Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 55
Pipeline Hazards (Part 1)

In the last lecture we have seen the pipeline structure for the MIPS32 processor. You may say that it is an ideal pipeline as we have not considered many of the real problems or conflicts that can arise and cause so-called hazards during instruction execution.

(Refer Slide Time: 00:57)

NPTEL ONLINE CERTIFICATION COURSES

NPTEL

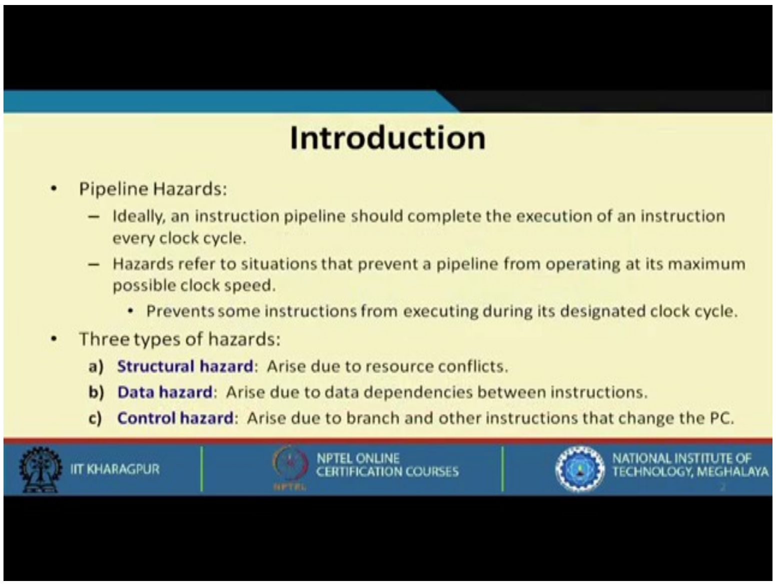
IIT KHARAGPUR | NIT MEGHALAYA

Lecture 55: PIPELINE HAZARDS (PART 1)

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

In the next few lectures we shall be looking at these kinds of conflicts or hazards that can impact or degrade the performance on a pipeline to a great extent, and what are the approaches we can use to avoid such degradation to the extent possible. We start our discussion on pipeline hazards in this lecture.

(Refer Slide Time: 01:23)



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.

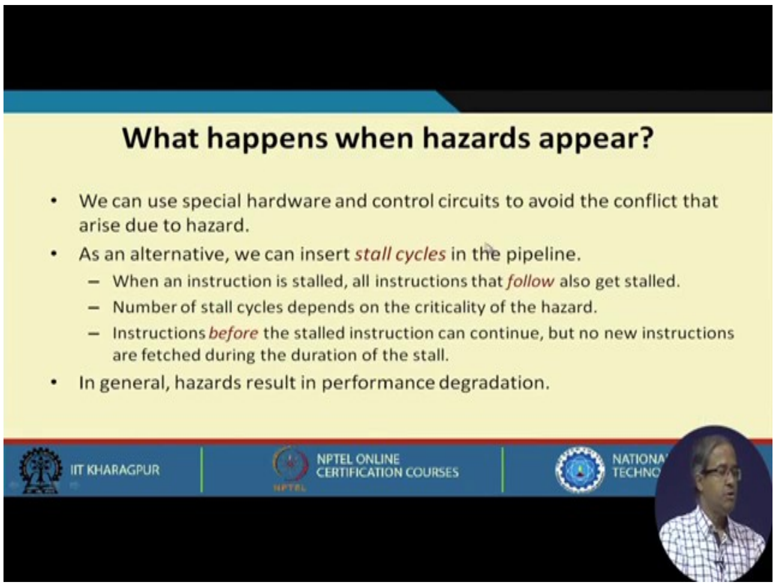
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us try to first understand what is a pipeline hazard. We have seen an ideal pipeline. An instruction pipeline in the ideal sense should complete the execution of one instruction every clock cycle, but when we say there is a hazard, it means some kind of a scenario because of which we cannot operate the pipeline at its maximum possible speed. Because you see there are several instructions that have entered the pipe already, several instructions are in various stages of execution, there can be some dependencies between the instructions. They may be trying to access some common resources because of which there can be a conflict, because of branch instructions we may have to wait till we know the outcome of the branch and the target address. There are several such instances that may prevent a pipeline to work with its maximum possible capability or speed.

Loosely speaking such situations are called hazards. They prevent a pipeline from operating at its maximum possible clock speed, which means that we cannot feed an instruction every clock cycle. Such hazards can prevent some instructions from executing during its designated clock cycle. This means, suppose an instruction was supposed to be entering the pipeline now, but maybe it will have to wait for one clock cycle. Broadly speaking hazards can be classified into 3 types; structural, data and control. Structural hazard arises due to resource conflicts, like one example you have already seen in memory access, IF and MEM, two instructions may be trying to access instructions and data leading to structural hazard. If there was a single memory module then one of the instructions will have to wait, but because we have use separate instruction and data caches they can proceed together.

Because of data dependencies between instructions, say one instruction is producing a result that the next instruction is using, data hazards may arise. Control hazard may arise due to branch and other instructions like interrupts that change the program counter.

(Refer Slide Time: 04:54)



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.

