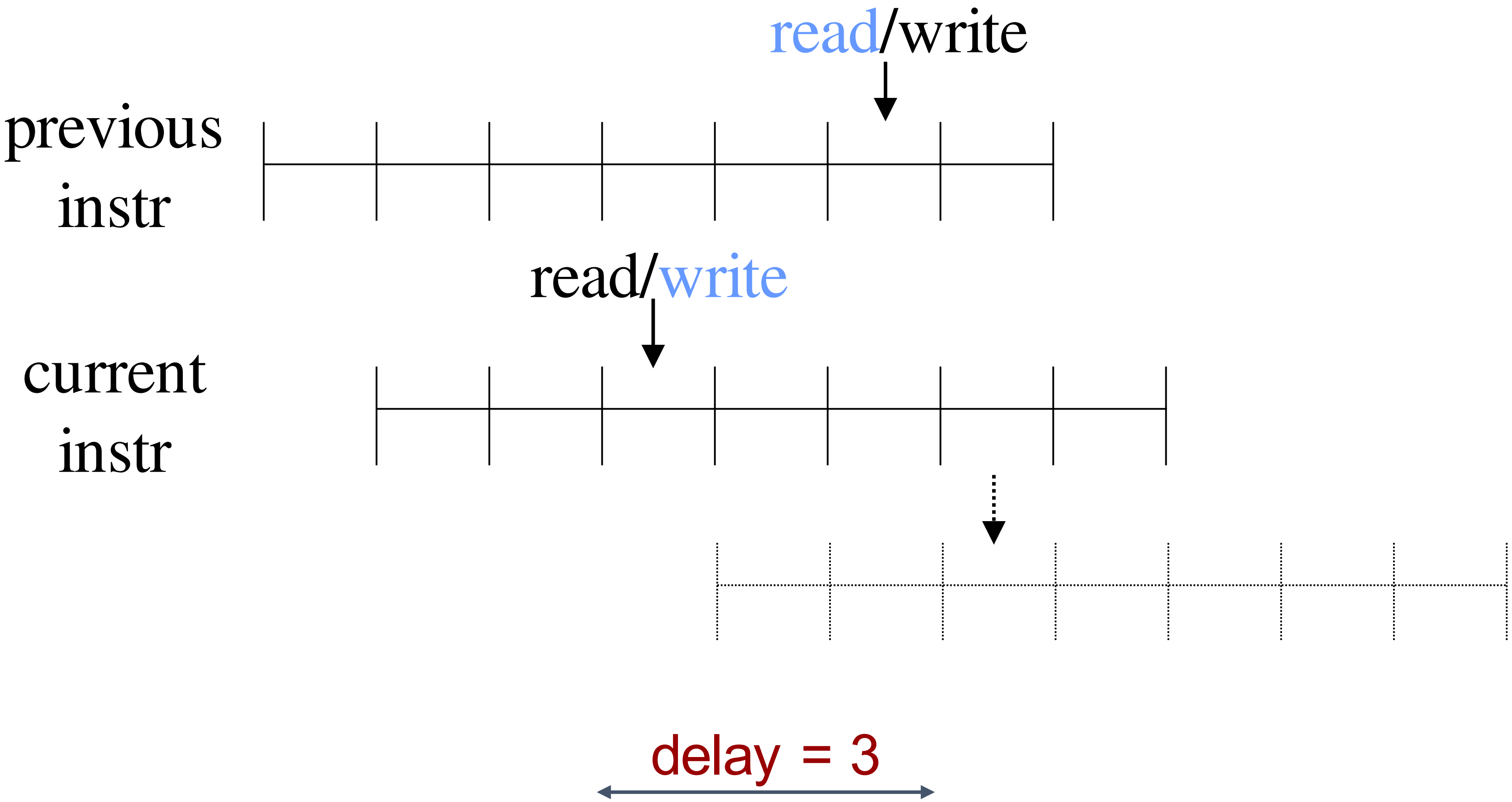
Pipelined Architecture
CSEN 3104
Lecture 8

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)

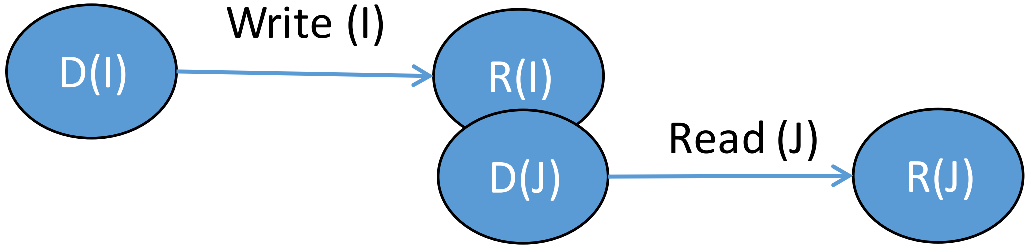
- Instruction (I): Read data object X
- Instruction (J): Write (or modify) data object X
- J tries to write (modify) data object before it is read by I
- So, the data object read by instruction I is incorrect
- It was desired that instruction I would read the old value of X and then the data object would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

