


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


Lecture – 56
Pipeline Hazards (Part 2)

In the last lecture we started our discussion on hazards in a pipeline. If you recall we talked about three kinds of hazards: structural hazard, data hazard and control hazard. We were discussing about the data hazards, and the examples that we took were for a case where there were data dependencies between ALU instructions. One of the ALU instructions was generating a result in a register, while the subsequent instructions were using the register value as inputs. We saw there was a problem that requires stalls, but we also came up with the solution. We proposed two things, first was some kind of a data forwarding hardware. The earlier instruction can compute the result in the EX stage, so that data value can be taken directly from the output of the EX stage, and through some multiplexers can be forwarded to the input of the following instructions. And the second thing that we talked about was that we are permitting something called split access to the register bank. The clock cycle is divided into two halves, during the first half we are allowing writes to happen; in the second half we are allowing reads to happen.

(Refer Slide Time: 02:11)



NPTEL ONLINE
CERTIFICATION COURSES



IIT KHARAGPUR



NIT MEGHALAYA

Lecture 56: PIPELINE HAZARDS (PART 2)

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

We continue with the discussion here. In this lecture we shall again we talking about data hazard to start with.

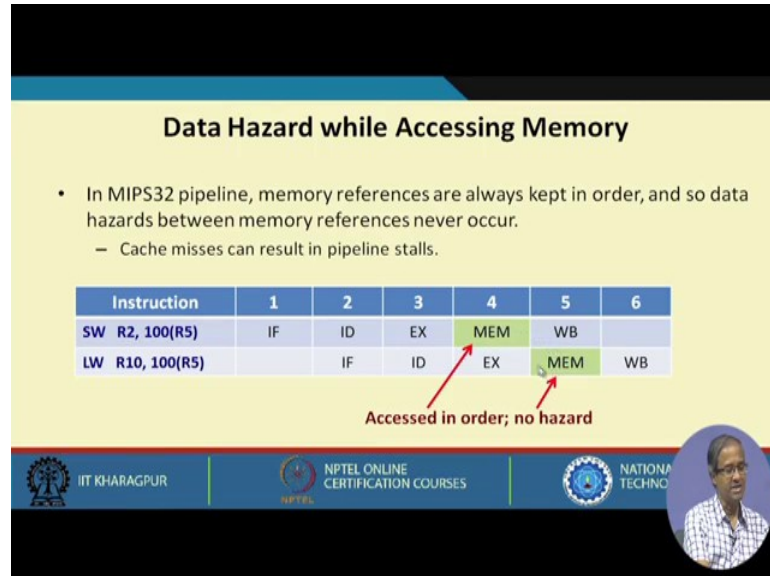
(Refer Slide Time: 02:18)

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.

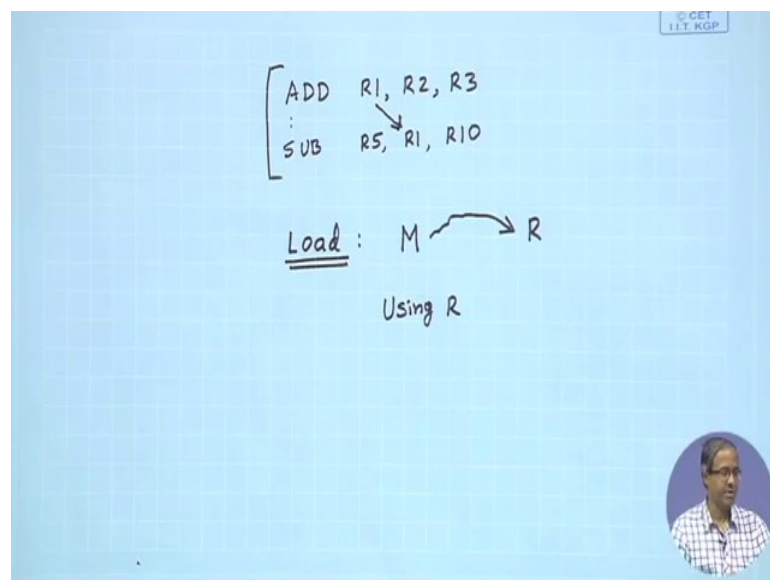
Instruction	1	2	3	4	5	6
SW R2, 100(R5)	IF	ID	EX	MEM	WB	
LW R10, 100(R5)		IF	ID	EX	MEM	WB

Accessed in order; no hazard



But now we will be considering the situation when the hazard is occurring while accessing memory.

