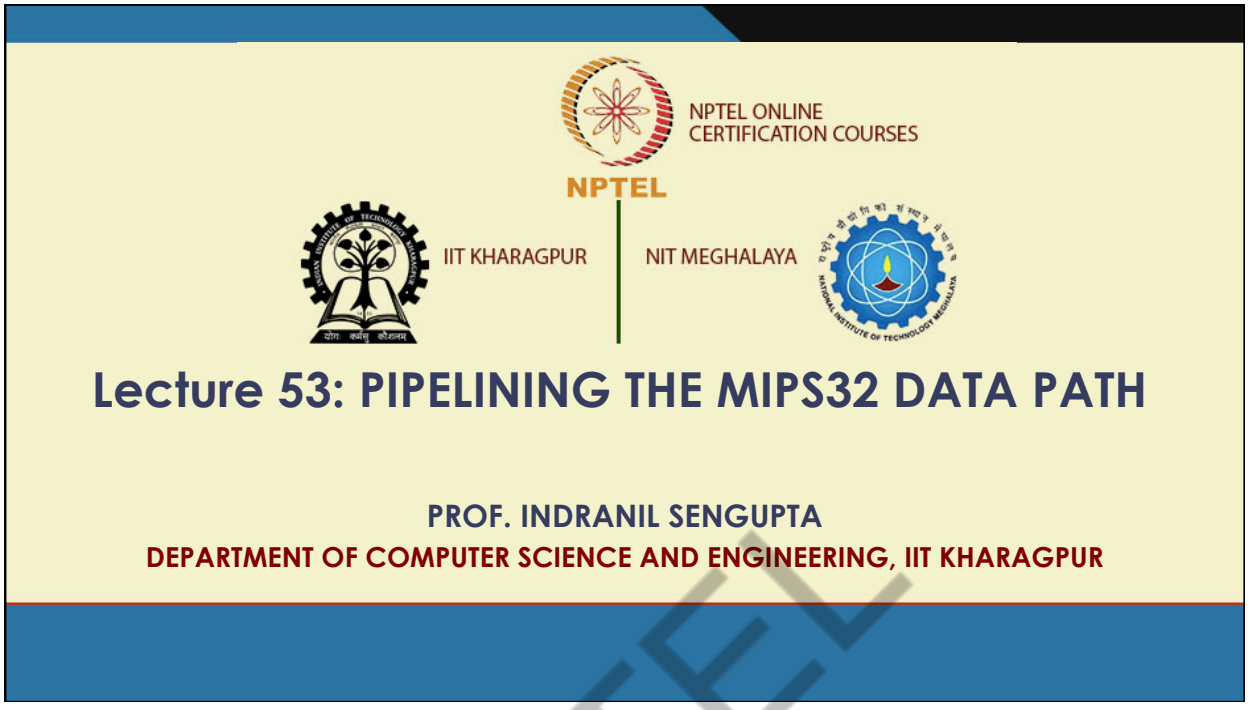


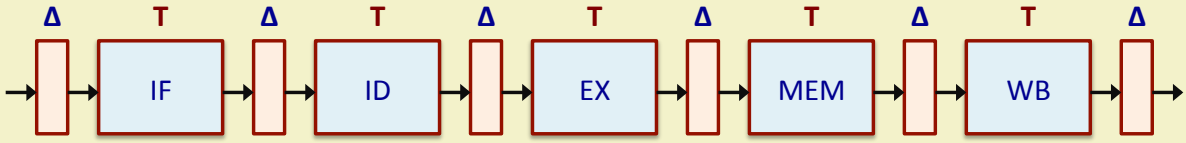
Introduction

- 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 detail.

5T

IF + ID + EX + MEM + WB
(Non-pipelined)

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

Instr-i finishes

Instr-(i+1) finishes

Instr-(i+2) finishes

Instr-(i+3) finishes

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:

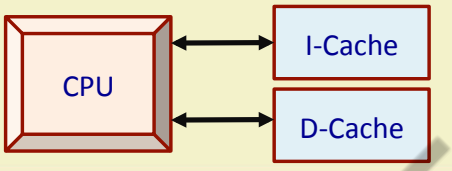
- IF & MEM: In clock cycle 4, both instructions i and i+3 access memory.
  - *Solution: use separate instructions and data cache*.
- ID & WB: In clock cycle 5, both instructions i and i+3 access register bank.
  - *Solution: allow both read and write access to registers in the same clock cycle*.

Advantages of Pipelining

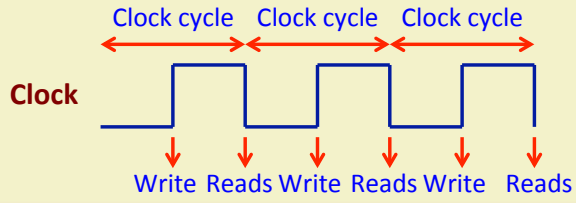
- In the non-pipelined version, the execution time of an instruction is equal to the combined delay of the five stages (say, *5T*).
- In the pipelined version, once the pipeline is full, one instruction gets executed after every *T* time.
  - Assuming all state delays are equal (equal to *T*), and neglecting 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.
  - To be discussed.

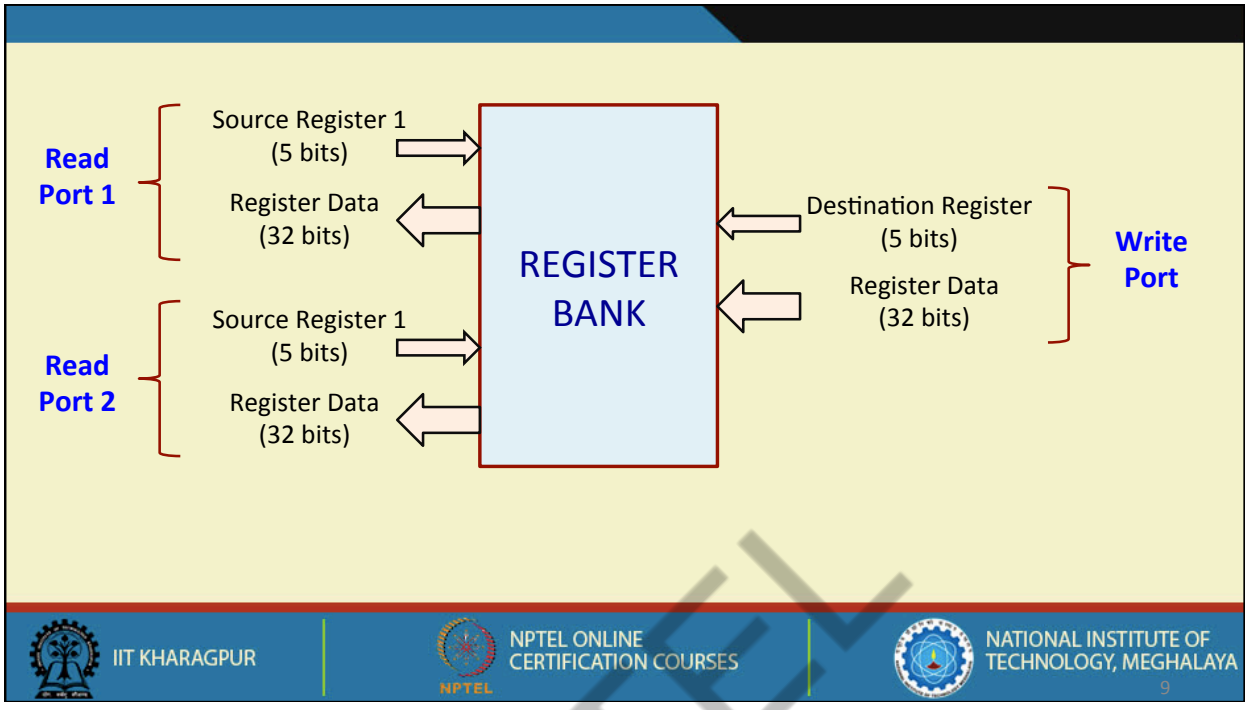
Some Observations

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.



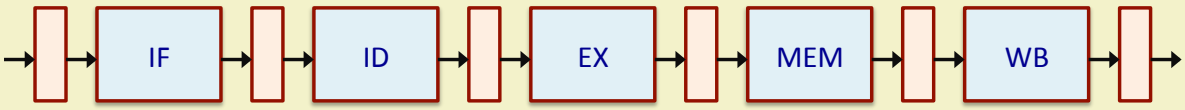


c) Since a new instruction is fetched every clock cycle, it is required to increment the PC on each clock.

- 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.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
10

Basic Performance Issues in a Pipeline



- Register stages are inserted between pipeline stages, which increases the execution time of an individual instruction.
  - Because of overlapped execution of instructions, throughput increases.
- The clock period T has to be chosen suitably:
  - Slowest stage in the pipeline.
  - Clock skew and jitter.
  - Register setup time: minimum time the register input must be held stable before the active clock edge arrives.

Example 1

- Consider the 5-stage MIPS32 pipeline, with the following features:
  - Pipeline clock rate of 1GHz (i.e. 1 ns clock cycle time).
  - For a non-pipelined implementation, ALU operations and branches take 4 cycles, while memory operations take 5 cycles.
  - Relative frequencies of ALU operations, branches and memory operations are 50%, 15%, and 35% respectively.
  - In the pipelined implementation, due to clock skew and setup time, the clock cycle time increases by 0.25 ns.
  - Calculate the estimated speedup of the pipelined implementation.