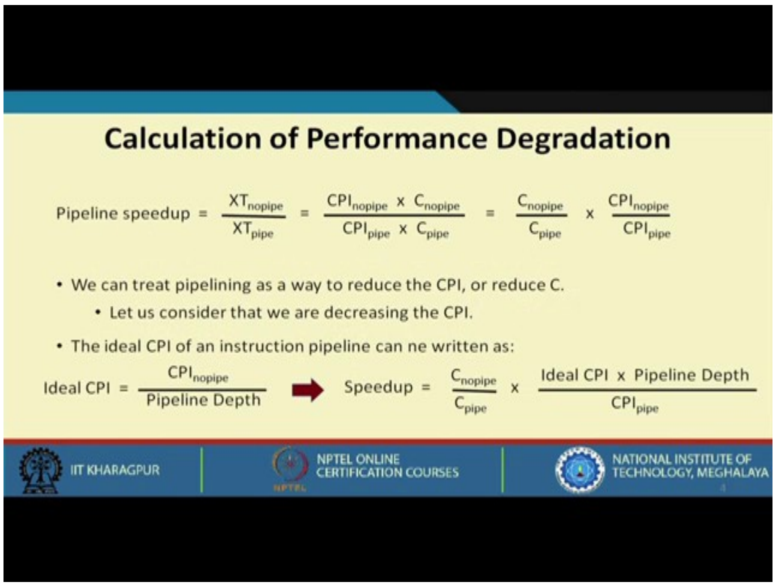
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONA TECHNO

What happens when such hazards show up? Well there are several alternate strategies we shall be looking at, some of them use some special hardware to detect the hazard. To detect hazards whenever they appear, and to try and avoid the conflict; one alternative is to have some hardware, which will be trying to detect such situations that can result in hazard, and try to overcome them automatically. But as an alternative we can simply insert stall cycles without using special hardware; this is a cheaper alternative.

Suppose I find that there is a hazard, if I allow the next instruction to enter the pipe there will be an clash somewhere. So I hold the instruction back, maybe I feed it in the next clock, not in the present clock. I am inserting a stall cycle, for one cycle I am not feeding anything in the pipeline. I am stalling the pipeline for one cycle and then again I am feeding the next instruction after that. Then the hazard will not show up. If I follow this principle and I stall one instruction, then all the instructions that follow this particular instruction will also get stalled because unless this instruction enters the pipe those instructions will not be able to enter the pipe also, and depending on the criticality of the hazard number of stall cycles can vary.

The important thing to notice is that whenever we are inserting stall cycles like this, the understanding is that the instructions that have already entered the pipeline run in the process of execution, they can proceed normally; there is no problem, but I am holding back the instructions that are following, I am inserting one or more stall cycles. They will be delayed, and after the stall cycles they will be fed into the pipeline. This is what I just now said, instruction before the stall instruction can continue, but no new instructions can be fetched during the duration of the stall. We may have to insert such stall cycles and this will result in performance degradation, because we are not allowing instructions to be fed or entering the pipe in every clock cycle, that is why we are not able to get the full pipeline performance that is possible in the ideal case.

(Refer Slide Time: 08:38)



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:

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

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

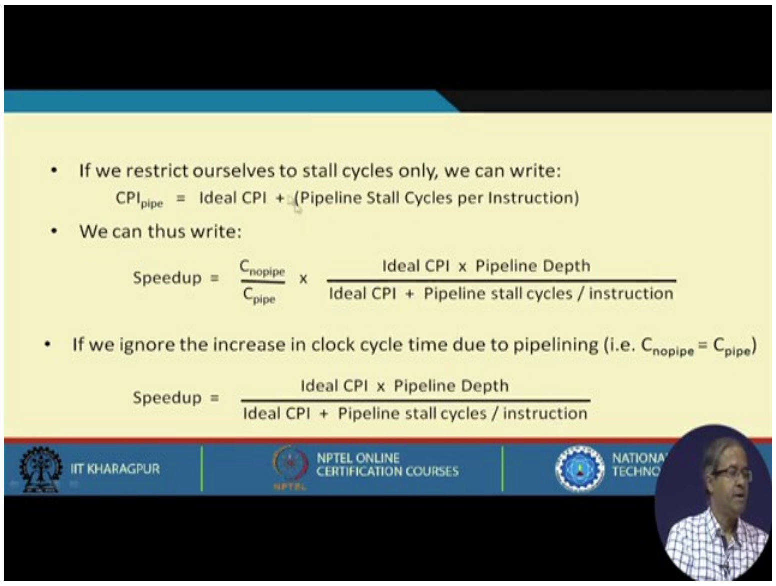
Let us make a simple calculation. With respect to a non-pipelined version suppose you are trying to estimate the speedup in a pipeline. A simple measure of speedup is to divide the execution time of the non-pipelined version with the execution time of the pipelined version. How I can calculate execution time? One simple way will be to multiply the cycles per instruction of the non-pipelined version with the clock cycle time assuming that the number of instructions are the same in the both the cases. Similarly for the pipelined version CPI for the pipelined version multiplied by the clock cycle time.

So, I can separate this out -- the clock cycle times and the CPIs. Now when we talk about pipelining we can either say that we are trying to reduce the CPI, or we can argue that we are trying to reduce C. But actually reducing C is not true because we are not really reducing the clock cycle time, but you can argue in some way that in a non-pipelined version you had the whole computation as a block. As if your clock was running, let us say, five times lower.

But now by making a pipeline with five stages, you are making the clock run five times faster that may be one argument, but a better argument is that we are not saying we are making clock faster, but we are saying we are making CPI smaller. You can argue in either way.

Another thing is the ideal CPI. In the absence of any hazards it will be equal to the CPI of the non-pipelined version divided by pipeline depth. Pipeline depth means number of stages in the pipeline; in our example pipeline depth was 5. In the ideal case we are getting a speedup of 5. So, CPI was reducing by 5 times.

