

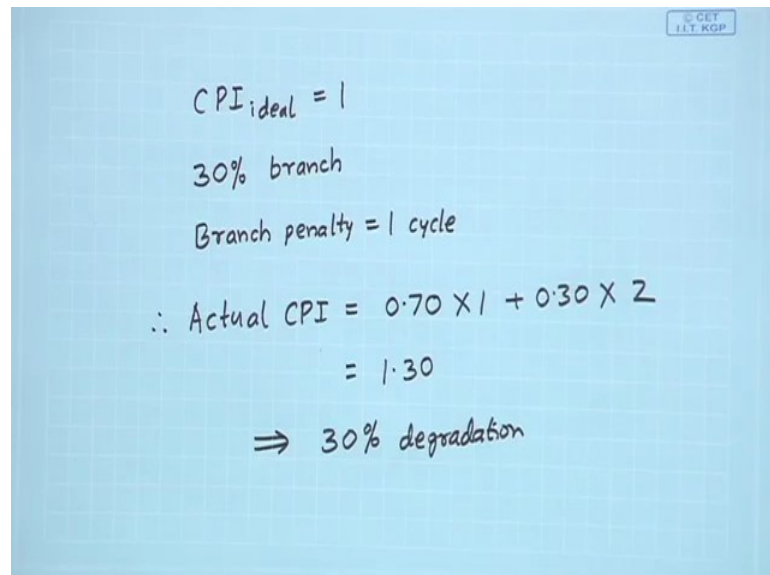
**Computer Architecture and Organization**  
**Prof. Indranil Sengupta**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 57**

**Pipeline Hazards (Part 3)**

In the last lecture we were looking into control hazards and associated problems. We had seen that for MIPS32 pipeline implementation, we can reduce the branch penalty to 1 cycle. Because we mentioned that at the end of the ID stage we can know both the outcome of a branch whether it is taken or not taken, and also the branch target address. So, in the worst case we have to incur a penalty of 1 cycle. You should say that 1 cycle is fine for load followed by use, we have 1 cycle here also. Let us make a simple calculation.

(Refer Slide Time: 01:09)



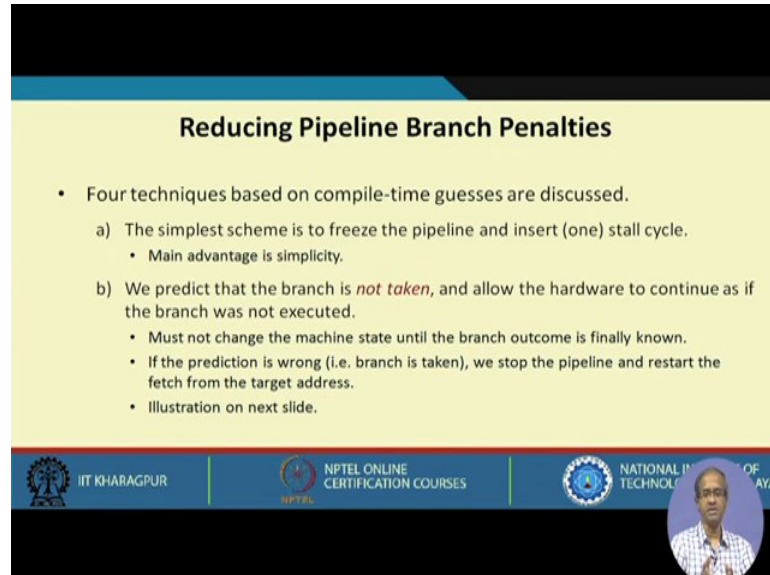
© CET  
I.I.T. KGP

$$\begin{aligned} \text{CPI}_{\text{ideal}} &= 1 \\ &30\% \text{ branch} \\ \text{Branch penalty} &= 1 \text{ cycle} \\ \therefore \text{Actual CPI} &= 0.70 \times 1 + 0.30 \times 2 \\ &= 1.30 \\ &\Rightarrow 30\% \text{ degradation} \end{aligned}$$

We saw in the example we took in the last lecture that the ideal CPI was 1, 30% of branch, but here we are saying that branch penalty is 1 cycle. So, what will be the actual CPI? In 70% of the case when there is no branch CPI will be 1, and for 30% of the cases there will be 1 cycle penalty. So, CPI will be 1.30. So, you see still you have a 30% degradation that is still quite substantial.

