Computer Architecture and Operating System

Pipelining

Prof. Indranil Sengupta

Department of Computer Science and Engineering

IIT Kharagpur

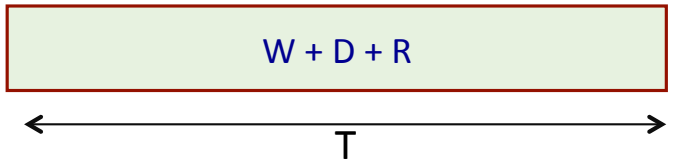
Pipelining

What is Pipelining?

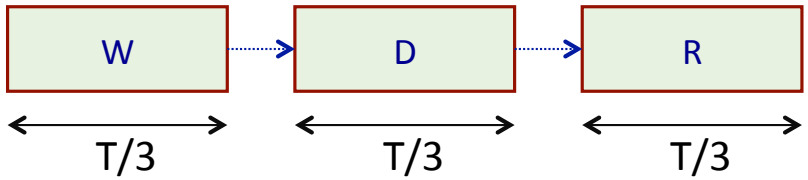
- A mechanism for overlapped execution of several input sets by partitioning some computation into a set of *k* sub-computations (or stages).
  - Very nominal increase in the cost of implementation.
  - Very significant speedup (ideally, *k*).
- Where are pipelining used in a computer system?
  a) **Instruction execution**: Several instructions executed in some sequence.
  b) **Arithmetic computation**: Same operation carried out on several data sets.
  c) **Memory access**: Several memory accesses to consecutive locations are made.

A Real-life Example

- Suppose we built a machine *M* that can wash (*W*), dry (*D*), and iron (*R*) clothes, one cloth at a time.
  - Total time required is *T*.
- As an alternative, we split the machine into three smaller ones M_W, M_D and M_R, which can perform the specific task only.
  - Time required by each of the smaller machines is *T/3* (say).

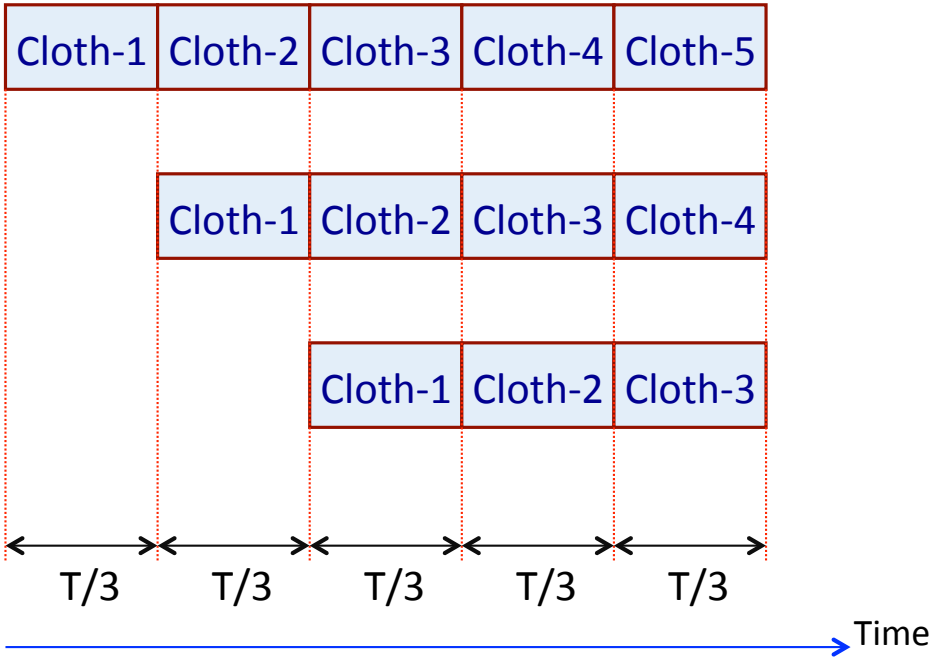


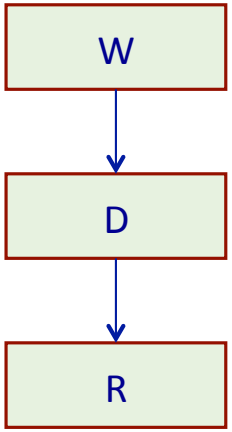
For N clothes, time $T_1 = N.T$



For N clothes, time $T_3 = (2 + N).T/3$

How does the pipeline work?





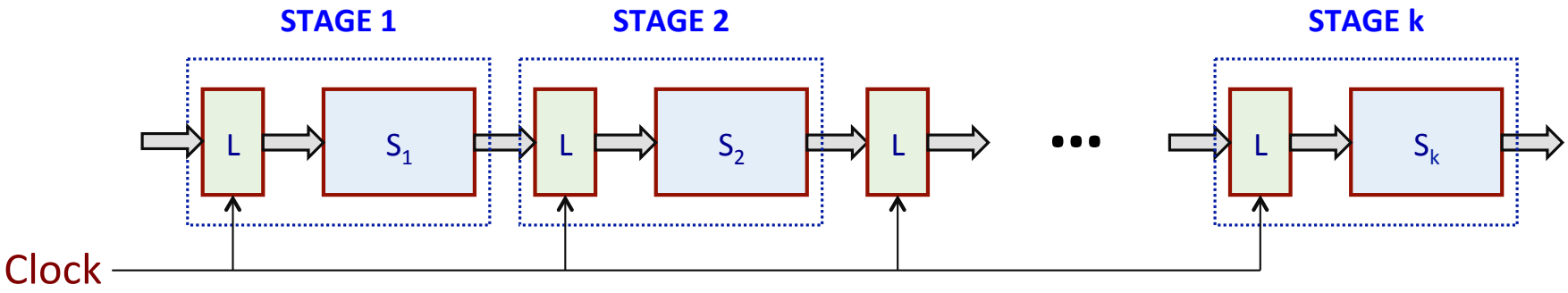
Finishing times:

- Cloth-1 – 3.T/3
- Cloth-2 – 4.T/3
- Cloth-3 – 5.T/3
- …
- Cloth-N – (2 + N).T/3

Extending the Concept to Processor Pipeline

- The same concept can be extended to *hardware pipelines*.
- Suppose we want to attain *k* times speedup for some computation.
  - *Alternative 1*: Replicate the hardware *k* times → cost also goes up *k* times.
  - *Alternative 2*: Split the computation into *k* stages → very nominal cost increase.
- Need for *buffering*:
  - In the washing example, we need a *tray* between machines (W & D, and D & R) to keep the cloth temporarily before it is accepted by the next machine.
  - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

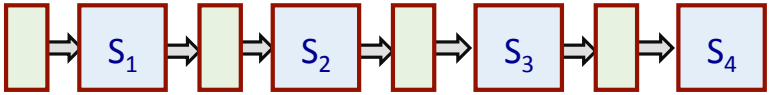
Model of a Synchronous k-stage Pipeline



- The latches are made with master-slave flip-flops, and serve the purpose of isolating inputs from outputs.
- The pipeline stages are typically combinational circuits.
- When *Clock* is applied, all latches transfer data to the next stage simultaneously.

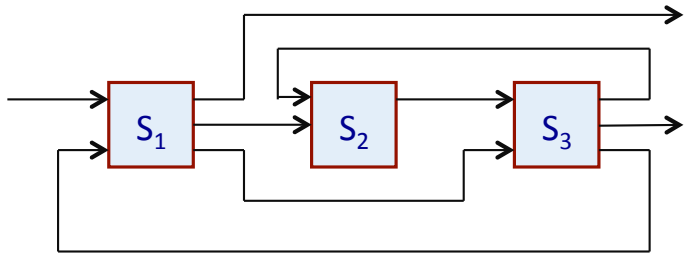
Reservation Table

- The Reservation Table is a data structure that represents the utilization pattern of successive stages in a synchronous pipeline.
  - Basically a *space-time diagram* of the pipeline that shows precedence relationships among the stages.
    - X-axis shows the time steps
    - Y-axis shows the stages
  - Number of columns give evaluation time.
  - The reservation table for a 4-stage linear pipeline is shown.



	1	2	3	4
S_1	X			
S_2		X		
S_3			X	
S_4				X

- Reservation table for a 3-stage multi-function pipeline is shown.
  - Contains feedforward and feedback connections.
  - Two functions *X* and *Y*.
- Some characteristics:
  - *Multiple X's in a row* :: repeated use of the same stage in different cycles.
  - *Contiguous X's in a row* :: extended use of a stage over more than one cycles.
  - *Multiple X's in a column* :: multiple stages are used in parallel during a clock cycle.



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

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

Pipeline Speedup

Some notations:

- τ :: clock period of the pipeline
- t_i :: time delay of the circuitry in stage S_i
- d_L :: delay of a latch

Maximum stage delay $\tau_m = max\ \{t_i\}$

Thus, $\tau = \tau_m + d_L$

Pipeline frequency $f = 1 / \tau$

If one result is expected to come out of the pipeline every clock cycle, f will represent the *maximum throughput* of the pipeline.

- The total time to process *N* data sets is given by

$$T_k = [(k-1) + N].\tau$$

(k – 1) τ time required to fill the pipeline
1 result every τ time after that → total N.τ

