

High Performance Computing
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

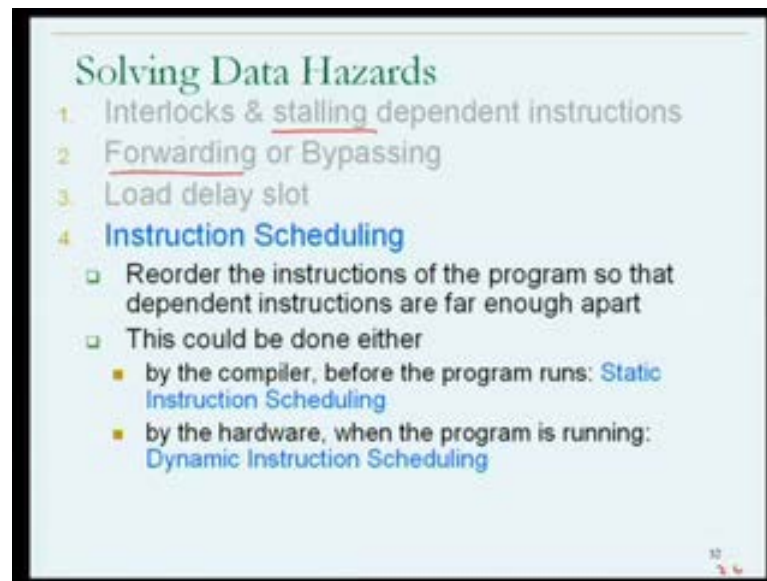
Module No. # 05

Lecture No. # 24

Welcome to lecture number twenty four of the course on high performance computing. We are looking at the idea of pipeline processor and are currently trying to understand the different kinds of hazards that may have been taken into account in the design of the processor though hazards are situations where instructions cannot proceed at the maximum throughput rate of the processor. Recall that pipeline processors try to get high performance by having the maximum possible throughput rate at which instructions are executed.

So, we are currently looking at the problem of data hazards. Let me just remind you that we have looked at few possible solutions to the data hazard problem. One of them was that the pipeline could be built to include pieces of hardware called interlocks. They check for data hazards and when a data hazard is detected, the hardware would cause the dependent instruction to be stalled and what we mean by stalling an instruction is, preventing it from proceeding through the pipeline as per schedule.

(Refer Slide Time: 01:04)



Another possibility we saw was the idea of forwarding and the idea here was if the result that the dependent instruction is awaiting is available, may be not in the register file but, somewhere else in the processor, then, if it can be provided through short circuit path to the point where the dependent instruction requires it then, there may be no need for the delay of stalling the instruction. And, we look at the special case for load instructions called the load delay slot.


Now, we are going to look at 4th possibility and that is the possibility of scheduling the instructions. In other words, reordering the instructions so that the dependent instructions are far enough apart so that the hazardous situation is resolved and the scheduling of instructions could either be done by the compiler; in other words, before the program runs.

In other words, if you looked at the executable file the **a dot out**, you would find the instructions in their rescheduled order; not the order which had the problem with the hazard. The alternative would be and we look at this only briefly that the hardware could be designed, the pipeline processor hardware could be designed. To do this rescheduling, the reordering of instructions dynamically, we will start by looking at the idea of static instruction scheduling.

(Refer Slide Time: 02:33)

Static Instruction Scheduling

- Reorder the instructions of the program to eliminate data hazards ...
 - or in general to reduce the execution time of the program
- Reordering must be safe
 - should not change the meaning of the program
- Two instructions can be exchanged if they are independent of each other




So, the idea here is that the instructions of the program could be reordered to eliminate the data hazards and we realize that arbitrary reordering of instructions may not be safe and that the reordering must be done in such a way that the meaning of the program is not changed. Why? If it is determined the two instructions are essentially independent of each other then, those two instructions could be interchanged and that, that would form a basis for trying to do the reordering of instructions.

(Refer Slide Time: 03:04)

Example: Static Instruction Scheduling

Program fragment:	Scheduling:
<pre>LW R3, 0(R1) ADDI R5, R3, 1 ADD R2, R2, R3 LW R13, 0(R11) ADD R12, R13, R3</pre>	<pre>LW R3, 0(R1) LW R13, 0(R11) ADDI R5, R3, 1 ADD R2, R2, R3 ADD R12, R13, R3</pre>
2 stalls	0 stalls



Now, we will understand static instruction scheduling through an example and here is the code example. Now, you will recognize the MIPS instructions in this code fragment. What I mean by a code fragment is that clearly, this is not a complete program in itself; it is just an extract from a program and we are looking at an extract of interest to us. So, you will notice that it starts with loading the contents of some memory location into register R 3. Subsequently, the value in that register is incremented by 1 result goes into R 5 then the same value is involved in an add instruction R 3 being the value I am referring to after this as a load word instruction and another add instruction.