- **Solution:**
  a) For non-pipelined processor:
     - Average instruction execution time = Clock cycle time x Average CPI
       = 1 ns x (0.50 x 4 + 0.15 x 4 + 0.35 x 5) = 4.35 ns

  b) For pipelined processor:
     - Clock cycle time = 1 + 0.25 = 1.25 ns
     - In the steady state, one instruction will get executed every clock cycle.
     - Speedup = 4.35 / 1.25 = 3.48

Micro-operations for Non-pipelined MIPS32

IF

IR ← Mem [PC];
NPC ← PC + 4;

ID

A ← Reg [rs];
B ← Reg [rt];
Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$
Imm1 ← $IR_{25..0}$ ## 00

EX

ALUOut ← A + Imm;
(a) Memory

ALUOut ← A func B;
(b) R-R ALU

ALUOut ← A func Imm;
(c) R-IMM ALU

ALUOut ← NPC + (Imm << 2);
cond ← (A op 0);
(d) Branch

MEM

PC ← NPC;
LMD ← Mem [ALUOut];
(a) Load

PC ← NPC;
Mem [ALUOut] ← B;
(b) Store

if (cond) PC ← ALUOut;
else PC ← NPC;
(c) Branch

(d) Others
PC ← NPC;

WB

Reg [rd] ← ALUOut;

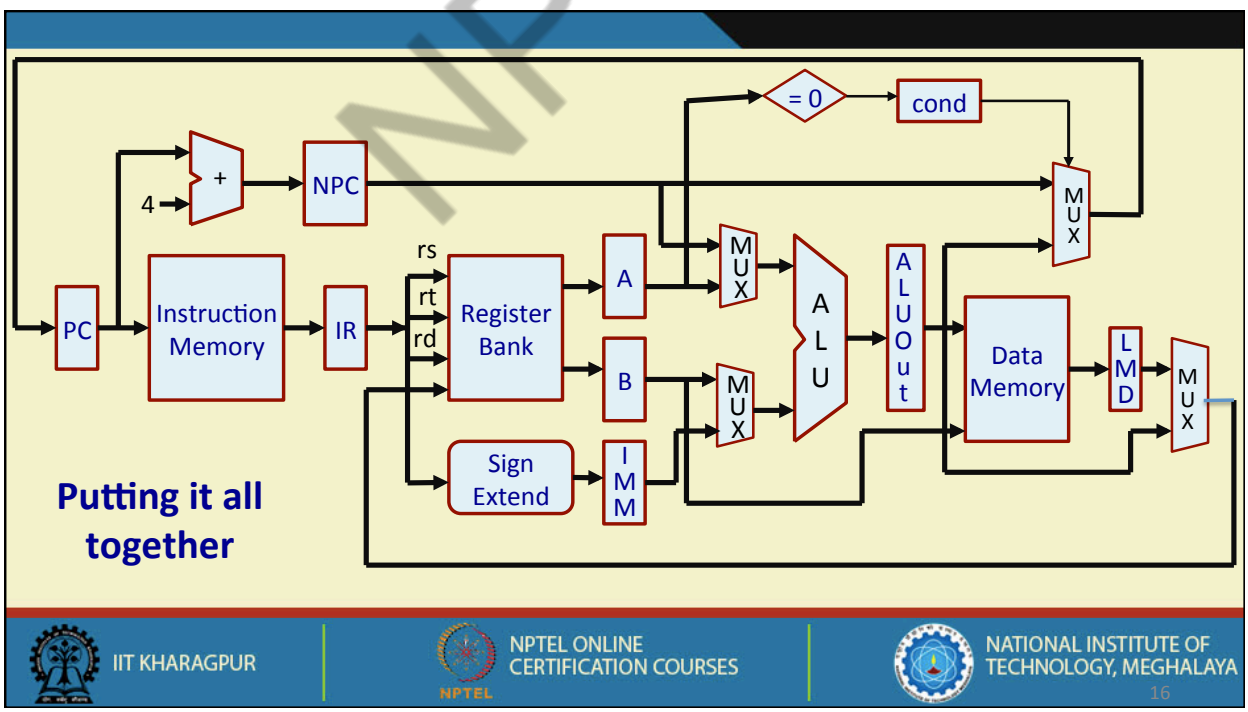
(a) R-R ALU

Reg [rt] ← ALUOut;

(b) R-IMM ALU

Reg [rt] ← LMD;

(c) Load



END OF LECTURE 53

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

17

NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 54: MIPS32 PIPELINE (Contd.)

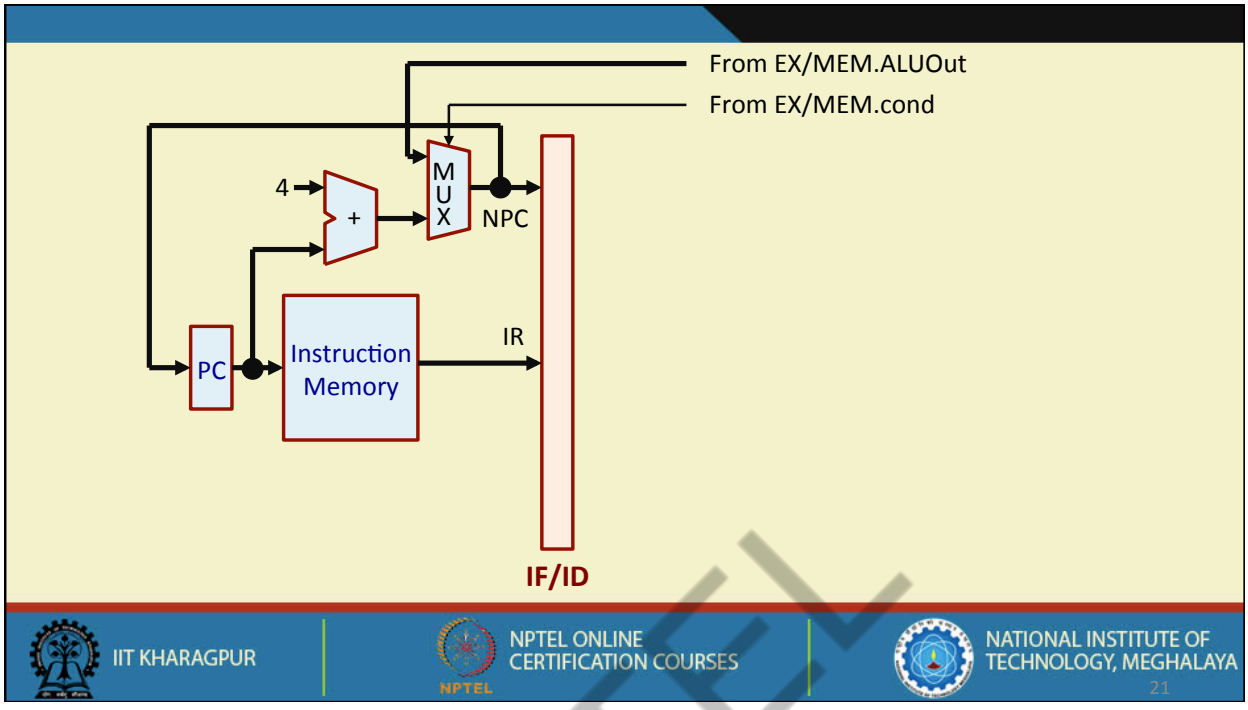
PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KHARAGPUR

Micro-operations for Pipelined MIPS32

- Convention used:
  - Many of the temporary registers required in the data path are included as part of the inter-stage latches.
  - IF/ID: denotes the latch stage between the IF and ID stages.
  - ID/EX: denotes the latch stage between the ID and EX stages.
  - EX/MEM: denotes the latch stage between the EX and MEM stages.
  - MEM/WB: denotes the latch stage between the MEM and WB stages.
- Example:
  - ID/EX.A means a register A that is implemented as part of the ID/EX latch stage.

(a) Micro-operations for Pipeline Stage IF

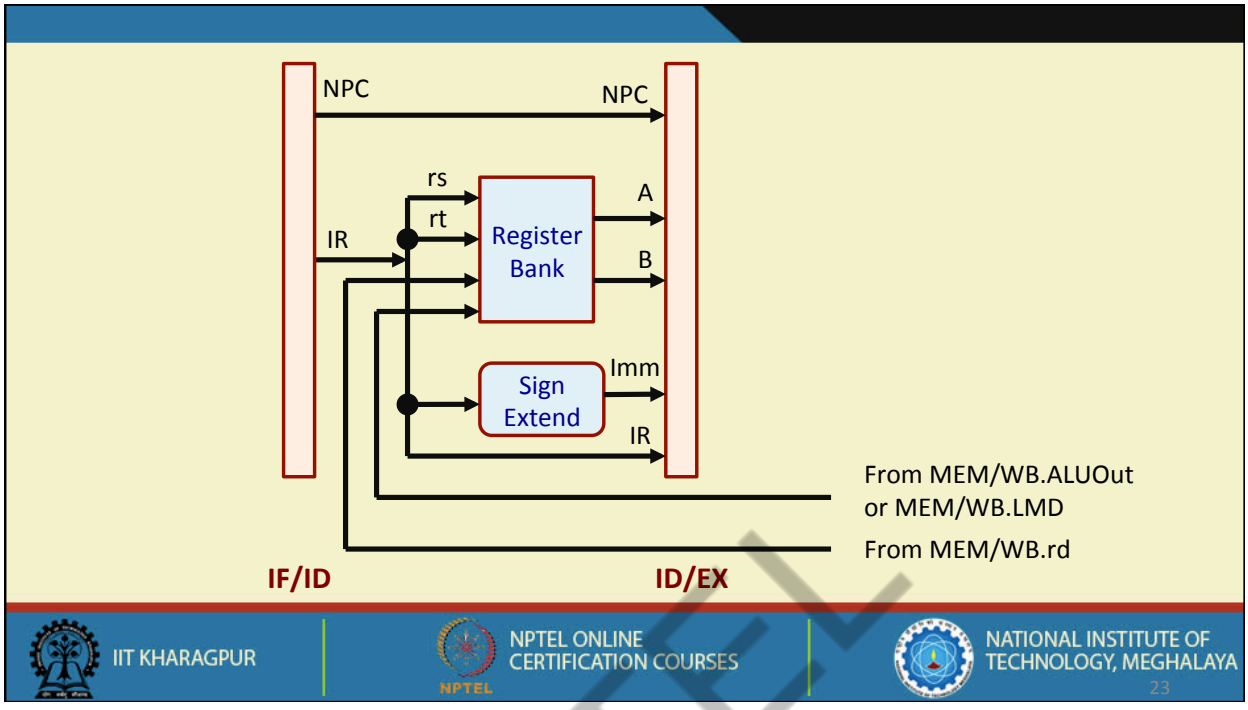
```
IF/ID.IR      ← Mem [PC];
IF/ID.NPC,PC ← ( if ((EX/MEM.opcode == branch) & EX/MEM.cond)
                        { EX/MEM.ALUOut}
                  else  {PC + 4} );
```