Let us see how we can reduce this further. This is the topic of our discussion in this lecture. Our specific target here will be to reduce the pipeline branch penalties.

(Refer Slide Time: 02:14)



**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.

The slide footer contains logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY DELHI, along with a small portrait of a man in a blue shirt.

We start by discussing four broad techniques. First is a very naive approach; we have seen that a branch will incur 1 cycle penalty. So, whenever there is a branch you freeze the pipeline, insert one stall cycle; this is the simplest approach. The main advantage is simplicity, but as I have shown in the example the overhead can be significantly high.

The other approaches are based on some kind of prediction. The second approach says we predict that the branch is not taken, and you allow the hardware to continue as if the branch is not executed at all. That means, the next instruction is fetched, it is executed and we continue as if nothing has happened, until the branch outcome is actually known to us. When the branch outcome is actually known to us, then we can know whether our prediction was right or wrong. If we see that our prediction was right, we do not do anything. The next instruction that already entered into the pipe, which was already executing, let it continue to execute --- we do not incur any stall cycle for that. But only if we find that our decision was wrong, it was actually a taken branch we are assuming not taken, we have to stop that, insert a stall, and fetch the new instruction from the target.


(Refer Slide Time: 04:15)

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.

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

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY KHARAGPUR

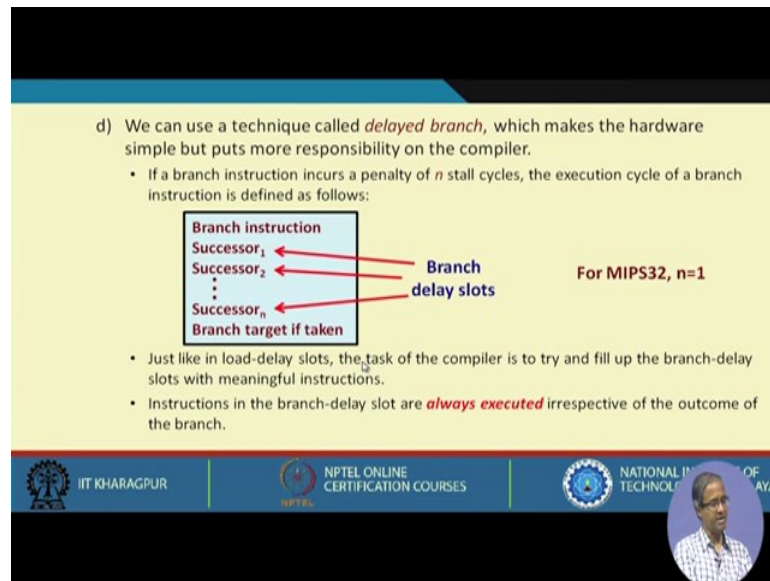


Let us take an example. If it is a not taken branch then there is no penalty; that means, it is a branch instruction, but the branch is not taken. So, the sequentially following instruction will be executing after that. The hardware will be assuming that the branch is not taken, and it will fetch the consecutive instruction without any stall. For such cases there will be no penalty, but if the branch is actually taken, then at the end of the ID stage you will come to know that your prediction was wrong, because it is here you come to know that your branch is taken or not taken by decoding the registers. If it is wrong then you will be incurring a 1-cycle penalty. Here for some of the cases there will be no penalty, for some cases there will be one cycle penalty.

The third approach is we predict that the branch is taken, just the reverse. We assume that the branch is always taken. But unfortunately for MIPS32 you will see that this prediction does not help because if we predict that the branch is taken; that means, you will always be fetching the next instruction from the target address, and the target address is known only at the end of ID. So, only after ID you can start the fetch. So, anyway this 1 cycle will get lost. Irrespective of whether it is taken or not taken branch for MIPS32, this 1 cycle penalty will always be there.