(Refer Slide Time: 02:42)



Let us say this ADD was generating a result in register R1, and possibly a subsequent instruction was using the value of R1. Here there was a data dependency, now what we

are saying that these kind of data dependencies can also happen because of load instruction.

LOAD instruction means from the memory you are loading some value into some register, and then some subsequent instruction is using this register. This is what we mean by data hazard during memory access.

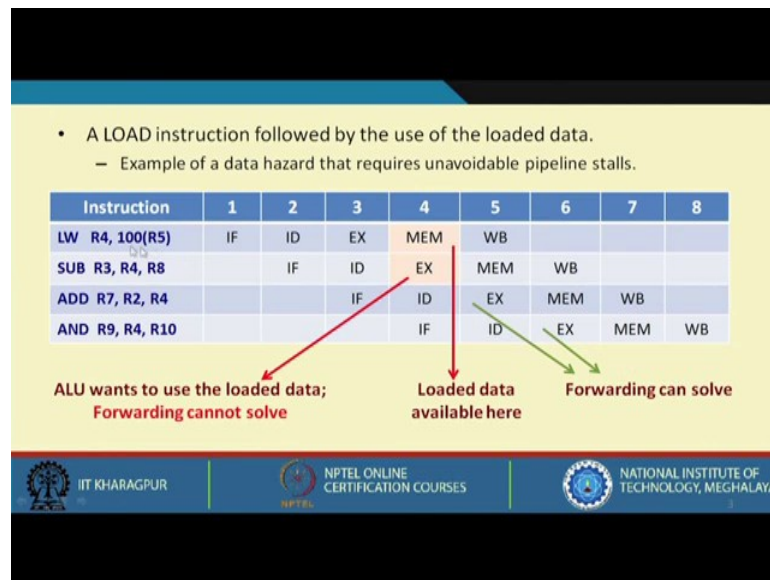
There are two situations we shall be looking at. The first situation is that, there is a STORE instruction followed by a LOAD instruction. Here the hazard or the data dependency is with respect to the memory location. The first instruction is writing some value, the content of register R2 into a memory location whose address is $R5 + 100$, while the second instruction is trying to load that same value from that same memory location, and load into some other register R10.

If you look at the MIPS32 pipeline structure you will see that all memory accesses can be either load or store. They take place only during the MEM stage; memory accesses do not take place or occur during the other stages in the pipeline. The previous instruction will reach the MEM stage earlier, the next instruction will reach the MEM stage one cycle later. So, that dependency is automatically maintained.

In this example the first instruction was using MEM. This store will occur here, load will occur here. So there is no conflict, however, one thing is missing of course, here if there is caches miss this thing we are ignoring here. If there is a cache miss, the store instruction can take longer, then possibly this kind of hazard may occur, but again this cache miss is a relatively infrequent event. So, we always try to design a memory hierarchy that can give you a hit ratio, which will about 98-99%.

The occurrence of cache misses is very rare. We ignore that occurrence for the time being. As I said the data are accessed in order.

(Refer Slide Time: 06:44)



Let us look at another situation where there is a LOAD instruction in the beginning. The load instruction is loading the value of a memory location $R5 + 100$ into a register R4, and the subsequent instructions are using R4. So, this is a load instruction followed by the use of the loaded data. Here you can observe one thing; the load instruction will be getting the data from the memory only at the end of the MEM stage.

But the next instruction SUB is requiring this R4, and the subtraction is supposed to take place during the EX stage. So, the value of R4 is required at the beginning of the EX only. This is not possible because the memory value loaded here it will be available only at the end, while the second instruction is trying to use that same value at the beginning of the EX.