(b) Micro-operations for Pipeline Stage ID

ID/EX.A ← Reg [IF/ID.IR [rs]];
ID/EX.B ← Reg [IF/ID.IR [rt]];
ID/EX.NPC ← IF/ID.NPC;
ID/EX.IR ← IF/ID.IR;
ID/EX.Imm ← sign-extend ($IF/ID.IR_{15..0}$);

IIT KHARAGPUR
NPTEL ONLINE
CERTIFICATION COURSES
NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA
22



(c) Micro-operations for Pipeline Stage EX

EX/MEM.IR ← ID/EX.IR;
EX/MEM.ALUOut ← ID/EX.A func ID/EX.B;

R-R ALU

EX/MEM.IR ← ID/EX.IR;
EX/MEM.ALUOut ← ID/EX.A func ID/EX.Imm;

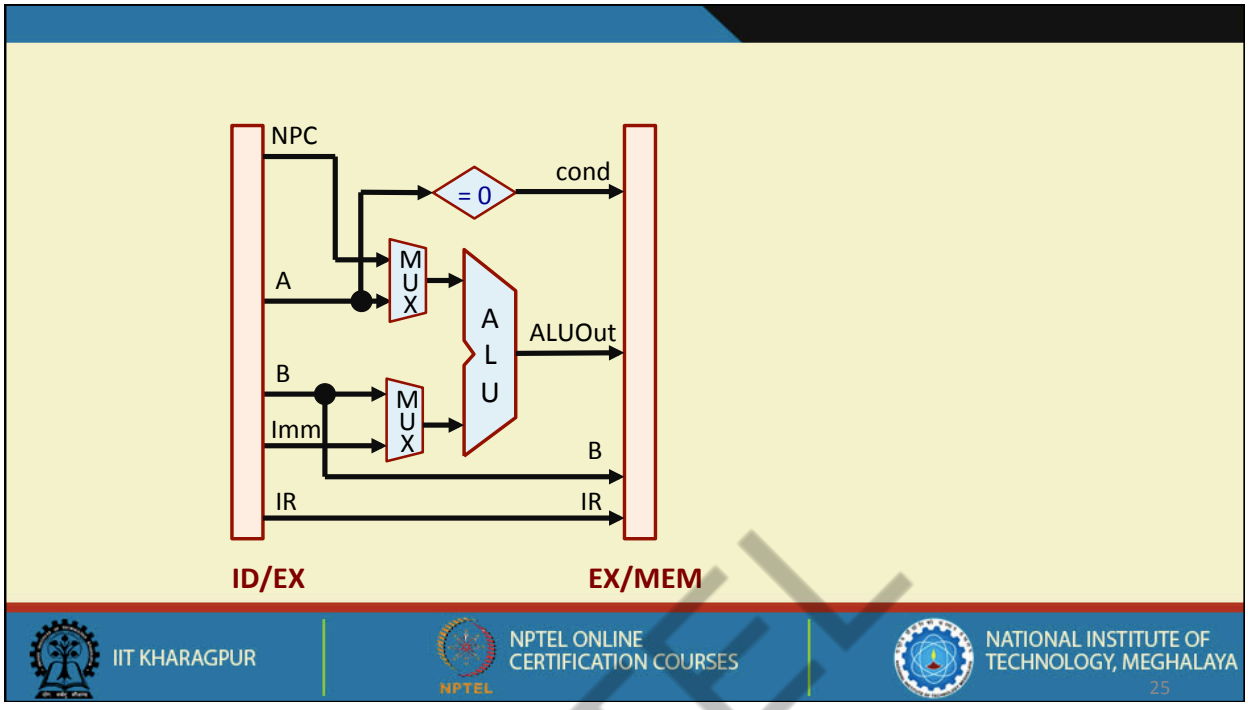
R-M ALU

EX/MEM.IR ← ID/EX.IR;
EX/MEM.ALUOut ← ID/EX.A + ID/EX.B;
EX/MEM.B ← ID/EX.B;

LOAD / STORE

EX/MEM.ALUOut ← ID/EX.NPC +
(ID.EX.Imm << 2);
EX/MEM.cond ← (ID/EX.A == 0);

BRANCH



(d) Micro-operations for Pipeline Stage MEM

MEM/WB.IR ← EX/MEM.IR;
MEM/WB.ALUOut ← EX/MEM.ALUOut;

ALU

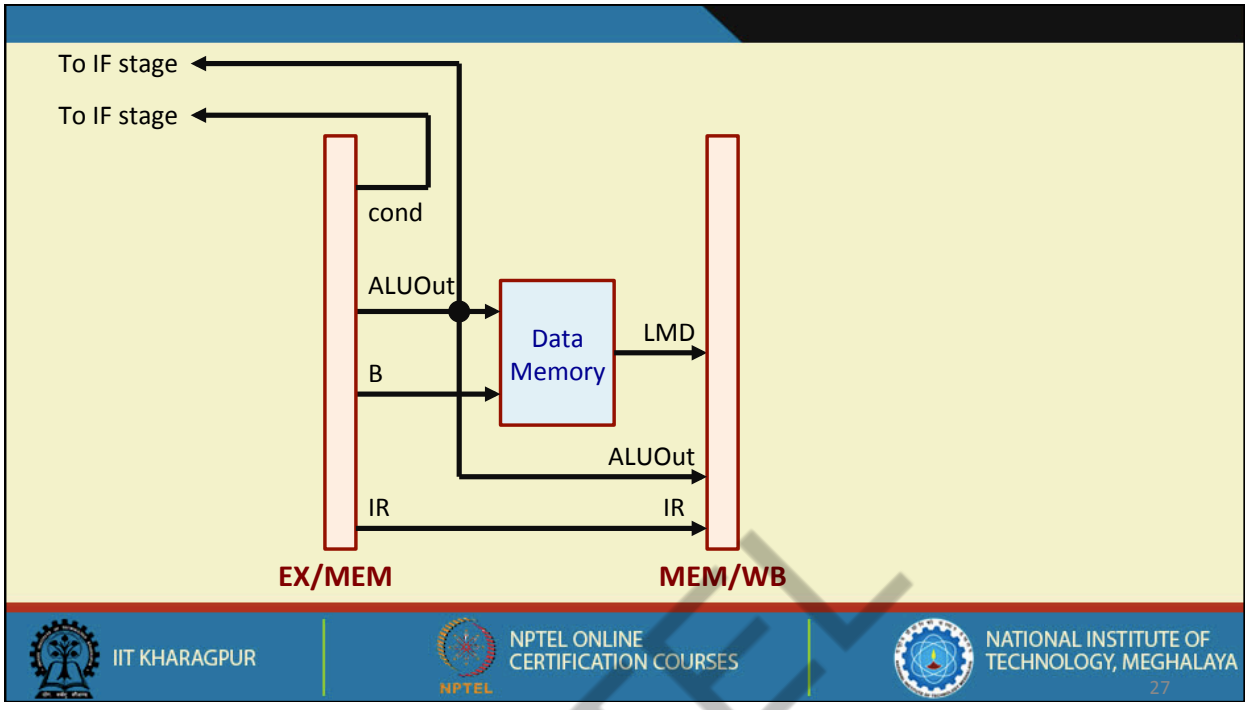
MEM/WB.IR ← EX/MEM.IR;
MEM/WB.LMD ← Mem [EX/MEM.ALUOut];

LOAD

MEM/WB.IR ← EX/MEM.IR;
Mem [EX/MEM.ALUOut] ← EX/MEM.B;