As I had said for MIPS32 we know the branch outcome and the target address both together. So, there is no advantage in this approach because in both cases there is 1 cycle penalty; for more complex instruction machines may be this will help.

(Refer Slide Time: 06:52)



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:


Branch instruction  
Successor<sub>1</sub>  
Successor<sub>2</sub>  
⋮  
Successor<sub>n</sub>  
Branch target if taken

Branch delay slots

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.

Logos: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, NATIONAL INSTITUTE OF TECHNOLOGY KANPUR



In such complex instructions, possibly the branch address is known earlier, but whether it is a taken or non taken branch is known much later in the instruction execution cycle. For those kind of instructions this strategy will work better, but not for MIPS.


The last approach is called delayed branch. It says it says that there is a branch instruction, let that branch execute -- it can be a taken branch or not taken branch. The assumption is that the slots that are following that branch instruction where you are normally inserting stall cycles, for MIPS there was 1 stall cycle. We call it as a branch delay slot. What we are saying that the instruction that is there in the delay slot, i.e. the instruction we fetch after the branch, will always be executed and the compiler knows that irrespective of the branch is taken or not taken. So, the compiler will try to put some instruction in the delay slot, which is supposed to be executed every time the branch is executed, irrespective of it is a taken or non taken.

This is called delayed branch. Here we are making the hardware simple, but we are putting all the responsibility on the compiler. In general a branch instruction can have  $n$  stall penalties, but for MIPS it is only one. In general I am showing  $n$ , after this  $n$  penalty the next target address will be known. The next instruction can be fetched after that. These  $n$  successor instructions are called branch delay slots; for MIPS it is 1. The compiler will try to move instructions around and try to fill them up, , these instructions are always executed irrespective of the outcome of the branch, whether you take it or do not take it.

(Refer Slide Time: 09:13)

**Some Examples**

<pre>ADD R3,R4,R5 BEQZ R2,L DELAY SLOT ... L:</pre>	<pre>L: ADD R1,R2,R8 SUB R3,R4,R5 BEQZ R3,L DELAY SLOT</pre>	<pre>ADD R1,R2,R8 BEQZ R3,L DELAY SLOT SUB R3,R4,R5 L:</pre>
↓	↓	↓
<pre>BEQZ R2,L ADD R3,R4,R5 ... L:</pre>	<pre>L: SUB R3,R4,R5 BEQZ R3,L ADD R1,R2,R8</pre>	<pre>ADD R1,R2,R8 BEQZ R3,L SUB R3,R4,R5 L:</pre>




Let us take some examples. Suppose I have an ADD instruction followed by a BEQZ, and there is a delay slot after that. You see the ADD instruction is executed before branch always. There is no harm if you move this ADD to the delay slot because whenever there is a branch, the delay slot will also be executed. So, we are not wasting the delay slot, rather we are moving a useful instruction. Whenever the branch is taken or not taken, ADD will always be executed.

The other example is a little complex, and is illustrated here. Similarly, the third example.

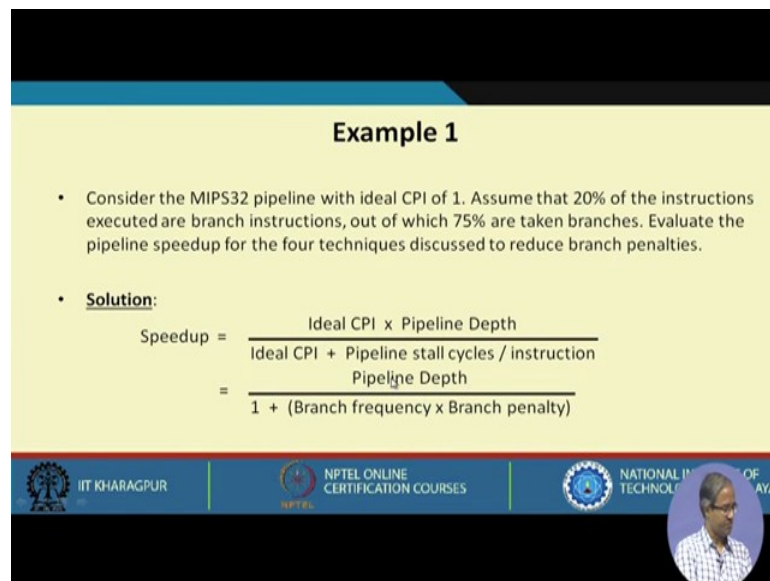
(Refer Slide Time: 12:02)

- 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.



