Computer Architecture

CSEN 3104

Lecture 6

Dr. Debranjan Sarkar

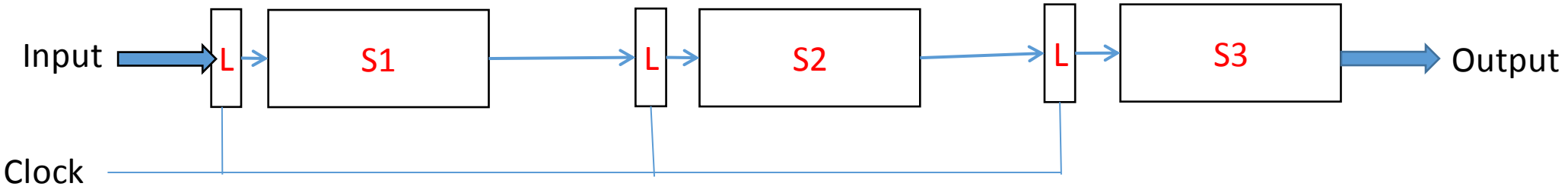
Basics of Pipelining

Basics of pipelining

- Modern CPUs employ a variety of speedup techniques viz.
  - cache memory
  - Pipelining etc..
- Pipelining allows the processing of several instructions to be partially overlapped (parallelism)
- Pipelining increases the overall instruction throughput.
- All the common steps involved in instruction processing by the CPU can be pipelined:
  - Instruction Fetching (IF)
  - Instruction decoding (ID)
  - Operand loading (OL)
  - Execution (EX)
  - Operand Storing (OS)

Principles of Pipelining

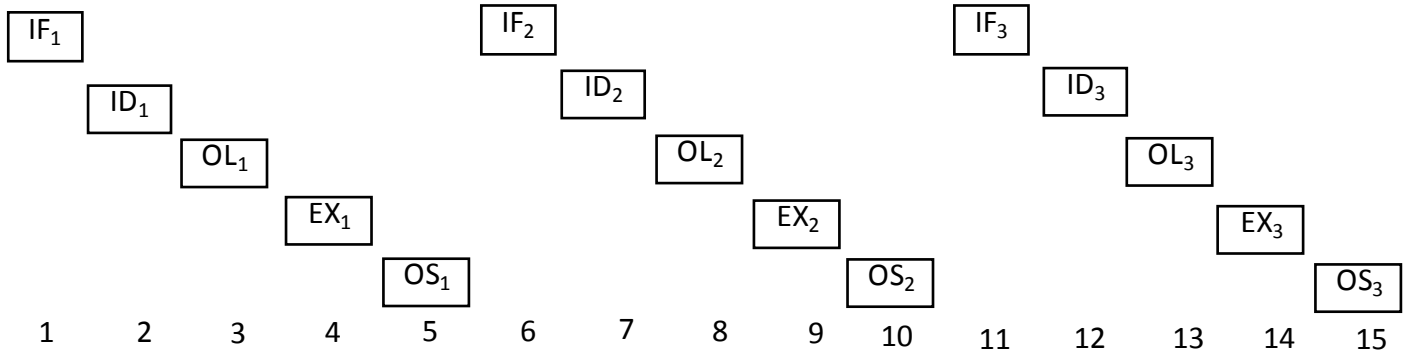
- Decomposes a sequential task into subtasks
- Each subtask is executed in a special dedicated stage
- These stages are connected with one another to form a pipe like structure.
- Instructions enter from one end and exit from another end
- Stages are pure combinational circuits for arithmetic or logic operations
- Result obtained from a stage is transferred to the next stage
- Final result is obtained after the instruction has passed through all the stages
- Stages are separated by high speed latches (or registers)
- All latches transfer data to the next stage simultaneously (on clock)



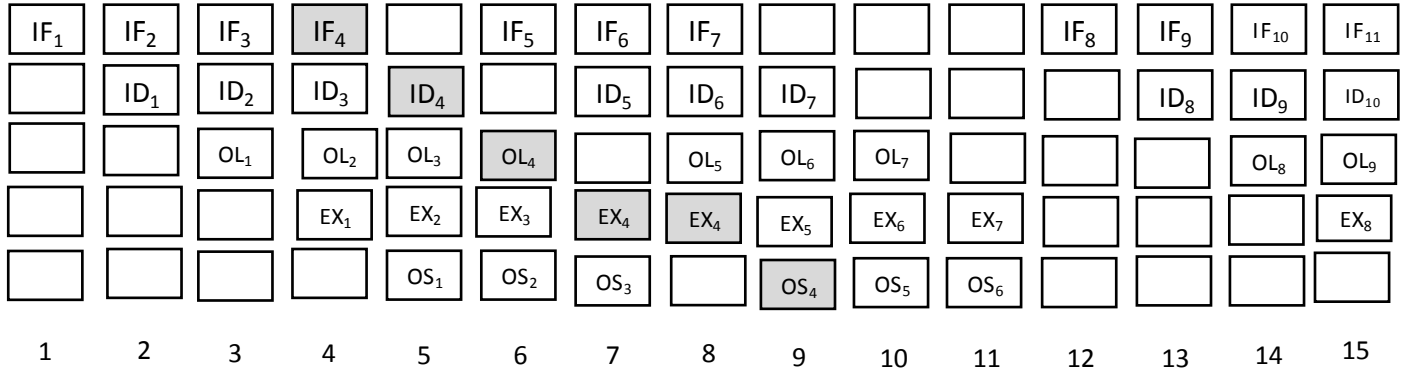
Instruction Processing :

(a) Non-pipelined (b) Pipelined

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Operand Load (OL)
- Execution (EX)
- Operand Store (OS)
- Time (clock cycles)



- Instruction Fetch (IF)
- Instruction Decode (ID)
- Operand Load (OL)
- Execution (EX)
- Operand Store (OS)
- Time (clock cycles)



Pipelining

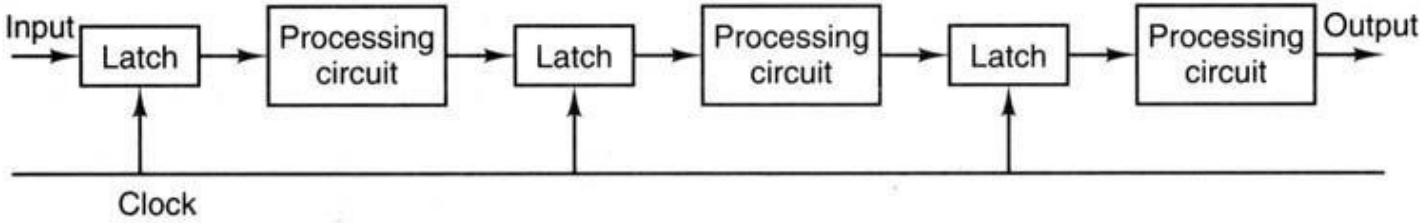
- In the example, upto five instructions can be overlapped,

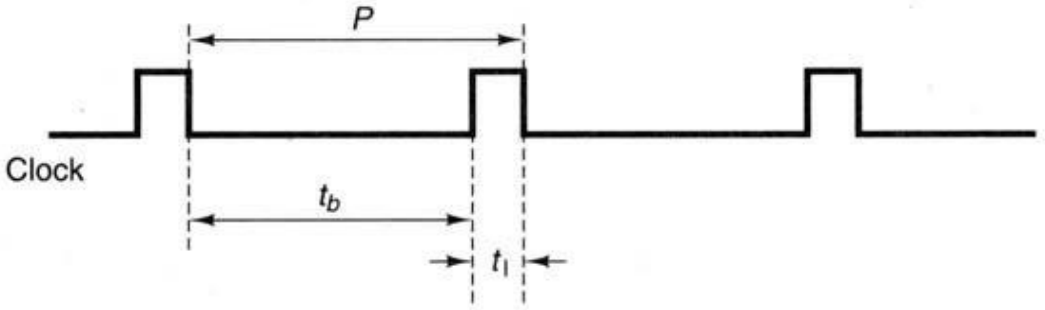
  provided, necessary pipeline stages are available

- Example of performance reducing delays
  - Instruction I_4 (shaded) uses the EX stage for two consecutive cycles
  - Instruction I_7 (branch), where the outcome of the I_7's EX step must be known before the location of the next instruction (I_8) to be processed can be identified.

Performance measure of a pipeline

- Speedup
- Efficiency
- Throughput





P: Clock period.
t_b: Maximum time for a stage to perform a function.
t_l: Time for latch to accept input data.

Speedup of a pipeline

- Let
  - *n* be the number of input tasks,
  - *m* the number of stages in the pipeline,
  - *P* the clock period
- The time required for the first input task to get through the pipeline = $1*m*P$
- The time required for the remaining tasks = $(n-1)*1*P$
- Note that after the pipeline has been filled, it generates an output on each clock cycle.
- Overall theoretical completion time (T_{pipe}) = $m*P + (n-1)*P$
- The pipeline will greatly outperform nonpipelined techniques, which require each task to complete before another task's execution sequence begins.

Speedup of a pipeline

- In a nonpipelined processor, the above sequential process requires a completion time of

  $T_{seq} = n * m * \tau$, where τ is the delay of each stage

- If we ignore the small storing time t_l that is required for latch storage (i.e., $t_l = 0$), then

  $T_{seq} = n * m * P$

- Now, *speedup* (S) may be represented as:

  $S = T_{seq} / T_{pipe} = n*m / (m+n-1)$

- The value S approaches m when $n \rightarrow \infty$. That is, the maximum speedup, also called ideal speedup, of a pipeline processor with m stages over an equivalent nonpipelined processor is m
- In other words, the ideal speedup is equal to the number of pipeline stages
- That is, when n is very large, a pipelined processor can produce output approximately m times faster than a nonpipelined processor
- When n is small, the speedup decreases. For n=1, the pipeline has the minimum speedup (= 1)

Efficiency and Throughput of a pipeline

- The efficiency E of a pipeline is defined as the speedup per stage
- If m is the number of stages, then

$$E = S/m = [n*m / (m+n -1)] / m = n / (m+n -1)$$

- The efficiency approaches its maximum value of 1 when $n \rightarrow \infty$
- When n=1, E will have the value $1/m$, which is the lowest obtainable value
- The throughput H, also called bandwidth, of a pipeline is defined as the number of input tasks it can process per unit of time
- When the pipeline has m stages, H is defined as

$$H = n / T_{pipe} = n / [m*P + (n-1)*P] = E / P = S / (mP)$$

- When $n \rightarrow \infty$, the throughput H approaches the maximum value of one task per clock cycle

Performance-cost-ratio of a pipeline

- The number of stages in a pipeline often depends on the tradeoff between performance and cost
- The optimal choice for such a number can be determined by obtaining the maximum value of a performance/cost ratio (PCR)
- PCR is defined as the ratio of maximum throughput to pipeline cost
- Throughput (H) =E /P
- As the maximum value of efficiency (E) is 1, so maximum throughput is 1/P
- Now P = $(t_{seq}/m) + t_l$
- Thus the maximum throughput that can be obtained with such a pipeline is

$$1/P = 1/[(t_{seq}/m) + t_l]$$

- The maximum throughput $1/P$ is also called the pipeline frequency
- The actual throughput may be less than $1/P$

Performance-cost-ratio of a pipeline

- The pipeline cost c_p can be expressed as the total cost of logic gates (c_g) and latches ($m*c_l$) used in m number of stages.

$$c_p = c_g + m*c_l$$

- Note that the cost of gates and latches may be interpreted in different ways:
  - Actual cost
  - design complexity
  - the area required on the chip or circuit board
- PCR = $1/\{[(t_{seq}/m) + t_l](c_g + m*c_l)\}$
- This equation has a maximum value when m = sqrt $\{(t_{seq} * c_g) / (t_l * c_l)\}$
- This value can be used as an optimal choice for the number of stages.