- For an equivalent non-pipelined processor (i.e. one stage), the total time is

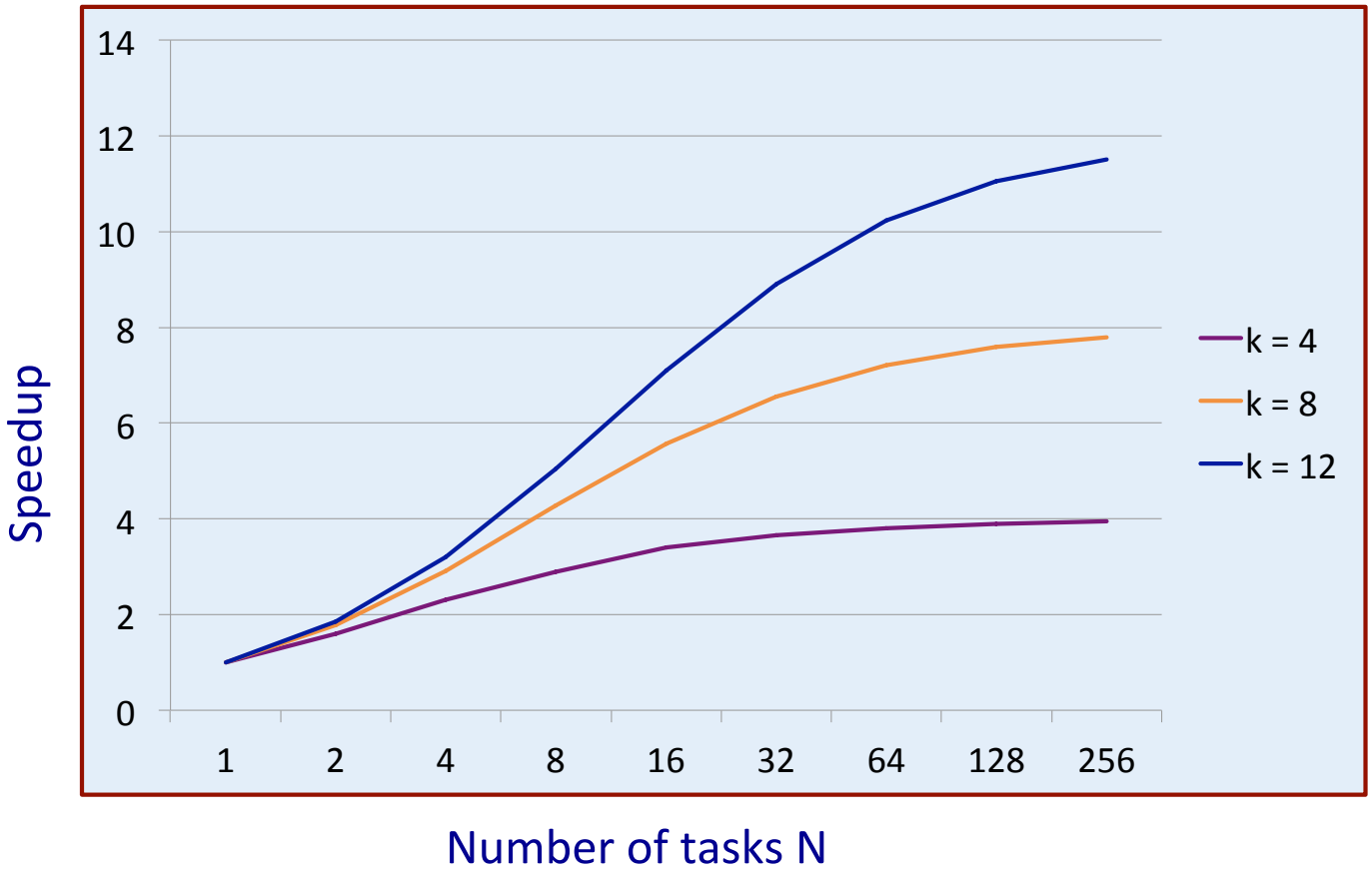
$$T_1 = N.k.\tau$$

(ignoring the latch overheads)

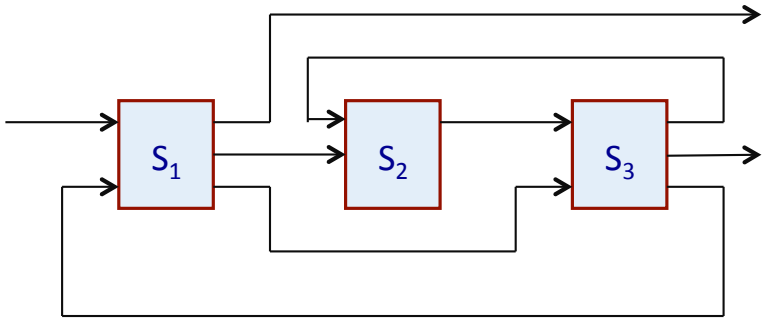
- Speedup of the *k*-stage pipeline over the equivalent non-pipelined processor:

$$S_k = \frac{T_1}{T_k} = \frac{N.k.\tau}{k.\tau + (N-1).\tau} = \frac{N.k}{k + (N-1)}$$

As N → ∞, S_k → k


Speedup
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
128
256
k = 4
k = 8
k = 12
Number of tasks N

Scheduling of Non-linear Pipelines



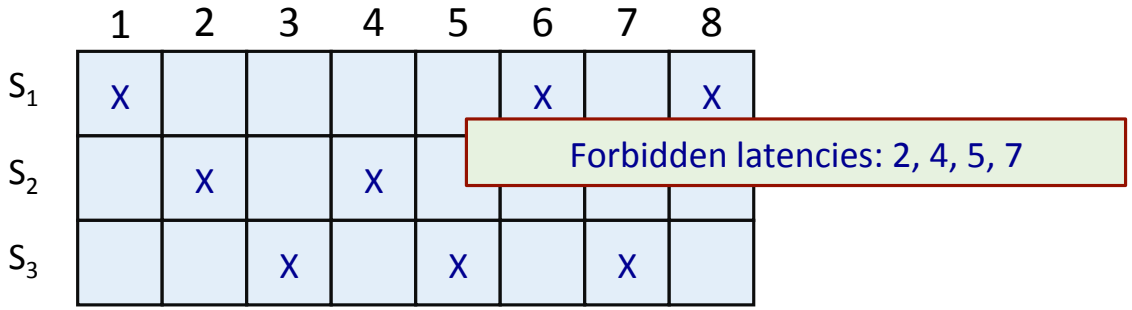
	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

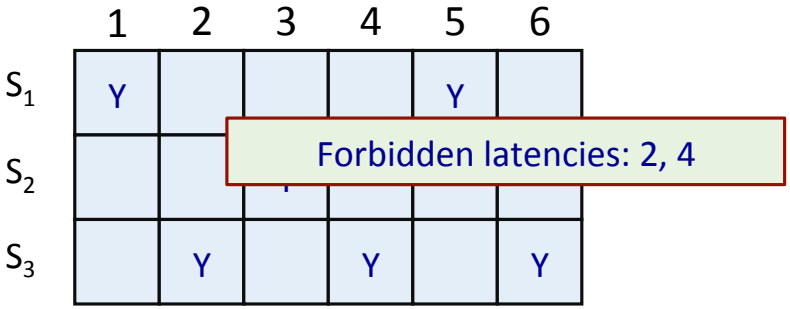
Two operations X and Y

- **X: 8 time steps to complete**
- **Y: 6 time steps to complete**

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

- **Latency Analysis**:
  - The number of time units between two initiations of a pipeline is called the *latency* between them.
  - Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.
  - The latencies that can cause collision are called *forbidden latencies*.
    - Distance between two X's in the same row of the reservation table.