For delayed branches there are some other difficulties, like you will be having multiple PCs, in fact, it will be  $n + 1$ . This means other than the target address, all the instruction in the delay slot also need to be saved if there is interrupt in between. So, there are multiple values of the PC that need to be saved, because the PC of the branch target and the PC of the delay slots are not sequential.

(Refer Slide Time: 12:47)



**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:**

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \frac{\text{Pipeline stall cycles / instruction}}{\text{Pipeline Depth}}}$$
$$= \frac{1}{1 + (\text{Branch frequency} \times \text{Branch penalty})}$$

The slide footer includes logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY DELHI, along with a small portrait of a man in a blue shirt.

Let us take an example. Consider pipeline with a ideally CPI of 1, let us say 20% of the instructions are branch and out of them 70% are taken, and the remaining 25% are not taken.

Using the four strategies we discussed let us evaluate the speedup. Speedup will be calculating using this formula, this ideal CPI multiplied by pipeline depth divide by ideal CPI plus stall cycles per instruction.

(Refer Slide Time: 13:40)

(a) Stall pipeline  
Branch penalty = 1  
Speedup =  $5 / (1 + 0.20 \times 1)$   
= 4.17

(b) Predict not taken  
Branch penalty = 1  
Speedup =  $5 / (1 + (0.20 \times 0.75))$   
= 4.35

(c) Predict taken  
Branch penalty = 1  
Speedup =  $5 / (1 + 0.20 \times 1)$   
= 4.17

(d) Delayed branch  
Branch penalty = 0.5  
Speedup =  $5 / (1 + (0.20 \times 0.5))$   
= 4.55

The slide footer includes logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA, along with a small portrait of a man.

Calculations for the four cases, namely, (a) stall pipeline, (b) predict not taken, (c) predict taken, and (d) delayed branch are shown.

For the last strategy I am assuming that there is 50% probability that the compiler will be able to fill up the delay slot. So, I am multiplying this by 0.5.

Now interrupts pose a more difficult problem in a pipeline. Let us see in the MIPS 5-stage integer pipeline whenever interrupt comes what are the issues.

(Refer Slide Time: 15:13)

### Dealing with Interrupts in MIPS32

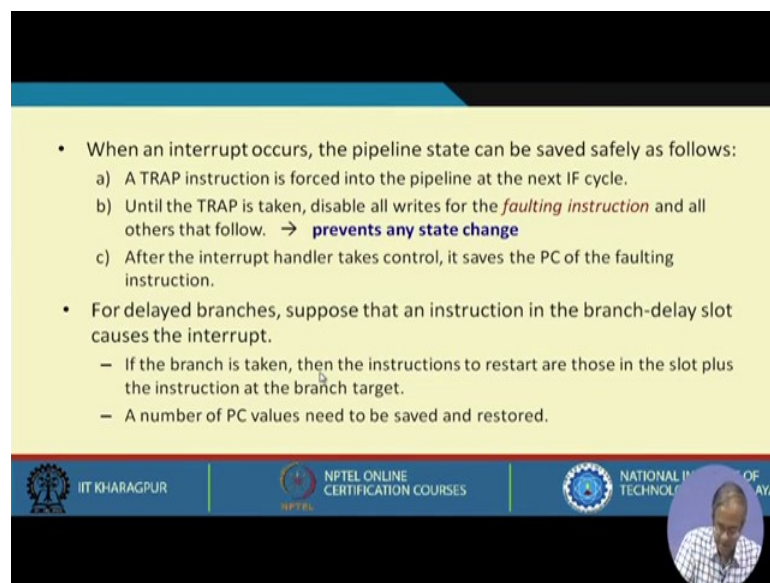
- Interrupts complicate the design of an instruction pipeline.
  - Overlapping of instruction executions make it more difficult to decide whether an instruction can safely change the state of the machine.
  - Some interrupts can force the machine to abort the instruction before it is completed (e.g.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

The slide footer includes logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA, along with the number 10.