Now, we first need to identify any problems with this piece of code in order to think it to consider what kind of reordering might be. Reordering of instructions might be useful. Currently, if you look at this piece of code and we have to bear in mind that there is this concept of the load delay slot. We observe that there is potentially a problem as far as the value R 3 being loaded and it being used in the... immediately following instruction. So, we suspect that if this was a processor which used interlocks then it would possibly have to stall the add I instruction by one cycle. So, I have inserted that comment that in the absence of some form of rescheduling of this code there would be the need for one cycle 1, cycle delay between the load word instruction going through the pipeline and the add I instruction going through the pipeline. Hence, my notation of the need for one stall in between.

Now, that is the primary problem with this piece of code. **we will** You will notice that there are others for example, in the second load word instruction which loads into register R 13 the immediately following instruction once again uses R 13 as a source operand. Therefore, there is a need to stall the pipeline or to stall the add instruction for at least one cycle before the value will safely be available in R 13 in register R 13 or before the value can be by passed appropriately. So, with this piece of code then we have these two problems as marked by the two red arrows and the notation of one stall.

Now, the question is if I wanted to reschedule the instructions, this is the existing order of the instructions. I need to try to do some reordering of the instructions in such a way that the problems disappear over the need for these two stall cycles is removed and I need to try to do this by changing the order of instructions.

Now, way in looking at the first two instructions very clearly there was no question of interchanging them because they the second instruction uses R 3 which is written by the first instruction and we could look similarly, at many of the instructions to determine which of them could be reordered.

But, let me just quickly cut through the analysis and point out that if I was to move the load word R 13 instruction up from its location near the bottom of the program, just to be over here, I have now moved it up there. What I have achieved? I have caused the need for the stall between the first load word instruction and the add I instruction to be eliminated. Now, the load word instruction which loads into R 13 and the first instruction that I am sorry which reads into R 3 and the add I instruction which is the first instruction, reduces the load value are separated by one instruction the load word R 13 instruction hence there is no need for that stall cycle.

By the same token I have separated the load word 13 instruction from the first instruction that uses the loaded value by at least two instructions and therefore, I have eliminated the need for the second stall cycle. Therefore, this movement of the load word R 13 instruction actually solves the problem. But, prior to doing that we do need to determine whether it is safe to interchange or to reorder these instructions in such a way that the load word R 13 instruction moves from its location towards the end of this code fragment up to the location which is where I had moved it.

So essentially, I need to determine that the load word R 13 instruction is independent of the two instructions which it is now going to precede and looking at these. For example, I look at the load word R 13 instruction; I notice that this instruction uses R 11 and modifies R 13. Whereas, the add instruction uses R 2 and R 3; in other words they have no registers in common.

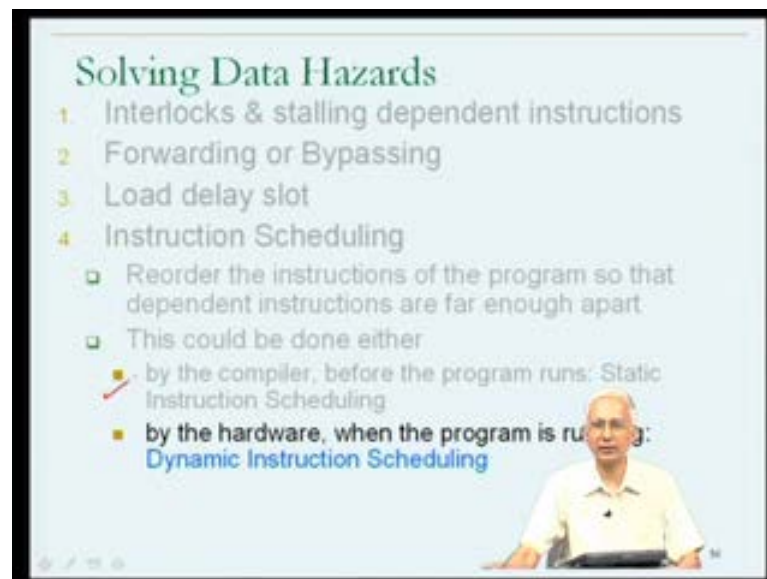
I have further, I know that these two instructions have no side effects. They do not have any implicit operands like R 31 and therefore, with some confidence I can say that the load word instruction is independent of the add R 2 instruction. I still have to check whether the load word R 13 instruction is independent of the add I instruction.