- A *latency sequence* is a sequence of permissible non-forbidden latencies between successive task initiations.
- A *latency cycle* is a latency sequence that repeats the same subsequence.

Function X

- Forbidden latencies: 2, 4, 5, 7
- Possible latency cycles:
  - (1, 8) = 1, 8, 1, 8, … (average latency = 4.5)
  - (3) = 3, 3, 3, … (average latency = 3.0)
  - (6) = 6, 6, 6, … (average latency = 6.0)

Function Y

- Forbidden latencies: 2, 4
- Possible latency cycles:
  - (1, 5) = 1, 5, 1, 5, … (average latency = 3.0)
  - (3) = 3, 3, 3, … (average latency = 3.0)
  - (3, 5) = 3, 5, 3, 5, … (average latency = 4.0)

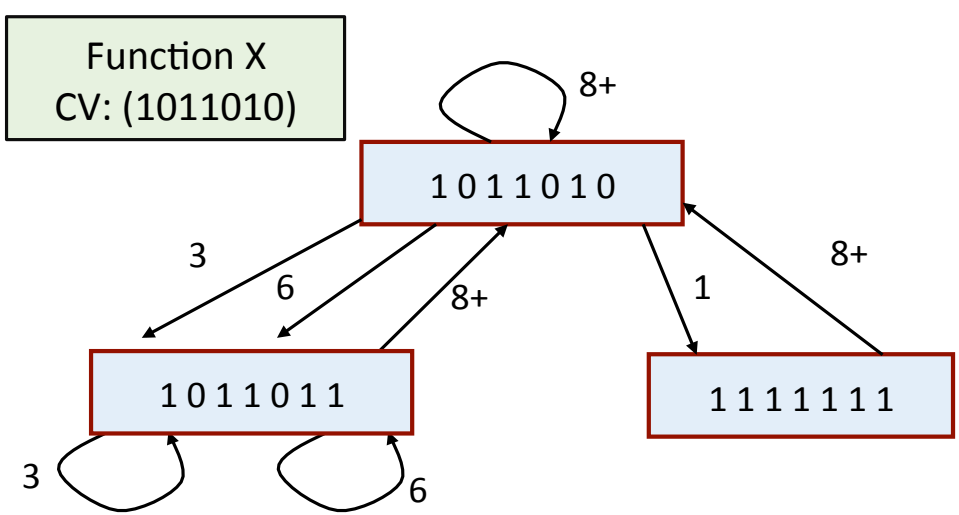
Constant Cycle

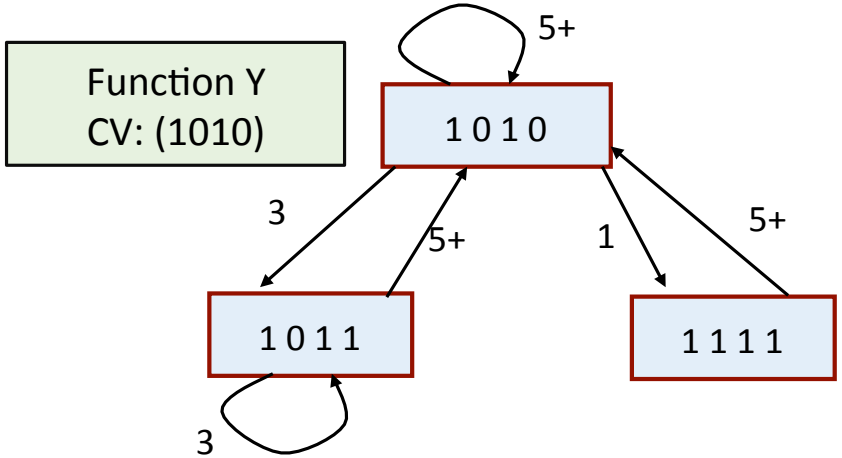
Collision Free Scheduling

- Main objective:
  - Obtain the shortest average latency between initiations without causing collisions.
- We define a *collision vector*.
  - If the reservation table has n columns, the maximum forbidden latency is $m \leq n-1$.
  - The permissible latencies p will satisfy: $1 \leq p \leq m-1$.
  - The collision vector is an m-bit binary vector $C = (C_m\ C_{m-1} \ldots C_2\ C_1)$, where $C_i = 1$ if latency i causes collision, and $C_i = 0$ otherwise.
  - C_m is always 1.

Function X: $C_X = (1011010)$
Function Y: $C_Y = (1010)$

- From the collision vector (CV), we can construct a *state diagram* specifying the permissible state transitions among successive initiations.
  - The collision vector corresponds to the initial state of the pipeline.
  - The next state at time *t+p* is obtained by shifting the present state *p*-bits to the right and OR-ing with the initial collision vector *C*.





- From the state diagram, we can determine latency cycles that result in *minimum average latency (MAL)*.
  - In a *simple cycle*, a state appears only once.
  - Some of the simple cycles are *greedy cycles*, which are formed only using outgoing edges with minimum latencies.