This is a scenario where you have a data hazard that is unavoidable. If you implement forwarding hardware you cannot forward it, because you are trying to go back in time. The first instruction is loading the result only at the end of cycle 4, and the next instruction is requiring that value at the beginning of cycle 4. So, as I said the loaded data will be available here at the end of cycle 4, while the ALU for the second instruction wants to use the loaded data at the beginning of cycle 4. Data forwarding will not be able to solve this problem. You recall when there was a dependency between ALU instruction, it was possible to solve this problem using data forwarding because for the ALU instructions the result is computed in the EX stage only, not in the MEM stage.

But here we are talking about one time step ahead, only at the end of the MEM stage will our data be made available to us. This is one scenario where this stall cycle will be unavoidable, but for the following instructions you can use forwarding, because the loaded value is already available at the end of R4, from there you can simply forward it to this EX, and also you can forward it to this EX because it is ahead in time. So, by using the same forwarding hardware we can solve the problem for the subsequent instructions. The only problem is for a load followed by its immediate use, there has to be one stall cycle in between.

(Refer Slide Time: 10:14)

- 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

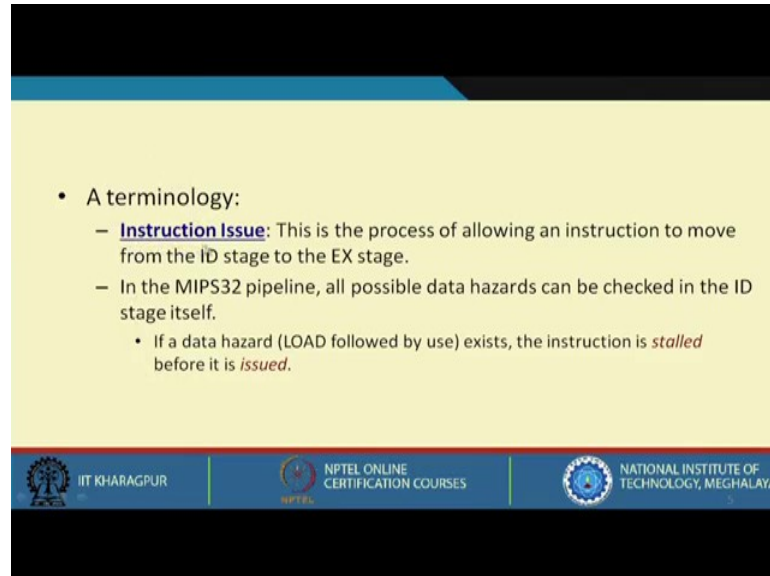
 IIT KHARAGPUR
  NPTEL ONLINE CERTIFICATION COURSES
  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Since we are not able to solve this problem using forwarding, we have to insert some kind of a stall cycle. What we do is that, there will be a special hardware that will be added to your decoding logic that will be called pipeline interlock. During the ID stage an instruction gets decoded. So, here only you will be able to know that your operands are R4 and R8; and already previous instruction is having R4 as the destination that is also known to the control unit. That hardware keeps track of that that, which register was the destination in the previous instruction and which are the registers that have been used in the current instruction.

If it finds that there is a conflict like that, there is a hazard, and then a stall will be inserted in the pipeline -- in this case a one cycle stall. The second instruction will be decoded, and find that it uses R4. So, it cannot start its EX here, it has to defer by one

cycle and it can start the EX here. When one instruction gets deferred all subsequent instructions will also get deferred.

(Refer Slide Time: 12:15)



The slide features a yellow background with a blue header and footer. The main content is a bulleted list defining 'Instruction Issue' and its implications in the MIPS32 pipeline. The footer contains logos for IIT Kharagpur, NPTEL, and the National Institute of Technology, Meghalaya.

- 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*.

Here we introduce a terminology called instruction issue.

You see the instruction that is operating on some data; they will typically be some ALU instruction, ADD, SUB like that. The actual operation is being carried out in the EX stage, before that in the ID stage you are decoding the instruction. You are trying to find out what the instruction type is, what are the input operands, and so on. When you are moving from ID to EX, it means that you are actually starting to execute the operation. It is during that phase you say that we are issuing the instruction.