(Refer Slide Time: 12:06).



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

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

- We can thus write:

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

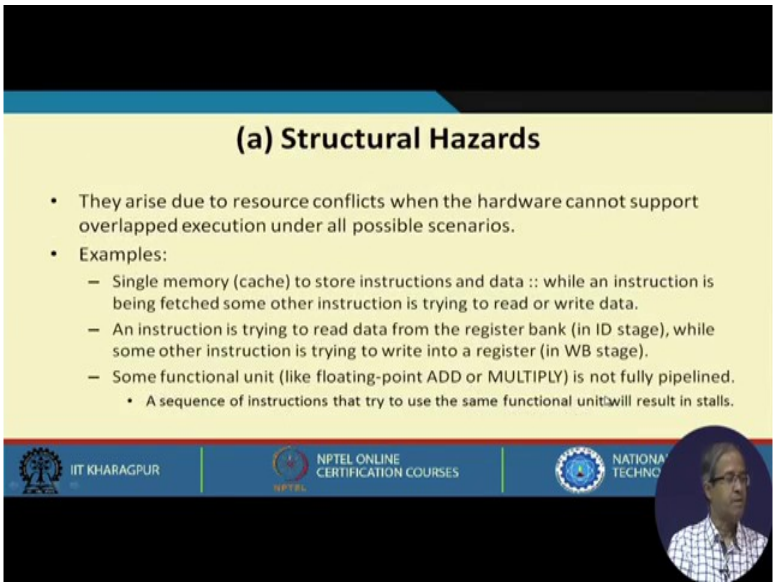
- If we ignore the increase in clock cycle time due to pipelining (i.e. $C_{nopipe} = C_{pipe}$)

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

Now suppose we are using the method of inserting stall cycles for removing hazards. The actual CPI of the pipeline will obviously be greater than the ideal CPI. It will be ideal CPI plus average pipeline stall cycles per instruction; this will be the impact of the hazard on the average case. So, CPI_pipe you can write like this.

Just remember this expression. We shall be using this later in some examples.

(Refer Slide Time: 13:55)



(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.

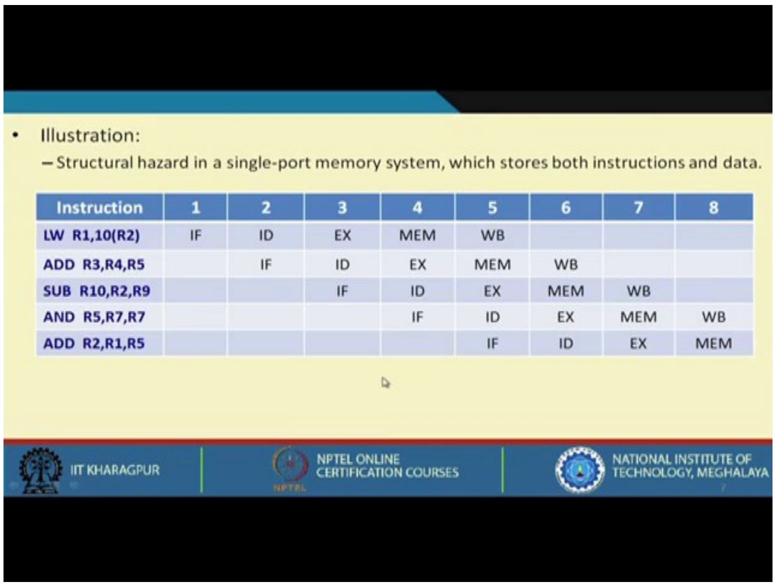
Let us look at the hazards now one by one. Structural hazards arise due to resource conflicts. Suppose I have a single copy of a resource and two instructions in two stages are trying to use the same resource. We have seen two examples, one is for memory and the other is for the register banks, the ID stage is trying to read from register, WB stage is trying to write into a register. These are examples of structural hazards. So, if the hardware does not support concurrent or overlapped execution, structural hazards will show up. If we have a single cache and not separate instruction and data, then when an instruction is being fetched and some other instruction is trying to read or write, the next instruction will have to wait.

So, you have to insert a stall cycle. Similarly an instruction is trying to read data from the register bank while some instruction is trying to write into a register. Again if you do not follow the principle I mentioned earlier, that you write in the first half of the clock and read in the second half of the clock, then the instruction that is trying to read data will have to wait.

So you see that if you do not take precautions or put in some additional hardware to detect and avoid this kind of a hazard, your stall cycles will get inserted automatically; otherwise your instruction execution process will not be correct. Another example I am giving. Let us say some of the functional units like floating point add or multiply, they are not fully pipelined. And suppose there are two consecutive instructions that are trying to use floating point add or floating point multiply. Because they are instructions that consume more than one cycles, they will result in stall cycles because when say the first instruction is doing multiply they will be using multiple clock cycles during the EX stage, because multiply cannot finish in one cycle.

So, it is consuming the EX stage for more than one cycle. If the next instruction is also requiring floating point multiplication, next instruction will have to wait and you will have to insert stall cycles.