Function X:

- Simple cycles: (3), (6), (8), (1,8), (3,8), (6,8)
- Greedy cycles: (3), (1,8)

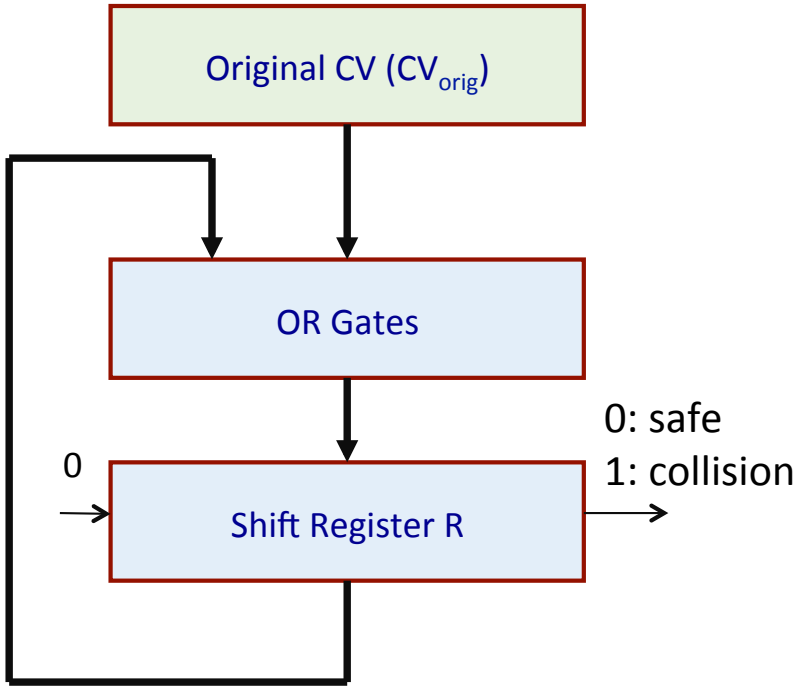
Function Y:

- Simple cycles: (3), (5), (1,5), (3,5)
- Greedy cycles: (3), (1,5)

MAL = 3 for both X and Y

The Scheduling Algorithm

```
Load collision vector (CV) in a shift register R;
If (LSB of R is 1) then
    begin
        Do not initiate an operation;
        Shift R right by one position with 0 insertion;
    end
else
    begin
        Initiate an operation;
        Shift R right by one position with 0 insertion;
        R = R or CV_orig;        // Logical OR with original CV
    end
```



Exercise 1

- For the following reservation tables,
  a) What are the forbidden latencies?
  b) Show the state transition diagram.
  c) List all the simple cycles and greedy cycles.
  d) Determine the optimal constant latency cycle, and the MAL.
  e) Determine the pipeline throughput, for τ = 20 ns.

	1	2	3	4
S_1	X			X
S_2		X		
S_3			X	

	1	2	3	4	5	6	7
S_1	X		X				X
S_2		X			X		
S_3				X		X	

Exercise 2

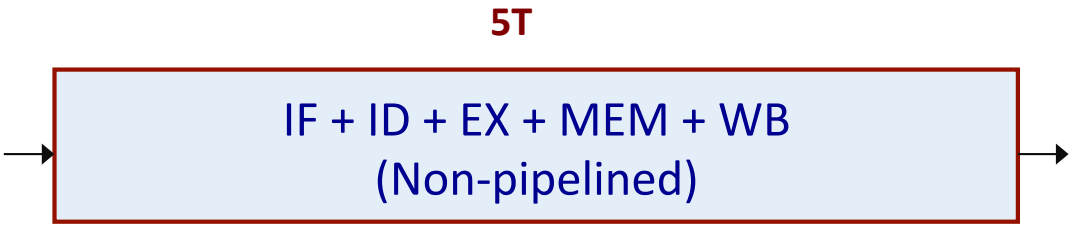
- A non-pipelined processor X has a clock frequency of 250 MHz and an average cycles per instruction (CPI) of 4. Processor Y, an improved version of X, is designed with a 5-stage linear instruction pipeline. However, due to latch delay and clock skew, the clock rate of Y is only 200 MHz.

  If a program consisting of 5000 instructions are executed on both processors, what will be the speedup of processor Y as compared to processor X?

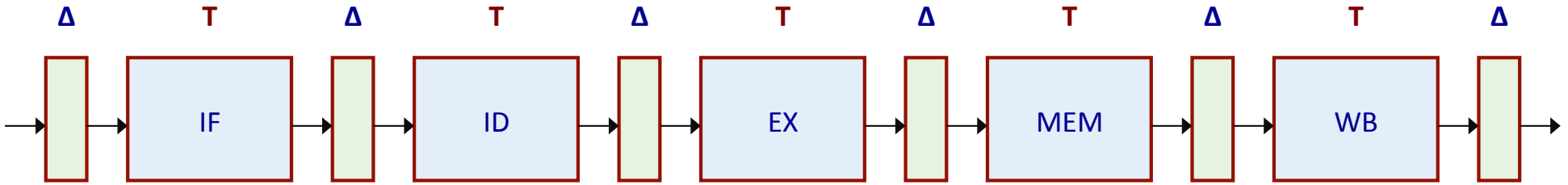
Pipelining the MIPS32 Data Path

Introduction

- Basic requirements for pipelining the MIPS32 data path:
  - We should be able to start a new instruction every clock cycle.
  - Each of the five steps mentioned before (IF, ID, EX, MEM and WB) becomes a pipeline stage.
  - Each stage must finish its execution within one clock cycle.
- Since execution of several instructions are overlapped, we must ensure that there is no conflict during the execution.
  - Simplicity of the MIPS32 instruction set makes this evaluation quite easy.
  - We shall discuss these issues in some detail.