When you say instruction issue, this is actually the process, when an instruction is moving from the ID stage to the EX stage. Now one good thing about the MIPS32 pipeline is because of the simplicity of the instructions, decoding is also very simple and the kinds of data hazards that are possible all can be detected at the end of the ID stage itself, because when you are decoding an instruction you have already know that this is an ADD instruction, R4 and R8 are input operands, and so on. You can check whether previous instruction is writing the value of R4 or R8; if so, there will be a hazard which will be detected.

So, it will know that at least one stall cycle is required. It will insert a stall cycle before it can be issued; before it can be moved to the EX stage.




(Refer Slide Time: 14:36)

- 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:

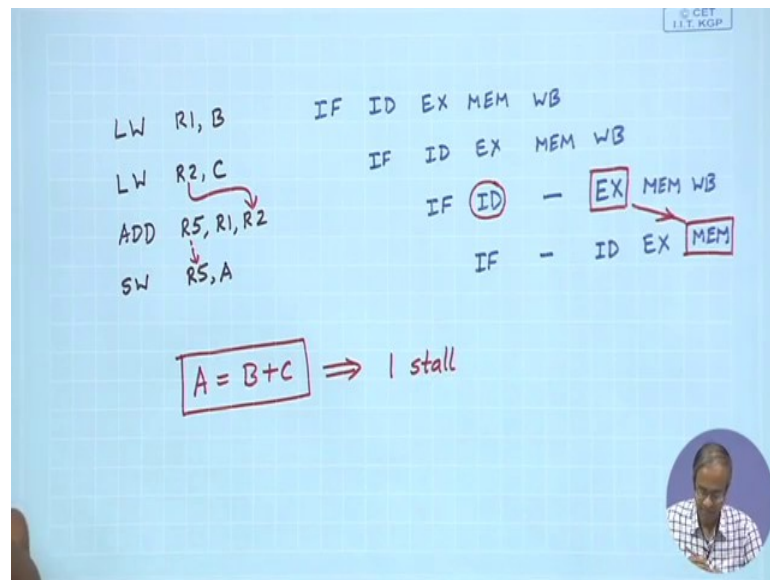
- Compiler tries to avoid generating code with a LOAD followed by an immediate use.

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

Let us look at a very common example that appears in typical programs. $A = B + C$ is just a representation of some kind of operation. MIPS32 instructions for this are written here. A straight forward implementation of this operation will be to load the value of B into some register, load the value of C into some register, add R1 and R2, put the result in R5, and finally, store R5 into A.

But here the second load and then add results in a data dependency, and there will be a hazard and there will a stall. This stall is unavoidable.

(Refer Slide Time: 16:09)



LW R1,B; the next instruction will be LW R2,C; then it is ADD R5,R1,R2 then there will be SW R5,A. Now you see here what we are doing the first load can proceed without any problem. For the second instruction also there is no dependency. But here there is a dependency between this loaded value of R2 and the use. So, when this ADD is fetched here and the ADD is decoded here, when you are decoding here itself your coming to know that this hazard is there. So, what we do? You insert a stall here and you issue the instruction; that means, you move it to the EX stage after this stall and so on.

So, only one stall will be required in this case.

(Refer Slide Time: 19:12)

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

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNICAL

Let us check another example because this will illustrate a few other things. Here there is not one, but two operations in sequence. When you translate this into MIPS code, this straightforward translation will be something like this.