Thank You

Pipelined Architecture

CSEN 3104

Lecture 7

Dr. Debranjan Sarkar

Types of parallelism

- Classification proposed by Wolfgang Handler (1977)
- Parallelism and pipelining in 3 distinct levels
  - Processor Control Unit (PCU) =>
    - Processor or CPU
  - Arithmetic Logic Unit (ALU) =>
    - Functional Unit or a processing element (PE) in an array processor
    - An element much smaller than a central processor and having much lower features than a processor
    - working under the control of the processor
    - there are many ALUs in a system, working in parallel to increase the speed of the system
  - Bit Level Circuit (BLC) =>
    - Combinational logic circuit needed to perform the bit operations in the ALU

3 Types of pipelining

- Instruction pipelines
  - Instruction Fetching (IF)
  - Instruction decoding (ID)
  - Operand loading (OL)
  - Execution (EX)
  - Operand Storing (OS)
- Arithmetic pipelines
- Processor pipelines
  - a cascade of processors each executing a specific module in the application program

Hazards of Instruction Pipeline

- Situations preventing execution of the next instruction in the instruction stream during its designated clock cycle
- The instruction is said to be stalled and a hazard is said to exist for that instruction
- The instructions later in the pipeline are also stalled
- Earlier instructions can continue
- No new instructions are fetched during the stall
- Pipeline cannot execute instructions at its peak rate

- 3 types of Hazards
  - (1) Structural hazards
  - (2) Data hazards
  - (3) Control hazards

Structural Hazards

- A structural hazard refers to a situation in which a required resource is not available (or is busy) for executing an instruction.
- Resource conflicts => Structural hazards
  - use of same resource in different stages
- Resource can be
  - Memory (data and instruction)
  - Functional Unit, which is not fully pipelined
- Example: Let the i^{th} instruction be ADD R4,X
- Here Penalty = 1 cycle

Clock	1	2	3	4	5	6	7	8	9	10
ADD R4,X	IF	ID	OF	EX	WB					
$(i+1)^{th}$ $inst^n$		IF	ID	OF	EX	WB				
$(i+2)^{th}$ $inst^n$			Stall	IF	ID	OF	EX	WB		
$(i+3)^{th}$ $inst^n$					IF	ID	OF	EX	WB	

Structural Hazards

- [1] If an execution unit that requires more than one clock cycle (such as multiply) is not fully pipelined or is not replicated, then a sequence of instructions that uses the unit cannot be subsequently (one per clock cycle) issued for execution.
- Replicating and/or pipelining execution units increases the number of instructions that can be issued simultaneously.
- [2] Another type of structural hazard that may occur is due to the design of register files.
- If a register file does not have multiple write (read) ports, multiple writes (reads) to (from) registers cannot be performed simultaneously.
- For example, under certain situations the instruction pipeline might want to perform two register writes in a clock cycle.
- This may not be possible when the register file has only one write port.
- The effect of a structural hazard can be reduced fairly simply by implementing multiple execution units and using register files with multiple input/output ports.

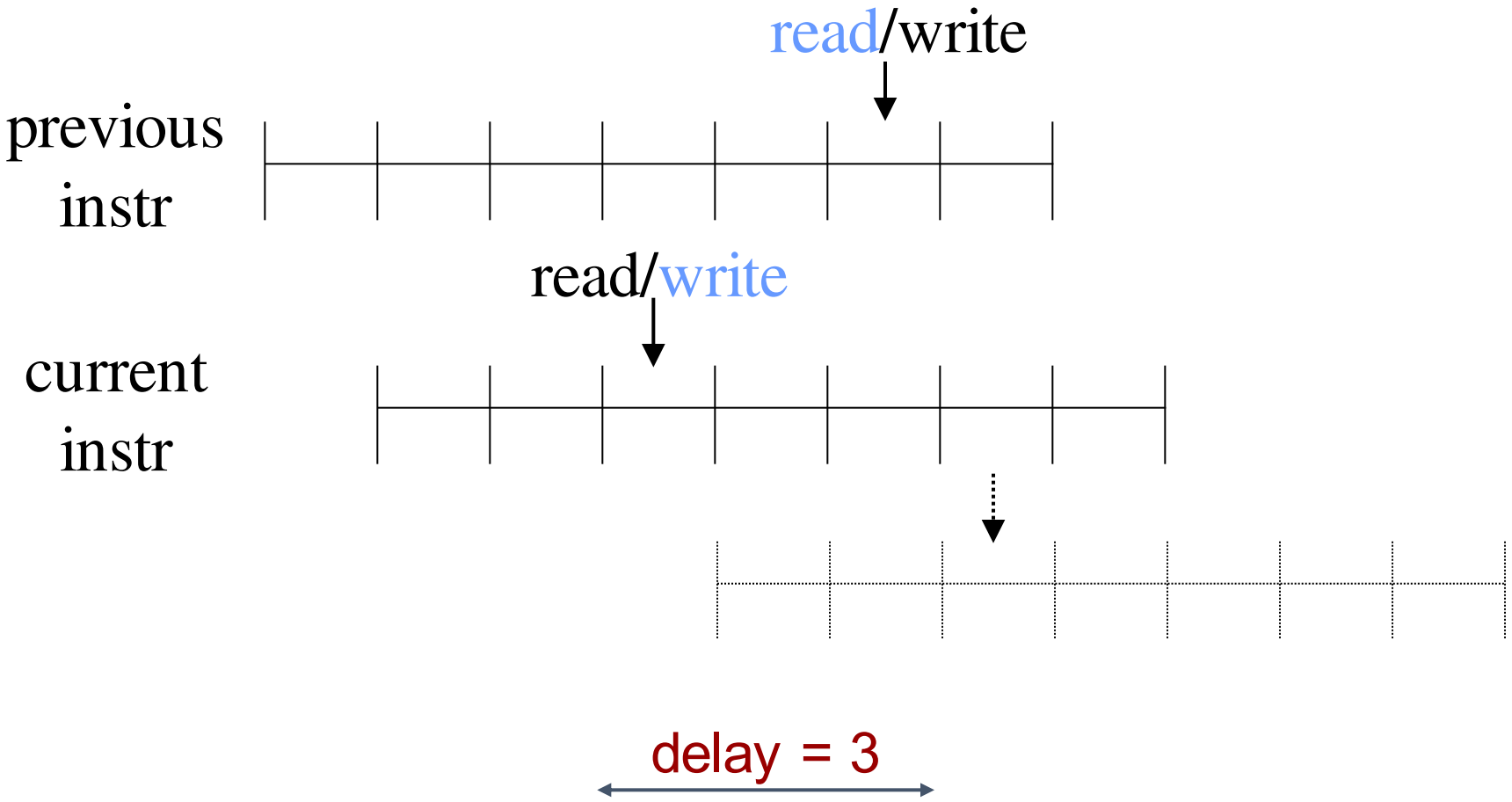
Techniques to solve Structural Hazards

- Certain resources are duplicated
- Functional Units (ALU, FP unit) can be pipelined themselves in order to support several instructions at a time
- Structural hazard due to memory conflict is avoided by providing separate data cache and instruction cache

Data Hazards

- A data hazard refers to a situation in which there exists a data dependency (operand conflict) with a previous instruction
- Data dependencies => Data hazards
- Example:
  - Two instructions I_1 and I_2 are in pipeline
  - The execution of I_2 can start before I_1 has terminated
  - If I_2 needs the result produced by I_1, then there is a data hazard
- Three classes of Data hazards
  - RAW (read after write)
  - WAR (write after read)
  - WAW (write after write)
- Read-after-Read (RAR) is not a hazard as no data is changed

Data Hazards



Data Hazards: RAW (Read after Write)

- Instruction (I): Write data object X
- Instruction (J): Read data object X
- J tries to read data before it is written by I
- So, J gets old value of data which is incorrect
- Program order must be preserved to ensure that J gets the correct data value

Data Hazards: WAW (Write after Write)

- Instruction (I): Write data object X
- Instruction (J): Write (or modify) data object X
- J tries to write (modify) data before it is written by I
- So, finally the data object written by instruction I prevails which is not desirable
- It was desired to have the final value of data object X written by instruction J and not by instruction I
- Program order must be preserved to ensure that J gets the correct data value

Data Hazards: WAR (Write after Read)

- Instruction (I): Read data object X
- Instruction (J): Write (or modify) data object X
- J tries to write (modify) data object before it is read by I
- So, the data object read by instruction I is incorrect
- It was desired that instruction I would read the old value of X and then the data object would be modified by instruction J
- Program order must be preserved to ensure that J gets the correct data value

Thank you

Pipelined Architecture

CSEN 3104

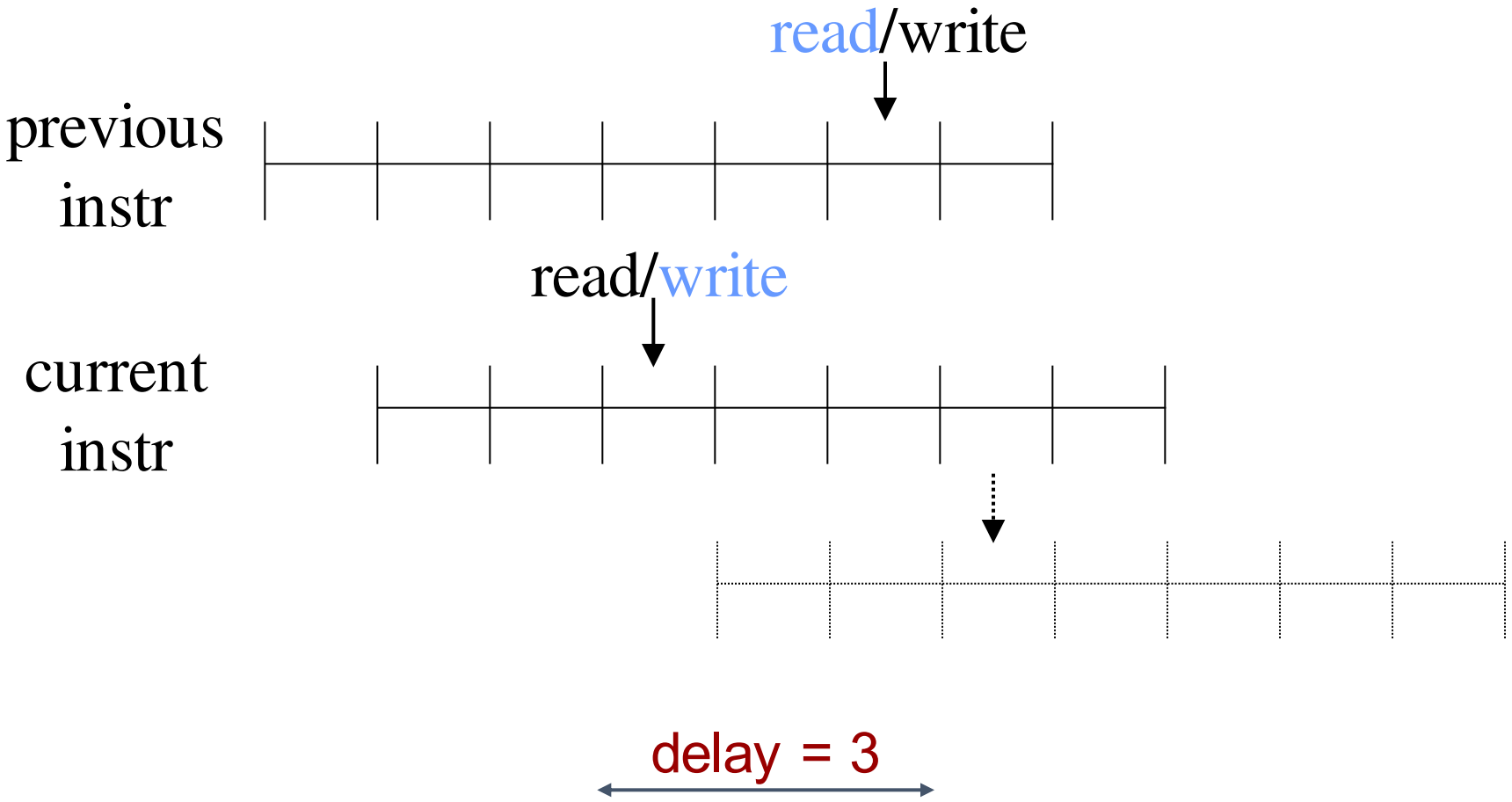
Lecture 8

Dr. Debranjan Sarkar

Data Hazards

- A data hazard refers to a situation in which there exists a data dependency (operand conflict) with a previous instruction
- Data dependencies => Data hazards
- Example:
  - Two instructions I_1 and I_2 are in pipeline
  - The execution of I_2 can start before I_1 has terminated
  - If I_2 needs the result produced by I_1, then there is a data hazard
- Three classes of Data hazards
  - RAW (read after write)
  - WAR (write after read)
  - WAW (write after write)
- Read-after-Read (RAR) is not a hazard as no data is changed