Interrupts can complicate the design of the pipeline. Overlapping of instruction execution makes it difficult to decide whether an instruction can modify something, when I say safely change the state it means either change the value of some register or something. Suppose an instruction changes the value of a register, later it is found that there is an interrupt that instruction has to be withdrawn, but already it has modified the register.


So, the instruction should not change the state of the machine before it is known that the interrupt has occurred or not. Some interrupts can force the machine to stop the instruction before it is completed, like page fault. Whenever you are trying to fetch an instruction that is not there in memory, it has to be fetched from disk; this is the example of a page fault. Under those cases the instruction has to be restarted. After the requested memory word is brought into memory, we will again execute that instruction. So, such interrupts are more difficult. The most difficult interrupts have the properties that they occur in between an instruction and they have to be restarted.

(Refer Slide Time: 17:09)



The slide contains a list of bullet points describing interrupt handling strategies. The first bullet point states that when an interrupt occurs, the pipeline state can be saved safely as follows, with three sub-points: 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. The second bullet point addresses delayed branches, stating that if an instruction in the branch-delay slot causes the interrupt, then if the branch is taken, the instructions to restart are those in the slot plus the instruction at the branch target, and a number of PC values need to be saved and restored.

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

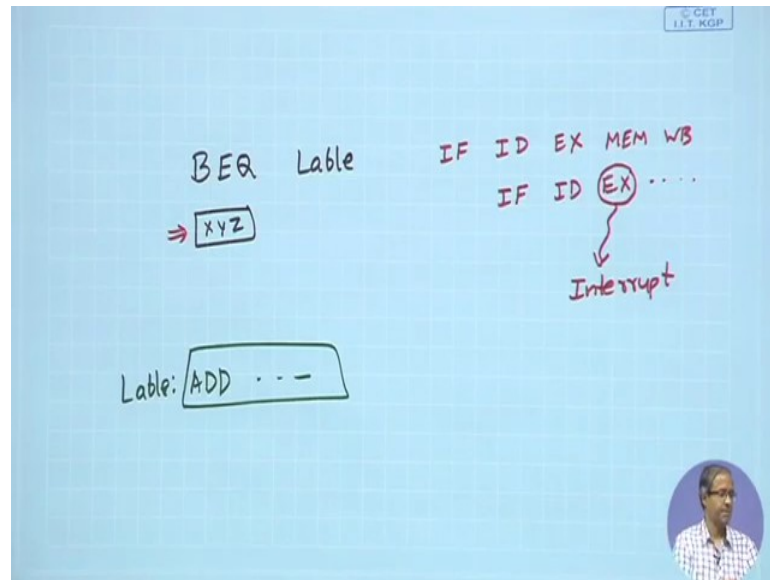


One strategy for interrupt handling can be like this. Whenever an interrupt occurs it can occur in any stage, in IF ID EX, etc. A special TRAP instruction or some kind of a flag is forced into the pipeline at the next instruction fetch cycle, which will indicate that an interrupt has occurred. The control unit will know that this TRAP instruction was inserted, and will not allow any writes to occur. Not only for the instruction that generates the exception at the interrupt, but for all the instruction which follow it and this will continue until the TRAP reaches the WB stage.



Therefore delayed branches it is a difficult condition, where the instruction in the branch delay slot may have cause the interrupt. There was an instruction that is presence in the delay slot,

(Refer Slide Time: 18:56)