Once again, the same reasoning tells me that they are independent. The load word instructions uses registers R 13 and R 11; the add I instruction uses registers R 5 and R 3.

They have nothing; they share no pieces of data and therefore, I determine that in fact, the load word R 13 instruction can be moved to the location that I had specified safely. the meaning of the program will not be effected in anyway by doing this I eliminate the need for the two stall cycles and in effect the sequence of instructions will run through the pipeline with each instruction immediately following the instruction before it in the code fragment.

So, this is the simple illustration of how by just changing the order of instructions after having thought through the consequences carefully, the need for stalling instructions could be eliminated and this is how static instructions scheduling could be done by a compiler to remove data hazards. We should know that the data hazards which have existed in the prescheduled code size do exist in the rescheduled code and therefore, we have successfully removed the data hazards.

(Refer Slide Time: 09:08)



Ok, now the example that we just looked at: assume that the work was done by the compiler and I had talked about the alternative where there could be processors which are designed so that the hardware does this rescheduling of instructions dynamically.

(Refer Slide Time: 09:31)

Kinds of Data Dependence

- True dependence
ADD R1, R2, R3
SUB R4, R1, R5
- Anti-dependence
ADD R1, R2, R3
SUB R2, R4, R5
- Output dependence
ADD R1, R2, R3
SUB R1, R4, R5
— final value of R1

I had suggested in the previous lecture that today such processors or not uncommon. I will just make a few comments about that but, prior to talking about the processors I like to clarify few kinds of data dependencies that the compiler or the hardware may have to check for. Now, the first kind of dependence between two instructions data dependence between two instructions that we had seen was of the kind where one of the instructions or let us say, the earlier instruction generates a value in a register. In this example which is used by the subsequent instruction and the problem now is that the second instruction is dependent on the first instruction for the value of R 1 and this kind of a dependence is known as a true dependence.

There are other kinds of dependence; data dependence that we did see and the one example that we did see for example, in the previous lecture was the situation where the subtract instruction is writing or using R 2 as destination and the add instruction which precedes it uses the same register as its source.

Now, there is no real danger as far as the execution of these instructions in this order is concerned. As far as we can see because the add instruction will read R 2 well before the subtract instruction updates register R 2 in a normal pipeline but, we do need to note that there is a dependence between these instructions in the sense that if I had thought of reordering these instructions. In other words, putting the subtract instruction before the add instruction then the meaning of the program would change because, if I put the

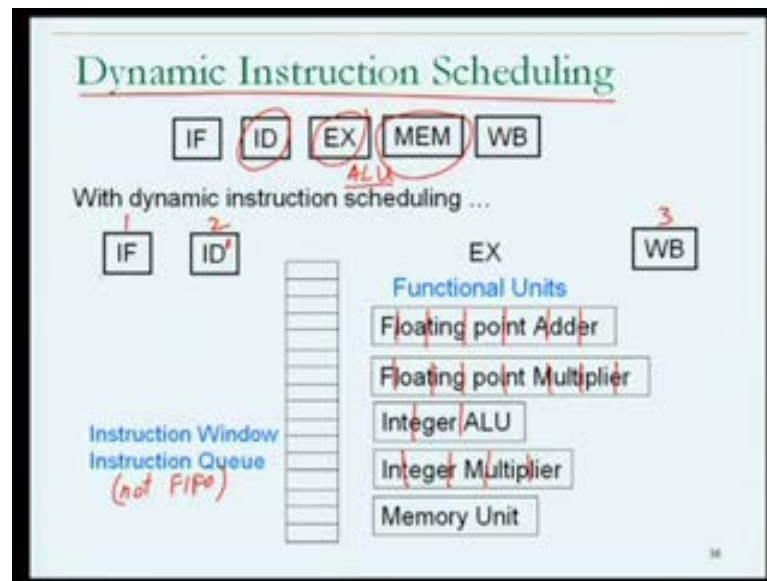
subtract instruction before the add instruction then the add instruction would receive the value of R 2 which had been created by the subtract instruction which was not the meaning of the original program.