(Refer Slide Time: 17:21)



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

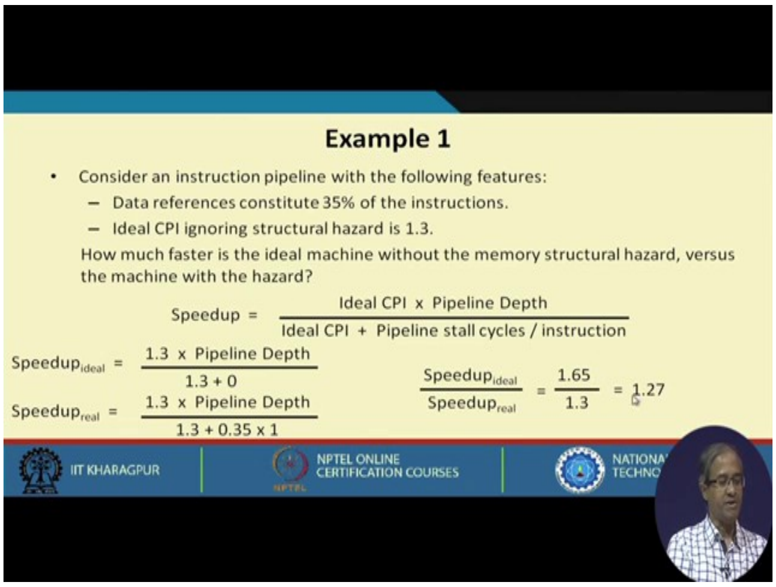
Instruction	1	2	3	4	5	6	7	8
LW R1,10(R2)	IF	ID	EX	MEM	WB			
ADD R3,R4,R5		IF	ID	EX	MEM	WB		
SUB R10,R2,R9			IF	ID	EX	MEM	WB	
AND R5,R7,R7				IF	ID	EX	MEM	WB
ADD R2,R1,R5					IF	ID	EX	MEM

Let us say we have a single port memory system; that means, a single memory system that stores both instruction and data. We do not have separate instruction cache and data cache. We have a sequence of instructions, a LOAD followed by a sequence of arithmetic and logic instructions. In the ideal case execution will proceed like this.

But as I said this LOAD instruction will be trying to access memory here, do a load, and this instruction will try to do the instruction fetch here in IF. So, there will be a clash; this will be a structural hazard. Suppose we have not replicated the memory. I can insert a stall cycle here. The hardware will automatically detect that there is a conflict, this instruction is trying to access memory, this was a LOAD instruction. So, you cannot do an instruction fetch here. It will automatically insert a stall cycle and it will resume the instruction fetch in the next cycle. All instructions that follow will also incur a one cycle delay.

Here we show that how we can insert a stall cycle to eliminate this structural hazard.

(Refer Slide Time: 19:30)



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?

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

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

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

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

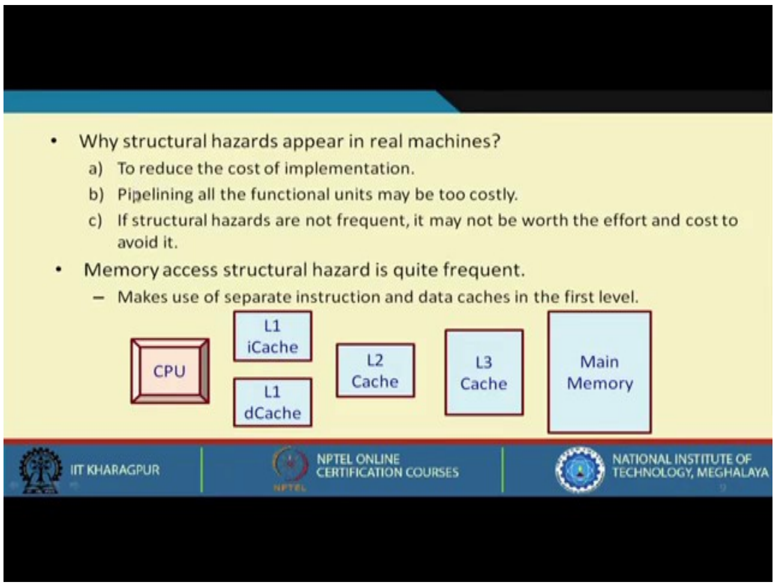
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONA TECHNO

Now, let us work out a simple example. We consider an instruction pipeline, where data references constitute 35% of the instructions, which means load and stores. And ideal CPI ignoring structural hazard is 1.3.

Now the question is how much faster will the ideal machine without the memory structural hazard be. We use this expression that was worked out earlier. For the ideal machine the ideal CPI is 1.3 and for the ideal machine there is no structural hazard. So, pipeline stall cycles per instruction will be 0, and for the real machine what will happen? For all such data reference instruction there will be one cycle delay because of this stall. So, ideal CPI is 1.3 and pipeline stall cycle per instruction it will happen 35% of the time and for each of that case there will be a 1 stall cycle inserted.

If you just calculate the speedup this becomes 1.65; 1.65 by 1.3 is 1.27. So, the ideal machine is 1.27 times faster than the real machine. This means that in reality whenever you have this kind of hazards, your performance degrades. There will be a 27% degradation in the performance. Now the question is why cannot we remove structural hazards by inserting additional hardware since this is so important. The question is how much is the cost to replicate the hardware. For I-cache and D-cache you can possibly do without much increase in cost, but there are a few things that you possibly cannot replicate. So, to reduce cost of implementation sometime structural hazards remain in certain cases. For instance, pipelining all the functional units may be too costly, for instance the divider.

