X					
S3				X				

Reservation Table

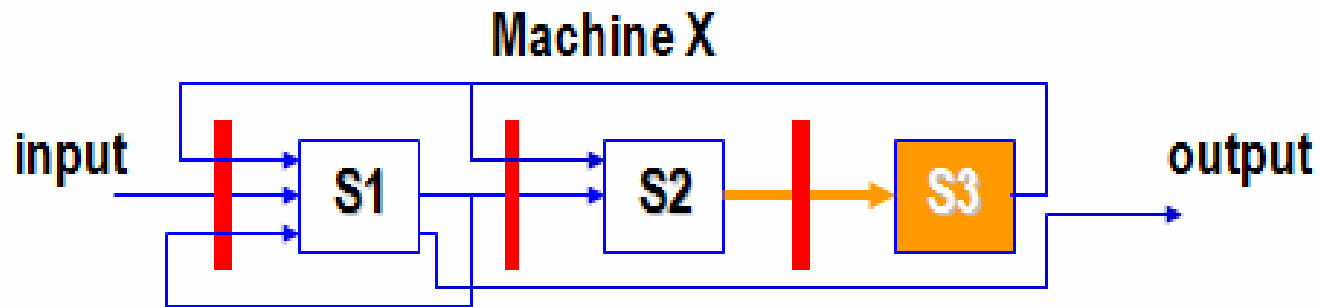


Reservation Table

Time →

	0	1	2	3	4	5	6	7
Stage →	S1	X	X					
S2			X		X			
S3				X				

Reservation Table



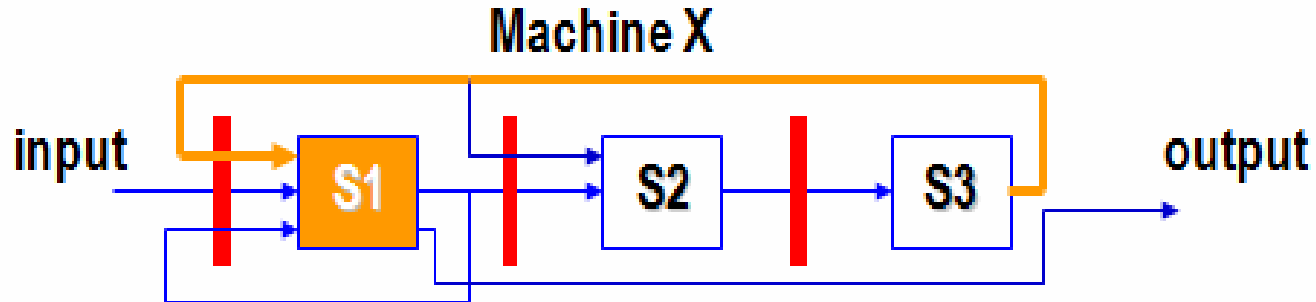
Reservation Table

Time →

Stage →

	0	1	2	3	4	5	6	7
S1	X	X						
S2			X		X			
S3				X		X		

Reservation Table

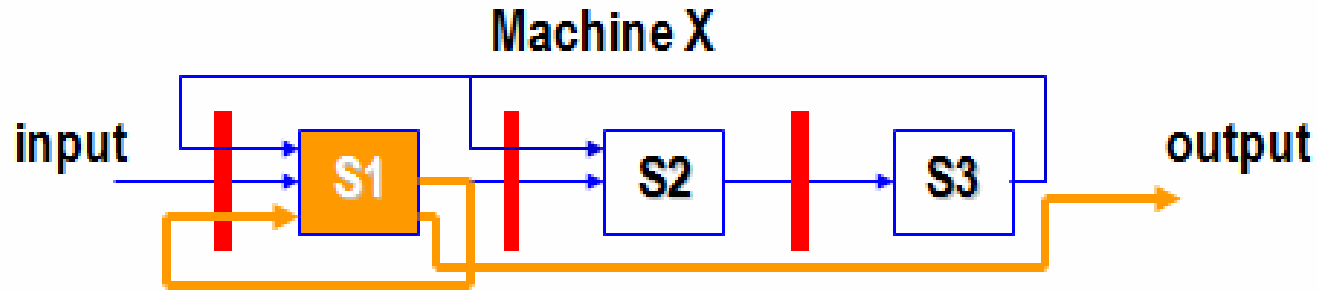


Reservation Table

Time →

Stage →		0	1	2	3	4	5	6	7
	S1	X	X					X	
	S2			X		X			
	S3				X		X		

Reservation Table



Reservation Table

Time →

Stage →		0	1	2	3	4	5	6	7
	S1	X	X					X	X
	S2			X		X			
	S3				X		X		

Exercise 1: Design the pipeline for both cases.

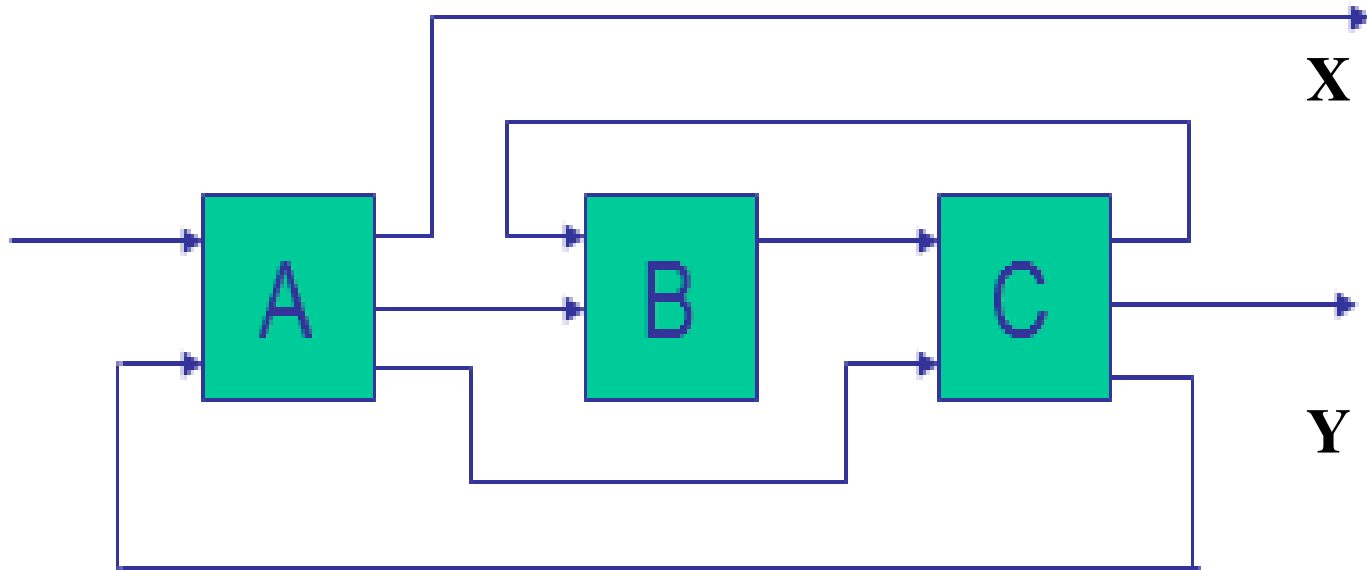
1.

X					X		X
	X		X				
		X		X		X	

2.

	1	2	3	4	5	6	7
A	X		X	X			
B		X				X	
C					X		X
D				X			

Non-linear Pipeline With More Than One Functions



2 Streamline Connections
2 Feedback Connections
1 Feedforward Connection

Reservation Table

Reservation Table for X

	1	2	3	4	5	6	7	8
A	X					X		X
B		X		X				
C			X		X		X	

- Columns represent the evaluation time for a given function
- Multiple checkmarks in a row, means repeated usage of the same stage in different cycles

Reservation Table for X
for Y

	1	2	3	4	5	6	7	8
A	X				Y	X		X
B		X	Y	X				
C		Y	X	Y	X	Y	X	

Latency Analysis

- **Latency:** The number of time units (clock cycles) between two initiations of a pipeline is the *latency between them*.
- A latency value k means that two initiations are separated by k clock cycles.
- Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.
- A collision implies resource conflicts between two initiations in a pipeline.
- Latencies that cause collision are called *forbidden latencies*.
- Some latencies cause collision, some not.

Latency Analysis

- Maximum forbidden latency can be m
 - If n = no. of columns in the reservation table then $m \leq n-1$
- All the latencies greater than $m+1$ do not cause collisions.
- **Permissible Latency** p , lies in the range:
 - $1 \leq p \leq m-1$
 - Value of p should be as small as possible
 - Permissible latency $p=1$ corresponds to an ideal case, can be achieved by a static pipeline

Latency Analysis

- **Latency Sequence:** a sequence of permissible latencies between successive initiations.
- **Latency Sequence Length:** the number of time intervals within the cycle.
- **Latency Cycle:** a latency sequence that repeats the same subsequence (cycle) indefinitely and without collision.
e.g. Latency Sequence: 1, 8
Latency Cycle $\rightarrow (1,8) \rightarrow 1, 8, 1, 8, 1, 8 \dots$
- **Minimum Average Latency (MAL of a latency cycle):**
sum of all latencies / number of latencies along the cycle
- **Constant Cycle:** One latency value.
- **Design Goal:** Obtain the shortest average latency between initiations without causing collisions.

Collision with latency 2 in evaluating X

	1	2	3	4	5	6	7	8	9	10	11
A	1		2		3	1	4	1,2	5	2,3	6
B		1		1,2		2,3		3,4		4,5	
C			1		1,2		1-3		2-4		

Collision with latency 5 in evaluating X

	1	2	3	4	5	6	7	8	9	10	11
A	1					1,2		1			2,3
B		1		1			2		2		
C			1		1		1	2		2	

Collision Vectors

- Combined set of permissible and forbidden latencies.
- m-bit binary vector $C = (C_m C_{m-1} \dots C_2 C_1)$
- The value of $C_i = 1$ if the latency i causes a collision;
- $C_i = 0$ if the latency i is permissible.
- $C_m = 1$, always; it corresponds to the maximum forbidden latency $m \leq n-1$.

Reservation Table for X

	1	2	3	4	5	6	7	8
A	X					X		X
B		X		X				
C			X		X		X	

Forbidden latencies for function X are 2,4,5,7
Latencies 1,3,6 do not cause collision.

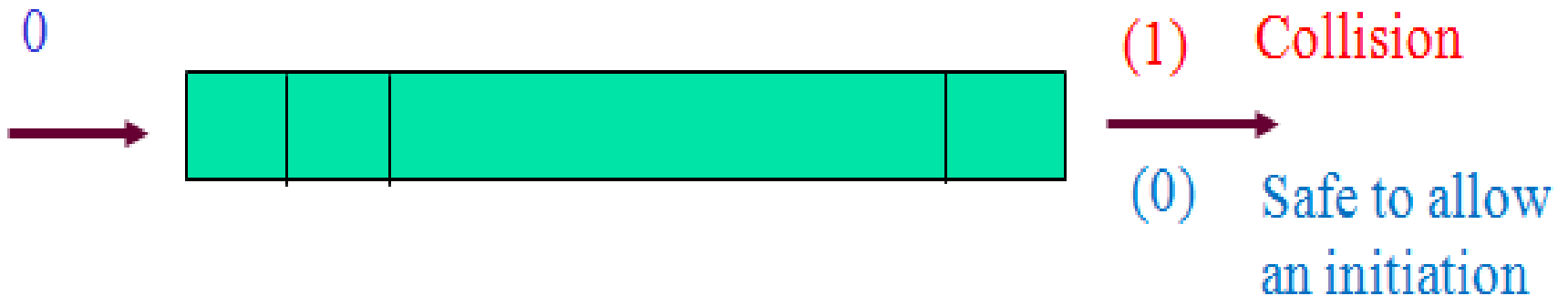
Therefore, the collision vector is = **1011010**

State Diagram

- It specifies the permissible state transitions among successive initiations.
- Collision vector corresponds to the initial state at time $t = 1$ (**initial collision vector**).
- The next state comes at time $t + p$, where p is a permissible latency in the range $1 \leq p < m$.

Right Shift Register

- The next state of the pipeline at time $t+p$ can be obtained by using a bit-right shift register.
- Initial CV is loaded into the register.
- The register is then shifted to the right
 - When a 0 emerges from the right end after p shifts, p is a permissible latency.
 - When a 1 emerges, the corresponding latency should be forbidden.
- Logical 0 enters from the left end of the shift register.



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

The Next State

- The next state after p shifts is obtained by bitwise-ORing the initial CV with the shifted register contents.
- This bitwise-ORing of the shifted contents is meant to prevent collisions from the future initiations starting at time $t+1$ and onward.