Hence, we must view this kind of a pair of instructions as also being dependent on each other and this kind of dependence is known as anti-dependence. So you can clearly see the difference between the two in the true dependence. The first instruction in the pair generated a value which was read by the second instruction in the pair. Whereas, in the anti-dependence the opposite was true; the first instruction of the pair use the value or register which was updated by the second instruction in the pair.

Now, there is a third kind of data dependence that hardware may have to be aware of and check for if one is assuming that the hardware does the rescheduling of instructions and that is something called the output dependence as illustrated by this example. And, here we notice that there are two instructions both of which have the same destination register and if we think about this we realize that there are consequences of reordering these two instructions would be that the add instruction would update R 1 after the subtract instruction updated R 1 which would change the meaning of the final value inside the register R 1.

So, the final value of R 1 would change if I reorder these instructions and hence the meaning of the program would change. Therefore, that would not be a legitimate reordering from the perspective of instruction scheduling. Therefore, in general they are these 3 kinds of dependencies that the compiler or the hardware depending on who is doing the reordering of instructions would have to take into account.

(Refer Slide Time: 12:47)



Now, when it comes to hardware to do the reordering of instructions we are still talking about a pipeline processor but, I will sort of generalize the capabilities of the processor to give you a better perspective on what pipeline processors today might actually look like. In the kind of pipelining that we were discussing up to now, we talked about 5 pipeline stages: instruction fetch, instruction decode, execute, memory and write-back and the assumption that we use was that there was no instruction reordering, no instruction scheduling. And therefore, the instructions would go through the pipeline as specified by the program, sometimes benefiting from bypassing, sometimes having to be stalled depending on the way that the data hazards are being handled.

Now, let us talk a little bit about how things could change with dynamic instruction scheduling. Now, if there is dynamic instruction scheduling there will still be a need to fetch and decode the instruction. Hence, we suspect that there would still be in instruction fetch stage in the pipeline of the dynamically dynamic instruction scheduling processor and there would still be the need for instruction decode stage to decode the instruction. However, it may not be necessary, it may not be possible to actually do the fetching of all the operands within the instruction decode stage and therefore, that functionality may be removed for the instruction decode stage.

So, I will just put a prime over the ID to indicate that this is different from the instruction decode stage; that we had in the pipeline that we were discussing up to now. Recall that

that in the pipeline that we were discussing up to now the instruction decode stage had hardware to do the decoding of the instruction and the fetching of the register operands.

Now, if I am talking about the dynamic instruction scheduling there may be the need for an instructions operands to be fetched after waiting for the operand values to become available in the registers and hence, that particular functionality may not be present in the ID stage hence the notation ID prime.

Now, if the hardware is going to be built to dynamically scheduled, the instructions that would imply that it is not proceeding it is not allowing the instructions to proceed to the next stage in the pipeline in the order in which they were fetched and decoded and hence there must be a pool of instructions or a piece of hardware memory within the processor where instructions that are under consideration for being scheduled next could be contained. Subsequently, all of those instructions one by one would execute but, during the time that they are being considered for being scheduled as a next instruction to execute they would have to be remembered and hence there would be some kind of a hardware data structure; some kind of a memory storage with associated logic within the processor where instructions could be saved after they had been decoded for consideration for scheduling.

This kind of a structure in a pipeline would be referred to as an instruction window or in some cases, as an instruction queue though it is not going to be treated as a first in first out queue. Clearly, if it was the first in first out queue, that would imply that the instructions enter the queue in some order and come out of the queue in the same order. In the same order which is not the clearly not the case here because, we are talking about dynamically scheduling the instructions in some different order.

So, basically the instructions get fetched, they get decoded without operands being fetched completely and then they are put into this instruction queue where the hardware of the dynamic instructions scheduling which implements a dynamic instruction scheduling determines which of the instructions should be allowed to execute next and need not be the instruction which is the... which was first fetched and first decoded but, could be one of the later ones as long as meaning of the program does not change.

So, those considerations would have to be taken into account by the hardware that does this scheduling and that would be the dynamic instruction scheduling component of the pipeline.

Now, to just give you a different perspective on how the rest of the pipeline might be organized and my discussion from this point on does not specifically relate to dynamic instruction scheduling but, it is just to give you more general perspective on what the rest of the pipeline might look like. We should note that when we talked about the simple pipeline we had the execution of instruction but, we really had only one piece of hardware that was doing the execution. Specifically, there was an ALU which was doing the execution of the instruction and it was used for add instructions. It was used for subtract logical instructions but, it was also used for load word instructions to calculate the effective address and further it was also used for branch instructions to calculate the branch target. However, we do know that in the, even in our MIPS 1 instruction set there are instructions which cannot be executed by the ALU. For example, they could be floating point instructions.