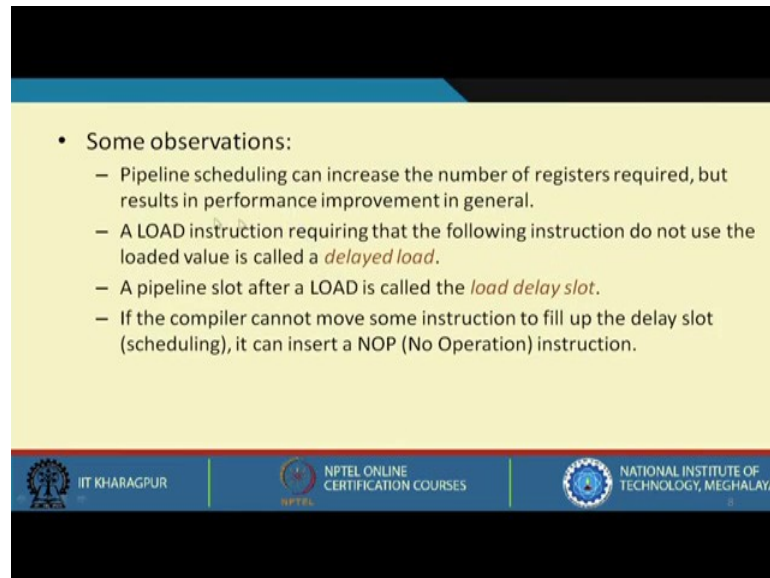
This is a straightforward implementation, but you see where are the problems? Here I find there is a load followed by a use; R2 is used here, here there is another load R2 followed by use. So, there will be a one stall cycle required here, and one stall cycle required here. There will be two load interlocks. If you run this code just like this the control unit will be inserting two stall cycles. So, the total time that we require to execute this code will be two clock cycles extra.

Now, let us do something here. Suppose we have lot of registers with us; so, what we do is something like this. We call this instruction scheduling and this modified code is called scheduled MIPS code. The first modification is that, we are not using the same registers. The first instruction is using R1 and R2, but for this second one we are using R3 and R4. This is a task that the compiler will be doing; the compiler will be moving some instructions around.

Now, what is the purpose of moving it here? You see that problematic data hazards have been eliminated. So, the problem case was a load followed by its immediate use. So, here I have inserted another load instruction in between, and so on.

The compiler can do this kind of code analysis and can move instructions around, these are very interesting problems for the compiler to solve, and if we do this then both the load interlocks are eliminated. This modified code will be able to run without any stall cycles; you can save two clock cycles in the execution of this code. This is the advantage of scheduling of the code.

(Refer Slide Time: 24:05)



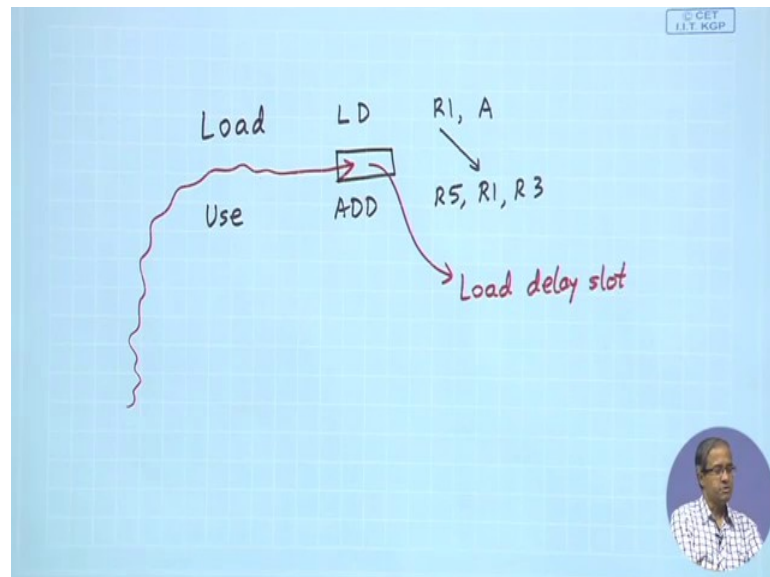
• Some observations:

- 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.

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

What we saw is that scheduling can improve performance, but in general it increases the number of registers required. In the previous case we had to use two additional registers R3 and R4.