STORE

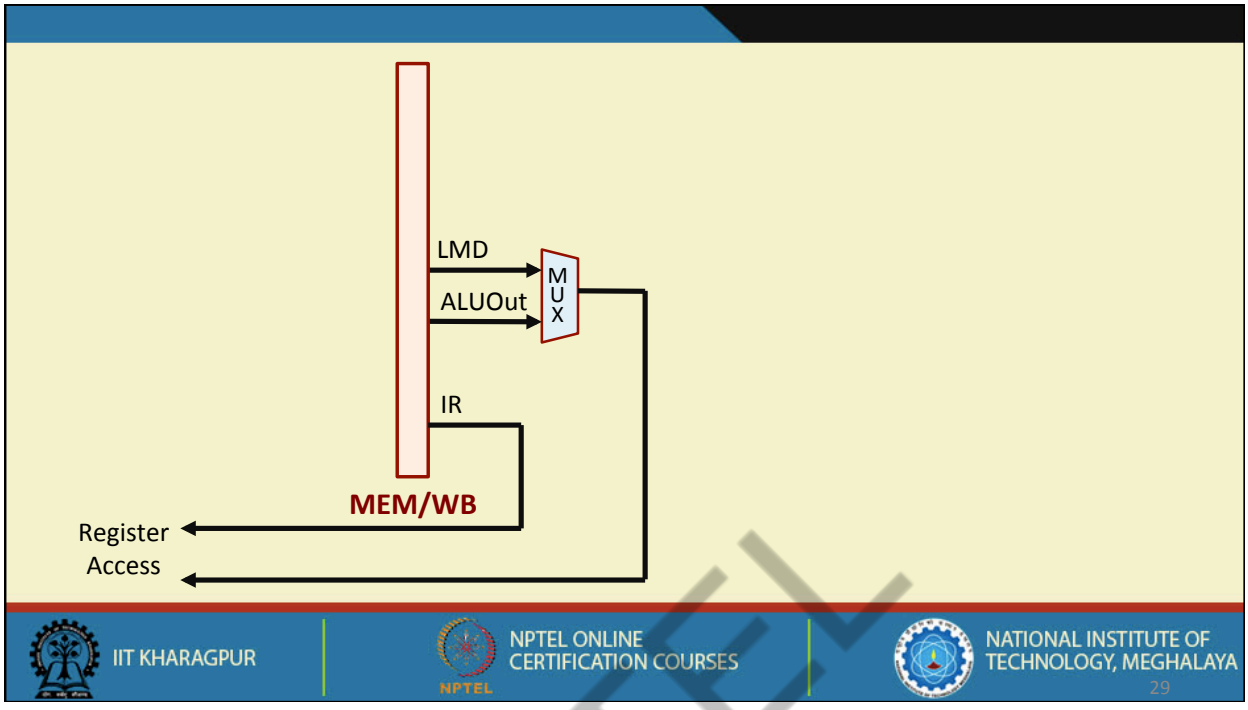
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
26

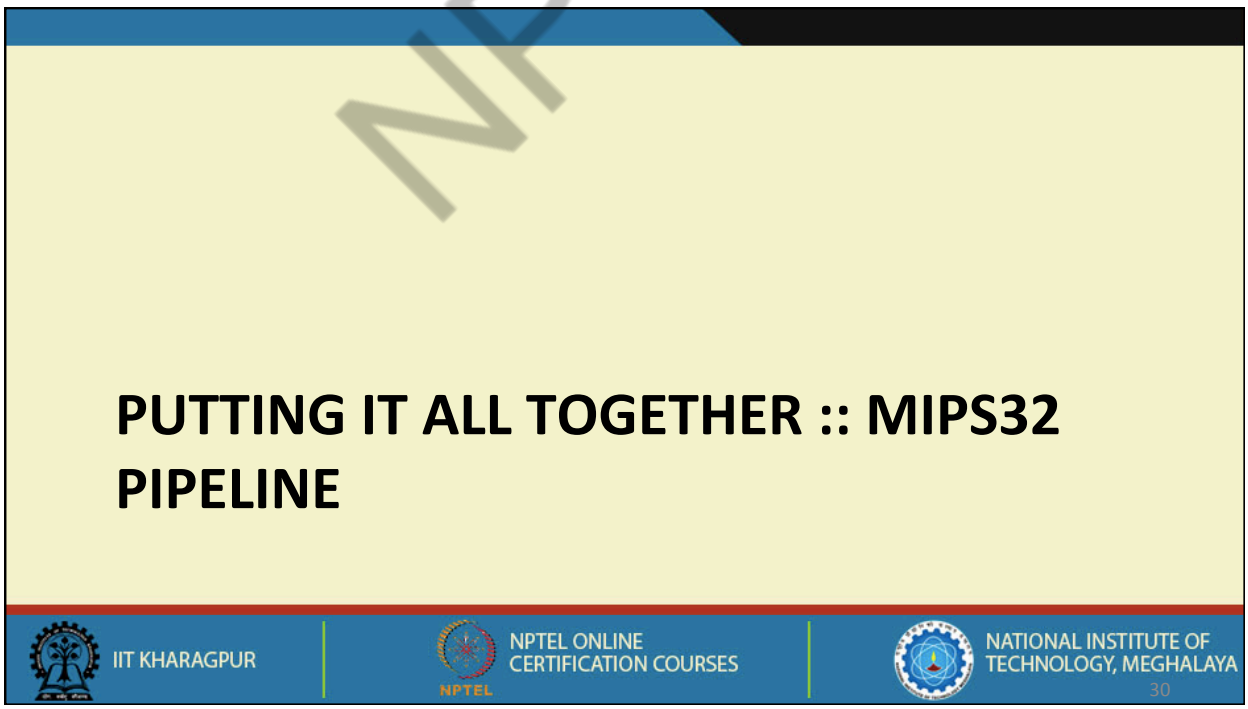


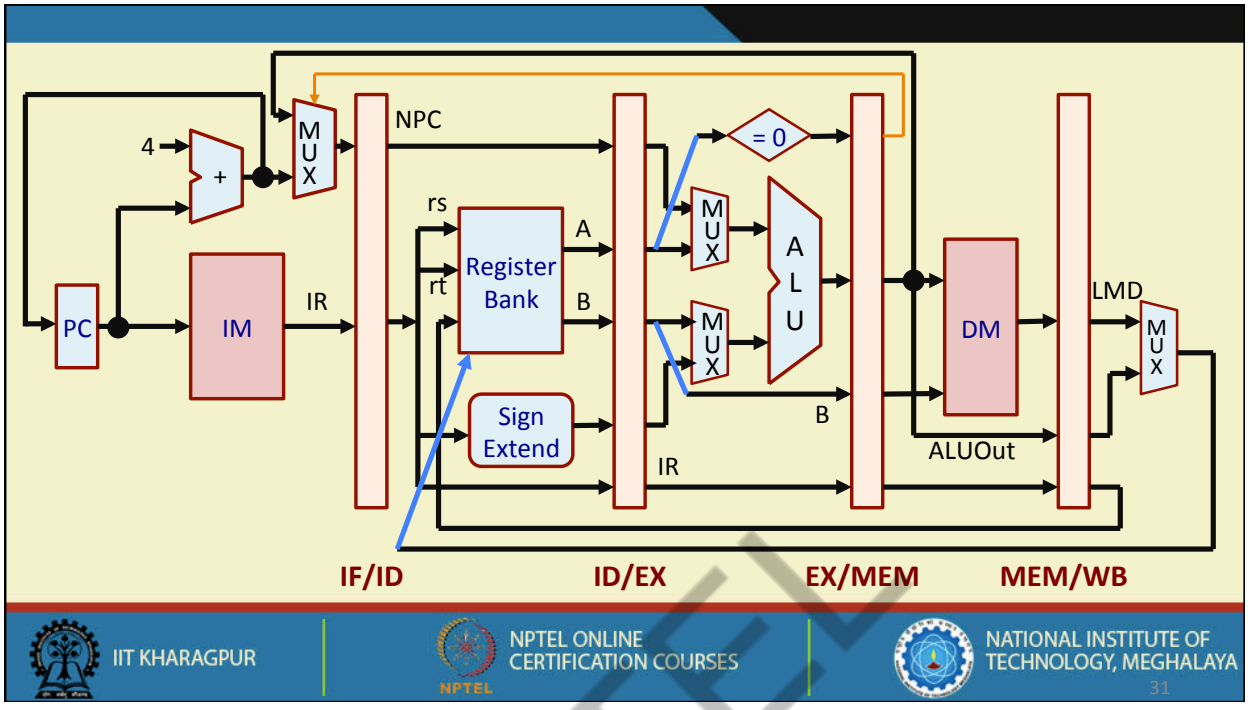
(e) Micro-operations for Pipeline Stage WB