(Refer Slide Time: 23:14)

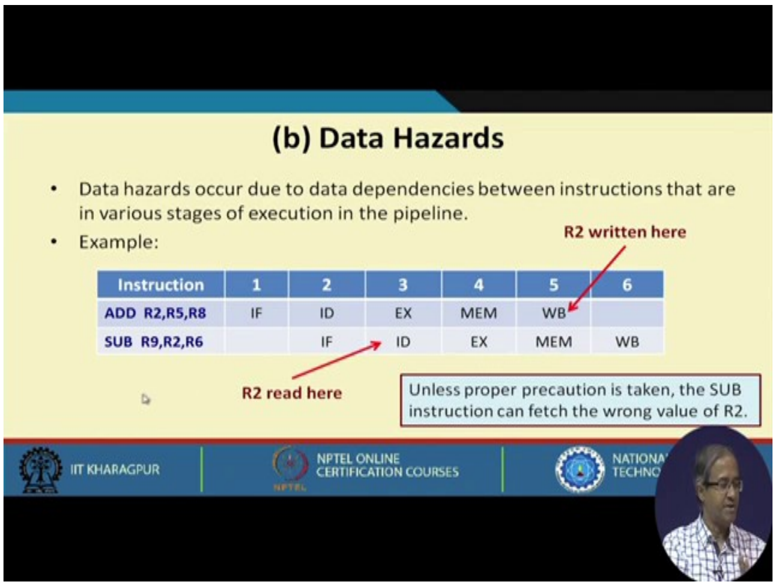


- Why structural hazards appear in real machines?
  a) To reduce the cost of implementation.
  b) Pipelining all the functional units may be too costly.
  c) If structural hazards are not frequent, it may not be worth the effort and cost to avoid it.
- Memory access structural hazard is quite frequent.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

CPU
L1 iCache
L1 dCache
L2 Cache
L3 Cache
Main Memory
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

And third thing is that you can bank on Amdahl's law again you see that how frequently such structural hazard scenarios occur. If you see that such structural hazards are not that frequent then you may avoid the effort and cost; you may see that well I find floating point division in a program appears very rarely. So, let me not invest anything to improve that to remove that hazard, let that hazard remain. Whenever there is a floating point division followed by another division let stall cycles be inserted, but for most of the scenarios such occurrences will not happen. We are trying to make the common case fast, but because memory access structural hazard is very frequent, we replicate the hardware because this is something we cannot compromise, because if we use a single integrated cache for every memory access we may have to use one stall cycle that is too expensive.

(Refer Slide Time: 24:30)



(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

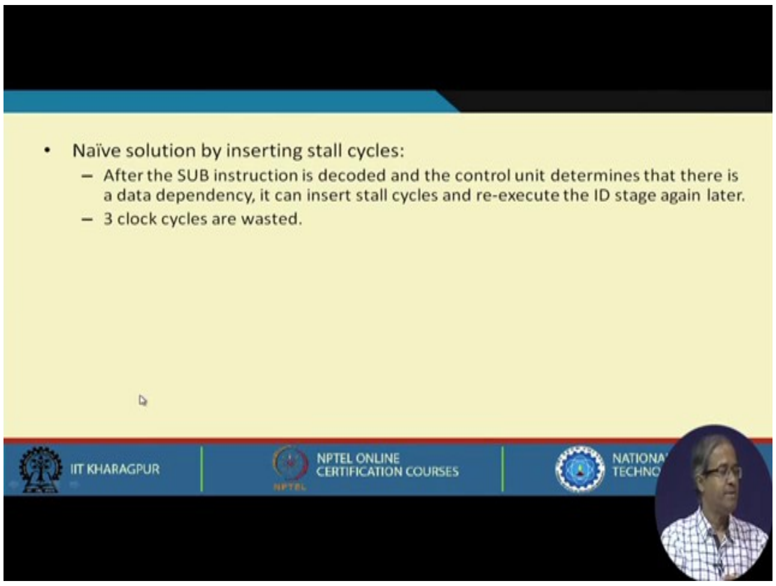
Unless proper precaution is taken, the SUB instruction can fetch the wrong value of R2.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONA TECHNO

Now, let us come to data hazards. Data hazard means there is some dependency among the instructions. Let us say I have an ADD instruction that produces a result in R2, there is a following instructions which uses the value of R2. So, unless you take proper precaution you see normally what happens. Normally this ADD instruction will be writing the value of R2 in WB stage and SUB instruction will be prefetching all register operands in ID. So, it is trying to prefetch before the value has been written.

If you do not take proper precaution this SUB instruction can fetch the old value of R2 and not the latest one, because R2 is written here and you are trying to read R2 in the ID phase here; this is what data hazard means. Apparently what it means is that you possibly have to insert two or more stall cycles, because unless WB completes you cannot use ID, maybe this ID you have to shift here, but you will see that we can do better.

(Refer Slide Time: 26:06)

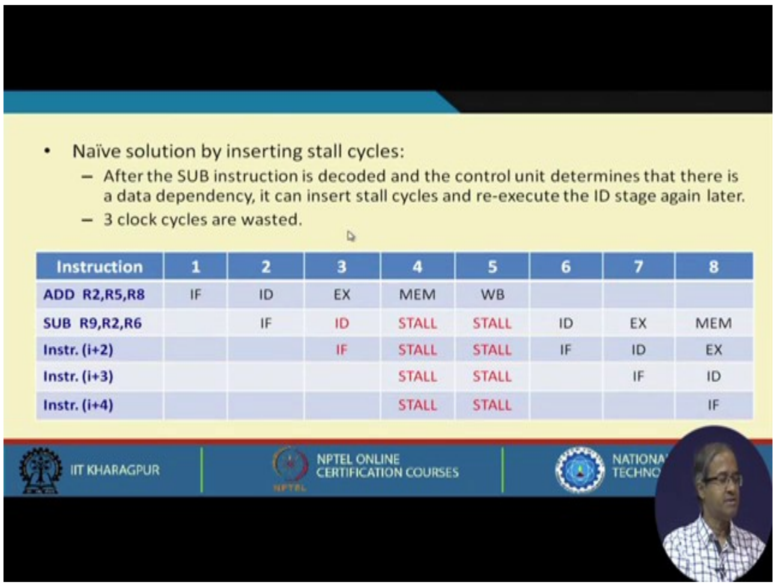


- Naïve solution by inserting stall cycles:
  - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
  - 3 clock cycles are wasted.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

A naïve solution is to insert stall cycles; after the SUB is decoded, we see that here also we are using R2 ne. So, we will be waiting till WB is completed -- insert stall cycles. So, in the naïve implementation 3 clock cycles stall should be used like this.