Data Hazards



Data Hazards: RAW (Read after Write)

- Instruction (I): Write data object X
- Instruction (J): Read data object X
- J tries to read data before it is written by I
- So, J gets old value of data which is incorrect
- Program order must be preserved to ensure that J gets the correct data value

Data Hazards: WAW (Write after Write)

- Instruction (I): Write data object X
- Instruction (J): Write (or modify) data object X
- J tries to write (modify) data before it is written by I
- So, finally the data object written by instruction I prevails which is not desirable
- It was desired to have the final value of data object X written by instruction J and not by instruction I
- Program order must be preserved to ensure that J gets the correct data value

Data Hazards: WAR (Write after Read)

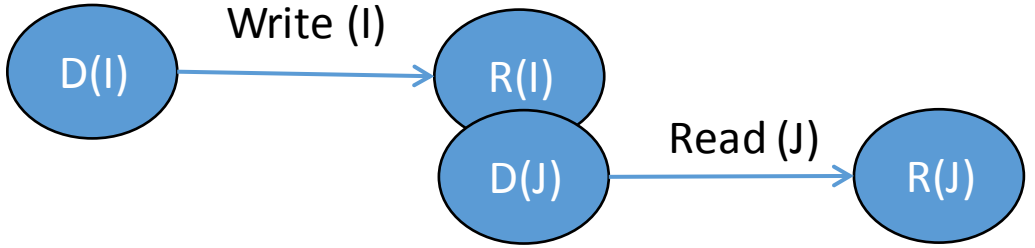
- Instruction (I): Read data object X
- Instruction (J): Write (or modify) data object X
- J tries to write (modify) data object before it is read by I
- So, the data object read by instruction I is incorrect
- It was desired that instruction I would read the old value of X and then the data object would be modified by instruction J
- Program order must be preserved to ensure that J gets the correct data value

Detection of Data Hazards

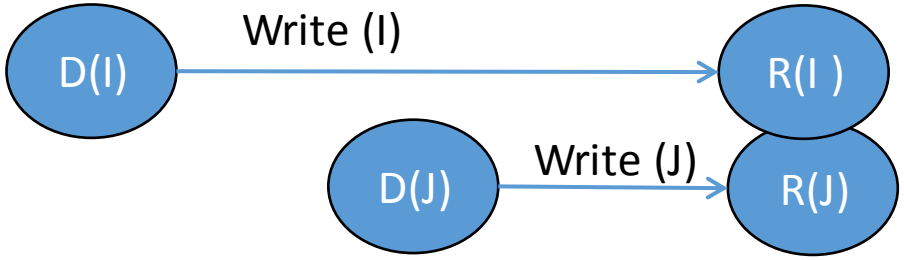
- Definitions:
  - Resource Object: Working registers, memory locations, flags
  - Data Object: Contents of the Resource Objects
  - Domain of an Instruction D(I):
    - Set of resource objects, whose data objects may affect the execution of the instruction I
    - Holds the operands to be read for execution of the instruction
  - Range of an Instruction R(I):
    - Set of resource objects, whose data objects may be modified by the execution of the instruction I
    - Holds the results produced after execution of the instruction

Conditions for data hazards

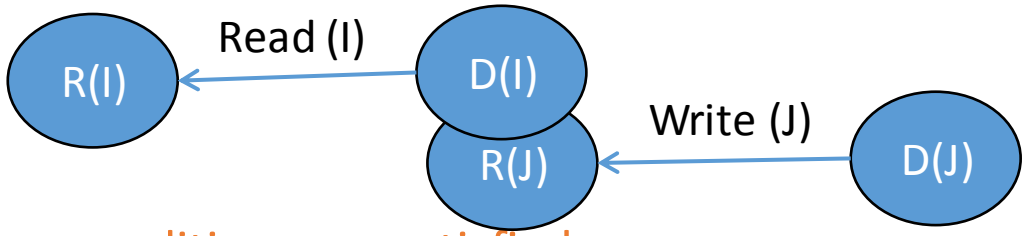
- I and J are two instructions in a program
- I occurs before J
- RAW Hazard: $R(I) \cap D(J) \neq \phi$



- WAW Hazard: $R(I) \cap R(J) \neq \phi$



- WAR Hazard: $D(I) \cap R(J) \neq \phi$



- These conditions are necessary (but not sufficient)
- Hazard may not appear even if one or more of the above conditions are satisfied
- If Hazard appears, then at least one of the above conditions must be satisfied
- If the two instructions are executed in right order, hazard will not occur

Detection of Data Hazard

- Data hazard is detected in the Instruction Fetch (IF) stage of a pipeline
- The domain and range of the fetched instruction is found out
- The domain and range of other instructions, being processed in the pipeline, is found out
- If any of the necessary conditions, stated earlier, is detected, a warning signal is issued to prevent the hazard from taking place

Solution of Data Hazard: stall the pipeline

- Let, a hazard has been detected between the current instruction (J) and a previous instruction (I)
- Simple solution:
  - Stall the pipeline and ignore the execution of the instructions J, J+1, J+2, …., until the instruction I has passed the point of resource conflict
- Advanced solution:
  - Ignore only instruction J and continue with the instructions J+1, J+2, ….
  - The potential hazards due to the suspension of J must be continuously tested as the instructions J+1, J+2, … execute prior to J
  - Multi-level hazard detection may be encountered, which requires very complex control policies

Solution of Data Hazard: stall the pipeline

- Example
- Let, a hazard has been detected between the current instruction (J) and a previous instruction (I)
- Instruction I: ADD R1, R2, R3
- Instruction J: SUB R4, R1, R6

	1	2	3	4	5	6	7
IF	I	J					
ID		ADD	SUB				
OF			R_2, R_3	~~R_1, R_6~~ Delay	Delay	R_1, R_6	
EX				R_2+R_3	~~R_1 R_6~~		R_1-R_6
WB					Res to R_1		

Delay 2 cycles

Solution of Data Hazard: data forwarding

- The data obtained at the Execution phase of the first instruction may be forwarded to the operand fetch unit of the 2nd instruction
- Instruction I: ADD R1, R2, R3
- Instruction J: SUB R4, R1, R6
- After data forwarding, the delay is reduced to 1 cycle

	1	2	3	4	5	6	7
IF	I	J					
ID		ADD	SUB				
OF			R_2, R_3	Delay	$R_{2+}R_3$, R_6		
EX				R_2+R_3		$R_{2+}R_3$-R_6	
WB					Res to R_1		

Delay Reduced to 1 cycle
By Data Forwarding

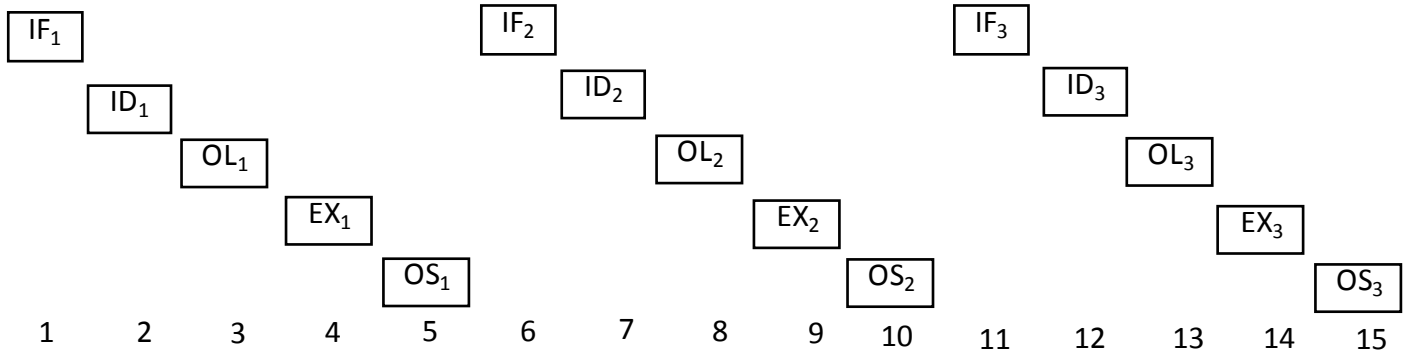
Control Hazards

- A control hazard refers to a situation in which an instruction, such as branch, causes a change in the program flow
- Procedural dependencies => Control hazards
  - Instructions that change the content of program counter
  - Example: unconditional and conditional branches, calls/returns
- Processor cannot determine which instruction to fetch next until the branch instruction is executed completely
- This causes delay in pipelined processors
- Control hazards occur less frequently than data hazards

Instruction Processing :

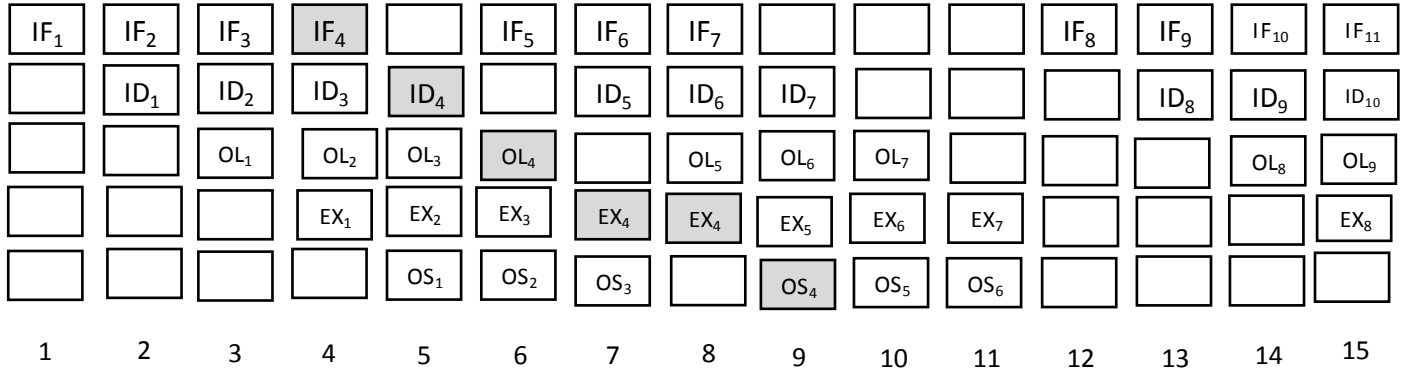
(a) Non-pipelined

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Operand Load (OL)
- Execution (EX)
- Operand Store (OS)
- Time (clock cycles)



(b) Pipelined with I_7 as Jump instruction

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Operand Load (OL)
- Execution (EX)
- Operand Store (OS)
- Time (clock cycles)