(Refer Slide Time: 24:34)



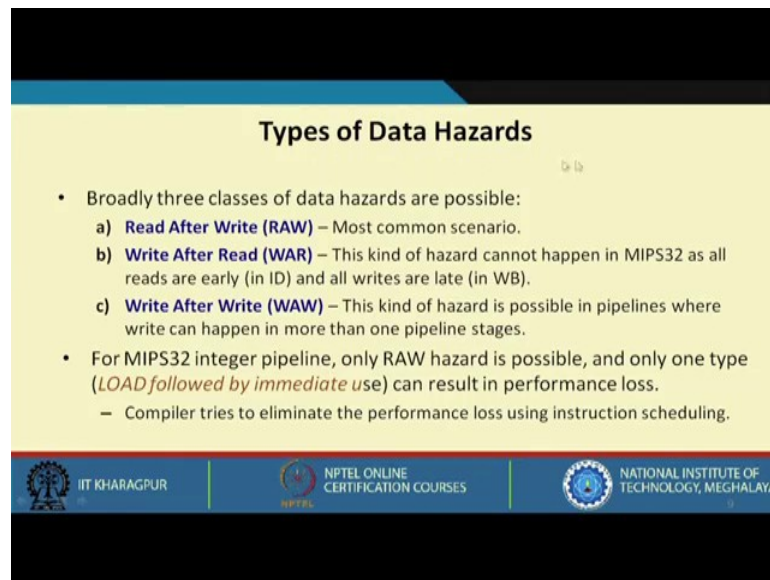
What I am saying whenever there is a load followed by its immediate use, as if there is a gap or slot coming here between the two instructions; this gap is called the load delay slot. If we do not do anything then the control circuit has to insert one stall cycle here.

But now the compiler knows that there is a load followed by its immediate use. So, it will try to move some other instruction from somewhere else into this delay slot. If it can do that then it can avoid that stall cycle. Coming back to this the load instruction that requires that the following instruction do not use the loaded value, we call it as delayed load and that slot is called delay slot or in this case load delay slot. As I had said the compiler will try to move instructions around and try to fill up this delay slot. If it cannot find any instruction to move, then it can insert a special NOP instruction, which does nothing.

The size of the code will increase, but there will be no hazard. So, the control unit becomes simpler; it does not have to check for any hazard here if the compiler takes the responsibility.

So, if you delegate the responsibility to the compiler and let the compiler check for the hazard situation, then you can solve this problem.

(Refer Slide Time: 27:39)



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 use*) can result in performance loss.
 - Compiler tries to eliminate the performance loss using instruction scheduling.

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

Generally speaking data hazards can be of three types, the first one is the one that you have seen already, read after write. Some instruction is writing a data somewhere register or memory, some other instructions is reading from there. There can be some other kind of data hazards also, write after read.

Reading takes place first writing takes place later; in MIPS32 this will never happen because reading of the registers take place always in the ID stage, and all writes take place in WB. So, it is never the case that you do the write before you do the read. Similarly for write after write, two instructions are writing, say the first instruction has written first, second instruction next, but the other thing can also happen. The first instruction is writing later, the second instruction is writing earlier. This will also not happen in the pipeline because all instructions take 5 cycles to complete.

Later on we shall see how we can extend the pipeline to handle multi cycle operations. There we will see that write after write hazard can also occur. So, in the MIPS32 integer pipeline only the first kind of hazard is possible and the only problem case which results in stall cycles is a load followed by immediate use. All others can be eliminated by forwarding, and we put lot of responsibility on the compiler.

(Refer Slide Time: 29:56)

(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 that an instruction is a branch, the next instruction has already been fetched.
 - We have to redo the fetch if it is a branch.



Now coming to control hazard; control hazards arise because there are branch instructions in a pipeline. Branch instruction means the next instruction to be executed can either be the sequentially next instruction or instruction from somewhere else. If the branch is actually taken it is a jump or a branch that is taken.

The next instruction to be executed will come from the target and not the next sequential instruction. This is the problem in a pipeline because the instructions are coming one by one; they will all be entering in the pipeline. Suddenly we find out that an early instruction was a branch it has to be taken. So, the following instructions all have to be ignored, they will have to be discarded and the new instruction fetch started from the target. When a branch is executed the value of the PC is normally changed to the old PC + 4.