Time of execute n instructions = *5Tn*



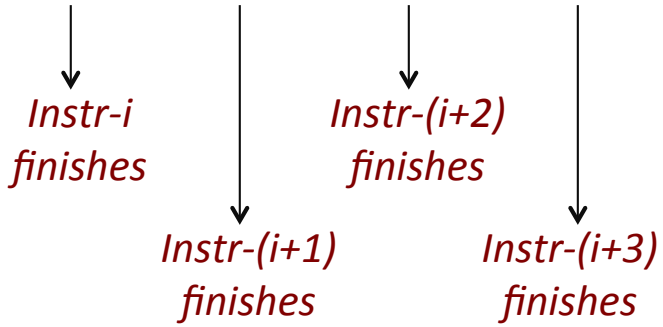
Time of execute *n* instructions = (4 + n).(T + Δ) ≈ (4 + n).T, if T >> Δ

Ideal Speedup = 5Tn / (4 + n)T ≈ 5, for large n.

In practice, due to various conflicts, speedup is much less.

Clock Cycles

Instruction	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
i + 1		IF	ID	EX	MEM	WB		
i + 2			IF	ID	EX	MEM	WB	
i + 3				IF	ID	EX	MEM	WB



Clock Cycles

Instruction	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
i + 1		IF	ID	EX	MEM	WB		
i + 2			IF	ID	EX	MEM	WB	
i + 3				IF	ID	EX	MEM	WB

Some examples of conflict:

- IF & MEM: In clock cycle 4, both instructions *i* and *i+3* access memory.
  - *Solution: use separate instructions and data cache*.
- ID & WB: In clock cycle 5, both instructions *i* and *i+3* access register bank.
  - *Solution: allow both read and write access to registers in the same clock cycle*.

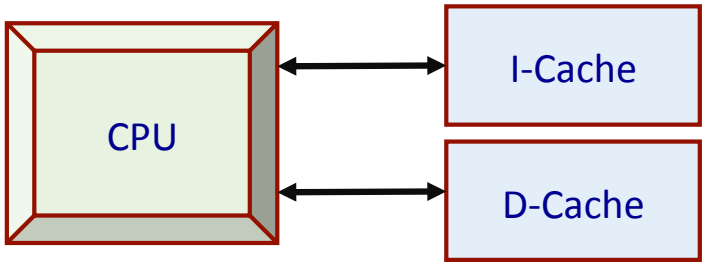
Advantages of Pipelining

- In the non-pipelined version, the execution time of an instruction is equal to the combined delay of the five stages (say, *5T*).
- In the pipelined version, once the pipeline is full, one instruction gets executed after every *T* time.
  - Assuming all state delays are equal (equal to *T*), and neglecting latch delay.
- However, due to various conflicts between instructions (called *hazards*), we cannot achieve the ideal performance.
  - Several techniques have been proposed to improve the performance.

Some Observations

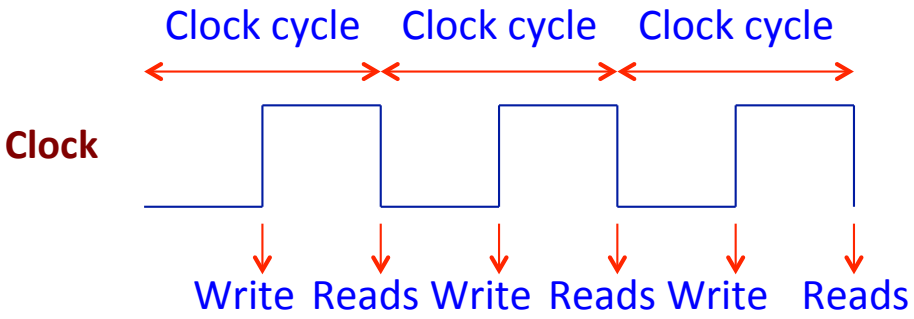
a) To support overlapped execution, peak memory bandwidth must be increased *5 times* over that required for the non-pipelined version.

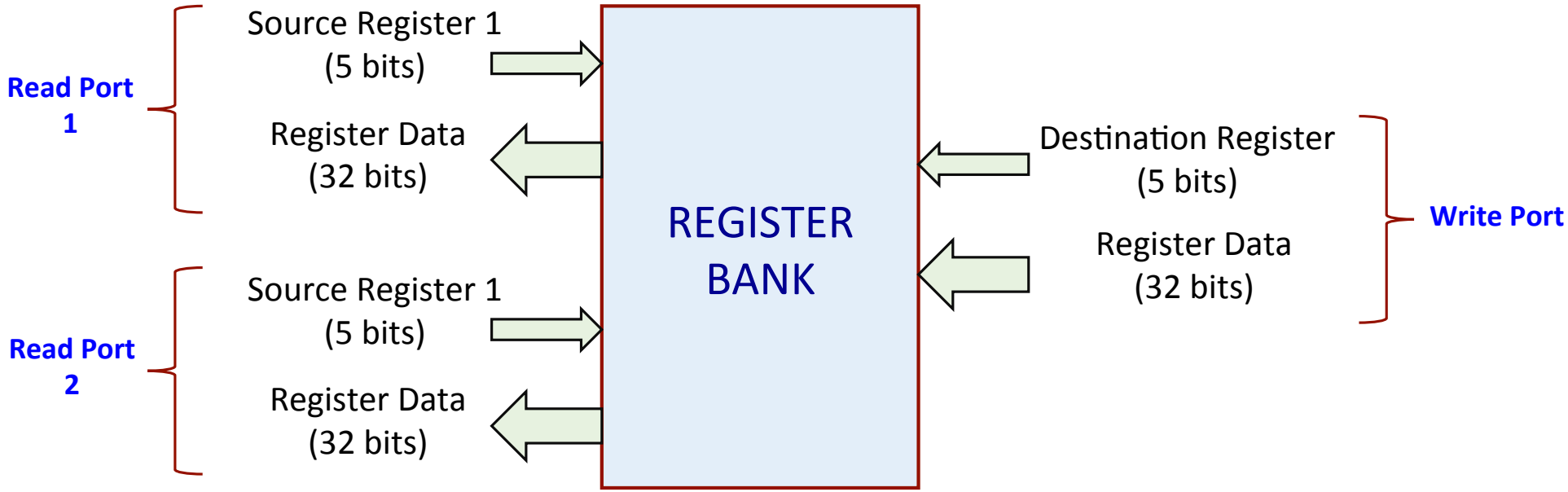
- An instruction fetch occurs every clock cycle.
- Also there can be two memory accesses per clock cycle (one for instruction and one for data).
- Separate instruction and data caches are typically used to support this.



b) The register bank is accessed both in the stages ID and WB.