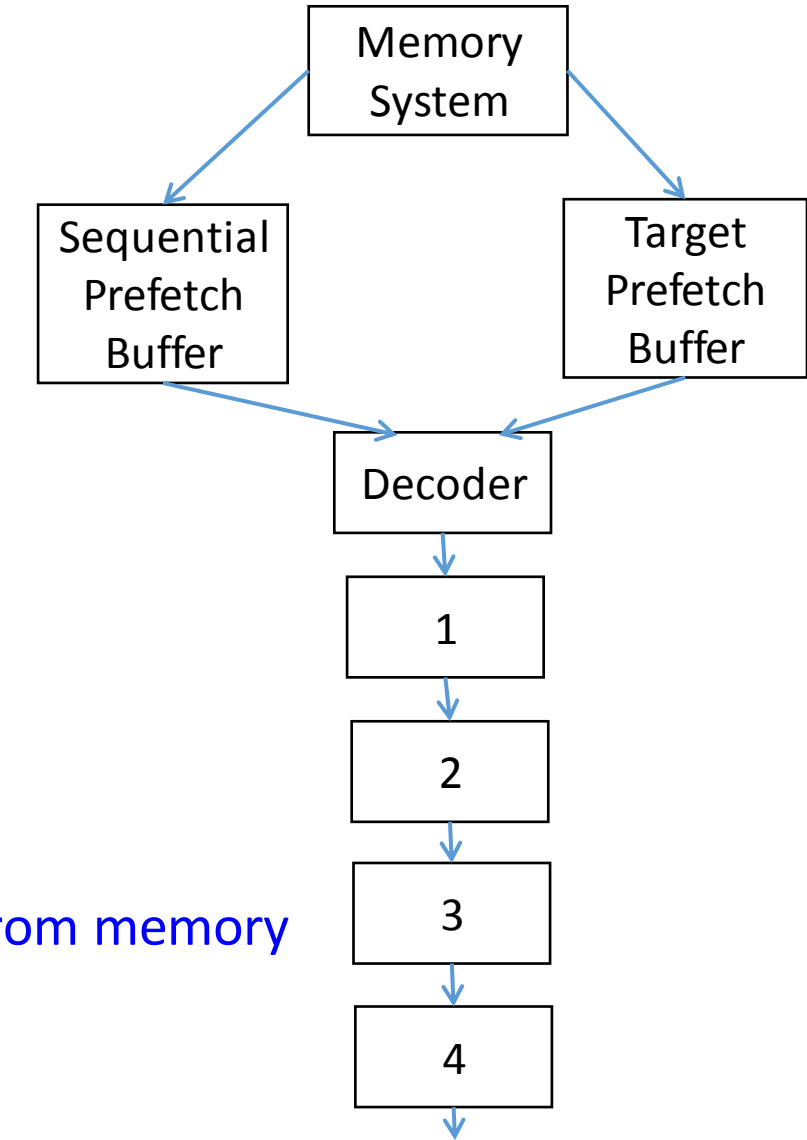
Penalty : 3 cycles

Techniques to mitigate Control Hazards: Freeze the pipeline

- Freeze the pipeline
  - until the branch outcome and target are known and then proceed with fetch
  - incurs a penalty equal to the number of stall cycles
  - unsatisfactory if
    - the instruction mix contains many branch instructions
    - the pipeline is very deep

Techniques to mitigate Control Hazards: Pre-fetching

- Pre-fetching
  - Pre-fetch both the target instruction part and the instructions following the branch
  - Multiport memory with concurrent access
  - If a conditional branch is successful,
    - Entire sequential prefetch buffer is invalidated
    - Target prefetch buffer is validated
  - And vice versa
  - When instruction is requested by the decoder,
    - It enters the decoder from one of the Prefetch Buffers depending on the situation
    - No delay is incurred
  - Otherwise, the decoder is idle until the instruction comes from memory



Techniques to mitigate Control Hazards: Branch Prediction

- Uses some additional logic (heuristic) to predict the outcome of a conditional branch instruction before it is executed
- Normally two strategies are followed
  - Static Branch Strategy: Probability of branch
  - Dynamic Branch Strategy: Branch history is taken into consideration
- Instructions are speculatively fetched and executed down the predicted path
- But results are not written back to the register file until the branch is executed and the prediction is verified
- When a branch is predicted, the processor enters a *speculative mode* in which results are written to another register file that mirrors the architected register file
- Another pipeline stage called the *commit* stage is introduced to handle writing verified speculatively obtained results back into the "real" register file
- Branch predictors cannot be 100% accurate
- So there is still a penalty for branches if the prediction is found to be incorrect

Techniques to mitigate Control Hazards: Delayed Branch

- The compiler detects the branch instruction and rearranges the machine language code sequence by inserting useful instructions or NOPs (No operations) that keep the pipeline operating without interruptions
- The Assembly Language Program developer must fill these branch delay slots
- This causes the computer to fetch the target instruction during the execution of the other useful instructions (or NOPs), allowing a continuous flow of the pipeline
- This solution does not extend well to deeper pipelines

Example of Delayed Branch

- I1: Load R1, A
- I2: Decrement R3, 1
- I3: Branch zero R3, I5
- I4: Add R2, R4
- I5: Subtract R5, R6
- I6: Store R5, B

Original Program

- I2: Decrement R3, 1
- I3: Branch zero R3, I5
- I1: Load R1, A
- I4: Add R2, R4
- I5: Subtract R5, R6
- I6: Store R5, B

Reordered Instructions

Example of Delayed Branch

- By reordering, I1, I4, and I5 are executed regardless of the branch outcome
- If the branch is unsuccessful,
  - Produces the same results as the original program
- If the branch is successful,
  - Execution of delayed instructions I1 and I5 needed anyway
  - Only one cycle is wasted in executing instruction I4, which is not needed
- In case of 5-stage instruction pipeline:
  - The delay slot is reduced to 1 for an unsuccessful branch
  - The delay slot is reduced to 2 for a successful branch

More about Delayed Branch

- Data dependencies between instructions moving across the branch and the remaining instructions being scheduled must be analyzed
- Instructions I1 and I4 are independent of the remaining instructions (I2, I3, I5 and I6), leaving them in the delay slot will not create data hazards
- Inserting NOP fillers does not save any cycles in the delayed branch operation
- Delayed branching is more effective in short instruction pipelines with about four stages
- Delayed branching is implemented in most RISC processors, including MIPS R4000

Classification of pipelines

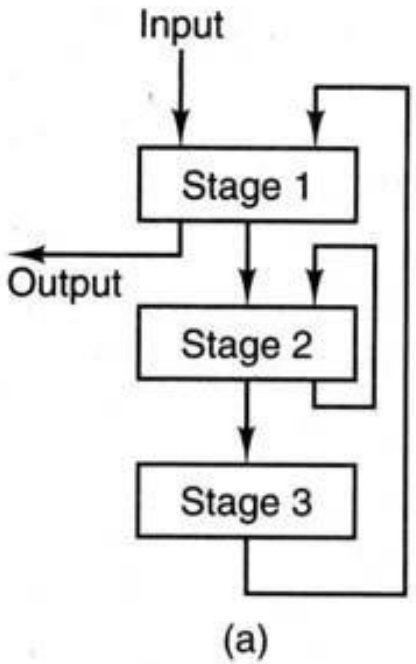
- As per Configuration, pipelines are of two types:
- Static Pipeline:
  - A static pipeline can perform only one function (or operation such as addition or multiplication) at a time
  - The operation of a static pipeline can only be changed after the pipeline has been drained
  - A pipeline is said to be drained when the last input data leave the pipeline
  - Thus, the performance of static pipelines is severely degraded when the operations change often
- Dynamic Pipeline:
  - a dynamic pipeline can perform more than one function at a time

Reservation Table

- A pipeline reservation table shows when stages of a pipeline are in use for a particular function
- Each stage of the pipeline is represented by a row in the reservation table
- Each column of the reservation table represents one clock cycle
- The number of columns indicates the total number of time units required for the pipeline to perform a particular function.
- To indicate that some stage S is in use at some time t_y, an X is placed at the intersection of the row and column in the table corresponding to that stage and time
- Multiple checkmarks in a row, means repeated usage of the same stage in different cycles

Reservation Table

- *3 stages, 5 clock cycles*
- The input data goes through the stages 1, 2, 2, 3 and 1 progressively



(a)

	Time t_0	t_1	t_2	t_3	t_4
Stage 1	X				X
Stage 2		X	X		
Stage 3				X	

(b)

A static pipeline and its corresponding reservation table.

Thank you

Pipelined Architecture

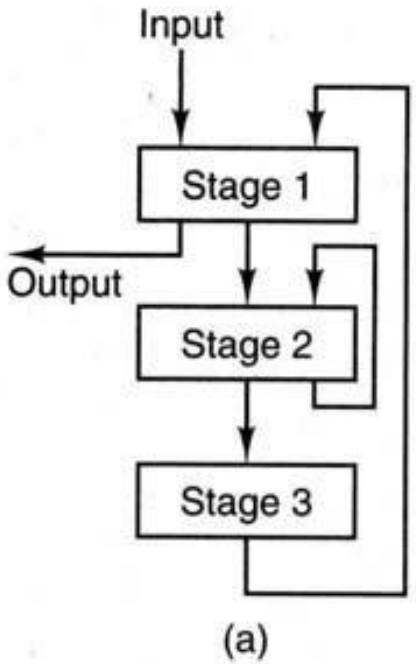
CSEN 3104

Lecture 9

Dr. Debranjan Sarkar

Reservation Table

- *3 stages, 5 clock cycles*
- The input data goes through the stages 1, 2, 2, 3 and 1 progressively



(a)

	Time t_0	t_1	t_2	t_3	t_4
Stage 1	X				X
Stage 2		X	X		
Stage 3				X	

(b)

A static pipeline and its corresponding reservation table.

Scheduling of Static Pipelines

Pipeline Control: Scheduling

- Controlling the sequence of tasks presented to a pipeline for execution is extremely important for maximizing its utilization.
- If two tasks are initiated requiring the same stage of the pipeline at the same time, a collision occurs, which temporarily disrupts execution.
- The reservation table can be used for determining the time difference between input data initiations so that collisions won't occur.
- Initiation of an input data refers to the time that the data enter the first stage of the pipeline.