But when a branch is taken it can be something else; the PC can be replaced by a target branch address, but that target branch address is normally not known till the end of the MEM cycle, which means that that you may require 3 stall cycles to take the decision that whether the branch is taken or not. By the time you take decision already three instructions have entered the pipe. So, there is a lot of loss in performance this is what meant by control hazard.

(Refer Slide Time: 31:45)

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 $CPI = 0.7 \times 1 + 0.3 \times 4 = 1.9$
Speed becomes almost half.

An example is shown here.

(Refer Slide Time: 33:07)

How to Reduce Branch Stall Penalty?

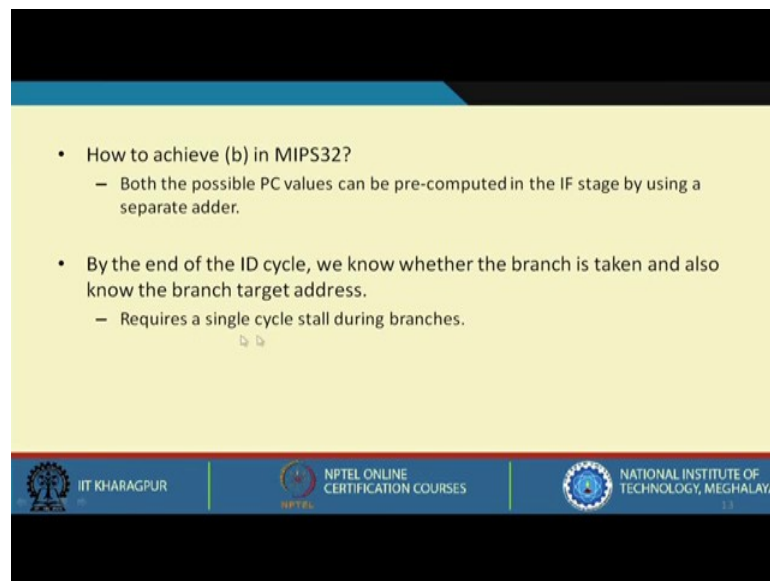
- The penalty can be reduced if both the following are achieved:
 - Determine whether the branch is taken earlier in the pipeline.
 - Compute the branch target address earlier in the pipeline.
- How to achieve (a) in MIPS32?
 - In MIPS32, the branches require testing a register for zero, or comparing 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.

There are many ways to reduce branch penalty. We have to tackle two sub-problems, determine whether the branch is taken or not, and what is the branch target address. Both of these have to be calculated early enough so that our stall cycles can be reduced. Now in MIPS32 because of the simplicity of the instructions, this is easier. For computing whether a branch is taken or not, we have to check whether the register is 0 or non-zero. Registers are already fetched in ID, and you can add a simple 0 comparator that hardly

takes any time; whenever you are fetching that register you also check whether it is 0 or non-zero.

The decision is already known at the end of ID itself. So, in MIPS32 the branches either require testing for 0 or comparing two registers because you are fetching all the registers in ID. So, you have to add some special comparison logic.

(Refer Slide Time: 34:33)



The slide contains the following text:

- 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.

At the bottom of the slide, there are three logos: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA.

And the second one is to compute the branch target address. This you can also calculate earlier by using a separate adder. At the end of the ID cycle you know now whether the branch is taken or not taken, and also you know the branch target address. You do not need to wait for three cycles, you need to wait for a single cycle only.

(Refer Slide Time: 35:05)

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

Earlier in the new approach we have said that you have to wait till MEM, but now we are saying that because of the simplicity of the MIPS, at the end of ID we will come to know that whether the branch is taken and also what is the address. We can start fetching after a maximum of one stall cycle. This can be ignored and you can start fetching from here. So, one stall cycle will be required. We shall see later how this branch penalty can be further reduced in general.

We have come to the end of this lecture. In the next lecture we shall continue with this discussion on control hazard because there are many ways to reduce the penalty due to branch instructions. We shall discuss these things in our next lecture.

Thank you.