- ID requires 2 register reads, and WB requires 1 register write.
- We thus have the requirement of *2 reads and 1 write* in every clock cycle.
- Two register reads can be supported by having two register read ports.
- Simultaneous reads and write may result in clashes (e.g., same register used).
  - Solution adopted in MIPS32 pipeline is to perform the write during the *first half* of the clock cycle, and the reads during the *second half* of the clock cycle.




Read Port 1
Source Register 1 (5 bits)
Register Data (32 bits)
Read Port 2
Source Register 1 (5 bits)
Register Data (32 bits)
REGISTER BANK
Destination Register (5 bits)
Register Data (32 bits)
Write Port

c) Since a new instruction is fetched every clock cycle, it is required to increment the *PC* on each clock.

- PC updating has to be done *during IF stage itself*, as otherwise the next instruction cannot be fetched.
- In the non-pipelined version discussed earlier, this was done during the MEM stage.

Pipeline Hazards

Introduction

- **Pipeline Hazards:**
  - Ideally, an instruction pipeline should complete the execution of an instruction every clock cycle.
  - Hazards refer to situations that prevent a pipeline from operating at its maximum possible clock speed.
    - Prevents some instructions from executing during its designated clock cycle.
- Three types of hazards:
  a) *Structural hazard*: Arise due to resource conflicts.
  b) *Data hazard*: Arise due to data dependencies between instructions.
  c) *Control hazard*: Arise due to branch and other instructions that change the PC.

What happens when hazards appear?

- We can use special hardware and control circuits to avoid the conflict that arise due to hazard.
- As an alternative, we can insert *stall cycles* in the pipeline.
  - When an instruction is stalled, all instructions that *follow* also get stalled.
  - Number of stall cycles depends on the criticality of the hazard.
  - Instructions *before* the stalled instruction can continue, but no new instructions are fetched during the duration of the stall.
- In general, hazards result in *performance degradation*.

(a) Structural Hazards

- They arise due to resource conflicts when the hardware cannot support overlapped execution under all possible scenarios.
- Examples:
  - Single memory (cache) to store instructions and data → while an instruction is being fetched some other instruction is trying to read or write data.
  - An instruction is trying to read data from the register bank (in ID stage), while some other instruction is trying to write into a register (in WB stage).
  - Some functional unit (like floating-point ADD or MULTIPLY) is not fully pipelined.
    - A sequence of instructions that try to use the same functional unit will result in stalls.

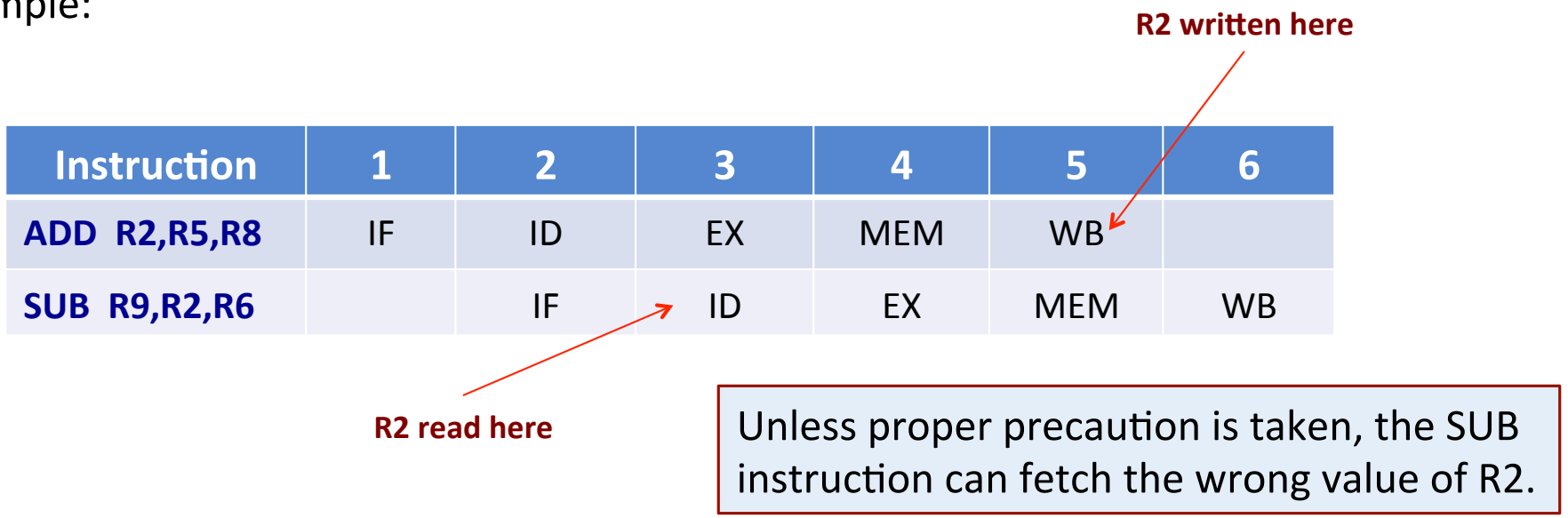
- Illustration:
  - Structural hazard in a single-port memory system, which stores both instructions and data.