Some Definitions

- Initiation: Launching of an operation into the pipeline. i.e. the time that the data enter the first stage of the pipeline

- Latency: the delay (or, the number of cycles) that elapse between two initiations of a pipeline
- Latency values must be non-negative integers
- A collision will occur if two pieces of input data are initiated with a latency equal to the distance between two X's in a reservation table
- For example, the reservation table, shown earlier, has two X's with a distance of 1 in the second row
- So, if a second piece of data is passed to the pipeline one time unit after the first, a collision will occur in stage 2
- A latency value *k* means that two initiations are separated by *k* clock cycles

Some definitions

- Collision
  - Collision occurs if a stage in the pipeline is required to perform more than one task at any time
  - Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.
  - A collision will occur if two pieces of input data are initiated with a latency equal to the distance between two X's in a reservation table
  - For example, the reservation table, shown earlier, has two X's with a distance of 1 in the second row
  - Therefore, if a second piece of data is passed to the pipeline one time unit after the first, a collision will occur in stage 2
  - A collision implies resource conflicts between two initiations in the pipeline, so it should be avoided

Forbidden Latency

- Latencies that cause collision are called forbidden latencies
- Forbidden latencies should be prohibited
- Otherwise, the two data would arrive at the same stage of the pipeline at the same time and lead to collision
- Forbidden latencies for the previous RT are 1 and 4
- Let the Maximum forbidden latency be m (here 4)
- n = no. of columns ($m \leq n-1$)
- With static pipelines, zero is always considered a forbidden latency, since it is impossible to initiate two jobs to the same pipeline at the same time.
- However, such initiations are possible with dynamic pipelines

Permissible Latency

- Latencies that do not cause any collision are called permissible latencies
- Permissible latencies for the previous RT are 2 and 3, as they do not cause collision
- Let the Maximum forbidden latency be m
- All the latencies greater than m 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

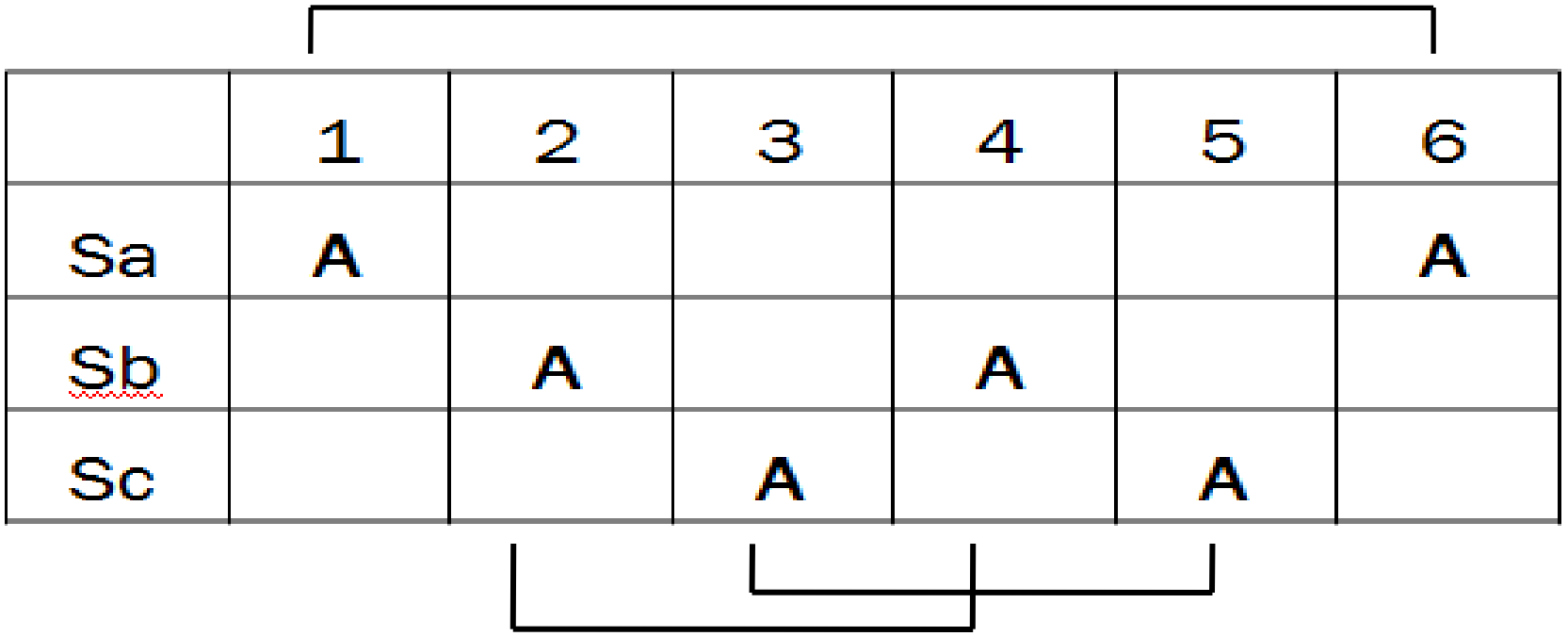
Forbidden Latency set or Forbidden List

- Reservation table, having more than one X's in any given row, has one or more forbidden latencies, which, if not prohibited, would allow two data to collide or arrive at the same stage of the pipeline at the same time
- Forbidden Latency set: the set of all possible column distances between two entries in a particular row of Reservation Table
- The forbidden list F is simply a list of integers corresponding to these prohibited Latencies
- As 0 is always considered a forbidden latency for static pipelines, 0 may be included in the Forbidden List

Collision Vectors

- A collision vector is a combined set of permissible and forbidden latencies.
- Let the Maximum forbidden latency be m (here 4) and n = no. of columns ($m \leq n-1$)
- Then the collision vector is an m-bit binary number $C = (C_m\ C_{m-1}....C_2\ C_1\ )$
- The initial collision vector, C, is created from the forbidden list in the following way:
  - each component C_i of C, for i=1 to m, is 1 if i is an element of the forbidden list.
  - Otherwise, C_i is zero

Example 1



	1	2	3	4	5	6
Sa	A					A
Sb		A		A		
Sc			A		A	

- Forbidden Latency Set, F = {5} U {2} U {2} = { 2,5}
- Permissible Latency List = { 1,3,4}
- Initial Collision Vector = 10010

State Diagram

- State Diagram is a graph of all the possible operation sequences through the pipeline
- State diagrams can be constructed to specify the permissible transitions among successive initiations
- State diagrams can be used to show the different states of a pipeline for a given time slice
- Once a state diagram is created, it is easier to derive schedules of input data for the pipeline that have no collisions

Procedure for construction of State Diagram

1. Start with the ICV
2. For each unprocessed state,

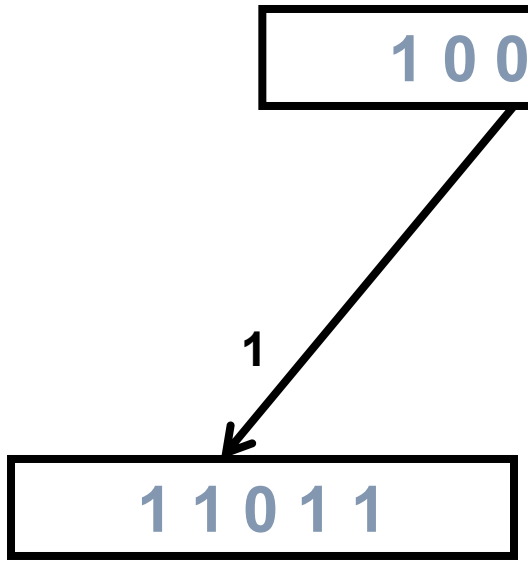
   For each bit i in the CV_i which is 0, do the following:

a. Shift CV_i right by i bits
b. Drop i rightmost bits
c. Append zeros to left
d. Logically OR with ICV
e. If step(d) results in a new state then form a new node for this state and join it with node of CV_i by an arc with a marking i.

- This shifting process needs to continue until no more new states can be generated.