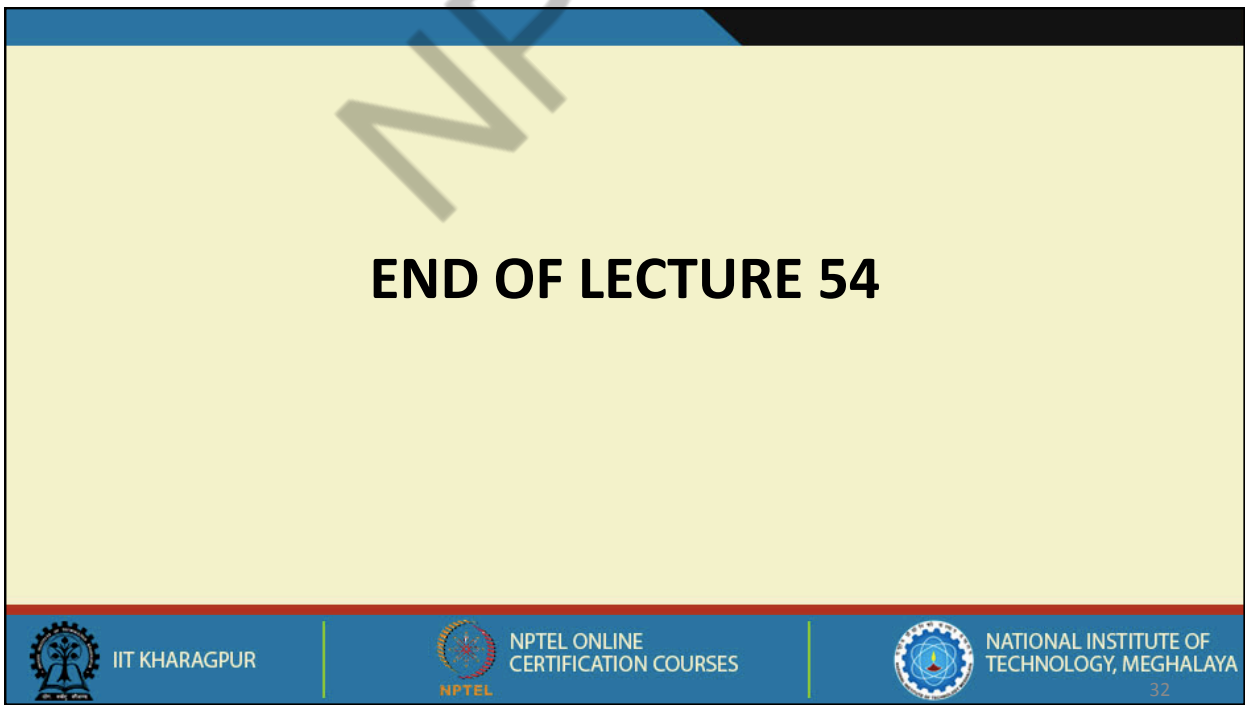
Reg [MEM/WB.IR [rd]] ← MEM/WB. ALUOut;	**R-R ALU**
Reg [MEM/WB.IR [rt]] ← MEM/WB. ALUOut;	**R-M ALU**
Reg [MEM/WB.IR [rt]] ← MEM/WB. LMD;	**LOAD**

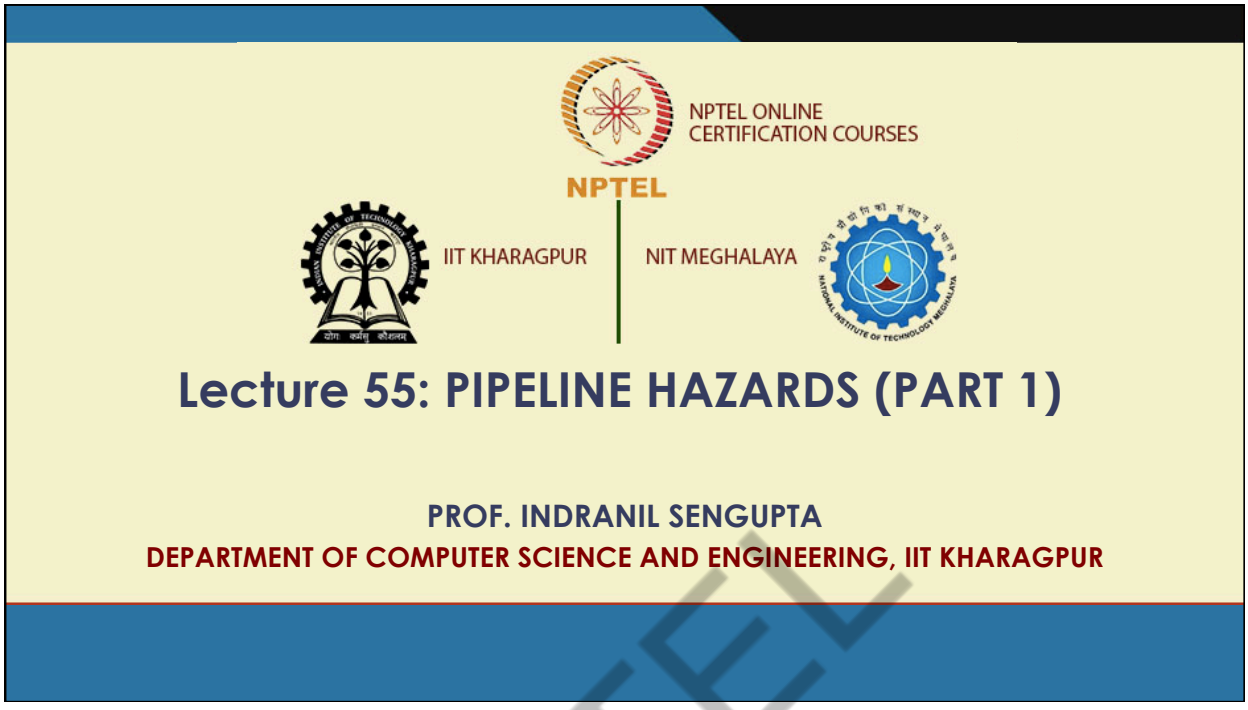
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
28


LMD
ALUOut
M
U
X
IR
MEM/WB
Register
Access
IIT KHARAGPUR
NPTEL ONLINE
CERTIFICATION COURSES
NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA
29


PUTTING IT ALL TOGETHER :: MIPS32
PIPELINE
IIT KHARAGPUR
NPTEL ONLINE
CERTIFICATION COURSES
NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA
30


4
+
MUX
NPC
PC
IM
IR
rs
rt
Register Bank
A
B
Sign Extend
IR
= 0
MUX
MUX
ALU
B
DM
ALUOut
LMD
MUX
IF/ID
ID/EX
EX/MEM
MEM/WB
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
31


END OF LECTURE 54
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
32



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.

Calculation of Performance Degradation

$$\text{Pipeline speedup} = \frac{XT_{nopipe}}{XT_{pipe}} = \frac{CPI_{nopipe} \times C_{nopipe}}{CPI_{pipe} \times C_{pipe}} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{CPI_{nopipe}}{CPI_{pipe}}$$

- We can treat pipelining as a way to reduce the CPI, or reduce C.
  - Let us consider that we are decreasing the CPI.
- The ideal CPI of an instruction pipeline can ne written as:

$$\text{Ideal CPI} = \frac{CPI_{nopipe}}{\text{Pipeline Depth}} \Rightarrow \text{Speedup} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{CPI_{pipe}}$$

- If we restrict ourselves to stall cycles only, we can write:

  CPI_{pipe} = Ideal CPI + (Pipeline Stall Cycles per Instruction)

- We can thus write:

$$\text{Speedup} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

- If we ignore the increase in clock cycle time due to pipelining (i.e. $C_{nopipe} = C_{pipe}$)

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

(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.

Example 1

- Consider an instruction pipeline with the following features:
  – Data references constitute 35% of the instructions.
  – Ideal CPI ignoring structural hazard is 1.3.

  How much faster is the ideal machine without the memory structural hazard, versus the machine with the hazard?

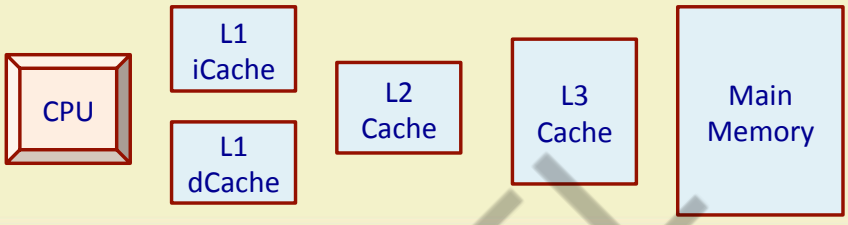
$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

$$\text{Speedup}_{ideal} = \frac{1.3 \times \text{Pipeline Depth}}{1.3 + 0}$$

$$\text{Speedup}_{real} = \frac{1.3 \times \text{Pipeline Depth}}{1.3 + 0.35 \times 1}$$

$$\frac{\text{Speedup}_{ideal}}{\text{Speedup}_{real}} = \frac{1.65}{1.3} = 1.27$$

- Why structural hazards appear in real machines?
  a) To reduce the cost of implementation.
  b) Pipelining all the functional units may be too costly.
  c) If structural hazards are not frequent, it may not be worth the effort and cost to avoid it.
- Memory access structural hazard is quite frequent.
  – Makes use of separate instruction and data caches in the first level.



(b) Data Hazards

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

R2 written here

Instruction	1	2	3	4	5	6
ADD R2,R5,R8	IF	ID	EX	MEM	WB	
SUB R9,R2,R6		IF	ID	EX	MEM	WB

R2 read here

Unless proper precaution is taken, the SUB instruction can fetch the wrong value of R2.

- 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?
  - We shall explore 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).

Reducing Data Hazards using Bypassing

- Basic idea:
  - The result computed by the previous instruction(s) is stored in some register within the data path (e.g. ALUOut).
  - Take the value directly from the register and forward it to the instruction requiring the result.
- What is required?
  - Additional data transfer paths are to be added in the data path.
  - Requires multiplexers to select these additional paths.
  - The control unit identifies data dependencies and selects the multiplexers in a suitable way.

Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	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	
OR R11, R4, R5					IF	ID	EX	MEM	WB

- The first instruction computes R4, which is required by all the subsequent four instructions.
- The dependencies are depicted by red arrows (result written in WB, operands read in ID).
- The last instruction (OR) is not affected by the data dependency.