C.V. = 1 0 1 1 0 1 0 \Rightarrow first state

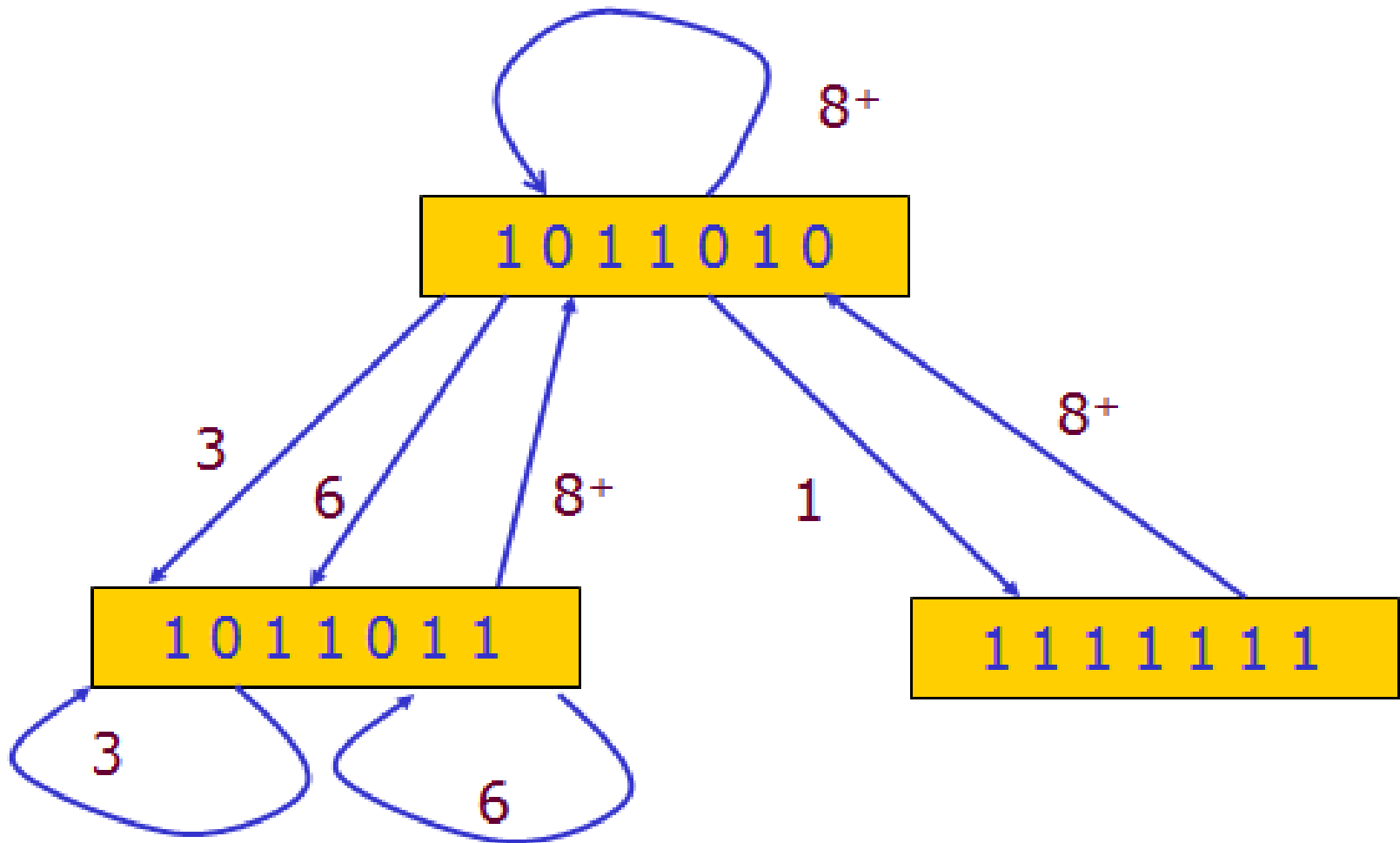
0 1 0 1 1 0 1 \Rightarrow C.V. 1-bit right shifted

1 0 1 1 0 1 0 \Rightarrow Initial C.V.

----- OR

1 1 1 1 1 1 1 \Rightarrow New CV

State Diagram for X



HW: Find the state diagram for Y

Cycles

- **Simple cycles:** each state appears only once
(3), (6), (8), (1, 8), (3, 8), and (6,8)
- **Greedy Cycles:** simple cycles whose edges are all made with minimum latencies from their respective starting states

(1,8), (3): one of them is MAL

Bounds on MAL

- MAL is lower-bounded by the maximum number of checkmarks in any row of the reservation table.
- MAL is lower than or equal to the average latency of any greedy cycle in the state diagram.
- Average latency of any greedy cycle is upper-bounded by the number of 1's in the initial CV plus 1.

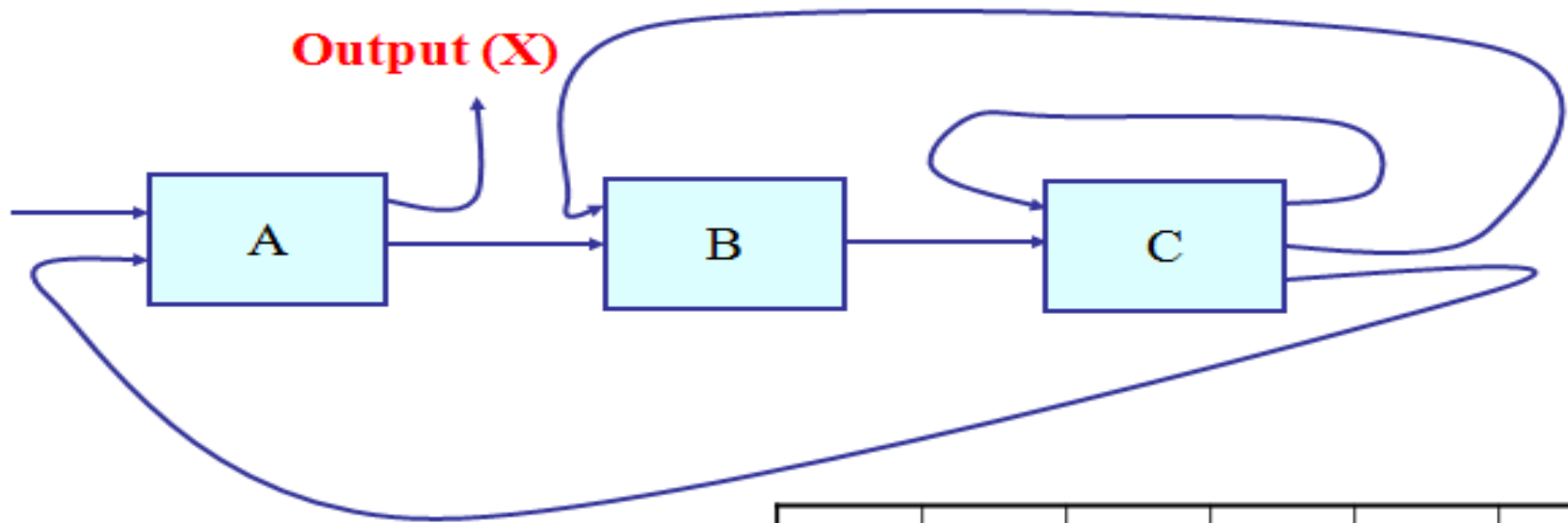
Collision-free scheduling

- Finding Greedy cycles from the set of Simple cycles.
- The Greedy cycle yielding the MAL is the final choice.

Optimization technique

- Insertion of Delay stages
 - Modification of reservation table
 - New CV
 - Improved state diagram
- To yield an optimal latency cycle
- Optimal latency cycle is selected from one of the lowest greedy cycles.

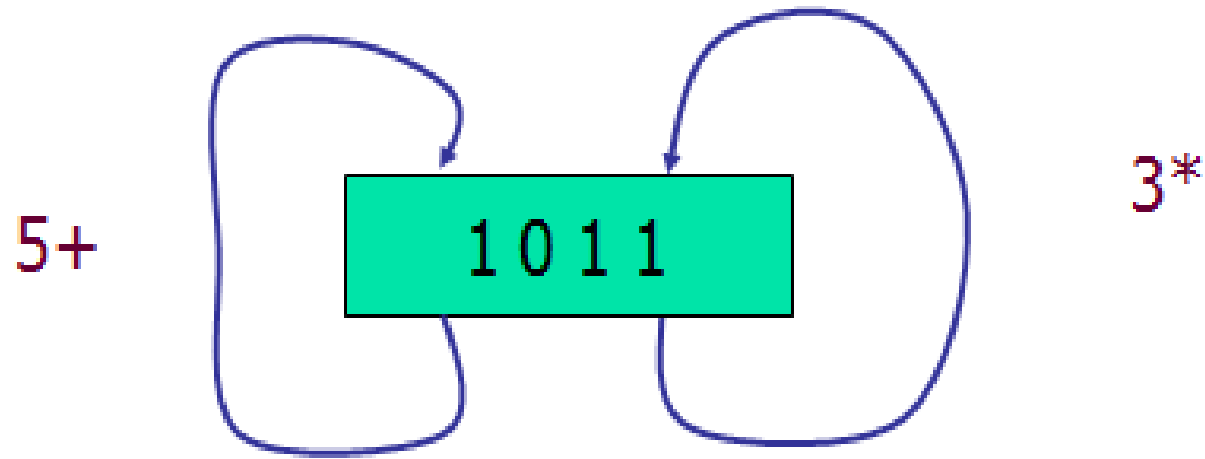
Example



	1	2	3	4	5
A	X				X
B		X		X	
C			X	X	

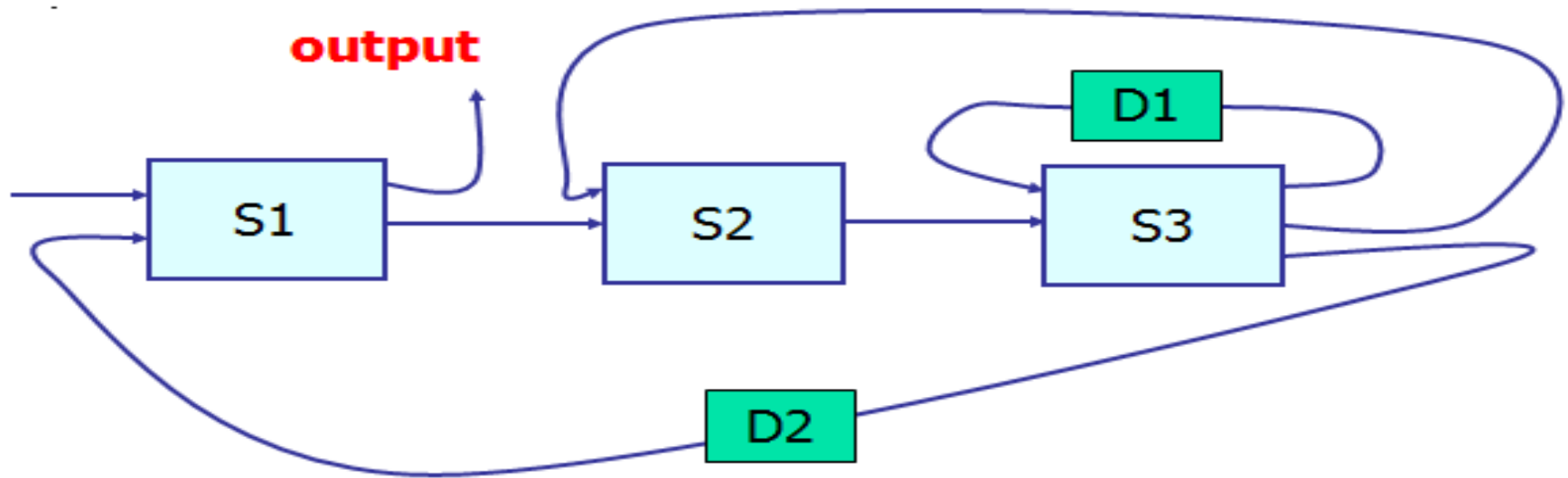
Forbidden Latencies: 1, 2, 4
C.V. → 1 0 1 1

Example(cntd.): State Diagram



MAL = 3

Example(cntd.): Delay Insertion



HW: Find the State Diagram

	1	2	3	4	5	6	7
S1	X						X
S2		X		X			
S3			X		X		
D1				X			
D2						X	

Forbidden: 2, 6
C.V. \rightarrow 1 0 0 0 1 0

Exercise 2:

	1	2	3	4
S1	x			x
S2		x		
S3			x	

- Find out
- (i) Forbidden latency
 - (ii) Initial collision vector
 - (iii) Draw the state diagram for scheduling the pipeline
 - (iv) Greedy cycle and MAL
 - (v) Bounds on MAL

Exercise 3:

	1	2	3	4	5	6
S1	x					x
S2		x			x	
S3			x			
S4				x		
S5		x				x

- Find out
- (i) Forbidden latency
 - (ii) Initial collision vector
 - (iii) Draw the state diagram for scheduling the pipeline
 - (iv) Greedy cycle and MAL
 - (v) Bounds on MAL

Classification of Pipelined Processors

- **Arithmetic Pipeline**
 - Divides an arithmetic operation into several steps each of which executed one-by-one.
 - e.g. 4-stage pipeline used in Star-100,
8-stage pipeline used in TI-ASC
- **Instruction Pipeline**
 - Basic stages are Fetch-Decode-Execute but can be expanded.
 - e.g. all high performance computers
- **Processor Pipeline**
 - processing of same data stream by cascade of processors
 - no practical example found

Arithmetic Pipeline

- combined multiply and add operations with a stream of numbers:

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \dots, 7$$

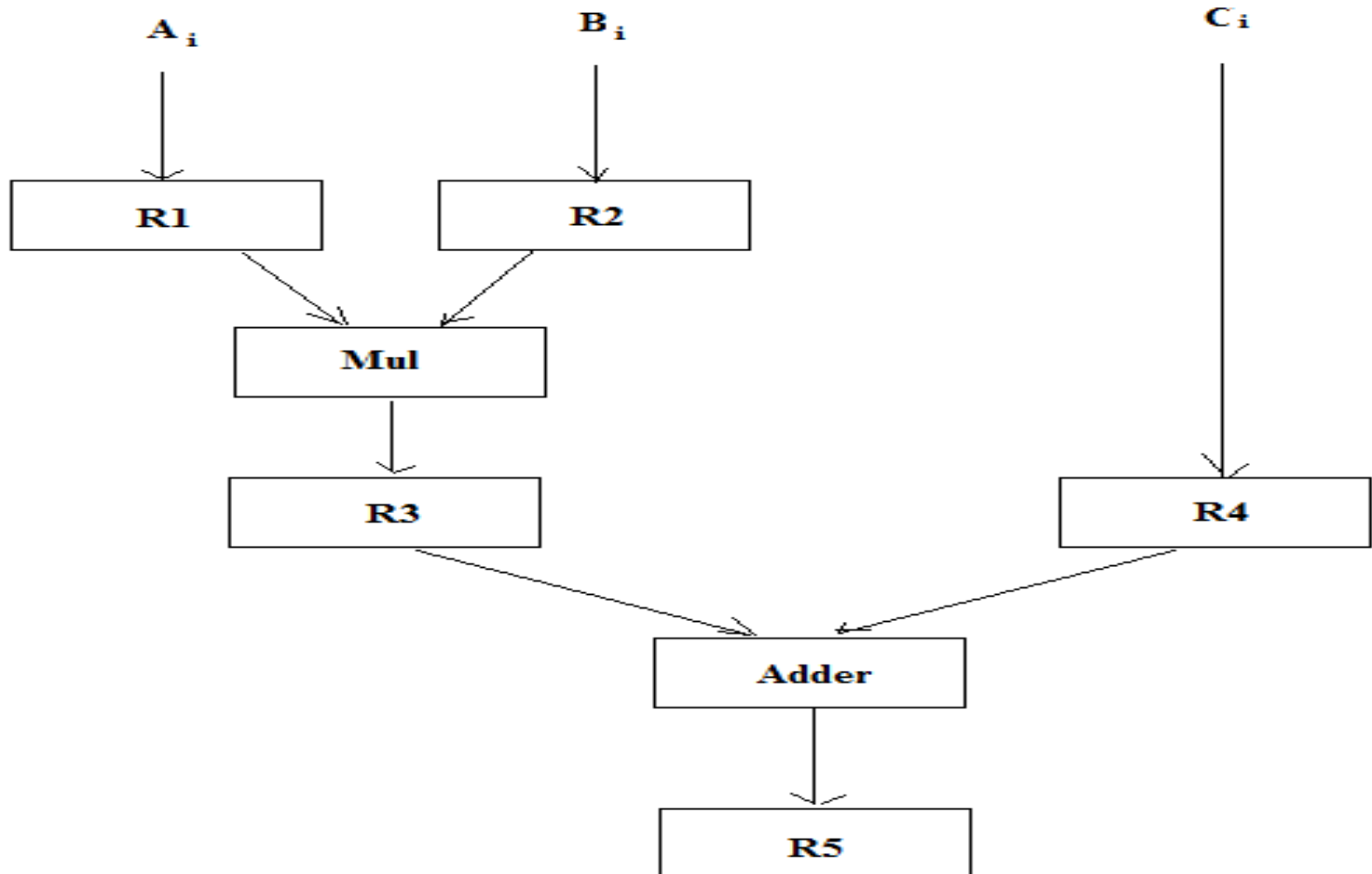
- The suboperations performed in each segment of the pipeline are as follows:

$$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$$

$$R3 \leftarrow R1 * R2 \quad R4 \leftarrow C_i$$

$$R5 \leftarrow R3 + R4$$

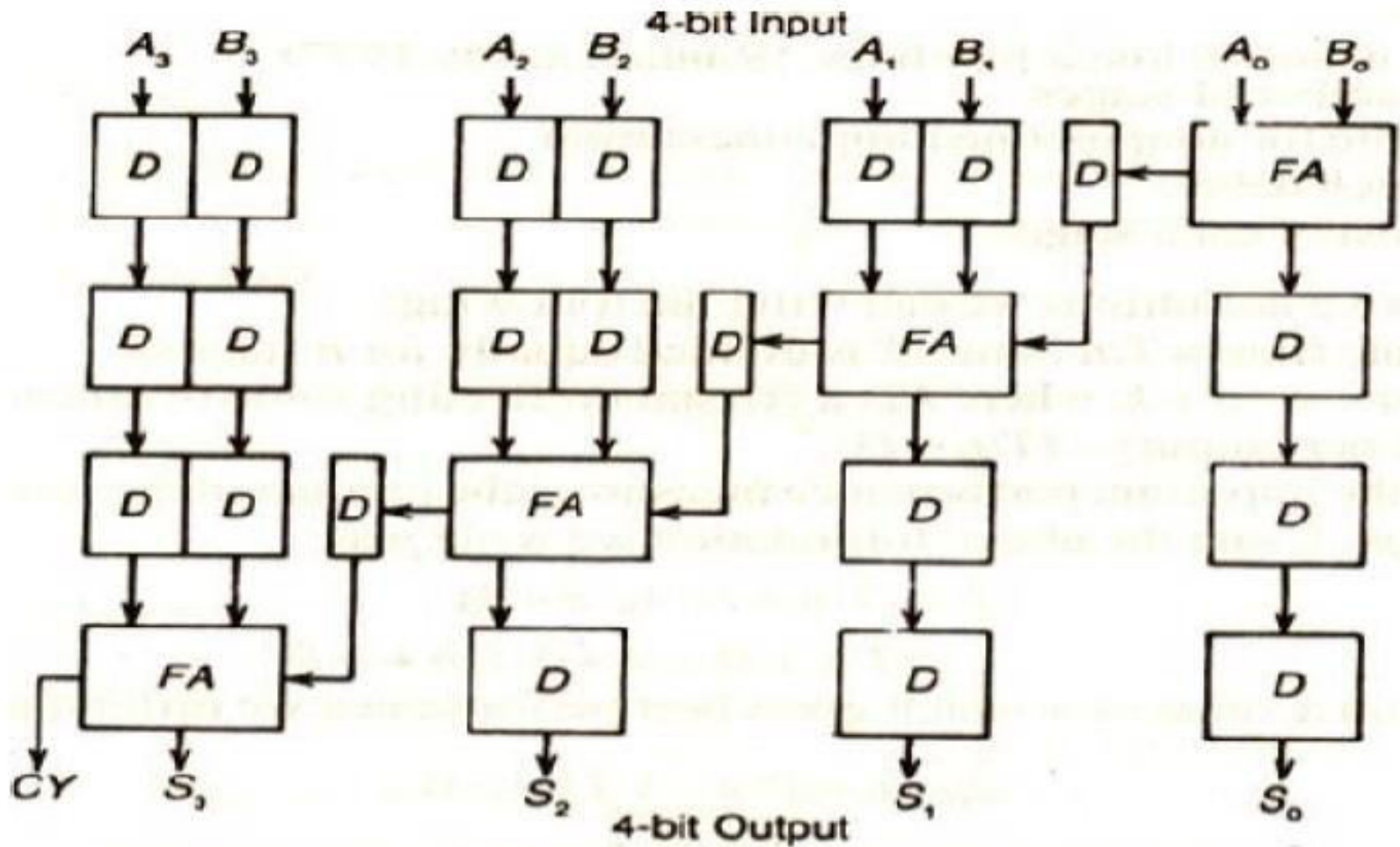
Arithmetic Pipeline



Arithmetic Pipeline

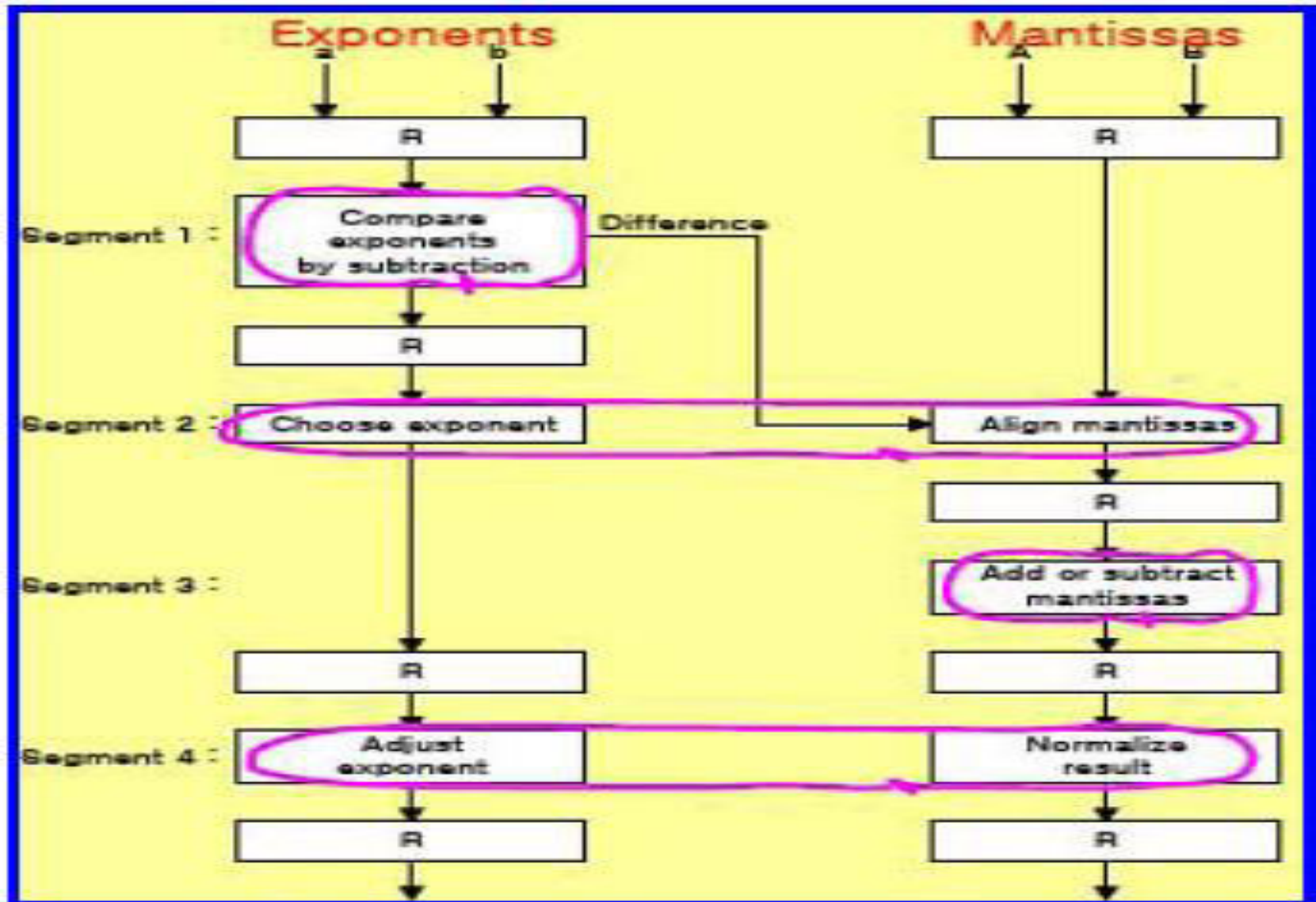
Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Fixed Point Addition Pipeline