Example 1: State Diagram

Example 1: State Diagram



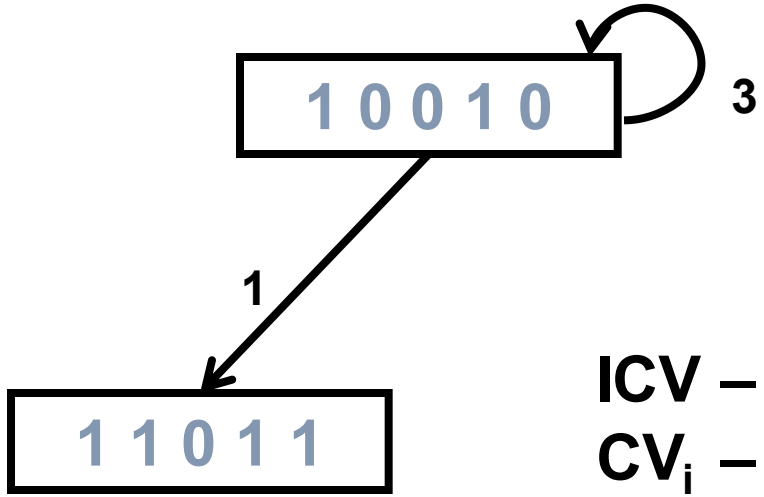
i = 1

ICV – 10010 OR

CV_i – 01001

CV* 11011

Example 1: State Diagram



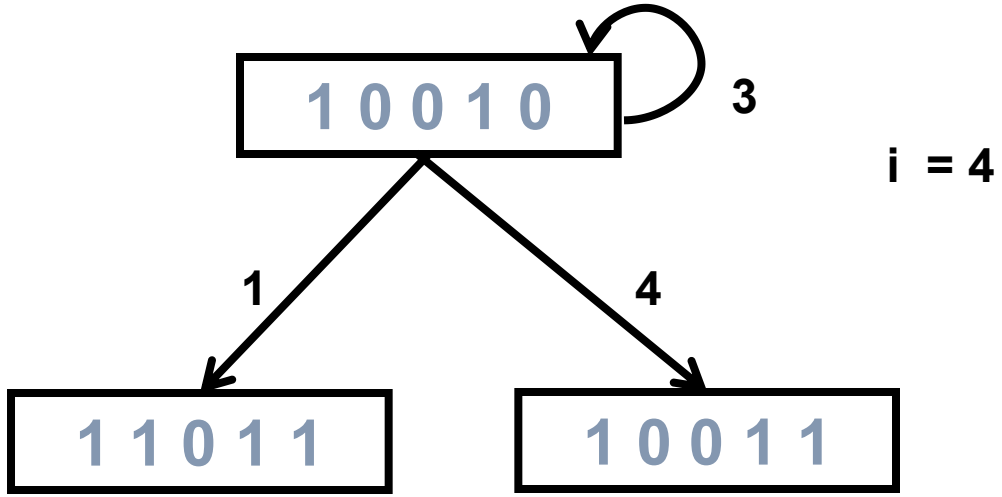
i = 3

ICV – 10010 OR

CV_i – 00010

CV* 10010

Example 1: State Diagram

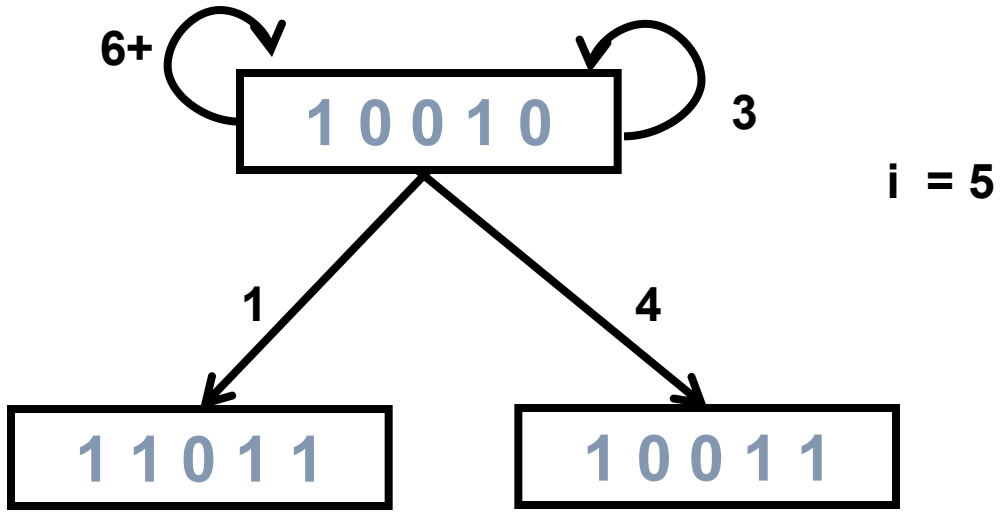


ICV – 10010 OR

CV_i – <u>00001</u>

CV* 10011

Example 1: State Diagram

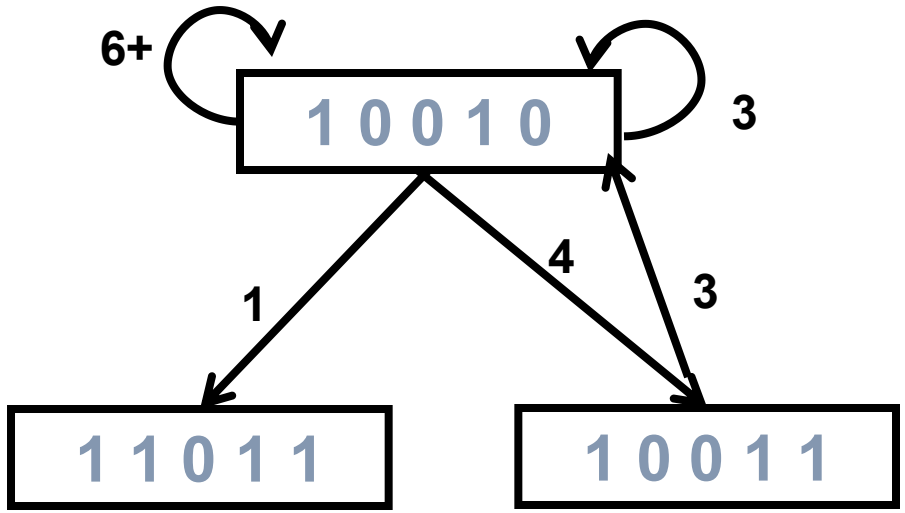


ICV – 10010 OR

CV_i – 00000

CV* 10010

Example 1: State Diagram



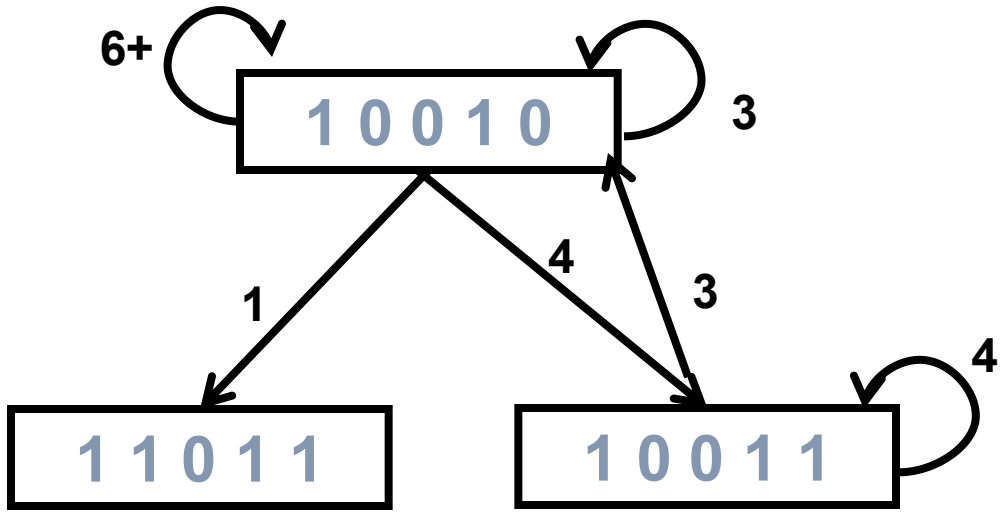
i =3

ICV – 10010 OR

CV_i – <u>00010</u>

CV* 10010

Example 1: State Diagram



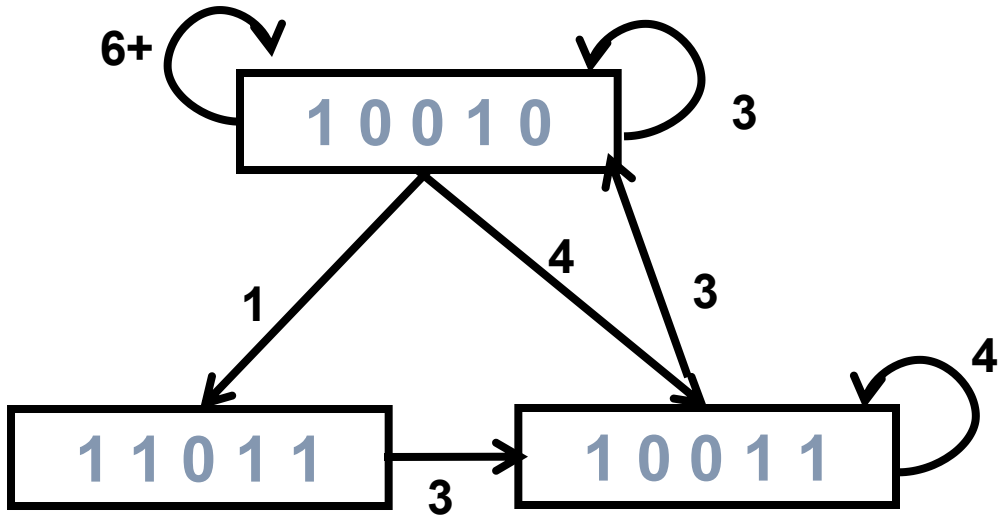
i =4

ICV – 10010 OR

CV_i – <u>00001</u>

CV* 10011

Example 1: State Diagram



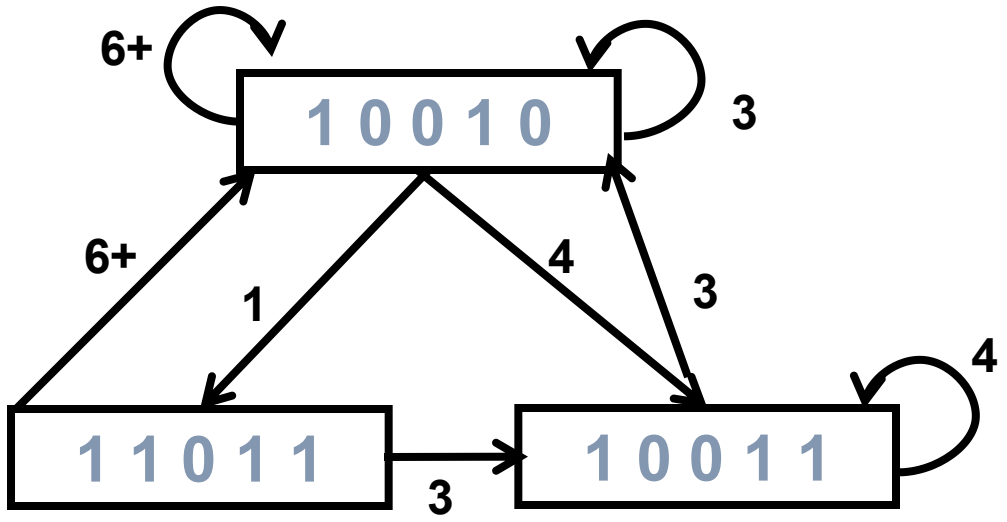
i =3

ICV – 10010 OR

CV_i – <u>00011</u>

CV* 10011

Example 1: State Diagram



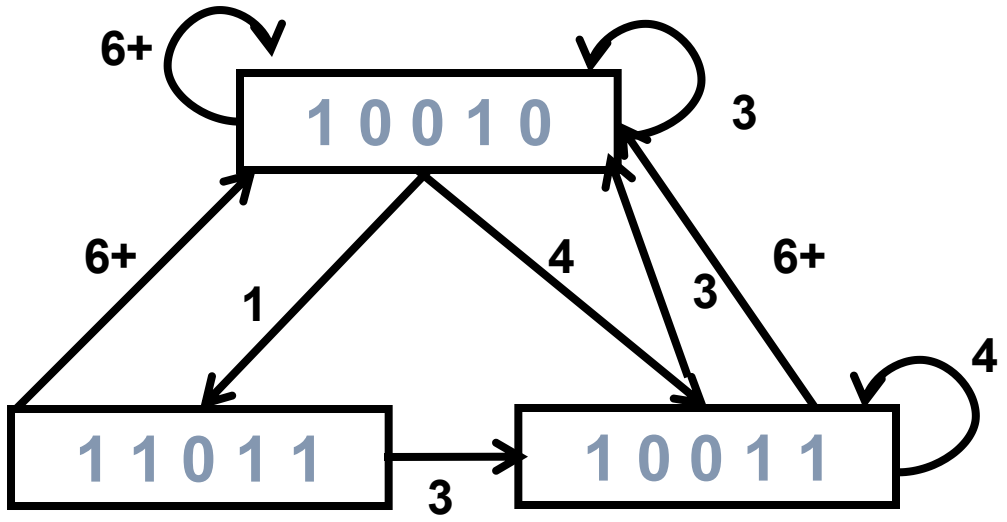
i = 5

ICV – 10010 OR

CV_i – <u>00000</u>

CV* 10010

Example 1: State Diagram



i = 5

ICV – 10010 OR

CV_i – <u>00000</u>

CV* 10010

Cycle, Average Latency and MAL

- A cycle in a state diagram is an alternating sequence of collision vectors and arcs: C0, a1, C1, ..., an, Cn
- Each arc ai connects collision vector Ci-1 to Ci
- All the collision vectors are distinct except the first and last
- A cycle is simply represented by a sequence of latencies of its arcs
- A Constant Cycle is a latency cycle which contains only one latency value
- The average latency for a cycle is determined by adding the latencies of the arcs of the cycle and then dividing it by the total number of arcs in the cycle
- Minimum Average Latency (MAL) is the minimum of all the average latencies of a state diagram

Example 1: Average Latency

- Let C0 is 10010, C1 is 11011 and C2 is 10011
- The cycle *C0, a1, C1, a2, C2, a3, C0*
- *a1 is an arc (latency = 1) from C0 to C1*
- *a2 is an arc (latency = 3) from C1* to *C2*
- *a3 is an arc (latency = 6) from C2* to *C0*
- *The cycle is represented as C=(1, 3, 6)*
- The average latency for a cycle is determined by adding the latencies of the arcs of the cycle and then dividing it by the total number of arcs in the cycle
- The cycle *C=(1, 3, 6) has the average latency:*

$$(1 + 3 + 6)/3 = 3.33$$

Example 1: Minimum Average Latency (MAL)

- The following are the average latencies of the different cycles in example 1
- 3/1= 3
- 6/1= 6
- (1 + 6)/2 =3.5
- (1 + 3 + 3)/3 =2.33
- (1 + 3 + 6)/3 = 3.33
- (4 + 3)/2 = 3.5
- (4 + 6) /2 = 5
- 4/1 = 4
- The minimum average latency is simply the minimum of these values
- So the MAL is 2.33

Thank you

Pipelined Architecture

CSEN 3104

Lecture 10

Dr. Debranjan Sarkar

Minimum Latency

- Although the cycle with the minimum average latency maximizes the throughput of the pipeline, sometimes a less efficient cycle may be chosen to reduce the implementation complexity of the pipeline's control circuit (i.e., a trade-off between time and cost)
- For example, the cycle *C=(1,3,3), which has the MAL of 2.33, requires a circuit that counts one unit of time,* then three units, again three units, and so on
- However, if it is acceptable to initiate an input datum after every 3 units of time, the complexity of the circuit will be reduced. Therefore, sometimes it may be necessary to determine the smallest latency that can be used for initiating input data at all times
- Such a latency is called the *minimum latency*

Example 1: Minimum Latency

- *One way to determine the minimum latency is to choose a cycle of* length 1 with the smallest latency from the state diagram. In this example, it is 3/1 = 3
- Another way is to find the smallest integer whose product with any arbitrary integer is not a member of the forbidden list.
- For example, for the forbidden list (2, 5), the minimum latency can be determined as follows:

Example 1: Minimum Latency

Minimum Latency	Times an integer	Product	Result
1	*1	=1	OK
1	*2	=2	No good
2	*1	=2	No good
3	*1	=3	OK
3	*2	=6	OK
4	*1	=4	OK
4	*2	=8	OK

- Forbidden Latency Set is { 2,5}
- Therefore the minimum latency for this pipeline is 3.