(Refer Slide Time: 26:29)

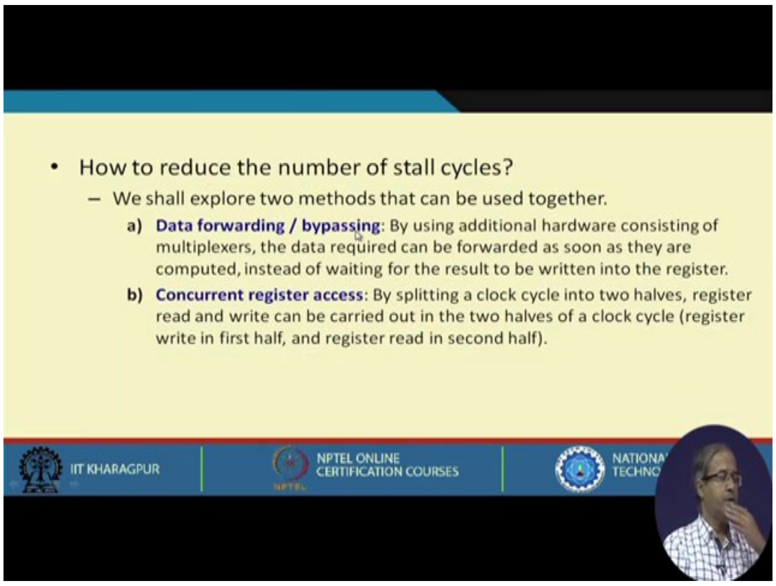


- Naïve solution by inserting stall cycles:
  - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
  - 3 clock cycles are wasted.

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

While decoding you find out that R2 is here and already the earlier instruction has R2 as target. So, three clock cycles are wasted.

(Refer Slide Time: 27:13)

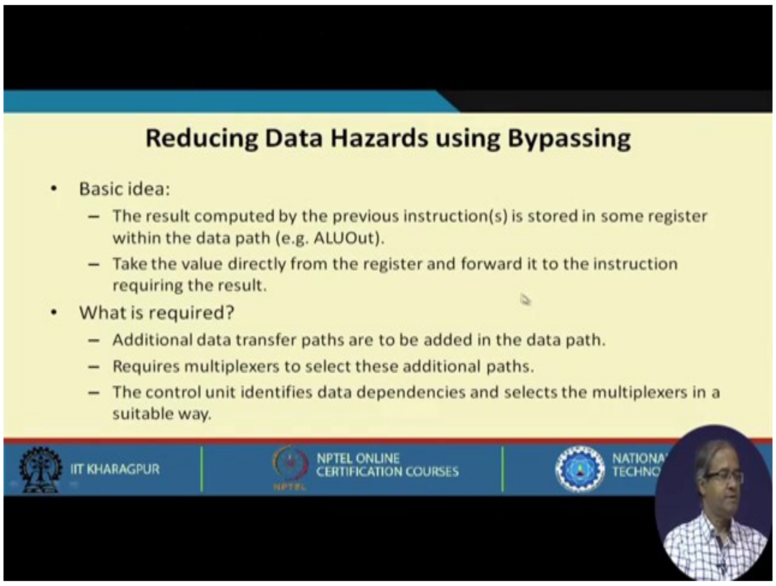


- How to reduce the number of stall cycles?
  - We shall explore two methods that can be used together.
    a) **Data forwarding / bypassing**: By using additional hardware consisting of multiplexers, the data required can be forwarded as soon as they are computed, instead of waiting for the result to 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).

But fortunately because of the simplicity of the instruction set we can reduce the number of stall cycles. How you can do that? We shall explore two methods; one is called data forwarding, and splitting a clock cycle will two halves, register writing is done in the first half and reading in the second half. The idea is very simple; see the first instruction which was generating the result R2, was writing the result in WB, but if you look at the instruction execution process, the value of the result was already calculated at the end of EX.

But the value is written into the register in WB. So, if we take the output of the ALU directly that is already calculated, and by using some additional MUX we forward that value to the next instruction. May be we will not have to wait for the WB stage, we may get the value earlier. This is called data forwarding or bypassing, by using additional hardware consisting of MUXes. The data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register file.

(Refer Slide Time: 29:08)

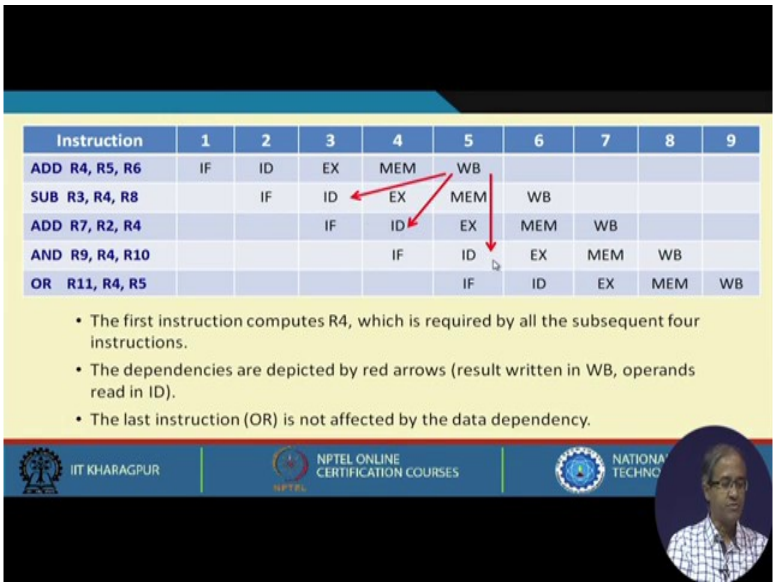


Reducing Data Hazards using Bypassing

- 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.