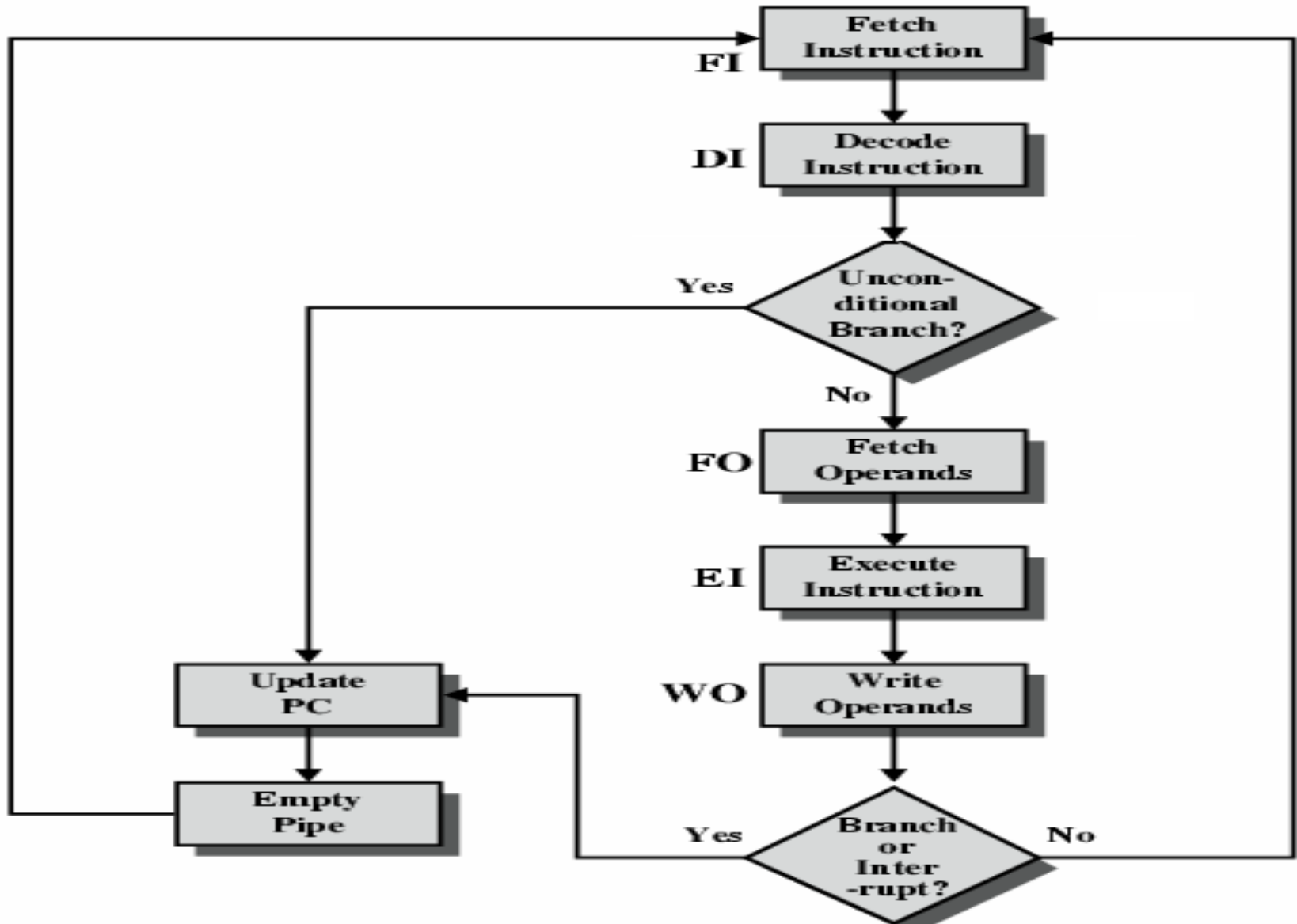


D = D flip-flops, which are all clocked by a common clock
 FA = Full Adder

Floating Point Addition/Subtraction Pipeline



5-Stage Instruction Pipeline



Self-review: Pipelining from CS303

Five stages in the datapath of MIPS

Five stage “RISC” load-store architecture

1. Instruction fetch (IF)

- get instruction from memory, increment PC

2. Instruction Decode (ID)

- translate opcode into control signals and read registers

3. Execute (EX)

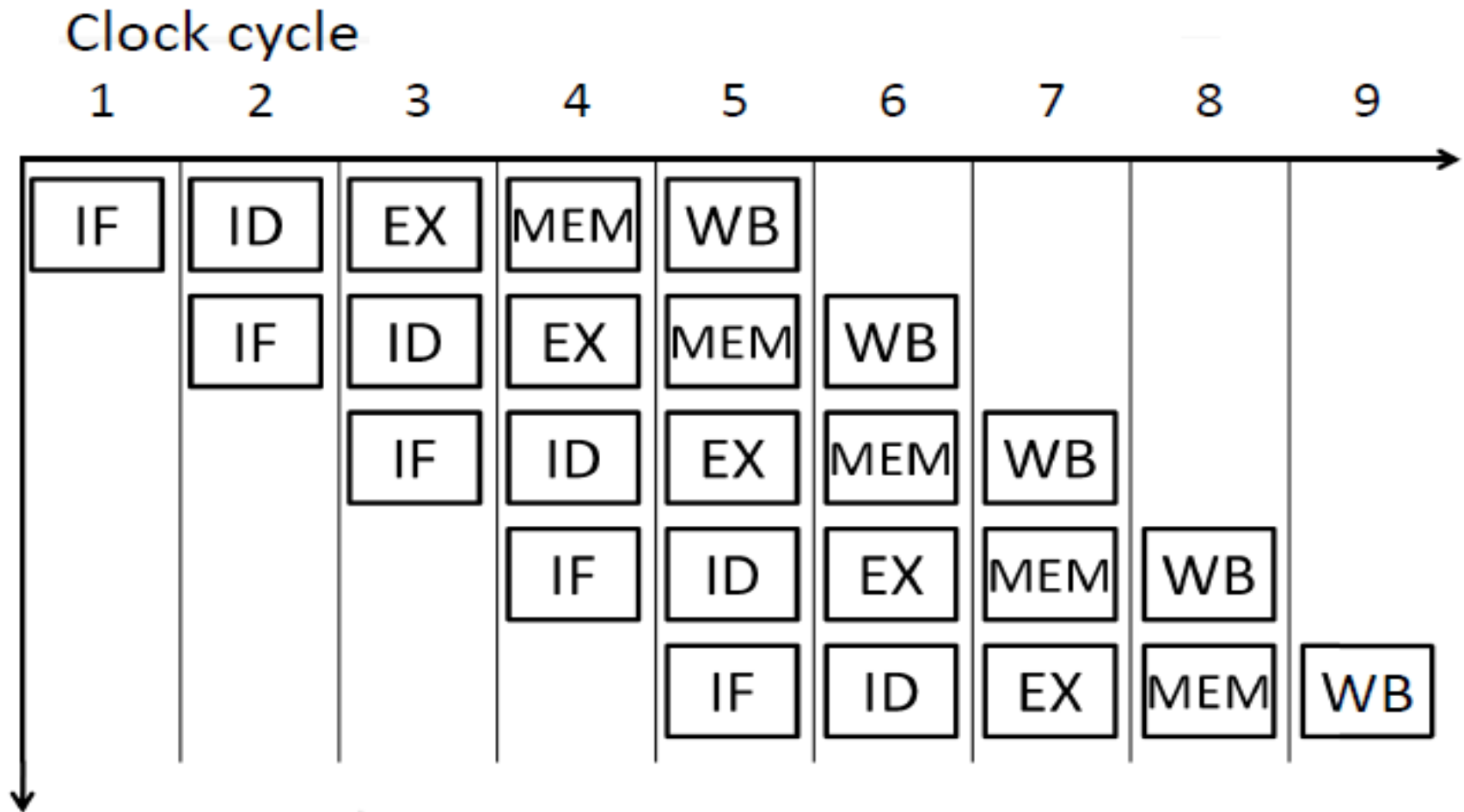
- perform ALU operation, compute jump/branch targets

4. Memory (MEM)

- access memory if needed

5. Writeback (WB)

- update register file



Latency: **5 cycles**

Throughput: **1 Instruction/cycle**

Concurrency: **5**

CPI : 1

Pipeline Conflicts/Hazards

- Data Dependency => Data hazard
- Branch Difficulties => Control hazard
- Resource Conflicts => Structural hazard

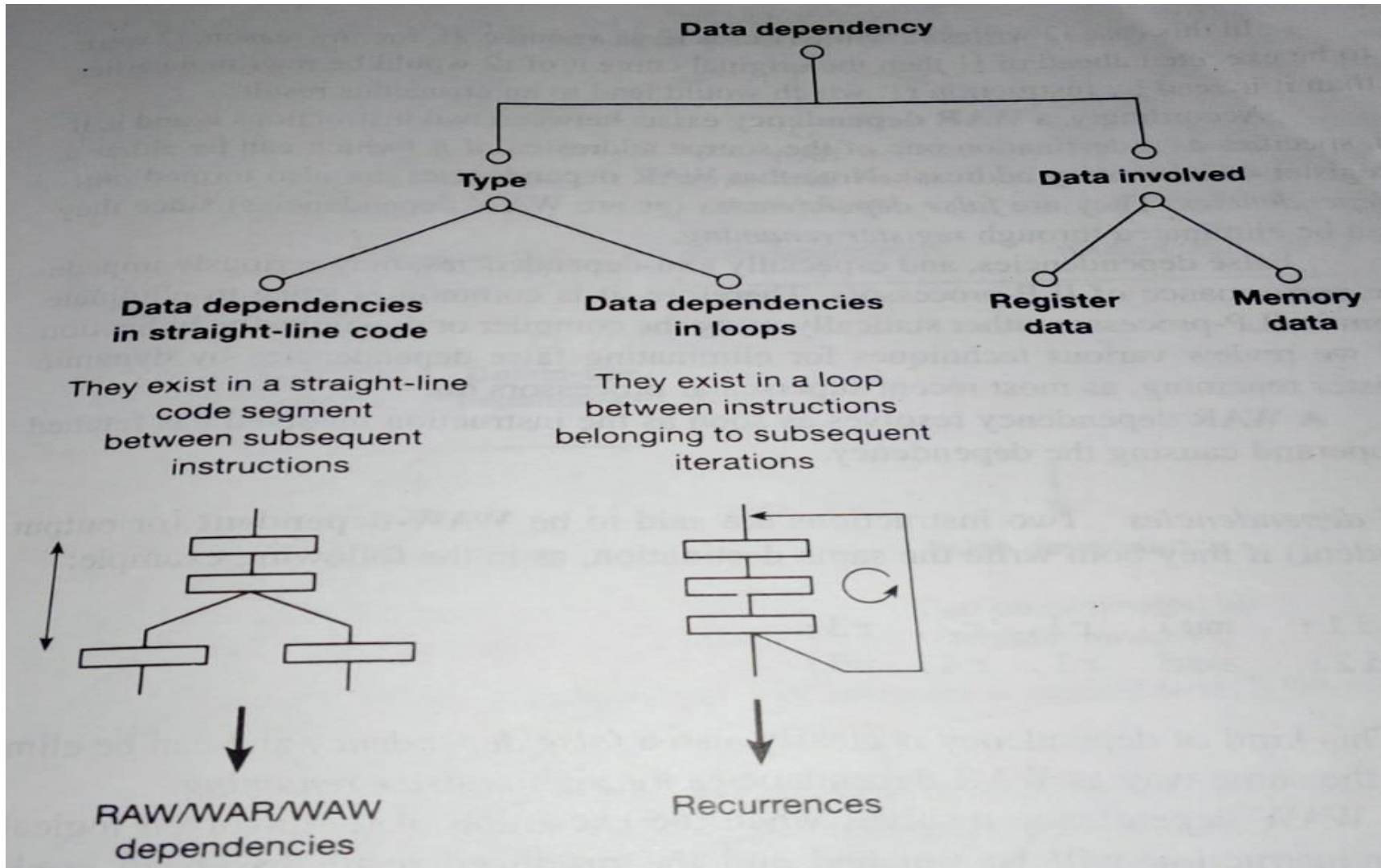
Data Hazard

Consider the instruction sequence:

ADD R1, R2, R3; results in R1
SUB R4, R5, R1;
AND R6, R1, R7;
OR R8, R1, R9;
XOR R10, R1, R11;

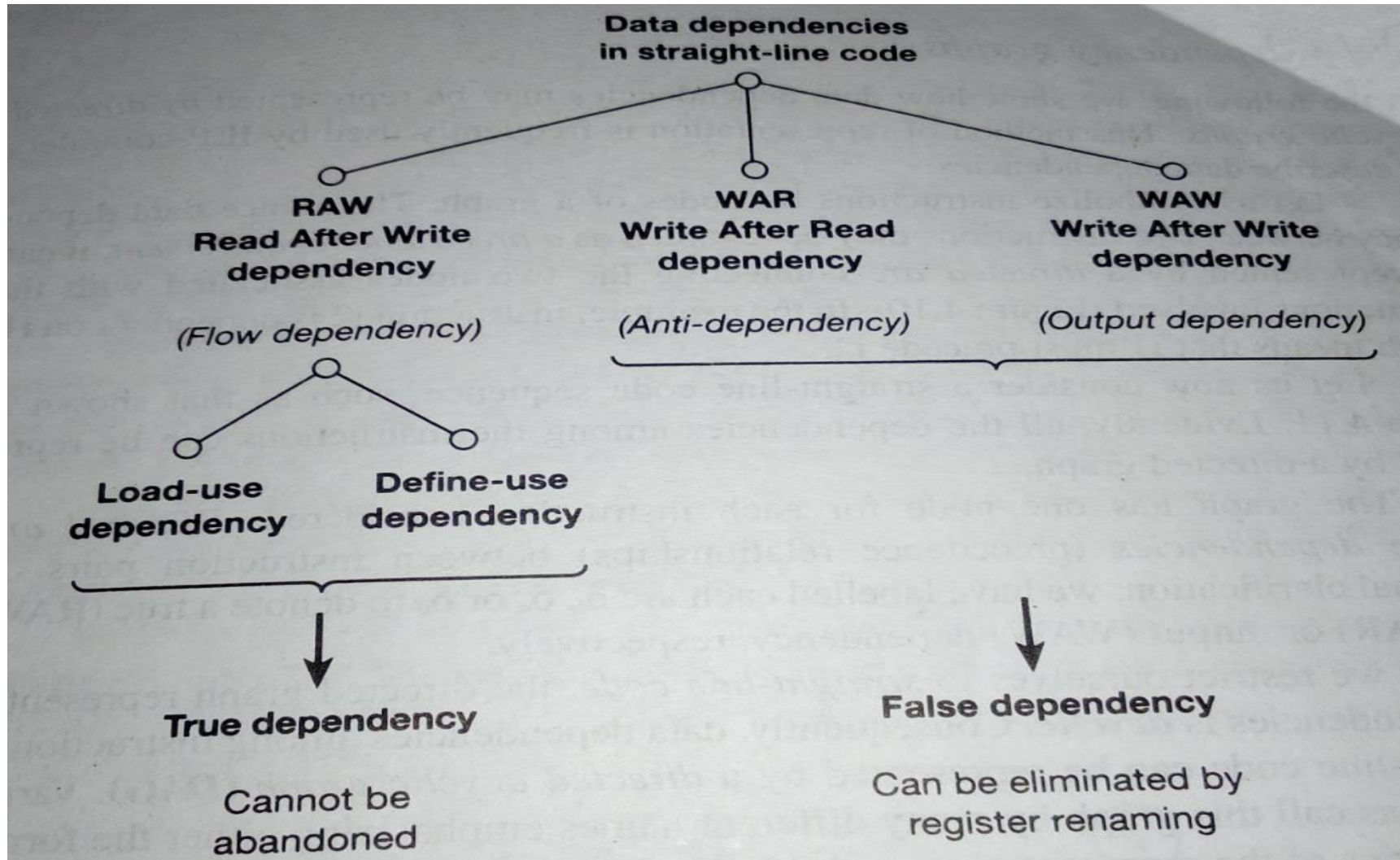
All instructions use R1 after the first instruction.