Simple cycle and Greedy cycle

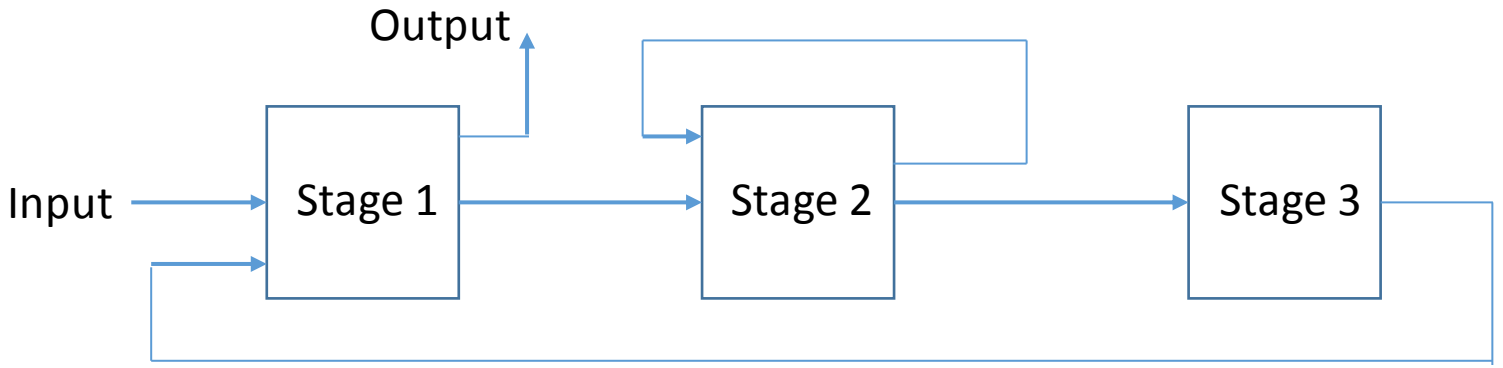
- **Simple Cycle:** latency cycle in which each state is encountered only once.
- In this example, the simple cycles are (3),(6), (1,6), (4,6), (4,3), (1,3,3), (1,3,6) and (4)
- The cycle (1,3,4,3) is not a simple cycle as the state 10011 is encountered twice
- **Greedy Cycle:** A simple cycle is a greedy cycle whose edges are all made with minimum latencies from their respective starting states
- At least one of the greedy cycles will lead to MAL.
- In this example, the Greedy cycle is (1,3,3)
- In the above example, the cycle that offers MAL is (1, 3, 3)

Upper and Lower Bounds of MAL

- Lower bound of MAL = the maximum number of checkmarks in any row of the Reservation Table
- MAL $\geq$ max no. of check marks in any row of the reservation table
- In this example, Lower bound of MAL = 2, since there are maximum 2 checkmarks in any row of the RT
- Upper bound of MAL= the number of 1's in the initial collision vector plus 1
- MAL $\leq$ avg latency of any greedy cycle
  $\leq$ no. of 1's in initial collision vector + 1
- In this example, ICV is 10010, so number of 1's = 2
- Upper bound of MAL = 2 + 1 = 3
- So $2 \leq$ MAL ≤ 3

Scheduling of Static Pipelines

Example 2



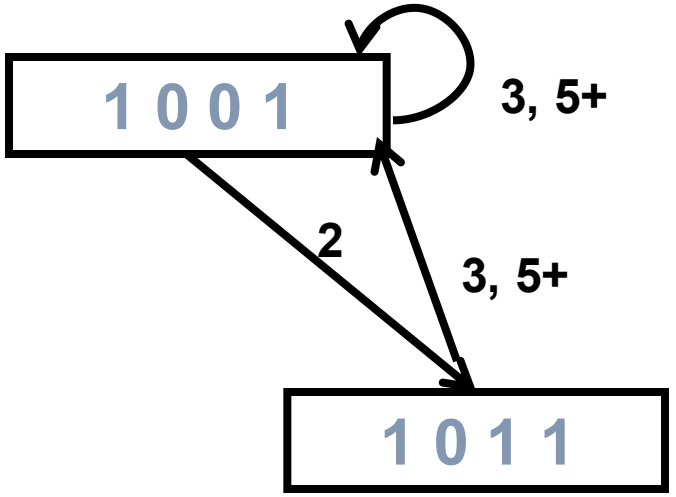
	1	2	3	4	5
Stage 1	X				X
Stage 2		X	X		
Stage3				X	

Forbidden Latencies: 1, 4

Initial Collision Vector: $C_4C_3C_2C_1$ = 1 0 0 1

Example 2

- State Diagram



- The following are the average latencies of the different cycles
- (2 + 3)/2 =2.5 (2 + 5)/2 =3.5 3/1= 3 5/1= 5
- The minimum average latency is 2.5

Example 2: Minimum Latency

- Although the cycle with the minimum average latency maximizes the throughput of the pipeline, sometimes a less efficient cycle may be chosen to reduce the implementation complexity of the pipeline's control circuit (i.e., a trade-off between time and cost)
- For example, the cycle *C=(2,3), which has the MAL of 2.5, requires a circuit that counts three units of time,* then two units, again three units, and so on.
- However, if it is acceptable to initiate an input datum after every three units of time, the complexity of the circuit will be reduced. Therefore, sometimes it may be necessary to determine the smallest latency that can be used for initiating input data at all times.
- Such a latency is called the *minimum latency*

Example 2: Minimum Latency

- One way to determine the minimum latency is to choose a cycle of length 1 with the smallest latency from the state diagram
- Another way is to find the smallest integer whose product with any arbitrary integer is not a member of the forbidden list.
- For example, for the forbidden list (1, 4), the minimum latency can be determined as follows:

Minimum Latency	Times an integer	Product	Result
1	* 1	= 1	No good
2	* 1	= 2	OK
2	* 2	= 4	No good
3	* 1	= 3	OK
3	* 2	= 6	OK
4	* 1	= 4	No good

- Therefore the minimum latency for this pipeline is 3

Example 2: Simple cycle and Greedy cycle

- **Simple Cycle:** latency cycle in which each state is encountered only once.
- In Example 2, the simple cycles are (3),(5), (2,3), (2,5)

- **Greedy Cycle:** A simple cycle is a greedy cycle whose edges are all made with minimum latencies from their respective starting states
- At least one of the greedy cycles will lead to MAL.
- In this example, the Greedy cycle is (2,3)
- In the above example, the cycle that offers MAL is (2, 3)

Example 2: Upper and Lower Bounds of MAL

- Lower bound of MAL = the maximum number of checkmarks in any row of the Reservation Table
- MAL $\geq$ max no. of check marks in any row of the reservation table
- In Example 2, Lower bound of MAL = 2, since there are maximum 2 checkmarks in any row of the RT
- Upper bound of MAL= the number of 1's in the initial collision vector plus 1
- MAL $\leq$ avg latency of any greedy cycle
  $\leq$ no. of 1's in initial collision vector + 1
- In example 2, ICV is 1001, so number of 1's = 2
- Upper bound of MAL = 2 + 1 = 3
- So 2 $\leq$ MAL $\leq$ 3

Example 3

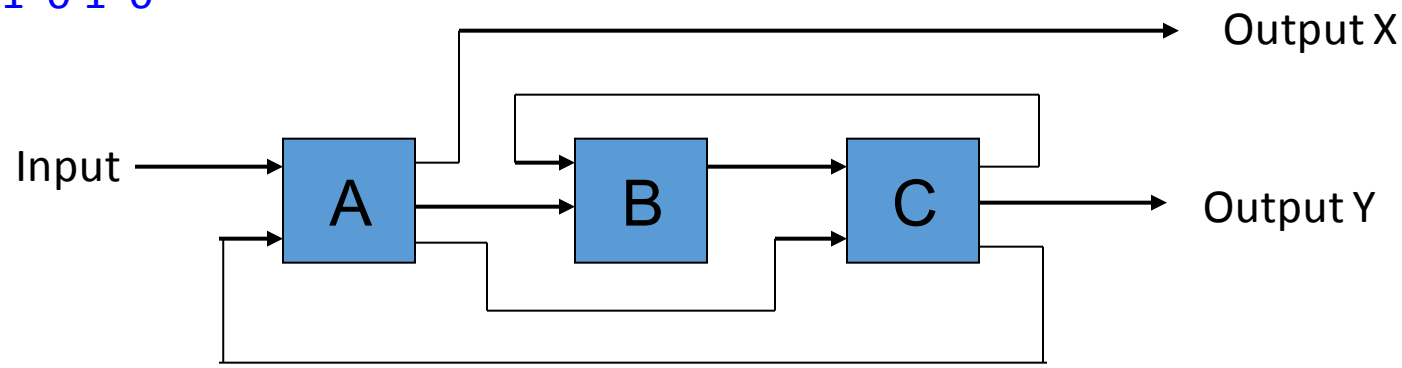
Forbidden Latencies:
{5,7,2} ?{2} ? {2,4} = {2,4,5,7}

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

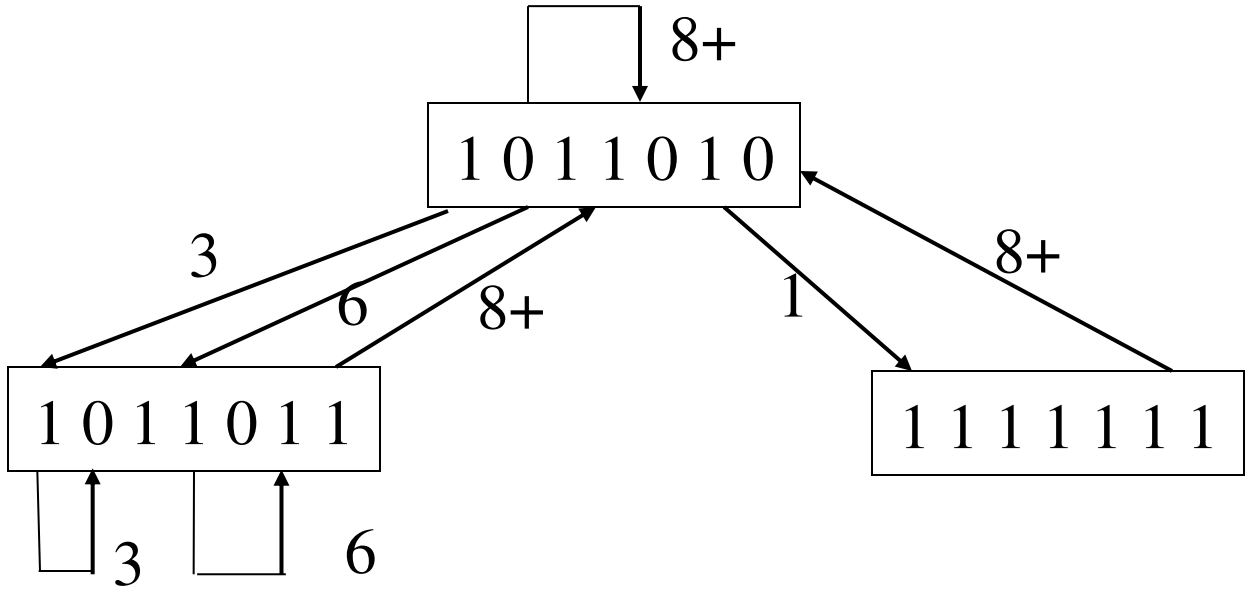
Sequence for output X: A, B, C, B, C, A, C, A

Initial Collision Vector:

$C_7C_6C_5\ C_4C_3C_2C_1$
1 0 1 1 0 1 0



Example 3: State Diagram for X



Example 3

- Simple Cycles: (3), (6), (8), (1, 8), (3, 8) and (6, 8) are simple cycles
- The cycle (6, 8, 1, 8) is not simple because the state (1011010) is encountered twice

- Greedy Cycles: The cycle (1, 8) and (3) are greedy cycles

- Minimum Average Latency (MAL): In greedy cycles (1, 8) and (3), the cycle (3) leads to MAL value 3

Example 3: Minimum Latency

Minimum Latency	Times an integer	Product	Result
1	*1	= 1	OK
1	*2	= 2	No good
2	*1	= 2	No good
3	*1	= 3	OK
3	*2	= 6	OK
3	*3	= 9	OK
4	*1	= 4	No good
5	*1	= 5	No good
6	*1	= 6	OK
6	*2	= 12	OK
7	*1	= 7	No good

- Forbidden Latency Set is { 2, 4, 5, 7}
- Therefore the minimum latency for this pipeline is 3.