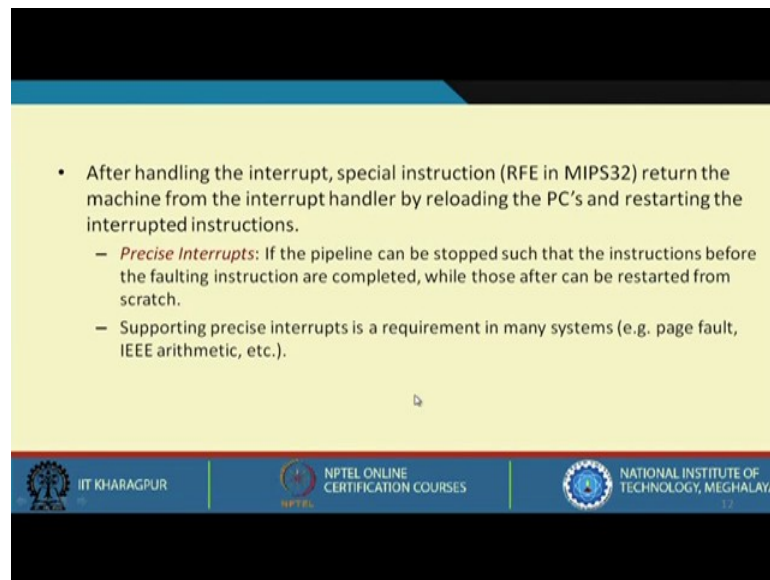


that can cause the interrupt. So, if the branch is taken, and there was a branch delay slot. There are some instructions here, say xyz, which has generated at interrupt. In the pipeline this branch instruction was already there. It was fetched, it was decoded, it was executed, it was supposed to do MEM and WB, and the next instruction is supposed to also execute along with it. So, xyz should also be fetched here, decoded here, executed here, and so on. What I am saying is that suppose this instruction during the EX stage generates an interrupt. We will have to stop everything not only this, but also this branch instruction. And when you come back you will have to not only restart this xyz, but also the instruction that is here.

So, multiple PC values have to be saved; the PC of this xyz, and also the PC of this. This is what is mentioned here. The instructions restarted are those in this slot plus the instruction at the branch target that requires a number of PC values to be saved and of course, restored. When the interrupt is handled after that there is a special instruction in MIPS32 called return from exception RFE, that will be reloading the PCs automatically and restart the interrupted instructions. So, for this kind of delayed branch kind of machines the RFE instruction has to do a lot of things. It will have to reload multiple PCs.

Let us define precise interrupts. Let us say an interrupt is occurred. If it is possible for the control unit to stop the pipeline such that all instructions before the instruction that generated the interrupt can complete, and all instructions that follow will wait they will be restarted later, then we say it is a precise interrupt.

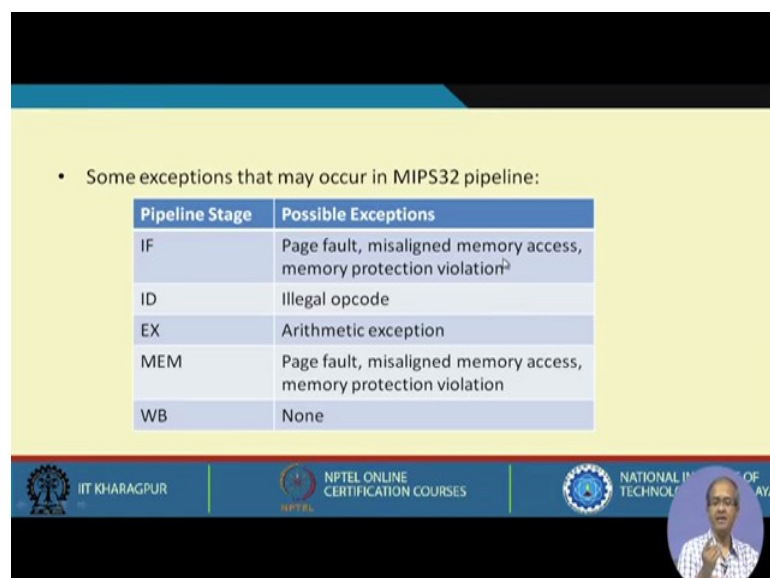
(Refer Slide Time: 22:00)



- 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.).

If you see the definition, if the pipeline can be stopped such that the instruction before the faulting instruction are completed, while those after can be restarted from scratch then we say it is a precise interrupt. Well there are cases like page faults where this precise interrupt is a necessity.

(Refer Slide Time: 22:33).