Data Dependency



Ref: "Advanced Computer Architecture", Peter Karsuk

Data Dependency



Ref: "Advanced Computer Architecture", Peter Karsuk

Data Hazard Classification

According to various data update patterns in instruction pipeline, there are 3 classes of data hazards exist:

RAW(Read After Write)

WAR(Write After Read)

WAW(Write After Write)

RAW(Read After Write)


Execution

Order is:

Instr_I

Instr_J

Instr_J tries to read operand before Instr_I writes it

 I: add **r1**,r2,r3
J: sub r4,**r1**,r3


- Caused by a “**Dependence**” (in compiler nomenclature).

WAR(Write After Read)

Execution
Order is:
Instr_I
Instr_J

Instr_J tries to write operand before Instr_I reads i

- Gets wrong operand




I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7

- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

WAW(Write After Write)

Execution
Order is:
Instr_I
Instr_J

Instr_J tries to write operand *before* Instr_I writes it
– Leaves wrong result (Instr_I not Instr_J)

 **I: sub r1,r4,r3**
J: add r1,r2,r3
K: mul r6,r1,r7

- Called an “**output dependence**” by compiler writers
This also results from the reuse of name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5

Data Hazard – Stalling (S/W Solution)

MUL R4,R2,R3

ADD R6,R4,R5

$R4 \leftarrow R2 + R3$

$R6 \leftarrow R4 + R5$



**Causes Data
Dependency**

Bubbling: Compiler can introduce the two-cycle delay needed between instructions I1 and I2 by inserting NOP (No-operation) instructions.

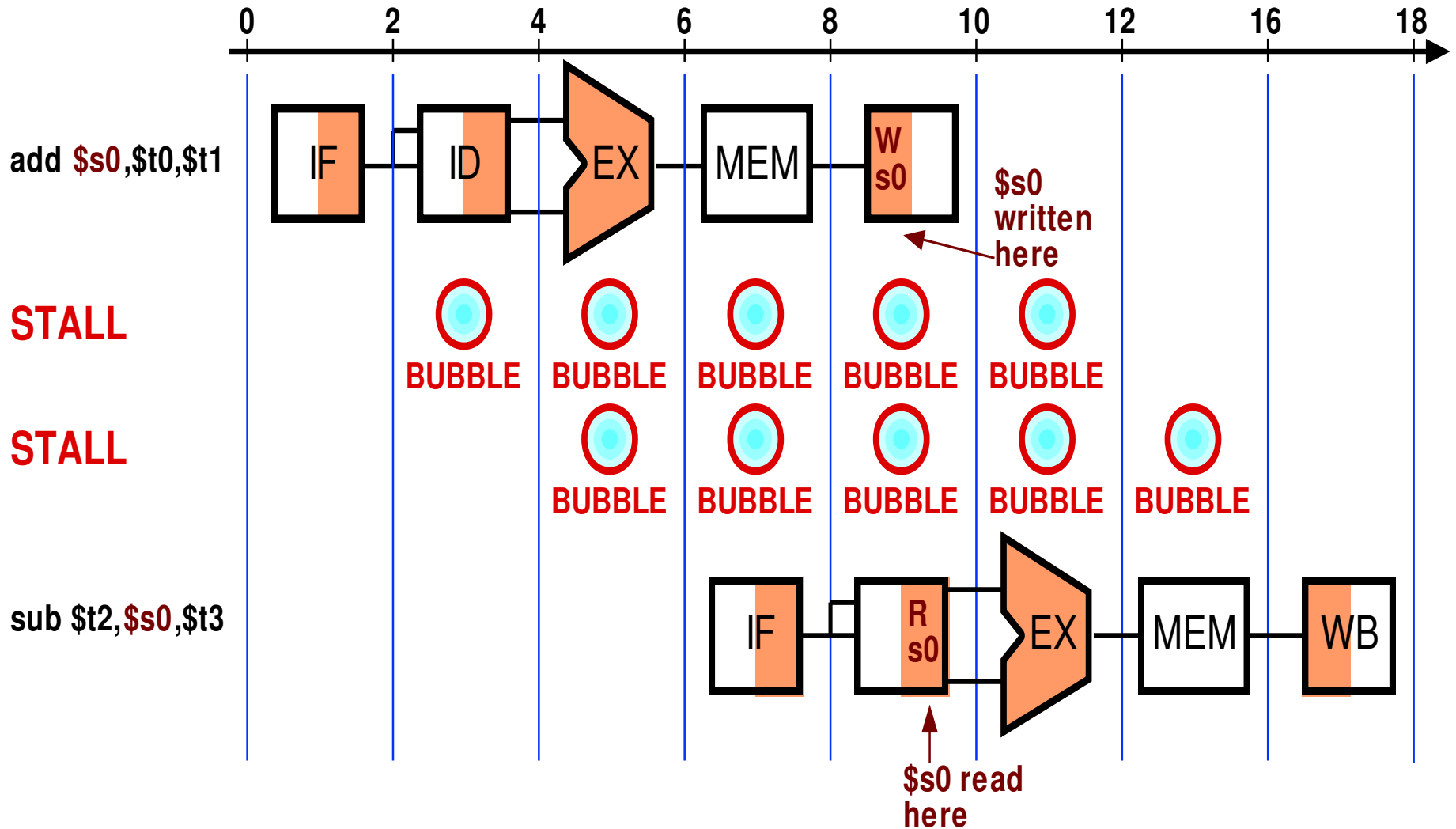
I1: Mul R4,R2,R3

NOP

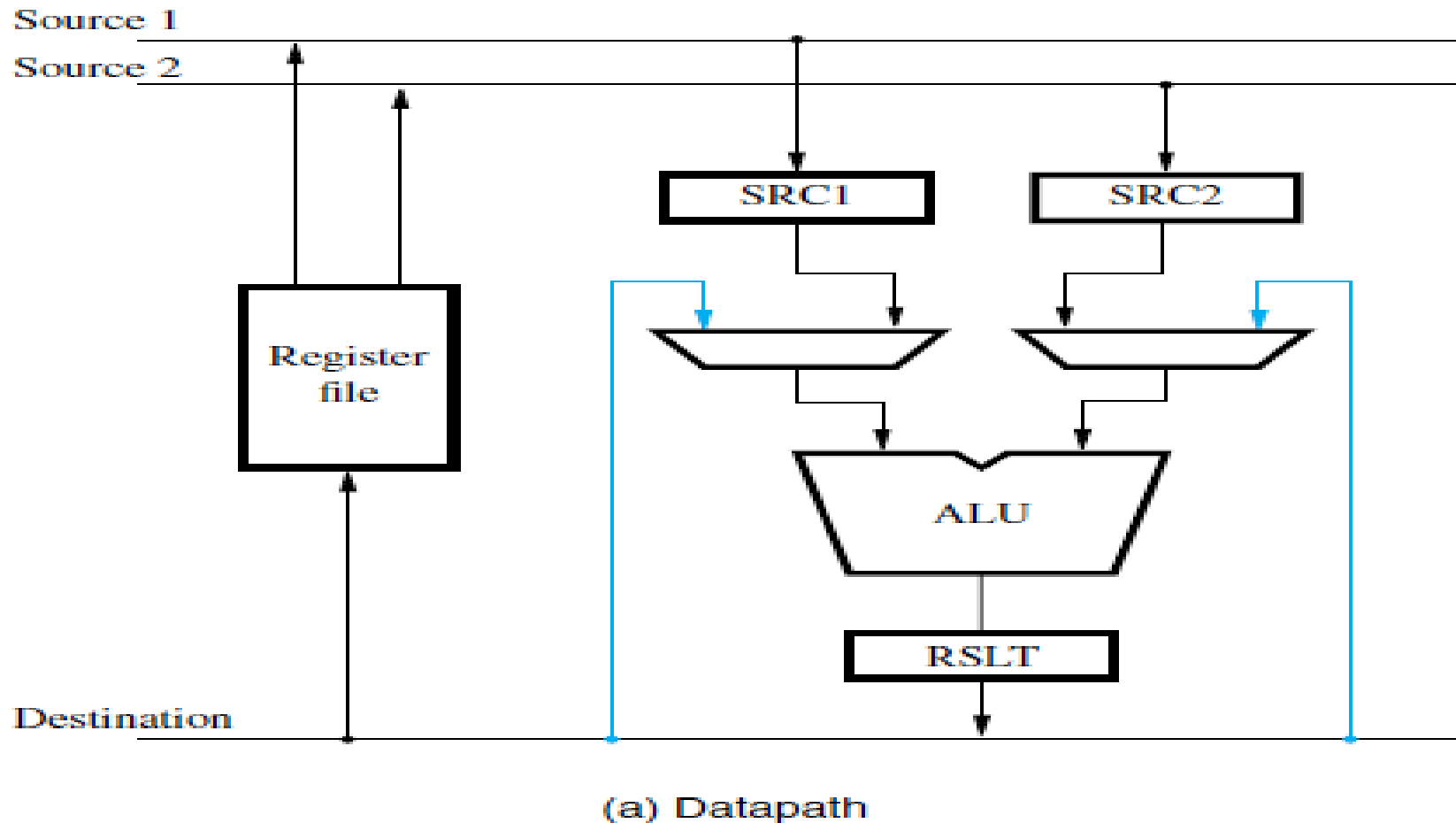
NOP

I2: Add R6,R4,R5

Data Hazard – Stalling (S/W Solution)

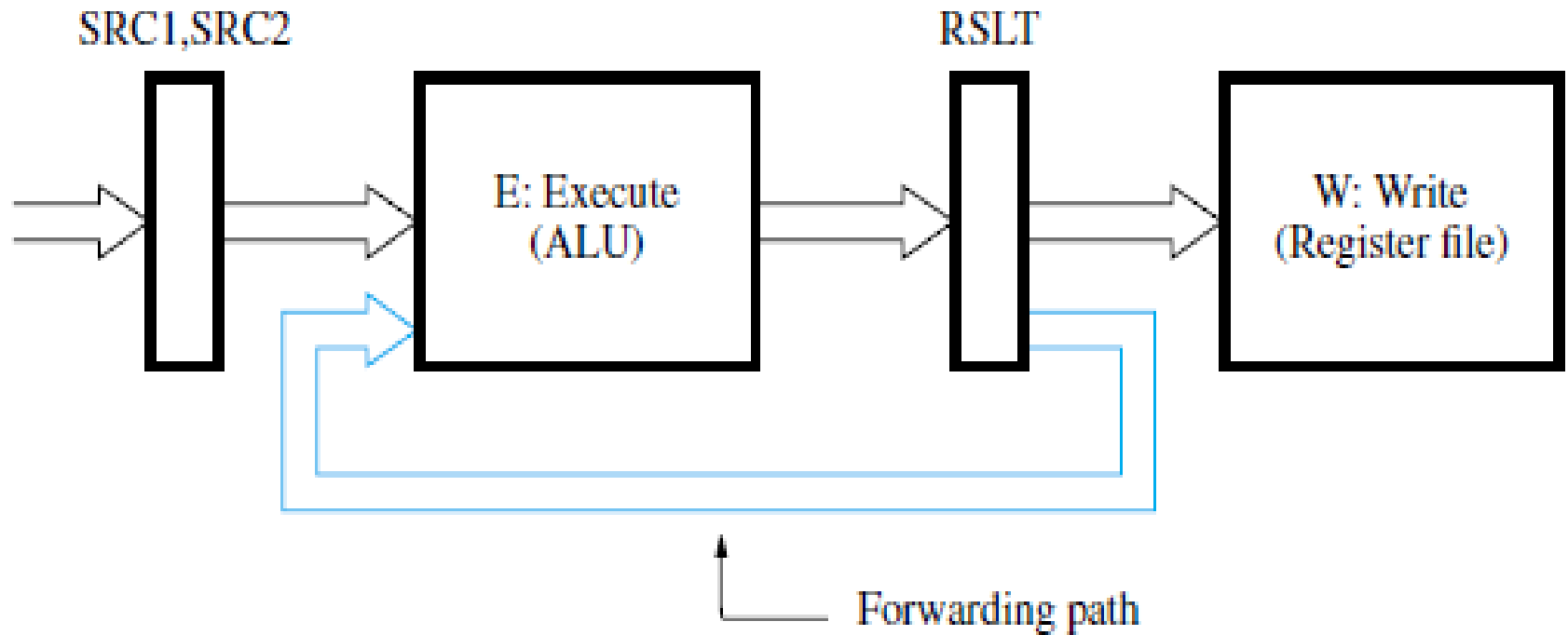


Data Hazards – Forwarding



1. Requires additional H/W
2. Registers SRC1, SRC2, and RSLT constitute the interstage buffers needed for pipelined operation.