So, in bypassing the result computed by the previous instruction is stored in some register, it is the just output of the ALU. You take the value directly from there, do not wait till WB, and forward to the instruction that require the result. To do this you need some additional connections, data transfer paths and some additional MUXes. Your control circuit also becomes a little complex; it will have to analyze the source and the destination registers of consecutive instructions, and it will automatically activate or deactivate this kind of forwarding paths whenever required.

(Refer Slide Time: 30:10)



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

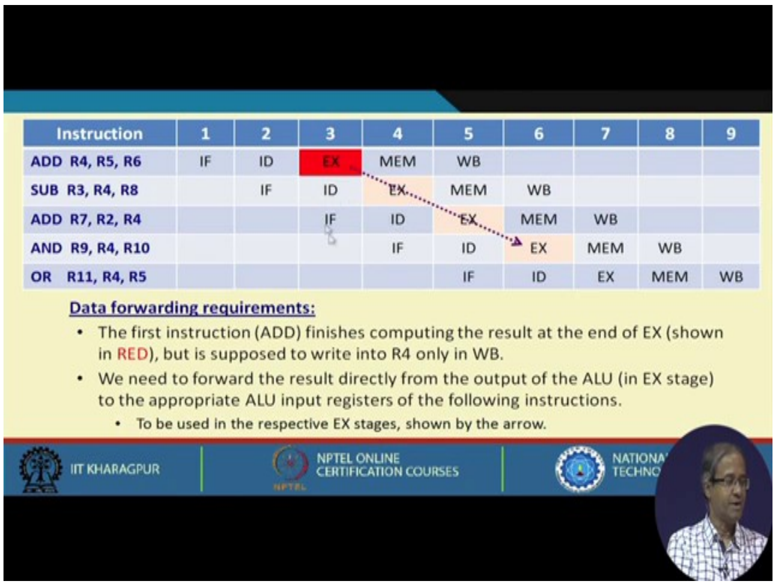
- The first instruction computes R4, which is required by all the subsequent four instructions.
- The dependencies are depicted by red arrows (result written in WB, operands read in ID).
- The last instruction (OR) is not affected by the data dependency.

Let us take an example here that shows lot of data dependency, like this ADD instruction generates R4, this R4 is used in all consecutive four instructions. In the normal case result will be written in WB, but these instructions are trying to read them here, here and here. So, first instruction computes R4 that is required by all subsequent three instructions; the dependencies are shown here, but the last instruction is using ID after WB. So, it is not affected. It is only the three consecutive instructions that need to be looked at.

We have already solve this by splitting the register access; we are saying writing is done in the first half of the clock cycle and reading is done in the second half of the clock cycle. So, this conflict is already resolved. Actually we are left with these two conflicts, but one thing you see it is possible to remove the two conflicts also. Why, because the first instruction is calculating R4 in the EX stage, the result is available here.

You need not have to wait till WB to be written into R4, you can directly take from here and forward it to the EX stage here, where this SUB will be requiring.

(Refer Slide Time: 32:11)



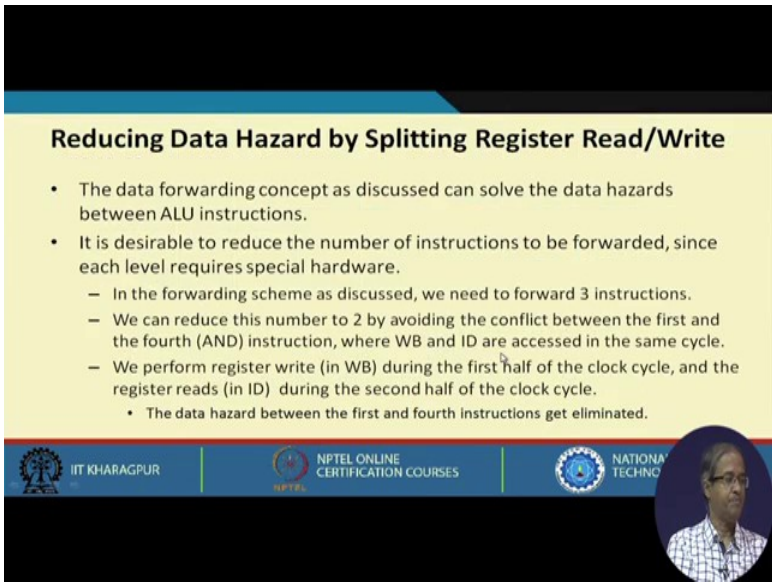
Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	EX	MEM	WB			
ADD R7, R2, R4			IF	ID	EX	MEM	WB		
AND R9, R4, R10				IF	ID	EX	MEM	WB	
OR R11, R4, R5					IF	ID	EX	MEM	WB

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.