- Some exceptions that may occur in MIPS32 pipeline:

Pipeline Stage	Possible Exceptions
IF	Page fault, misaligned memory access, memory protection violation <sup>2</sup>
ID	Illegal opcode
EX	Arithmetic exception
MEM	Page fault, misaligned memory access, memory protection violation
WB	None

The designers of the pipeline spent some effort to ensure that interrupt handling is precise. So, what are the kinds of interrupts that can be generated in the five stages? They are shown in the table.

(Refer Slide Time: 23:39)

• 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.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY Kharagpur

Take this example. For ID you can get illegal opcode, for EX you can get its a divide by 0 some arithmetic exception, MEM can generate page fault, misaligned access memory, protection violation, and WB there is nothing. Multiple interrupts may occur in the same cycle. Let us say a load instruction followed by add is here. The load instruction can generate a page fault during MEM, while this add can generate an arithmetic exception during EX. So, two interrupts are being generated in the same clock cycle.





One solution is that if this kind of a thing happens, you ignore the second instruction interrupt and only handle the first one and again restart. If the second instruction (add) generated an interrupt, it will generate it again. We deal only with the page fault and restart the execution; the second interrupt will occur again and will be handled at that time.

(Refer Slide Time: 24:26)

- 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.



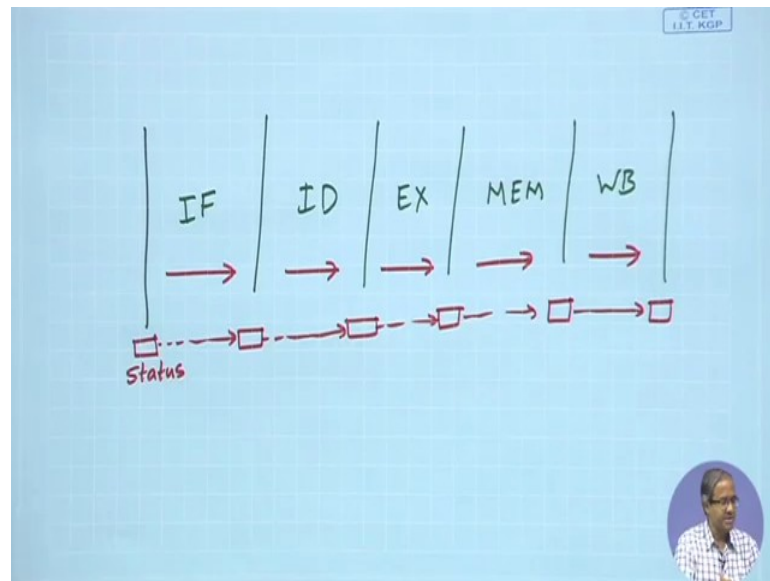
This is one solution you can think of. The second difficulty is that interrupts may appear out of order. Let us take another example. The first load instruction may be having a page fault here in MEM, and this add instruction can have page fault in IF. So, it is out of order. That means, the earlier instruction is generating interrupt later, later instruction is generating interrupt earlier. A possible solution to this we will be just discussing briefly. The solution is the hardware will post each interrupt in a status vector.

(Refer Slide Time: 25:08)

- 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.



(Refer Slide Time: 25:23)



When the instructions move from one stage to the next, there is also a special status register that is part of the inter-stage latches, this will also move from one stage to the next.

This status vector is carried along with instruction as it moves through the pipe, and in the status vector you set a bit indicating that there is an interrupt, and also the type of the interrupt. You do not do anything here, you let it move and when it reaches WB only then you process the interrupt. If you allow it to move till WB when it reaches here, you see that what is the type of the interrupt then you handle that interrupt.

So, when the instruction reaches WB, the interrupt status vector is checked and handled, but if you do it like this then preciseness of the interrupt is guaranteed because the first instruction will be reaching WB earlier, the following instruction will be reaching WB later. So, the first one to reach will be the earlier instruction. Maybe the interrupts are generated out of order, but the first instruction will always reach WB earlier than the next instruction. So, the interrupt for the first instruction will reach WB earlier this is the idea. So, this is what we mentioned here that interrupt handling will be carried out in precise order and with this we come to the end of this lecture.

In the next lecture we shall be discussing some more methods to improve the control hazard handling.

Thank you.