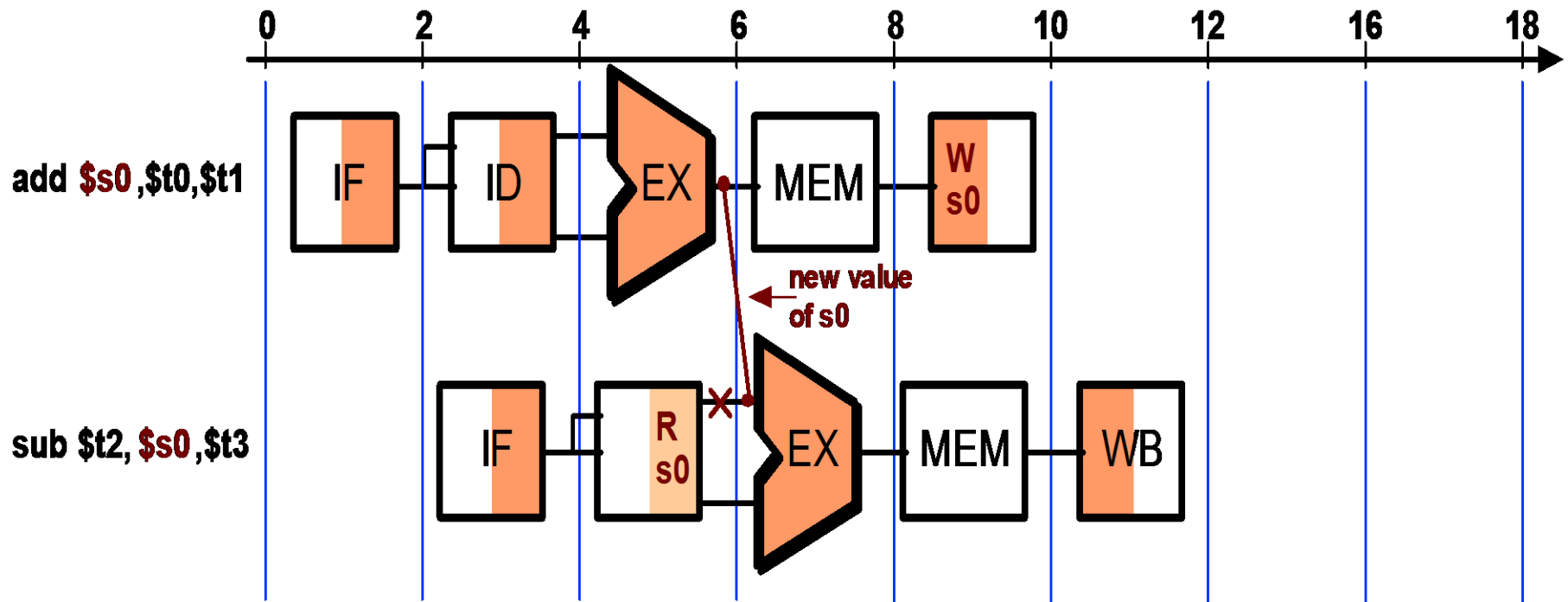
Data Hazards – Forwarding



(b) Position of the source and result registers in the processor pipeline

Data Hazards - Forwarding

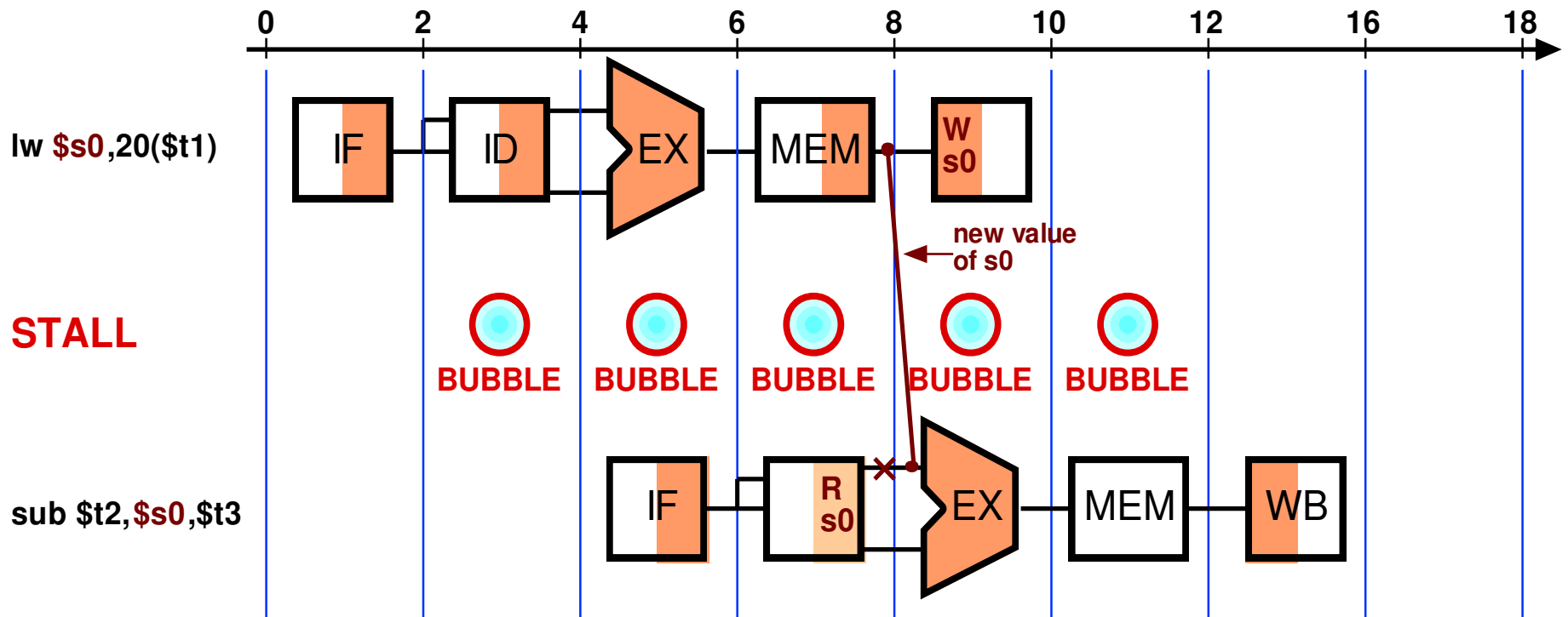
- **Key idea:** connect new value directly to next stage
- Still read s0, but ignore in favor of new result



Problem: what about load instructions?

Data Hazards - Forwarding

- STALL still required for load - data available after MEM
- MIPS architecture calls this delayed load, initial implementations required compiler to deal with this




Another Example: Stalling + Forwarding

LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

Data Hazards - Reordering Instructions

- Assuming we have data forwarding, what are the hazards in this code?


```
lw  $t0, 0($t1)
sw  $t0, 4($t1)
lw  $t2, 4($t1)
sw  $t2, 0($t1)
```



Data Hazard


- Reorder instructions to remove hazard:

```
lw  $t0, 0($t1)
lw  $t2, 4($t1)
sw  $t0, 4($t1)
sw  $t2, 0($t1)
```



Reordering

Control Hazard

- Delay in fetching instruction.
 - Cache miss
 - Branch Instructions.
- 
- Instruction/Control
Hazard

Solutions:

Prefetching

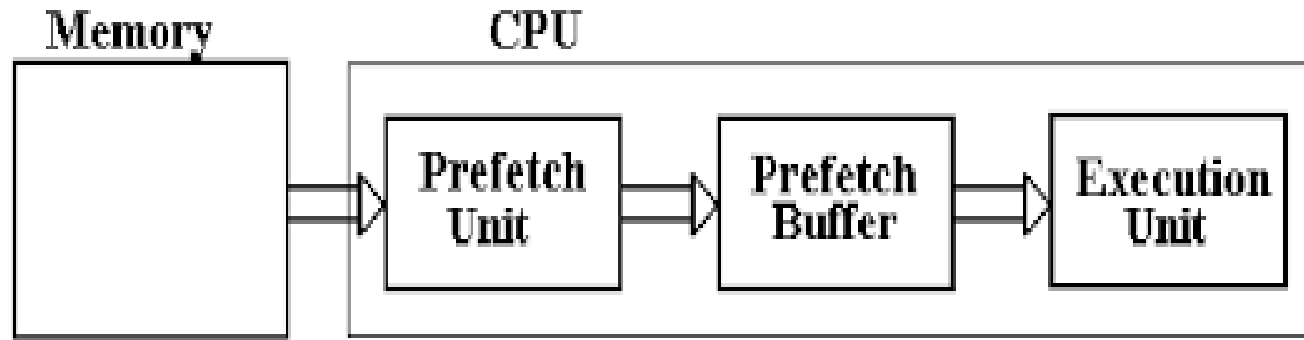
Delayed Branching

Branch Prediction

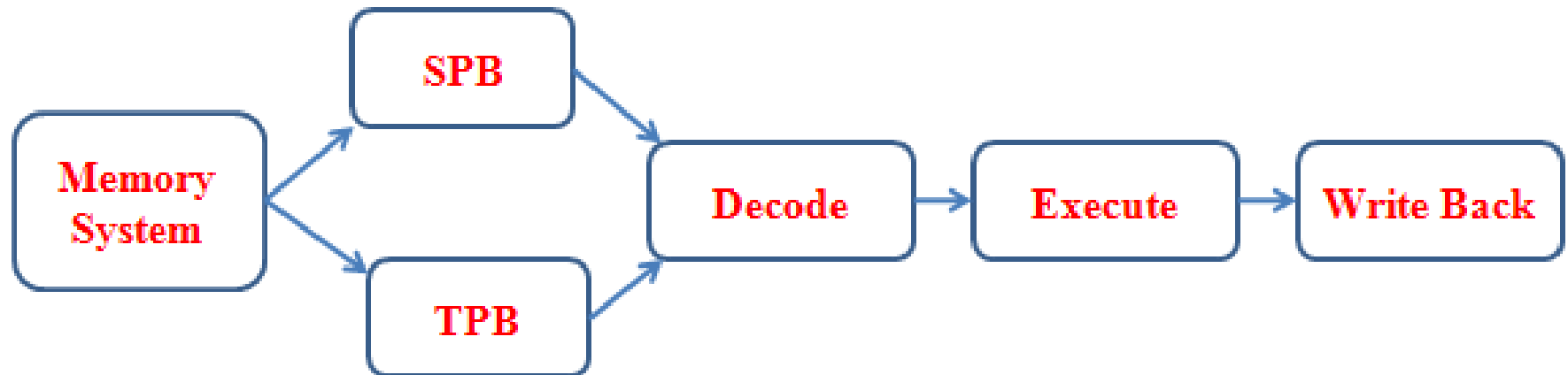
Prefetching

- Memory is assumed to be a collection of multiple modules.
- 2 pre-fetch buffers (caches)
- **Sequential Prefetch Buffer**
 - holds instructions fetched during the sequential part of a program.
- **Target Prefetch Buffer**
 - holds instructions fetched from the target of a conditional branch.

Pre-fetching



Sequential Instructions
Indicated by PC



Instructions from
Branched locations

Delayed Branching

branch instruction

sequential successor₁
sequential successor₂
.....
sequential successor_n

Branch delay of
length n (*Delay Slot*)

branch target if taken

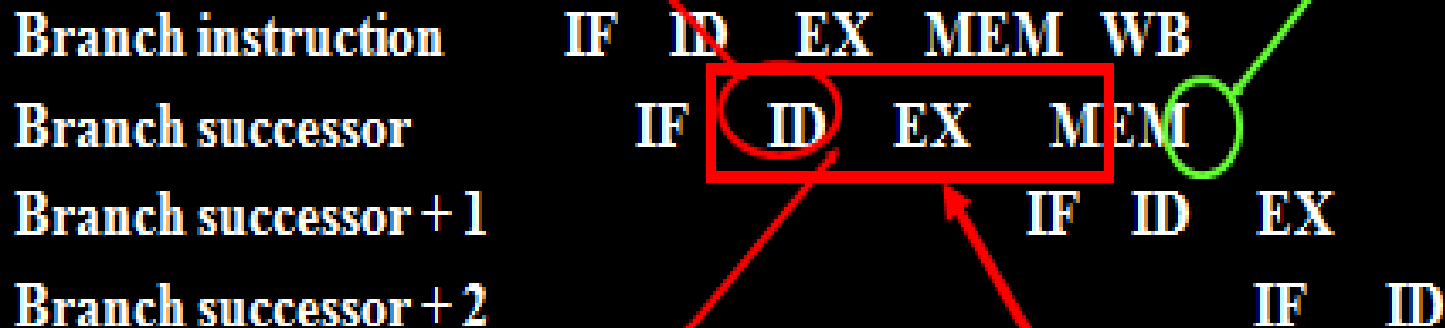
Solution:

- Possible to improve pipeline performance by rearranging instructions.
- Use of delay slots to execute independent instructions.

Delayed Branching

We don't know yet the instruction being executed is a branch.
Fetch the branch successor.