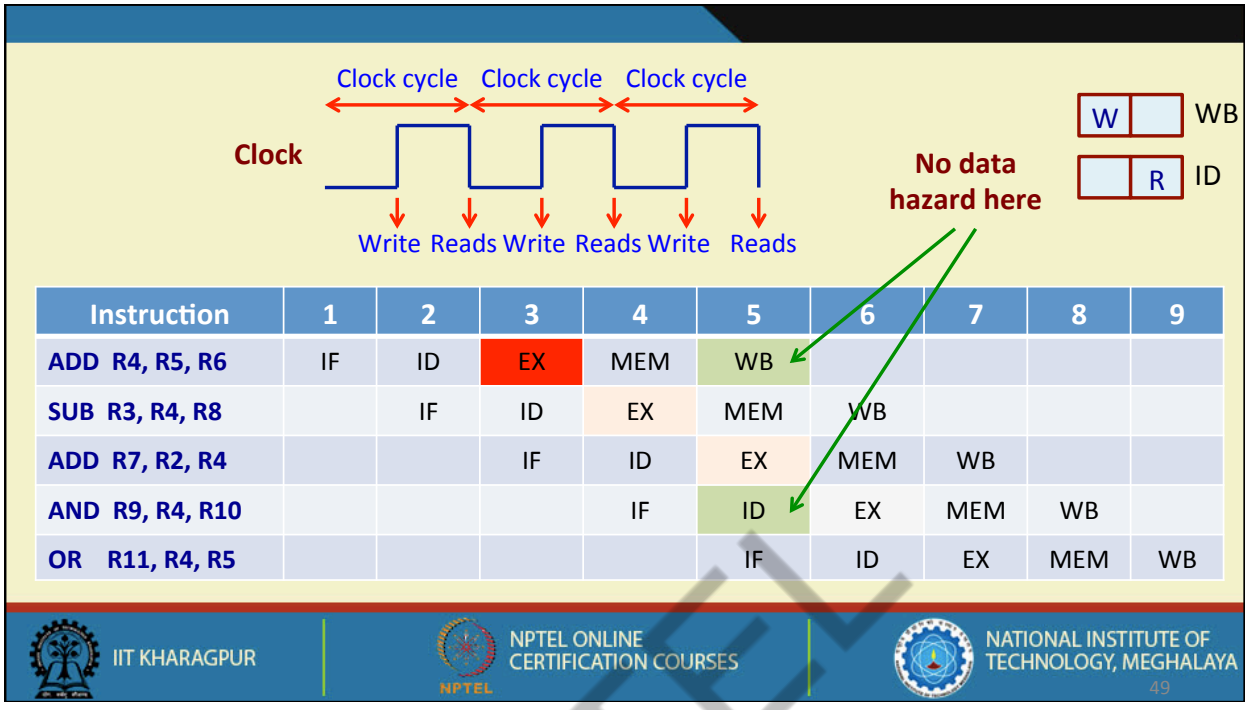
Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	EX	MEM	WB			
ADD R7, R2, R4			IF	ID	EX	MEM	WB		
AND R9, R4, R10				IF	ID	EX	MEM	WB	
OR R11, R4, R5					IF	ID	EX	MEM	WB

Data forwarding requirements:

- The first instruction (ADD) finishes computing the result at the end of EX (shown in RED), but is supposed to write into R4 only in WB.
- We need to forward the result directly from the output of the ALU (in EX stage) to the appropriate ALU input registers of the following instructions.
  - To be used in the respective EX stages, shown by the arrow.

Reducing Data Hazard by Splitting Register Read/Write

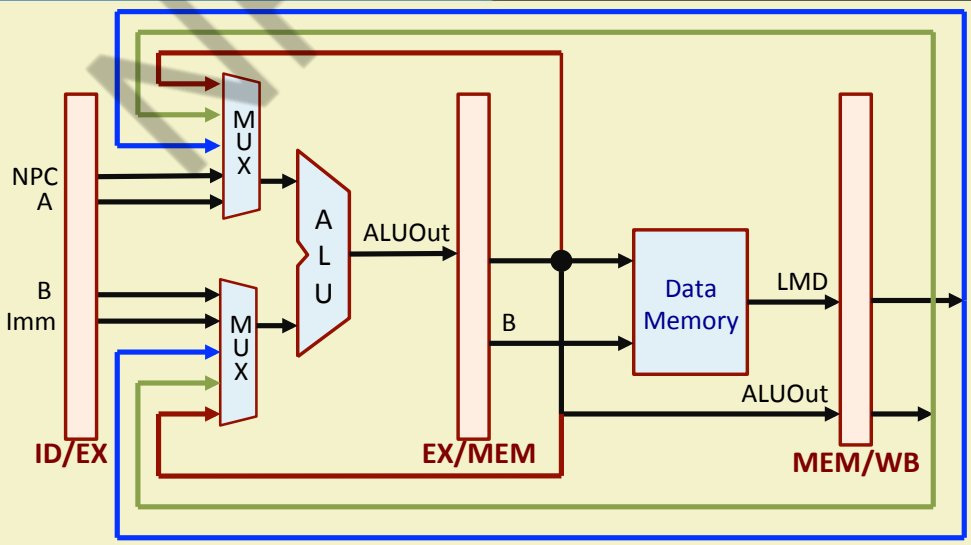
- The data forwarding concept as discussed can solve the data hazards between ALU instructions.
- It is desirable to reduce the number of instructions to be forwarded, since each level requires special hardware.
  - In the forwarding scheme as discussed, we need to forward 3 instructions.
  - We can reduce this number to 2 by avoiding the conflict between the first and the fourth (AND) instruction, where WB and ID are accessed in the same cycle.
  - We perform register write (in WB) during the first half of the clock cycle, and the register reads (in ID) during the second half of the clock cycle.
    - The data hazard between the first and fourth instructions get eliminated.



Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	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	
OR R11, R4, R5					IF	ID	EX	MEM	WB

Data Forwarding Hardware

Only the relevant parts of the MIPS32 pipeline data path is shown.



END OF LECTURE 55

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

51

NPTEL ONLINE CERTIFICATION COURSES

NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 56: PIPELINE HAZARDS (PART 2)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KHARAGPUR

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 1

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

- Some observations:
  - Pipeline scheduling can increase the number of registers required, but results in performance improvement in general.
  - A LOAD instruction requiring that the following instruction do not use the loaded value is called a *delayed load*.
  - A pipeline slot after a LOAD is called the *load delay slot*.
  - If the compiler cannot move some instruction to fill up the delay slot (scheduling), it can insert a NOP (No Operation) instruction.

Types of Data Hazards

- Broadly three classes of data hazards are possible:
  a) **Read After Write (RAW)** – Most common scenario.
  b) **Write After Read (WAR)** – This kind of hazard cannot happen in MIPS32 as all reads are early (in ID) and all writes are late (in WB).
  c) **Write After Write (WAW)** – This kind of hazard is possible in pipelines where write can happen in more than one pipeline stages.
- For MIPS32 integer pipeline, only RAW hazard is possible, and only one type (*LOAD followed by immediate u*se) can result in performance loss.
  - Compiler tries to eliminate the performance loss using instruction scheduling.

(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?
  - The value of PC may or may not change to something other than PC_{old} + 4.
  - 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

Example:
We have seen that 3 cycles are wasted for every branch.
Assume – 30% branch frequency, CPI_{ideal} = 1.

Actual CPI = 0.7 x 1 + 0.3 x 4
= 1.9
Speed becomes almost half.

How to Reduce Branch Stall Penalty?

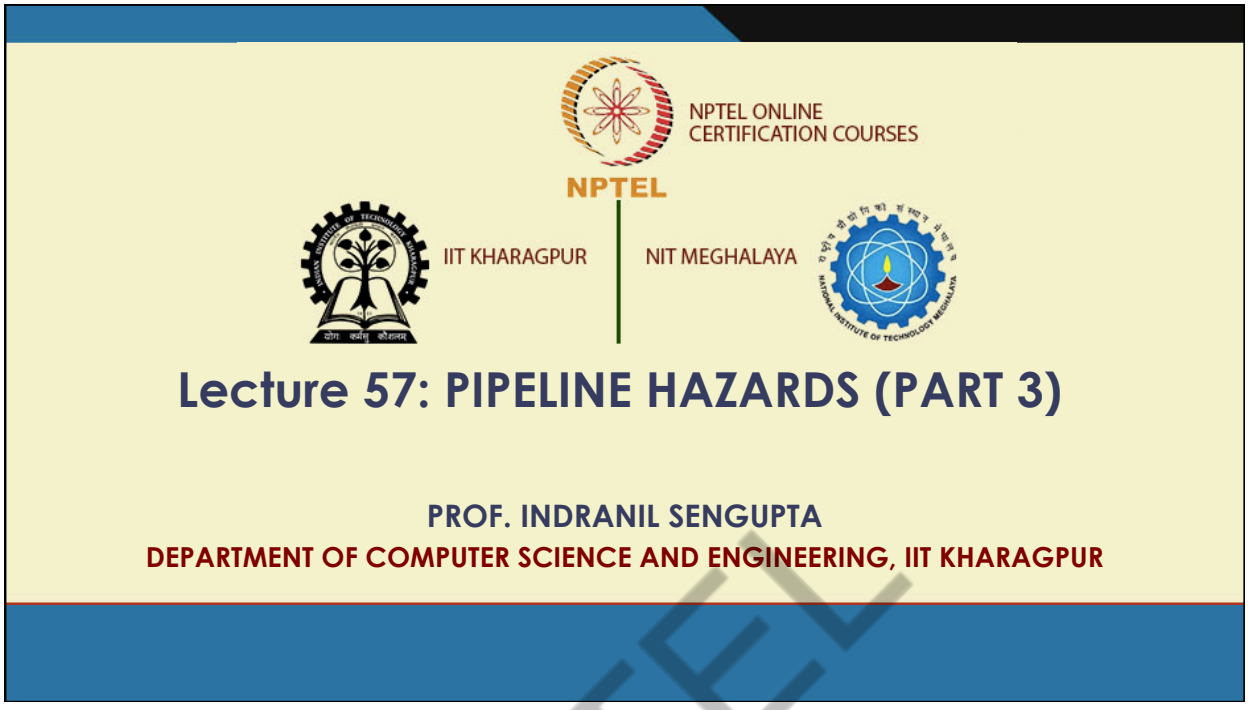
- The penalty can be reduced if both the following are achieved:
  a) Determine whether the branch is taken earlier in the pipeline.
  b) Compute the branch target address earlier in the pipeline.
- How to achieve (a) in MIPS32?
  - In MIPS32, the branches require testing a register for zero, or comparing the values of two registers.
  - Possible to complete this decision by the end of the ID cycle where the registers are pre-fetched, by adding special comparison logic.