Instruction	1	2	3	4	5	6	7	8
LW R1,10(R2)	IF	ID	EX	MEM	WB			
ADD R3,R4,R5		IF	ID	EX	MEM	WB		
SUB R10,R2,R9			IF	ID	EX	MEM	WB	
AND R5,R7,R7				STALL	IF	ID	EX	MEM
ADD R2,R1,R5					STALL	IF	ID	EX

Insert stall cycle to eliminate the structural hazard.

(b) Data Hazards

- Data hazards occur due to data dependencies between instructions that are in various stages of execution in the pipeline.
- Example:



- Naïve solution by inserting stall cycles:
  - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
  - 3 clock cycles are wasted.

Instruction	1	2	3	4	5	6	7	8
ADD R2,R5,R8	IF	ID	EX	MEM	WB			
SUB R9,R2,R6		IF	ID	STALL	STALL	ID	EX	MEM
Instr. (i+2)			IF	STALL	STALL	IF	ID	EX
Instr. (i+3)				STALL	STALL		IF	ID
Instr. (i+4)				STALL	STALL			IF

- How to reduce the number of stall cycles?
  - Two methods that can be used together:
    - **a)** **Data forwarding / bypassing**: By using additional hardware consisting of multiplexers, the data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register.
    - **b)** **Concurrent register access**: By splitting a clock cycle into two halves, register read and write can be carried out in the two halves of a clock cycle (register write in first half, and register read in second half).

Data Hazard while Accessing Memory

- In MIPS32 pipeline, memory references are always kept in order, and so data hazards between memory references never occur.
  - Cache misses can result in pipeline stalls.

Instruction	1	2	3	4	5	6
SW R2, 100(R5)	IF	ID	EX	MEM	WB	
LW R10, 100(R5)		IF	ID	EX	MEM	WB

Accessed in order; no hazard

- A LOAD instruction followed by the use of the loaded data.
  - Example of a data hazard that requires *unavoidable pipeline stalls*.

Instruction	1	2	3	4	5	6	7	8
LW R4, 100(R5)	IF	ID	EX	MEM	WB			
SUB R3, R4, R8		IF	ID	EX	MEM	WB		
ADD R7, R2, R4			IF	ID	EX	MEM	WB	
AND R9, R4, R10				IF	ID	EX	MEM	WB

ALU wants to use the loaded data; Forwarding cannot solve

Loaded data available here

Forwarding can solve

- What is the solution?
  - As we have seen the hazard cannot be eliminated by forwarding alone.
  - Common solution is to use a hardware addition called *pipeline interlock*.
    - The hardware detects the hazard and stalls the pipeline until the hazard is cleared.
  - The pipeline with stall is shown.

Instruction	1	2	3	4	5	6	7	8	9
LW R4, 100(R5)	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	STALL	EX	MEM	WB		
ADD R7, R2, R4			IF	STALL	ID	EX	MEM	WB	
AND R9, R4, R10				STALL	IF	ID	EX	MEM	WB

- A terminology:
  - **Instruction Issue**: This is the process of allowing an instruction to move from the ID stage to the EX stage.
  - In the MIPS32 pipeline, all possible data hazards can be checked in the ID stage itself.
    - If a data hazard (LOAD followed by use) exists, the instruction is *stalled* before it is *issued*.

- A common example:

  A = B + C

- Pipelined execution of the corresponding MIPS32 code is shown.

Instruction	1	2	3	4	5	6	7	8	9
LW R1, B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	STALL	EX	MEM	WB		
ADD R5, R1, R2			IF	STALL	ID	EX	MEM	WB	
SW R5, A				STALL	IF	ID	EX	MEM	WB

Instruction Scheduling or Pipeline Scheduling:

- Compiler tries to avoid generating code with a LOAD followed by an immediate use.

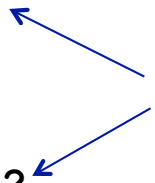
Example

A C code segment:

```
x = a – b;
y = c + d;
```

MIPS32 code:

```
LW   R1, a
LW   R2, b
SUB  R8, R1, R2
SW   R8, x
LW   R1, c
LW   R2, d
ADD  R9, R1, R2
SW   R9, y
```



Two load interlocks

Scheduled MIPS32 code:

```
LW   R1, a
LW   R2, b
LW   R3, c
SUB  R8, R1, R2
LW   R4, d
SW   R8, x
ADD  R9, R3, R4
SW   R9, y
```

Both load interlocks are eliminated

(c) Control Hazard

- Control hazards arise because of branch instructions being executed in a pipeline.
  - Can cause greater performance loss than data hazards.
- What happens when a branch is executed?
  - If the branch is taken, the PC is normally not updated until the end of MEM.
  - The next instruction can be fetched only after that (3 stall cycles).
  - Actually, by the time we know than an instruction is a branch, the next instruction has already been fetched.
    - We have to redo the fetch if it is a branch.

Instruction	1	2	3	4	5	6	7	8	9	10	11
BEQ Label	IF	ID	EX	MEM	WB						
Instr. (i+1)		*IF*	Stall	Stall	IF	ID	EX	MEM	WB		
Instr. (i+2)			Stall	Stall	Stall	IF	ID	EX	MEM	WB	
Instr. (i+3)				Stall	Stall	Stall	IF	ID	EX	MEM	WB
Instr. (i+4)					Stall	Stall	Stall	IF	ID	EX	MEM
Instr. (i+5)						Stall	Stall	Stall	IF	ID	EX
Instr. (i+6)							Stall	Stall	Stall	IF	ID