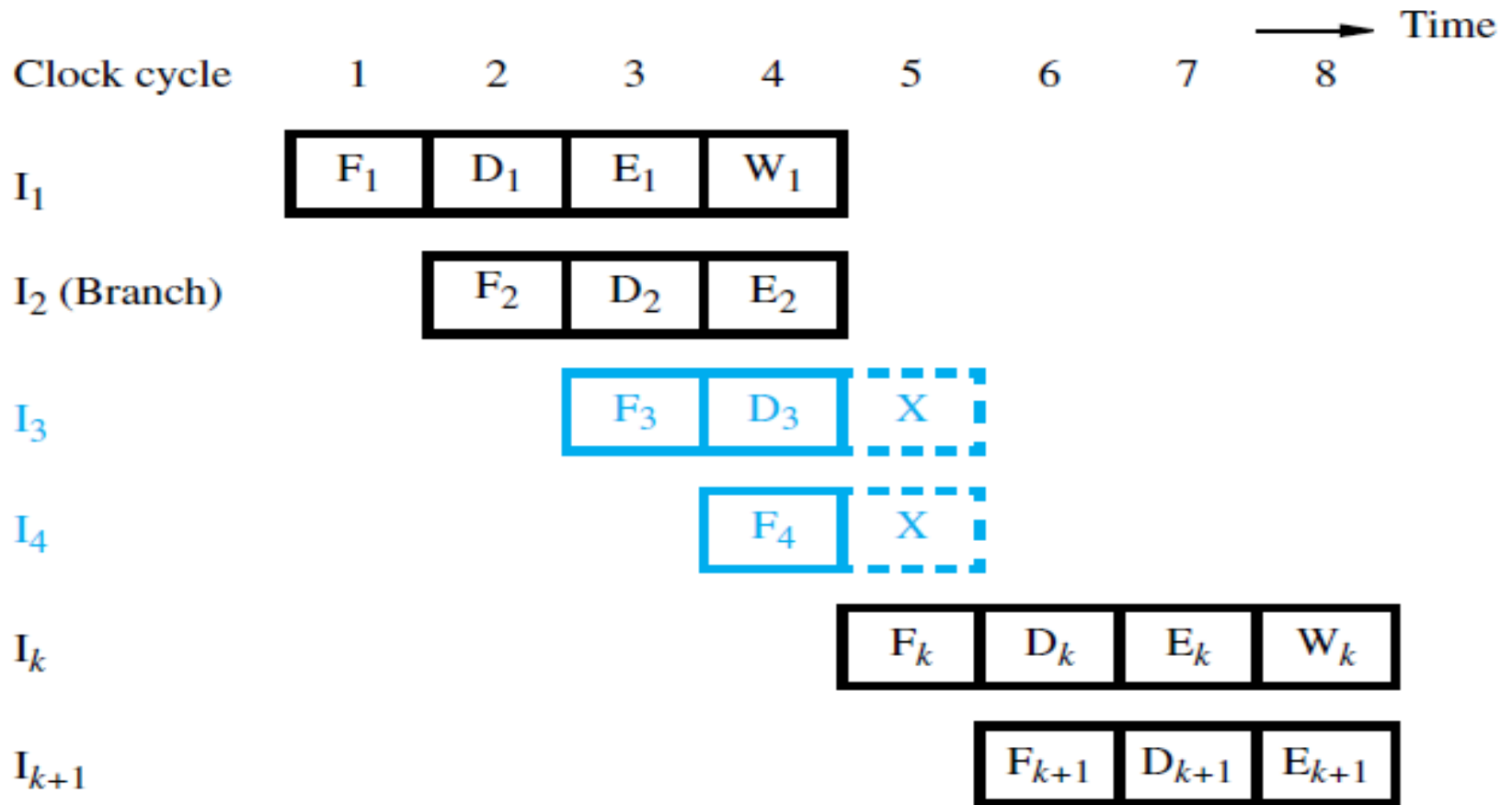
Now, target address is available.



Now, we know the instruction being executed is a branch.
But stall until branch target address is known.

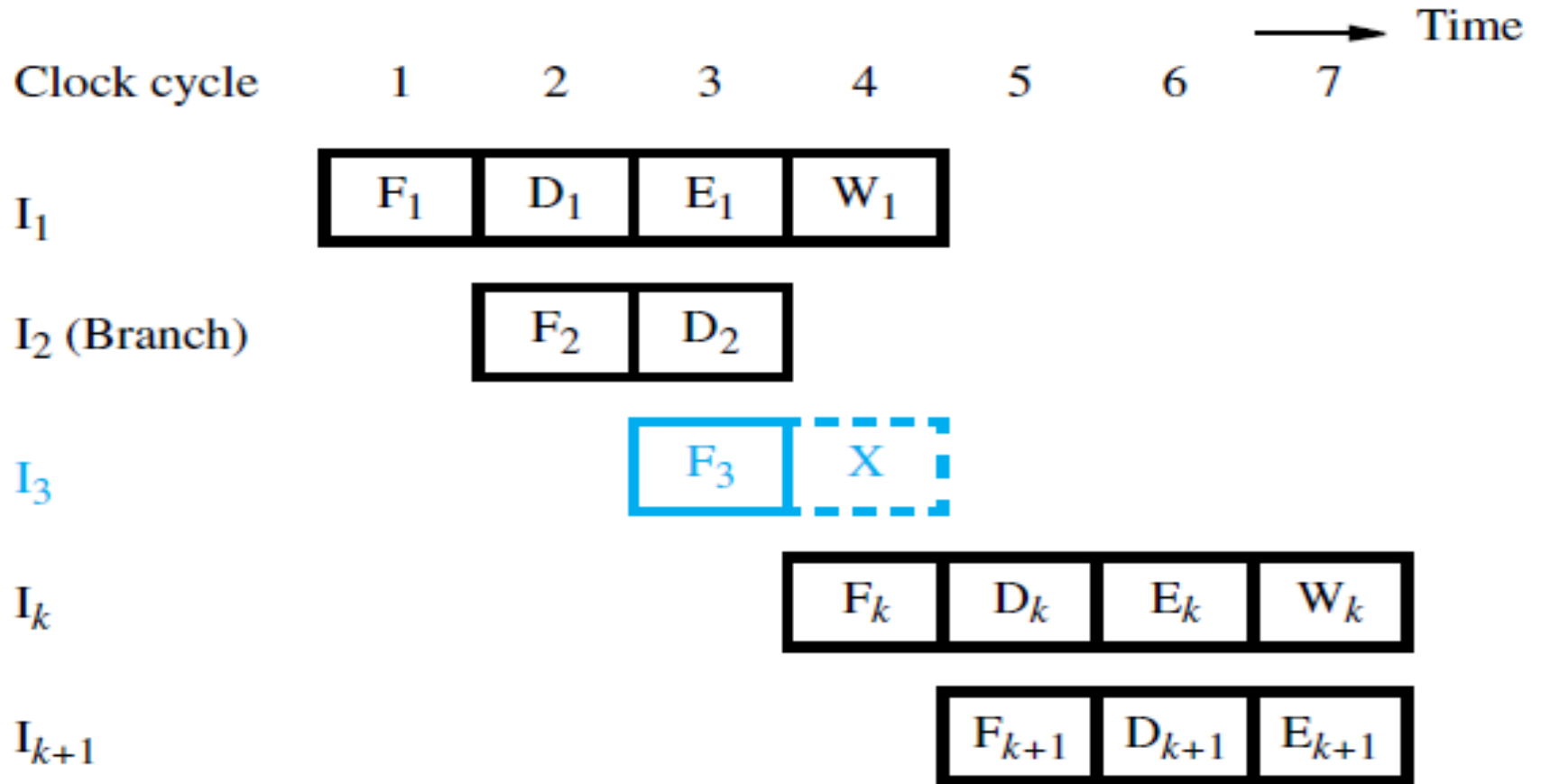
3 Wasted clock cycles for the TAKEN branch

Delayed Branching



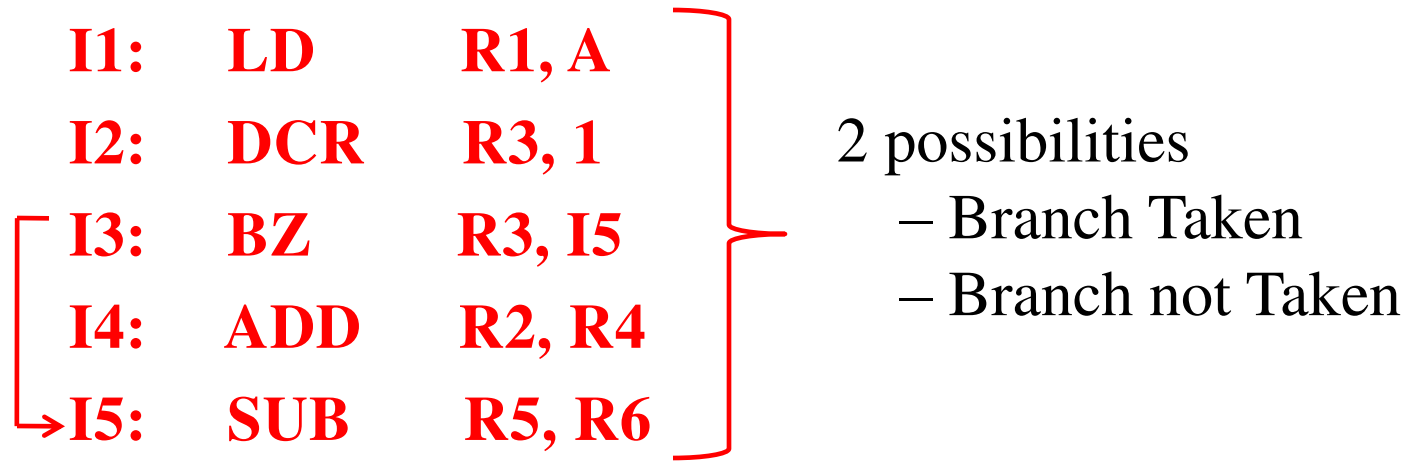
(a) Branch address computed in Execute stage

Delayed Branching



(b) Branch address computed in Decode stage

Example: Delayed Branching



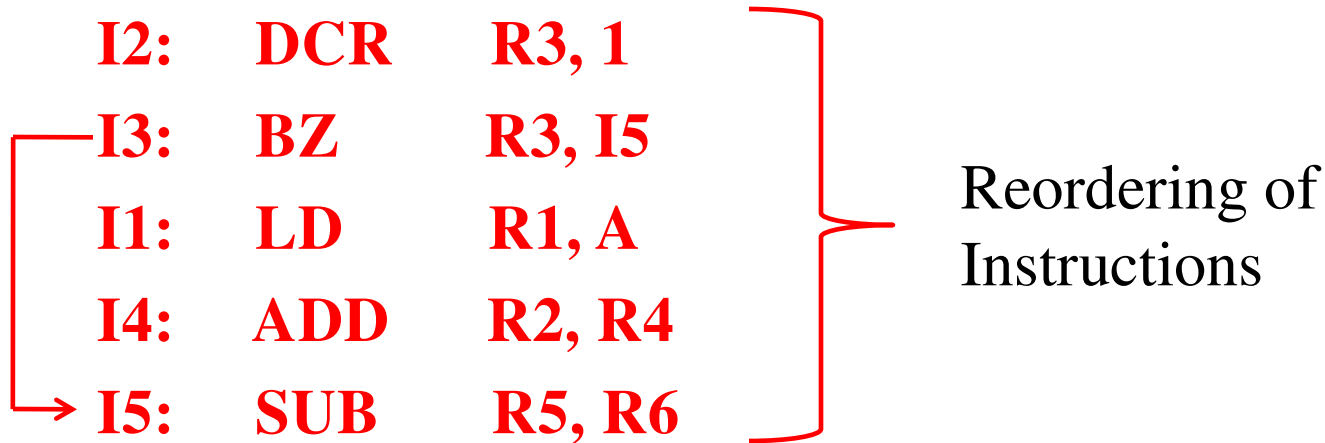
If Branch Taken => Delay Slots

If Branch not Taken => Normal execution

Example: Delayed Branching

	Time 										
	1	2	3	4	5	6	7	8	9	10	11
I1:	F1	D1	E1	M1	W1						
I2:		F2	D2	E2	M2	W2					
I3:			F3	D3	E3	<i>M3</i>	<i>W3</i>				
Stall (I4)				<i>F4</i>	<i>D4</i>	<i>E4</i>	<i>M4</i>	<i>W4</i>			
Stall					<i>O</i>	<i>O</i>	<i>O</i>	<i>O</i>	<i>O</i>		
I5:						F5	D5	E5	M5	W5	

Example: Delayed Branching



If Branch Taken => Use of Delay Slots

If Branch not Taken => Normal execution

Example: Delayed Branching

	Time 								
	1	2	3	4	5	6	7	8	9
I2:	F2	D2	E2	M2	W2				
I3:		F3	D3	E3	<i>M3</i>	<i>W3</i>			
I1:			<i>F1</i>	<i>D1</i>	<i>E1</i>	<i>M1</i>	<i>W1</i>		
I4:				<i>F4</i>	<i>D4</i>	<i>E4</i>	<i>M4</i>	<i>W4</i>	
I5:					F5	D5	E5	M5	W5

Branch Prediction

- Assume an outcome and continue fetching (undo if prediction is wrong) .
- Loss of cycles only on *mis-prediction*
- Performance penalty only when guess wrong
- Hardware required to "squash" instructions
- **Static Prediction:**
 - Prediction is made based on the type of branch codes.
 - cannot be changed once committed to the H/W.
- **Dynamic Prediction:**
 - Prediction is made based on the recent branch history.
 - Requires additional H/W to keep track of the past behavior of the branch instruction (e.g. **Branch Target Buffer** to hold recent branch information including the address of the branch target used).