Data forwarding means the value of R4 is actually getting calculated here, and you are forwarding the value like this. But as I have said you need to forward only to the two consecutive instructions, you do you really need beyond that. We need to forward the result directly from the output of the ALU in the EX stage, to the appropriate ALU register of the following instruction. When the next instruction enter the EX stage, this instruction will already be in the MEM stage. From there the result has to be fed back to the input by using some multiplexer and additional paths.

(Refer Slide Time: 33:06).

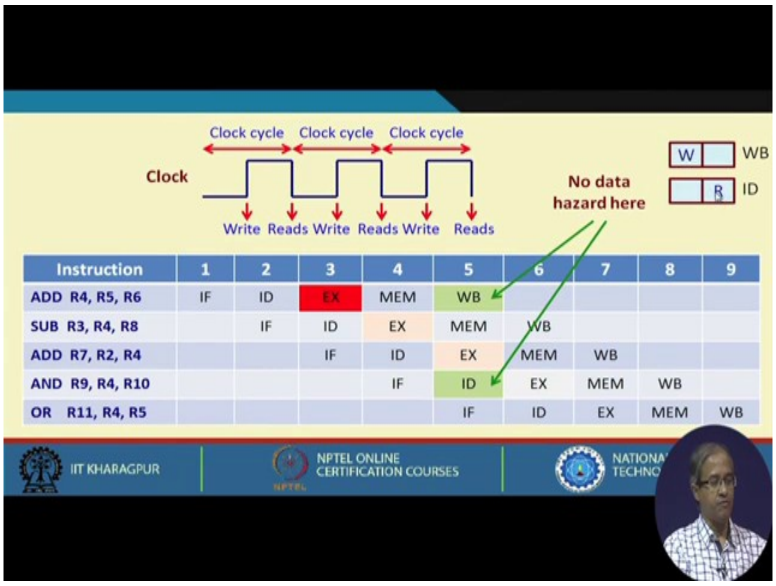


Reducing Data Hazard by Splitting Register Read/Write

- 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.

And this already we have set earlier. By splitting register read/write we have already avoided or reduced one dependency. In the naive forwarding there was the requirement of forwarding three instructions, but if you split register read/write we reduce it to two. This we have already discussed earlier in detail.

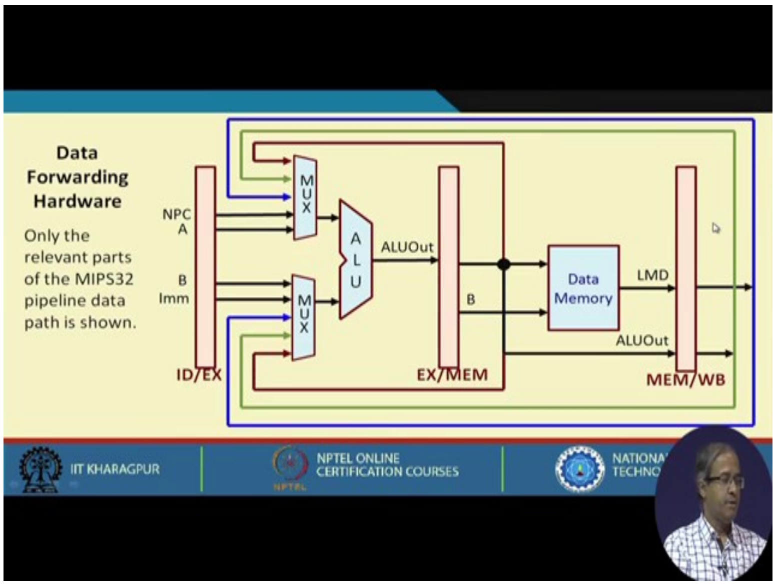
(Refer Slide Time: 33:37)



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

Because of this register splitting, this conflict has been avoided.

(Refer Slide Time: 34:05)



Data Forwarding Hardware

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

But for the others, this is the forwarding hardware we require. This is the EX stage, this is the ALU where the actual calculation is done. Normally the data input of the ALU is coming from the earlier stage, but because of the forwarding we use some additional connections from this ALUout from here. Similarly if there is a dependency of the second most instruction, the instruction will already be moved here. So, this ALUout is copied here, this value also is getting fed back here or here. Well here I am showing another feedback path also; this will have required for LOAD instruction.

But in the example you have not seen this yet, but this is the complete forwarding hardware. For LOAD instruction data will be read into the LMD, this LMD value is also forwarded. You see basically you need some additional inputs to the MUX; earlier it was 2:1 MUX now it has become a 5:1 MUX, this is the actually the forwarding hardware I am talking about. The first instructions that generates the result has come here, now that partial result is here, the next instruction has come here and is trying to use that result, you forward it from here or the instruction generating result has come here the next to next instruction has reached here. So, you forward it from here ALUout just like that. Here, all possible paths are shown. Either you forward it from ALUout here or for LOAD instruction from LMD.

With this we come to the end of this lecture. What we have seen here is the importance of handling and tackling hazards because they can cause a very significant degradation in performance. We have looked at structural hazards and mentioned how structural hazards can be avoided by replicating resources.

Then we looked at data hazards. Data hazards are important and apparently without anything you are requiring three stall cycles for at least the ALU instructions with data hazards. But what we saw is that using forwarding hardware and by splitting the register access we are able to totally eliminate data hazard related stalls for ALU instructions. In the next lecture we shall see what will happen if there is a data hazard for a LOAD followed by an ALU instruction, which uses the loaded value. There we shall see that it is not possible to totally eliminate the stall cycles, but we can try to reduce it again. This we shall discuss in the next lecture.

Thank you.