- How to achieve (b) in MIPS32?
  - Both the possible PC values can be pre-computed in the IF stage by using a separate adder.
- By the end of the ID cycle, we know whether the branch is taken and also know the branch target address.
  - Requires a single cycle stall during branches.

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

How to reduce pipeline branch penalty?

END OF LECTURE 56



Reducing Pipeline Branch Penalties

- Four techniques based on compile-time guesses are discussed.
  a) The simplest scheme is to freeze the pipeline and insert (one) stall cycle.
     - Main advantage is simplicity.
  b) We predict that the branch is *not taken*, and allow the hardware to continue as if the branch was not executed.
     - Must not change the machine state until the branch outcome is finally known.
     - If the prediction is wrong (i.e. branch is taken), we stop the pipeline and restart the fetch from the target address.
     - Illustration on next slide.

Not Taken Branch (no penalty)

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

Taken Branch (1 cycle penalty)

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

69

c) We predict that the branch is *taken.*

- As soon as the branch outcome is decoded and the target address computed, fetching of instructions can commence from the computed target.
- Since in MIPS32 pipeline, we come to know about the branch outcome and target address together (at the end of ID), there is *no advantage* in this approach.
- May be used for complex instruction machines where the branch outcome is known later than the branch target address.

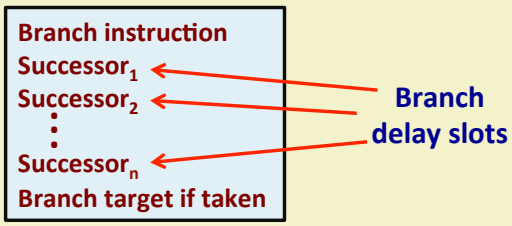
Taken or Not Taken Branch (1 cycle penalty)

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

70

d) We can use a technique called *delayed branch*, which makes the hardware simple but puts more responsibility on the compiler.

- If a branch instruction incurs a penalty of *n* stall cycles, the execution cycle of a branch instruction is defined as follows:



For MIPS32, n=1

- Just like in load-delay slots, the task of the compiler is to try and fill up the branch-delay slots with meaningful instructions.
- Instructions in the branch-delay slot are ***always executed*** irrespective of the outcome of the branch.

Some Examples

```
    ADD   R3,R4,R5
    BEQZ  R2,L
    DELAY SLOT
    …
L:
```

⇩

```
    BEQZ  R2,L
    ADD   R3,R4,R5
    …
L:
```

```
L:  ADD   R1,R2,R8
    SUB   R3,R4,R5
    BEQZ  R3,L
    DELAY SLOT
```

⇩

```
L:  SUB   R3,R4,R5
    BEQZ  R3,L
    ADD   R1,R2,R8
```

```
    ADD   R1,R2,R8
    BEQZ  R3,L
    DELAY SLOT
    SUB   R3,R4,R5
L:
```

⇩

```
    ADD   R1,R2,R8
    BEQZ  R3,L
    SUB   R3,R4,R5
L:
```

- Additional overhead for delayed branches:
  - Multiple PC's (one plus the length of the delay slot, i.e. *n + 1*) are needed to correctly restore the state when an interrupt occurs.
  - Consider a taken branch instruction followed by instruction(s) in the delay slot(s).
    - The PC of branch target and PC's of delay slots are not sequential.

Example 1

- Consider the MIPS32 pipeline with ideal CPI of 1. Assume that 20% of the instructions executed are branch instructions, out of which 75% are taken branches. Evaluate the pipeline speedup for the four techniques discussed to reduce branch penalties.

- **Solution**:

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

$$= \frac{\text{Pipeline Depth}}{1 + (\text{Branch frequency} \times \text{Branch penalty})}$$

(a) Stall pipeline

Branch penalty = 1
Speedup = 5 / (1 + 0.20 x 1)
= 4.17

(b) Predict not taken

Branch penalty = 1
Speedup = 5 / (1 + (0.20 x 0.75))
= 4.35

(c) Predict taken

Branch penalty = 1
Speedup = 5 / (1 + 0.20 x 1)
= 4.17

(d) Delayed branch

Branch penalty = 0.5
Speedup = 5 / (1 + (0.20 x 0.5))
= 4.55

Dealing with Interrupts in MIPS32

- Interrupts complicate the design of an instruction pipeline.
  - Overlapping of instruction executions make it more difficult to decide whether an instruction can safely change the state of the machine.
  - Some interrupts can force the machine to abort the instruction before it is completed (e.g. page fault).
  - The most difficult interrupts to handle in a pipeline have the properties:
    a) They occur within instruction
    b) They have to restarted

- When an interrupt occurs, the pipeline state can be saved safely as follows:
  a) A TRAP instruction is forced into the pipeline at the next IF cycle.
  b) Until the TRAP is taken, disable all writes for the *faulting instruction* and all others that follow. → **prevents any state change**
  c) After the interrupt handler takes control, it saves the PC of the faulting instruction.
- For delayed branches, suppose that an instruction in the branch-delay slot causes the interrupt.
  – If the branch is taken, then the instructions to restart are those in the slot plus the instruction at the branch target.
  – A number of PC values need to be saved and restored.

- After handling the interrupt, special instruction (RFE in MIPS32) return the machine from the interrupt handler by reloading the PC's and restarting the interrupted instructions.
  – *Precise Interrupts*: If the pipeline can be stopped such that the instructions before the faulting instruction are completed, while those after can be restarted from scratch.
  – Supporting precise interrupts is a requirement in many systems (e.g. page fault, IEEE arithmetic, etc.).

- Some exceptions that may occur in MIPS32 pipeline:

Pipeline Stage	Possible Exceptions
IF	Page fault, misaligned memory access, memory protection violation
ID	Illegal opcode
EX	Arithmetic exception
MEM	Page fault, misaligned memory access, memory protection violation
WB	None

- Multiple interrupts may also occur in the same clock cycle.
- An example:

Instruction	1	2	3	4	5	6
LW R1,100(R2)	IF	ID	EX	MEM	WB	
ADD R4,R5,R6		IF	ID	EX	MEM	WB

  - Can cause a data page fault (in MEM) and arithmetic exception (in EX) at the same time (clock cycle 4).
  - As a possible solution, we can deal only with the data page fault and then restart the execution. The second interrupt will occur again, and will be handled later.

- Difficulty: Interrupts may appear out of order.
  - LW causes a data page fault (cycle 4), and ADD causes an instruction page fault (cycle 2).

Instruction	1	2	3	4	5	6
LW R1,100(R2)	IF	ID	EX	MEM	WB	
ADD R4,R5,R6		IF	ID	EX	MEM	WB

- A possible solution to this problem is discussed next.

- Possible Solution:
  - The hardware posts each interrupt in a status vector, which is carried along with each instruction as it moves through the pipeline.
  - When an instruction reaches WB, the interrupt status vector is checked and handled if present.
  - Guarantees that all interrupt handling is carried out in *precise* order.

END OF LECTURE 57

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

83

NPTEL ONLINE CERTIFICATION COURSES

NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 58: PIPELINE HAZARDS (PART 4)

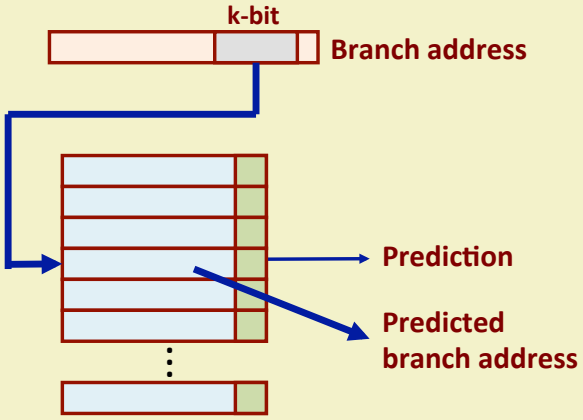
PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KHARAGPUR

Introduction

- What we have seen so far to reduce branch penalties?
  - Simple hardware-based schemes that uses a *static* prediction approach (assume *taken* or *not taken*).
  - Software-based scheme with compiler support (*delayed branches*).
- We shall now explore some hardware based approaches to *dynamically* predict the outcome of a branch.
  - Prediction changes with change in branch behavior.

Branch Prediction Buffer (BPB)

- The BPB is a small high-speed memory (like cache) that is indexed by the lower few bits of the branch instruction address.
- 1-bit prediction scheme:
  - The memory contains the target address of branch, and also a bit that says whether the branch was recently taken or not.