Branch Prediction

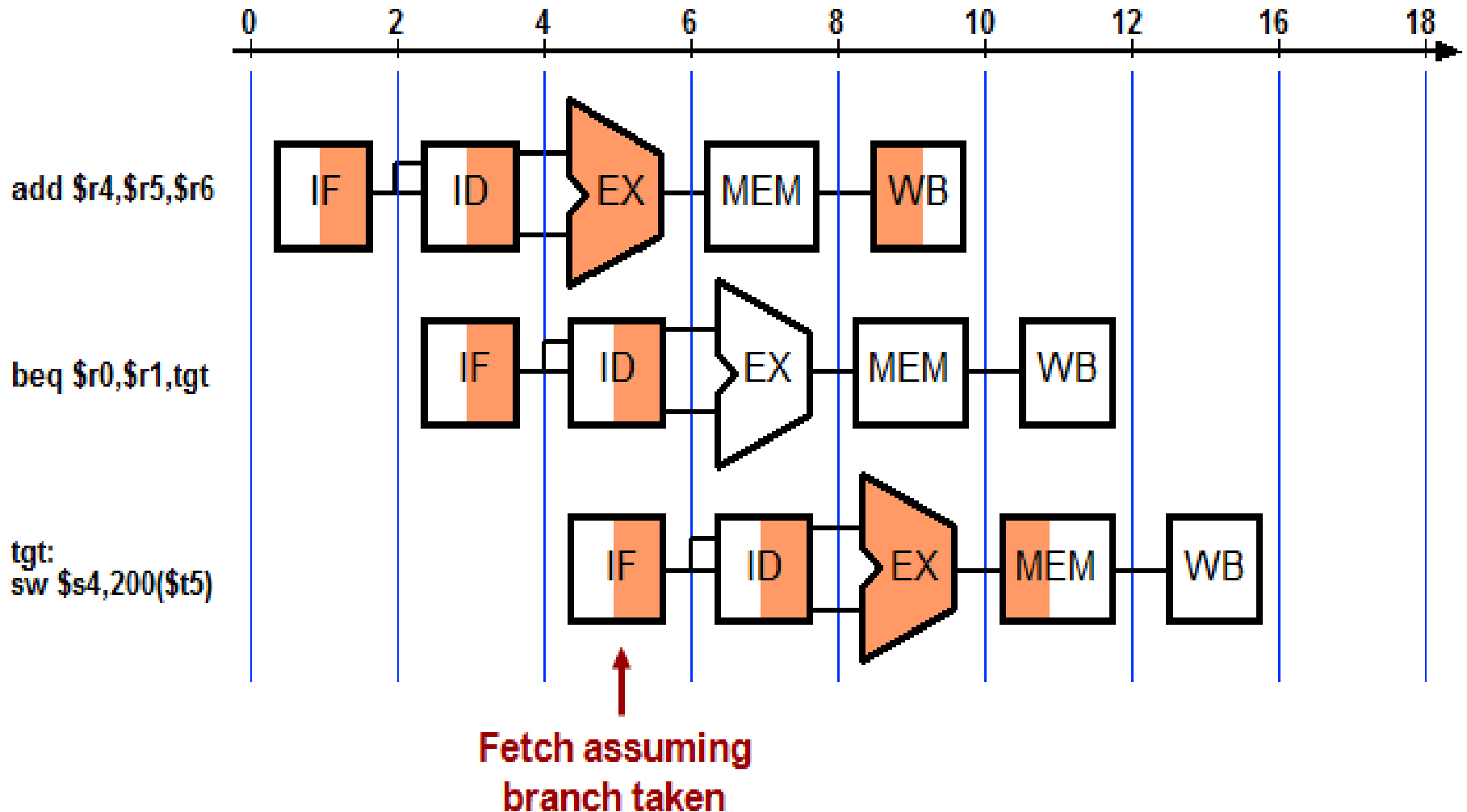
Predicting branch not taken:

1. Speculatively fetch and execute in-line instructions following the branch
2. If prediction incorrect flush pipeline of speculated instructions
 - Convert these instructions to NOPs by clearing pipeline registers
 - These have not updated memory or registers at time of flush

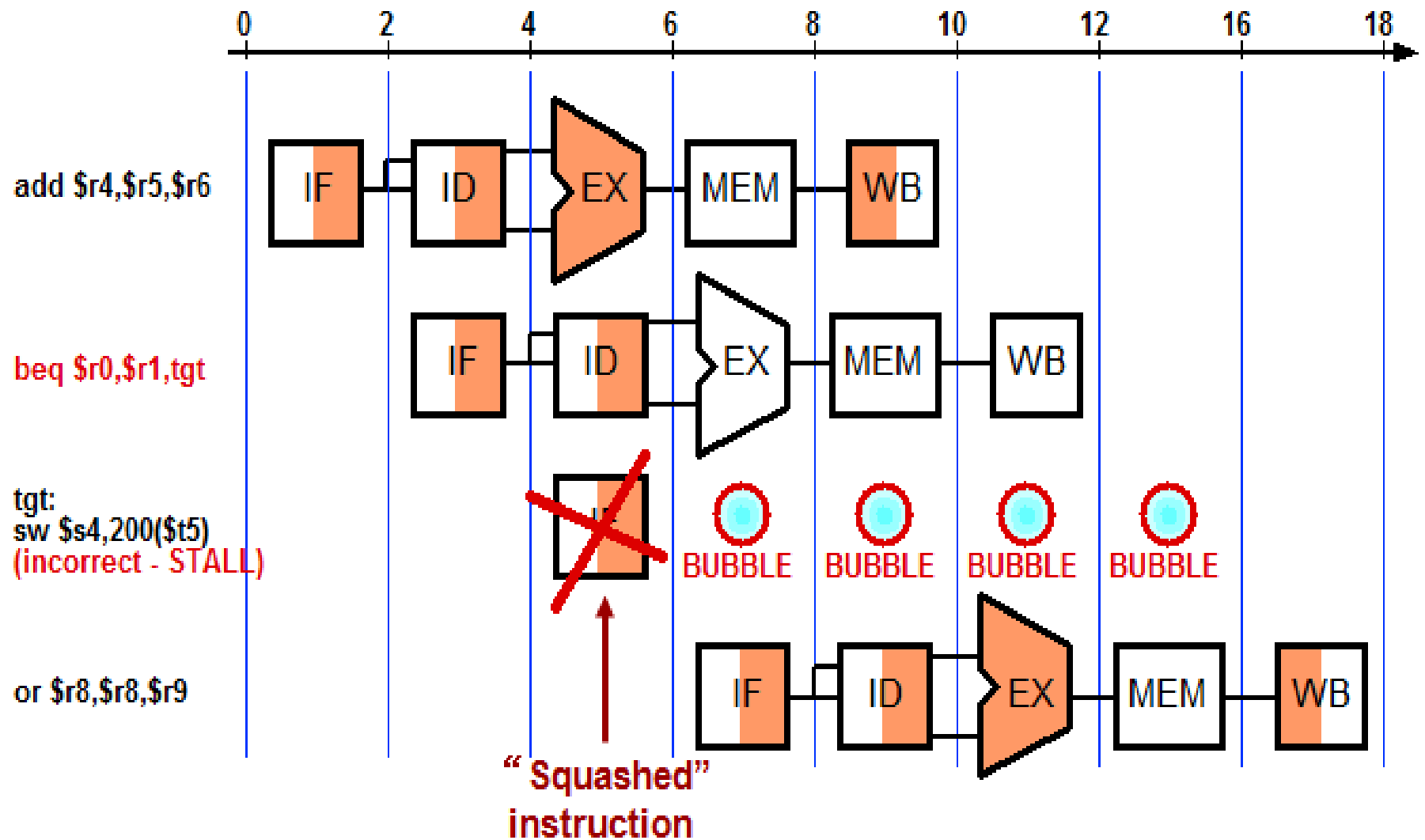
Predicting branch taken:

1. Speculatively fetch and execute instructions at the branch target address
2. Useful only if target address known earlier than branch outcome
 - May require stall cycles till target address known
 - Flush pipeline if prediction is incorrect
 - Must ensure that flushed instructions do not update memory/registers

Correct Prediction



Incorrect Prediction

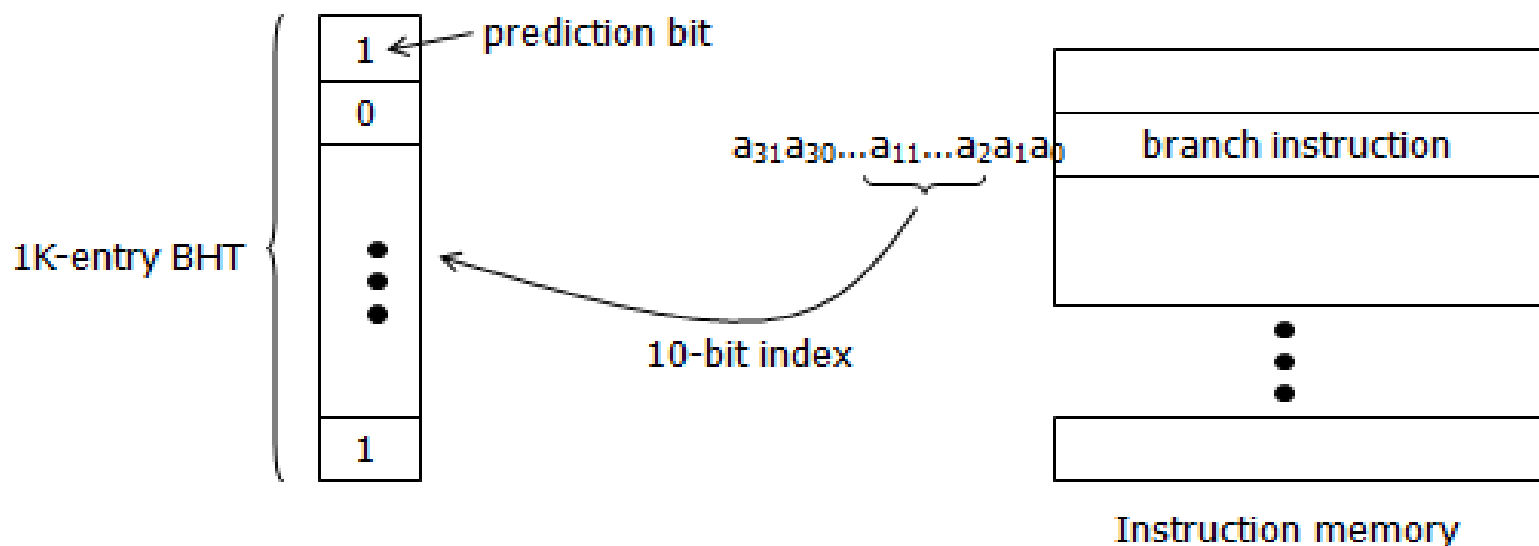


1-Bit Branch Prediction

- **Branch History Table (BHT): Lower bits of PC address index table of 1-bit values**
 - Says whether or not the branch was taken last time
 - No address check (saves HW, but may not be the right branch)
 - If prediction is wrong, invert prediction bit

1 = branch was last taken

0 = branch was last not taken



Hypothesis: branch will do the same again.

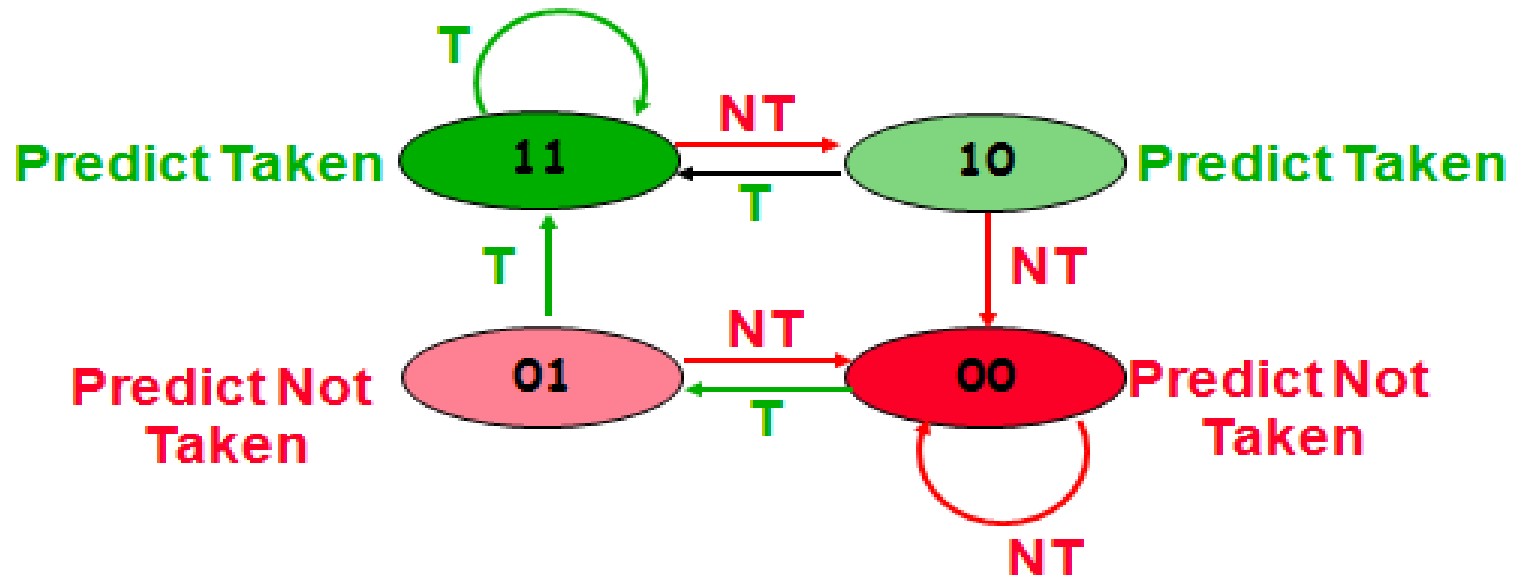
2-Bit Branch Prediction

(*Jim Smith, 1981*)

- A 2-bit scheme where prediction is changed only if mispredicted *twice*.

Red: stop, not taken

Green: go, taken



References

1. PPT on Pipelining from CS303 (3rd Semester)
2. Advanced Computer Architecture – Kai Hwang
3. Computer Organization – Carl Hamacher
4. Advanced Computer Architectures – Dezso Sima, Peter Karsuk
5. Computer Architecture & Organization – John P. Hayes
6. Computer System Architecture – M. Morris Mano
7. Computer Organization & Architecture – T. K. Ghosh
8. Computer Organization & Architecture – Xpress Learning

Thank You