Detection of Data Hazards

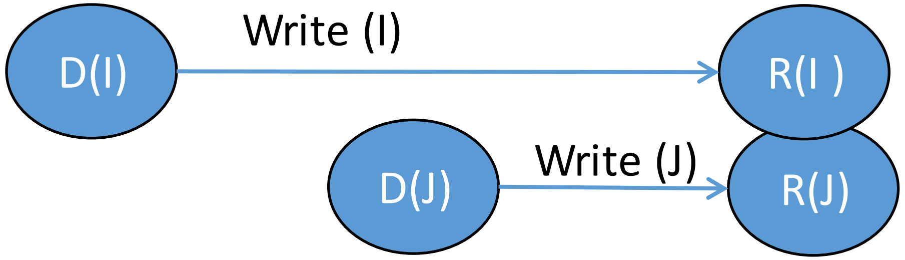
- Definitions:
  - Resource Object: Working registers, memory locations, flags
  - Data Object: Contents of the Resource Objects
  - Domain of an Instruction D(I):
    - Set of resource objects, whose data objects may affect the execution of the instruction I
    - Holds the operands to be read for execution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

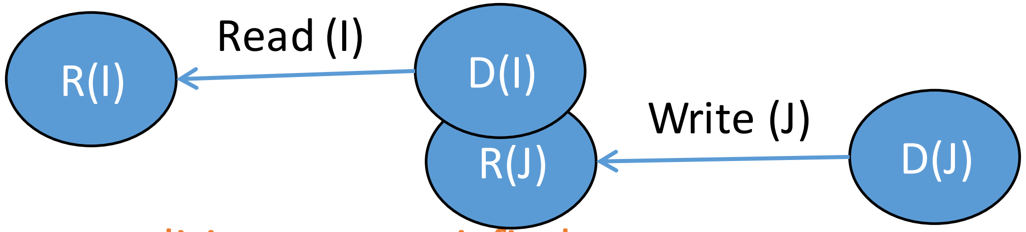
- I and J are two instructions in a program
- I occurs before J
- RAW Hazard: $R(I) \cap D(J) \neq \phi$



- 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 (b) Pipelined with I_7 as Jump instruction

• Instruction Fetch (IF)	IF_1					IF_2					IF_3				
• Instruction Decode (ID)		ID_1					ID_2					ID_3			
• Operand Load (OL)			OL_1					OL_2					OL_3		
• Execution (EX)				EX_1					EX_2					EX_3	
• Operand Store (OS)					OS_1					OS_2					OS_3
• Time (clock cycles)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

• Instruction Fetch (IF)	IF_1	IF_2	IF_3	IF_4		IF_5	IF_6	IF_7				IF_8	IF_9	IF_{10}	IF_{11}
• Instruction Decode (ID)		ID_1	ID_2	ID_3	ID_4		ID_5	ID_6	ID_7				ID_8	ID_9	ID_{10}
• Operand Load (OL)			OL_1	OL_2	OL_3	OL_4		OL_5	OL_6	OL_7				OL_8	OL_9
• Execution (EX)				EX_1	EX_2	EX_3	EX_4	EX_4	EX_5	EX_6	EX_7				EX_8
• Operand Store (OS)					OS_1	OS_2	OS_3		OS_4	OS_5	OS_6				
• Time (clock cycles)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