- 1-bit prediction scheme (contd.):
  - The prediction may be incorrect → it may correspond to some other branch instruction that has the same lower-order *k* address bits.
  - In this scheme, instruction fetching begins from the predicted address.
  - If the prediction is later found to be wrong, the prediction bit is inverted.
- Drawback of the 1-bit prediction scheme:
  - Consider a branch that is taken most of the time. When it is not taken, there will be two incorrect predictions, rather than one.
  - Same is true for the reverse case (i.e. a branch not taken most of the time).

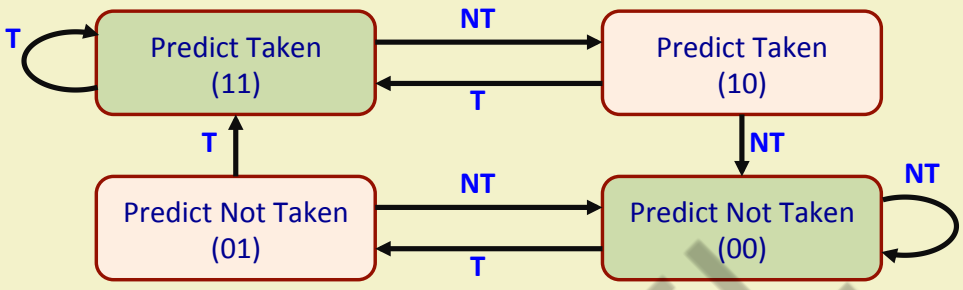
Example 1

- Consider the following nested loop where the inner loop iterates for 20 times before exiting.

```
for (loop=0; loop<1000; loop++)
{
    ……
    for (i=0; i<20; i++)
      A[i] = A[i] + 5;
}
```

- For every execution of the inner loop, there will be two mispredictions:
  a) Last iteration of the current loop.
  b) First iteration of the next loop.
- Though the branch is taken 95% of the time (19 out of 20), correct prediction occurs only 18 times.
  - 90% accuracy.

- 2-bit prediction scheme:
  - Tries to avoid two mispredictions per loop in the 1-bit prediction scheme.
  - Here, a prediction is changed (taken to not-taken, or vice versa) only after *two consecutive mispredictions*.
  - Can be captured by the following state transition diagram:

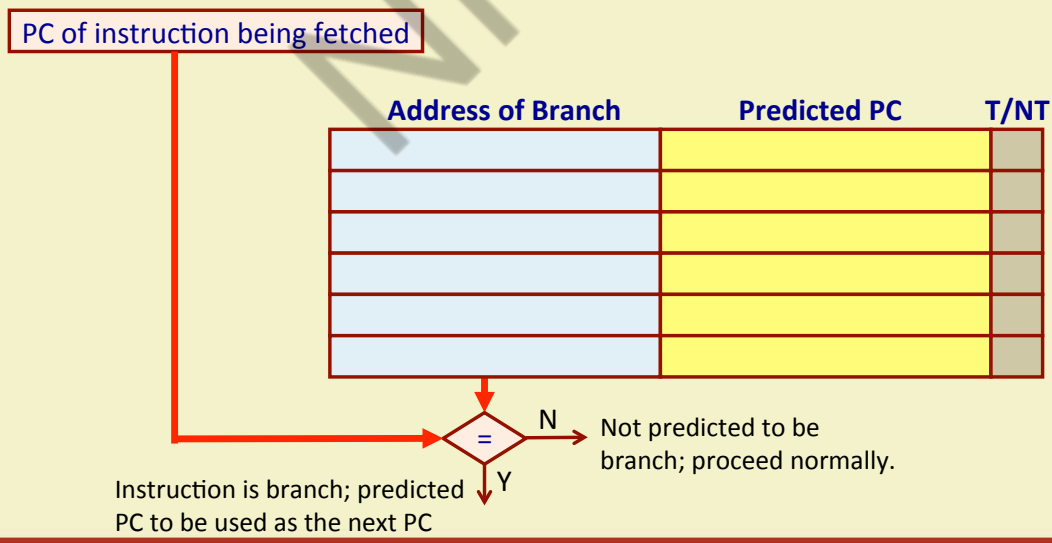


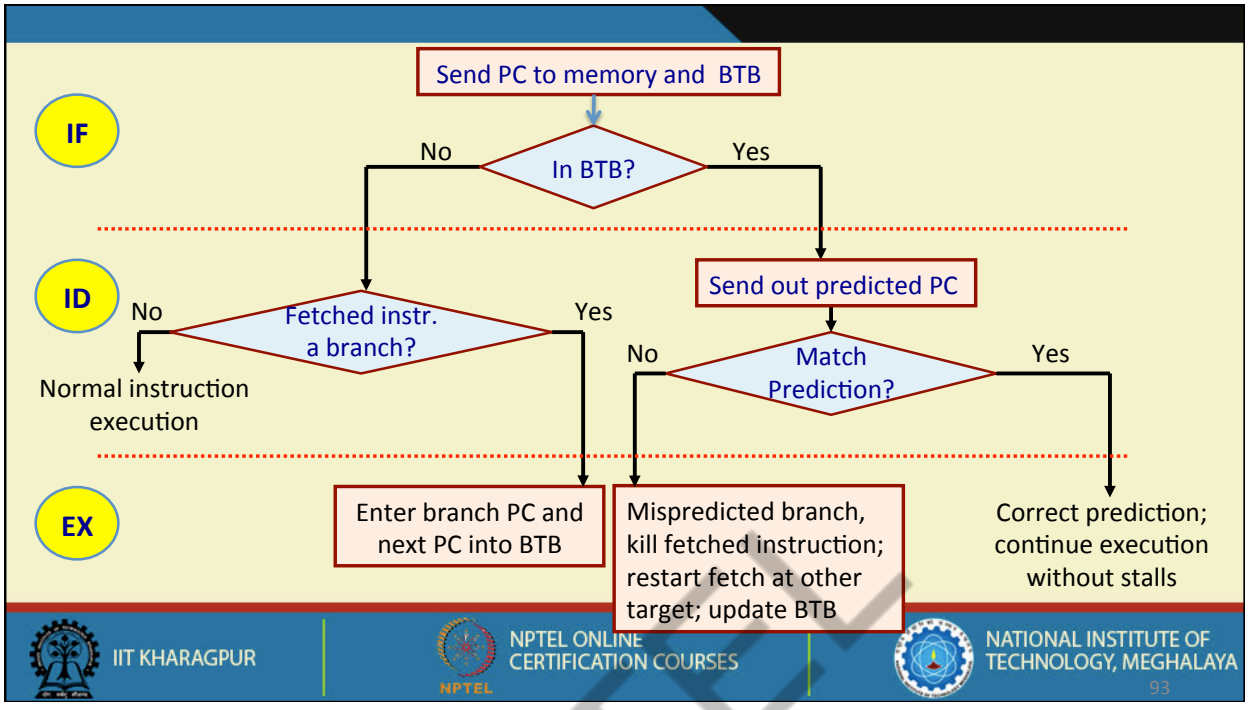
Each state is encoded in 2 bits.

- An observation:
  - It may be noted that the 2-bit prediction scheme as discussed cannot speed up the MIPS32 pipeline.
  - In MIPS32 pipeline, both the branch outcome and branch target address gets known together (viz. at the end of ID).
  - To improve MIPS32 pipeline performance, we need to know from what address to fetch by the end of IF.
    - Whether the instruction is a branch (not decoded yet).
    - What is the predicted next value of PC?

Branch Target Buffer (BTB)

- The BTB is a high-speed memory that stores the predicted address for the next instruction after a branch.
  - Since we are predicting the next instruction address from where fetching will start *before decoding* the instruction, it is required to know whether the fetched instruction is predicted as a taken branch.
  - The PC of the instruction being fetched is matched against a set of instruction addresses stored in the BTB → represent addresses of *known branches*.
  - BTB also stores whether the branch was predicted T or NT, and helps keep the misprediction penalty small.





- There will not be any branch penalty if an entry is found in the BTB and is correct.
  - Otherwise, there will be penalty of at least one clock cycle.
  - Since the BTB may also need to be updated, the penalty can be two clock cycles.

Found in BTB	Prediction	Actual Branch	Penalty Cycles
Y	T	T	0
Y	T	NT	2
Y	NT	NT	0
Y	NT	T	2
N	-	T	2
N	-	NT	1

Example 2

- Consider the following parameters of a MIPS pipeline with BTB:
  - 90% of the branches are found in BTB.
  - 8% of the predictions are incorrect.
  - 75% of the branches are taken.

  Compute the branch penalty.

Delayed branch requires penalty of 0.5 clock cycles on the average

Branch penalty = (% Branches found in BTB x % Mispredictions x 2)
+ (% Branches not found in BTB x % Taken branches x 2)
+ (% Branches not found in BTB x % Not-taken branches x 1)
= 0.90 x 0.08 x 2 + 0.10 x 0.75 x 2 + 0.10 x 0.25 x 1 = 0.32 clock cycles

Conditional Instructions

- Conditional instructions can help eliminate some branch instructions and hence also the corresponding branch penalties.
  - MIPS32 has a number of conditional instructions (e.g. conditional move).

An example code:

```
if (X == 0) A = B;
```

Assume:
- R1 holds X
- R2 holds A
- R3 holds B

Using branch

```
    BNEZ   R1,L
    ADDU   R2,R3,R0
L:
```

Conditional move

```
CMOVZ   R2,R3,R1
```

Essentially, we converted the control dependence into data dependence.

END OF LECTURE 58