Example 3: Upper and Lower Bounds of MAL

- Lower bound of MAL = the maximum number of checkmarks in any row of the Reservation Table
- In this example, Lower bound of MAL = 3, since there are maximum 3 checkmarks in any row of the RT
- Upper bound of MAL= the number of 1's in the initial collision vector plus 1
- MAL $\leq$ avg latency of any greedy cycle

  $\leq$ no. of 1's in initial collision vector + 1
- IN this example, ICV is 1011010, so number of 1's = 4
- Upper bound of MAL = 4 + 1 = 5
- So 3 $\leq$ MAL $\leq$ 5

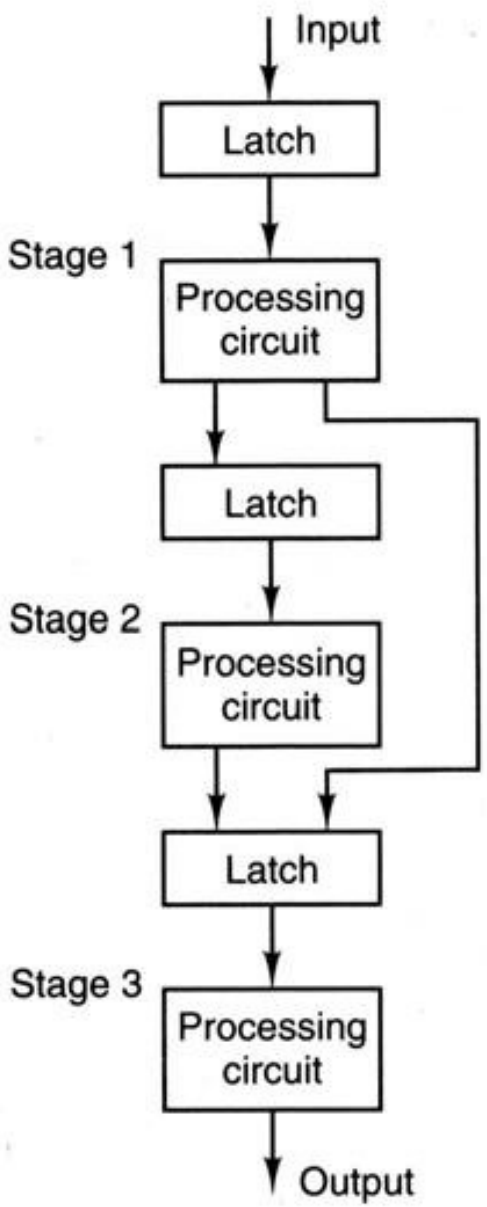
Assignment

Given the Reservation Table for output Y as follows:

	1	2	3	4	5	6
A	Y				Y	
B			Y			
C		Y		Y		Y

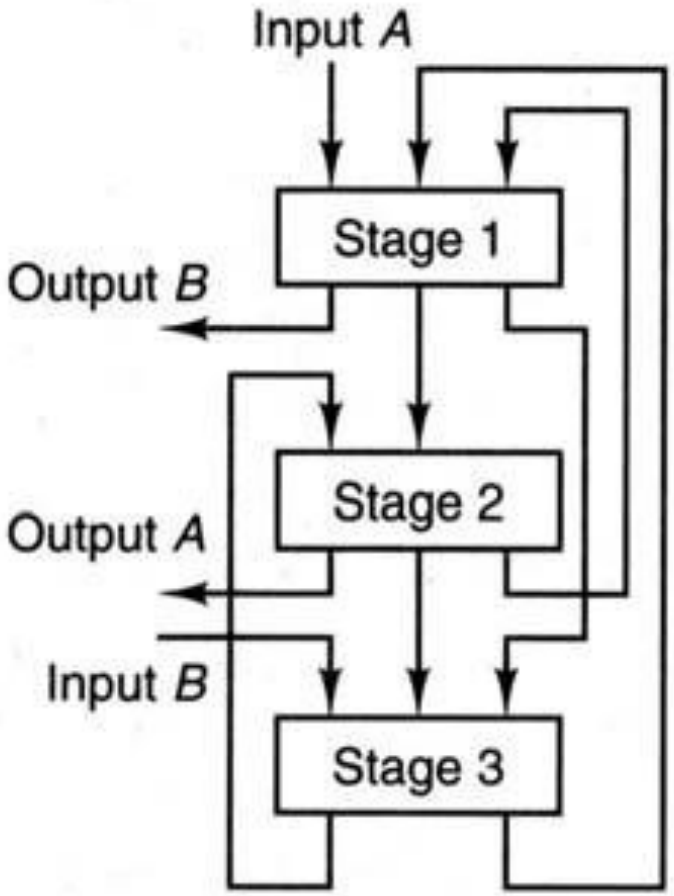
(a) Find out the (i) Forbidden latencies and (ii) Initial Collision Vector
(c) Draw the State Diagram for scheduling the pipeline
(d) Find out the simple cycle, greedy cycle and MAL
(e) What are bounds on MAL?

Dynamic Pipeline



- A dynamic pipeline can perform more than one operation at a time
- This 3-stage dynamic pipeline performs addition and multiplication on different data at the same time
- For multiplication, the input data must go through stages 1, 2, and 3
- For addition, the data only need to go through stages 1 and 3
- In dynamic pipeline, stage 1 can perform the first stage of the addition operation on an input data D1, and at the same time stage 3 can perform the last stage of the multiplication operation on another input data D2
- The time interval between the initiation of the inputs D1 and D2 to the pipeline should be such that they do not reach stage 3 at the same time; otherwise, there is a collision
- In general, in dynamic pipelines the mechanism that controls when data should be fed to the pipeline is much more complex than in static pipelines

A dynamic pipeline and its reservation tables



(a)

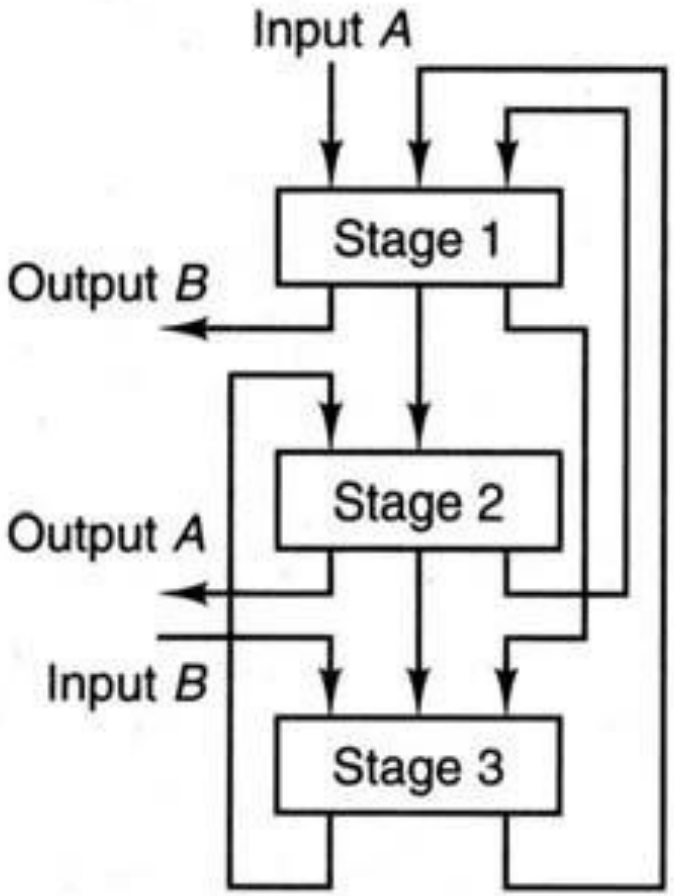
	t_0	t_1	t_2	t_3	t_4
Stage 1	A			A	
Stage 2		A			A
Stage 3			A		

(b)

	t_0	t_1	t_2	t_3	t_4
Stage 1		B			B
Stage 2				B	
Stage 3	B		B		

(c)

A dynamic pipeline and its reservation tables



(a)

	t_0	t_1	t_2	t_3	t_4
Stage 1	A			A	
Stage 2		A			A
Stage 3			A		

(b)

	t_0	t_1	t_2	t_3	t_4
Stage 1		B			B
Stage 2				B	
Stage 3	B		B		

(c)

Example of Overlaid Reservation Table

	t_0	t_1	t_2	t_3	t_4
Stage 1	A			A	
Stage 2		A			
Stage 3			A		A

For A

	t_0	t_1	t_2	t_3	t_4
Stage 1		B			B
Stage 2				B	
Stage 3	B		B		

For B

	t_0	t_1	t_2	t_3	t_4
Stage 1	A	B		A	B
Stage 2		A		B	
Stage 3	B		AB		A

Overlaid

Scheduling of Dynamic Pipeline

- When scheduling a static pipeline, only collisions between different input data for a particular function had to be avoided.
- With a dynamic pipeline, it is possible for different input data requiring different functions to be present in the pipeline at the same time.
- Therefore, collisions between these data must be considered as well.
- As with the static pipeline, however, dynamic pipeline scheduling begins with the compilation of a set of forbidden lists from function reservation tables.
- Next the collision vectors are obtained, and finally the state diagram is drawn.

Forbidden Lists of Dynamic Pipeline

- With a dynamic pipeline, in general there are p^2 forbidden lists and hence p^2 cross-collision vectors for a p-function pipeline
- In the earlier figure, there are 2 functions (A and B), so *the* number of forbidden lists is 4, denoted by *AA, AB, BA, and BB*
- Task A may collide with a previously initiated task B, if the latency between these two initiations is a member of the forbidden list
- AA = (2,3), AB = (1,2,4), BA = (2,4), and BB = (2,3),

Collision Vectors and Collision matrices

- The collision vectors are determined in the same manner as for a static pipeline
- 0 indicates a permissible latency and a 1 indicates a forbidden latency
- For the preceding example, the collision vectors are
- C_{AA} = (0 1 1 0), C_{BA} = (1 0 1 0), C_{AB} = (1 0 1 1), C_{BB} = (0 1 1 0)
- The collision vectors for the *A function form the collision matrix* M_A, *that is,* $M_A = [C_{AA}, C_{BA}]^T$
- The collision vectors for the *B function form the collision matrix* M_B*:* *that is,* $M_B = [C_{AB}, C_{BB}]^T$
- For the above collision vectors, the collision matrices are
- $M_A = [0110, 1010]^T$ $M_B = [1011, 0110]^T$

State Diagram

- The state diagram for the dynamic pipeline is developed in the same way as for the static pipeline
- The resulting state diagram is much more complicated than a static pipeline state diagram due to the larger number of potential collisions

Problems of pipeline

- In practice, making the delays of pipeline stages equal is a complicated and time-consuming process
  - It is essential to maximize performance that the stages be close to balanced.
  - It is done for commercial processors, although it is not easy and cheap to do
- Another problem with pipelines is the overhead in term of handling exception or interrupts
  - A deep pipeline increases the interrupt handling overhead.

What is Microprocessor without Interlocked Pipelined Stages?

- One of the main disadvantages of pipelining is that there can be various dependencies that occur between stages:
  - Data hazards (RaW, WaW and WaR)
  - Control Hazards
  - Structural Hazards
- In an interlocked pipeline, hardware is used to check for hazards between stages. Sometimes this may lead to deadlock.
- In a non interlocked pipeline, no hardware is used to check for hazards. However, the burden of checking for dependencies is left to the programmer/compiler. If they fail to check for hazards, the program might use invalid data and even create a deadlock requiring a reboot.
- MIPS CPU was originally intended to simplify processor design by eliminating hardware interlocks between the five pipeline stages
- Later versions of the MIPS ISA did have interlocking however.

Thank you