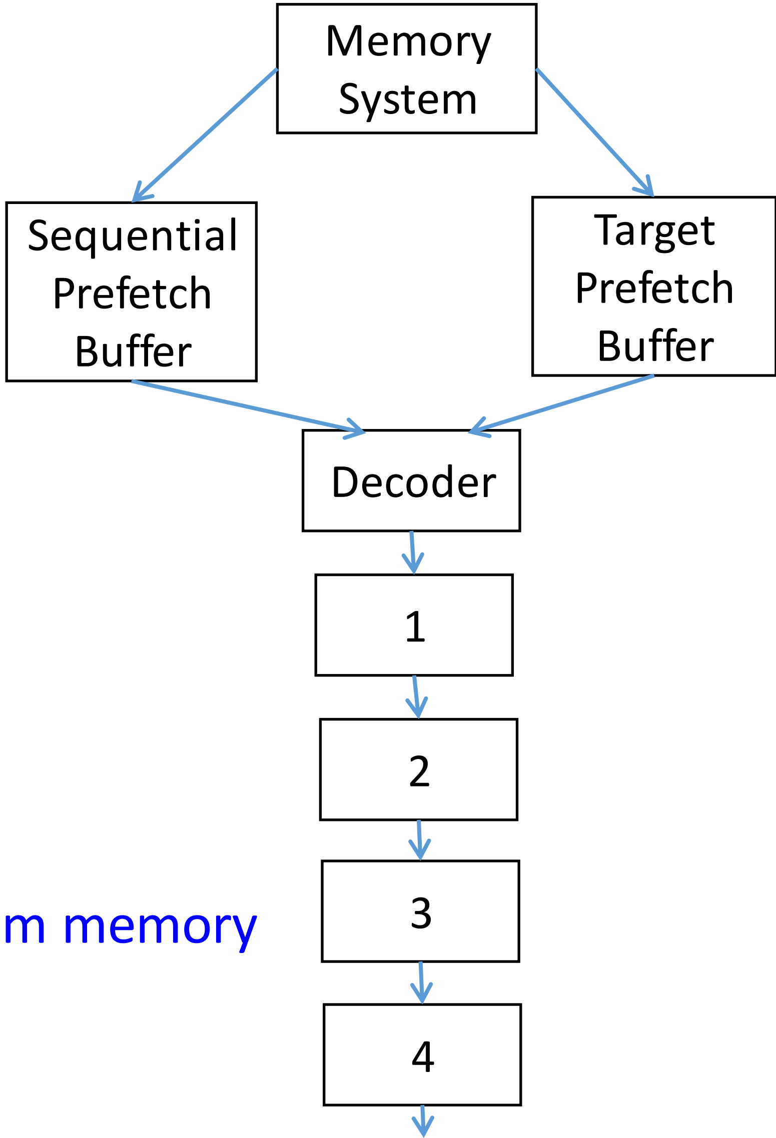
Penalty : 3 cycles

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

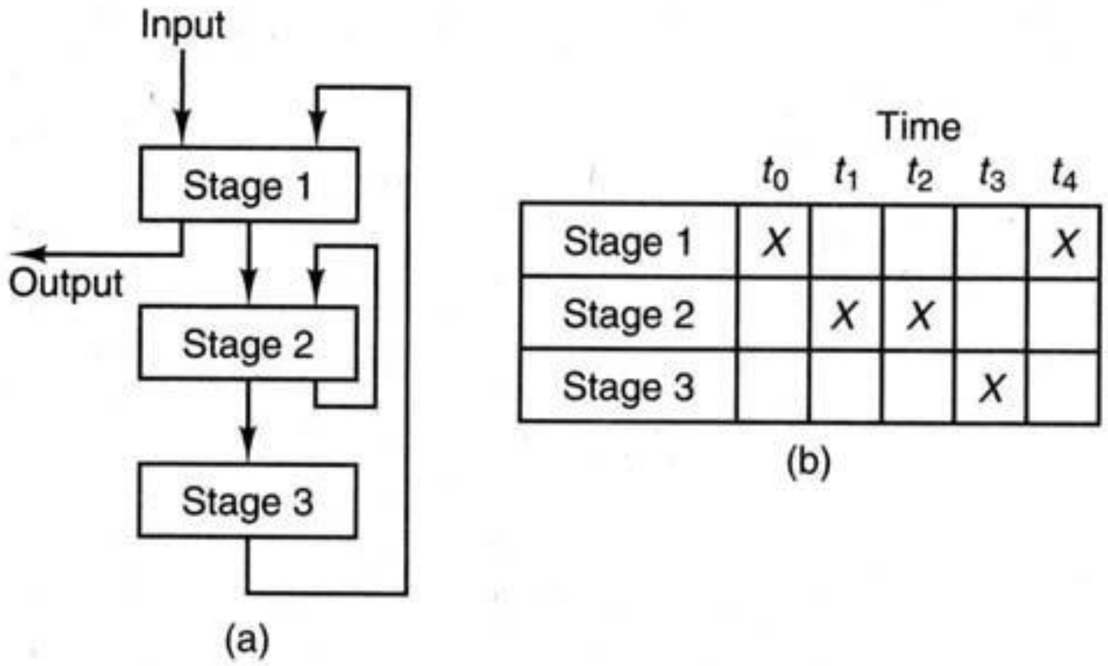
- As per Configuration, pipelines are of two types:
- Static Pipeline:
  - A static pipeline can perform only one function (or operation such as addition or multiplication) at a time
  - The operation of a static pipeline can only be changed after the pipeline has been drained
  - A pipeline is said 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)

	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