So, we suspect that in addition to the just the ALU for the execution of instructions, there may be several pieces of hardware - each specialized to do different kinds of operations and that therefore, when thinking about the processor pipeline it is not that thinking about the processor pipeline is just having an ALU is not realistic. Rather, we should think of a pipeline which has many different pieces of hardware for different functional capabilities as far as the instruction set is concerned. For example, if there is a... if there are floating point instructions there must be a piece of hardware they can add floating point values. Similarly, a piece of hardware they can multiply floating point values may be another piece of hardware to do the division of floating point values as far as the integers are concerned, they would need to be an integer ALU; there may also be the need for an integer multiplier.

You saw you may suspect that multiplication of integers is a little bit more complicated than addition of integers and therefore, they may be a separate piece of hardware to do that. Further, we could actually say that rather than having a separate memory stage we could view the operations associated with the memory stage from our prior description of the pipeline is actually just being another functional requirement which could be met by another functional unit which I label as the memory unit.

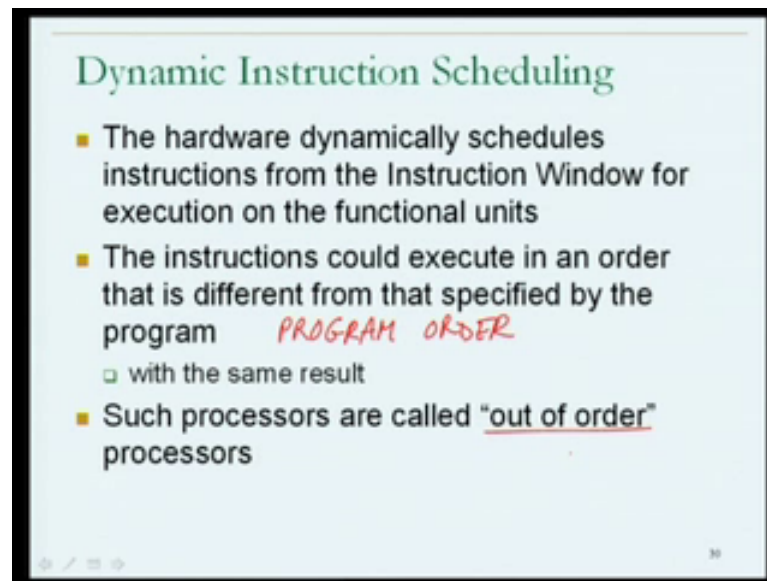
So they could be a large number of such functional units the number would depend on the nature of the instruction set and how much functionality could be built into each of the functional units subsequently when an instruction completes it would still have to write-back therefore, we would suspect that they would be a write-back stage.

Now, the question that you will ask at this point is how many stages are there in the pipeline that we have just drawn and the answer is we can clearly see that there is an IF stage there is an ID stage and there is a write-back stage but, as far as the execution stage is concerned you notice that I have shown the different functional units as being of different lengths and the suspension that you may have had is that I am using different lengths because the amount of time to execute an operation through that functional unit may be more for example, in the case of floating point multiplication then in the case of a single integer ALU which is why the floating point multiply unit itself is longer.

So the consequences that we may want to think about the functional units themselves as potentially being pipelined so this floating point adder might have a six stage pipeline it may be a pipeline floating point adder the floating point multiplier may be in eight stage or seven stage multiplier the integer ALU may be a 3 stage integer ALU, etcetera.

so these functional units could might themselves be pipelined and therefore, the number of cycles that it takes for an instruction to go through the pipeline might be one for IF one for ID the instruction might spend 3 or 4 cycles waiting in the instruction queue then it might spend 5 or 7 or 3 or 1 instruction of a cycle in the execution unit before spending one cycle in the write-back unit. So the thing to note here is that by doing dynamic instruction scheduling the order in which the instructions are fetched and decoded is not going to be the order in which they get executed right.

(Refer Slide Time: 21:01)



Dynamic Instruction Scheduling

- The hardware dynamically schedules instructions from the Instruction Window for execution on the functional units
- The instructions could execute in an order that is different from that specified by the program **PROGRAM ORDER**
 - with the same result
- Such processors are called "out of order" processors

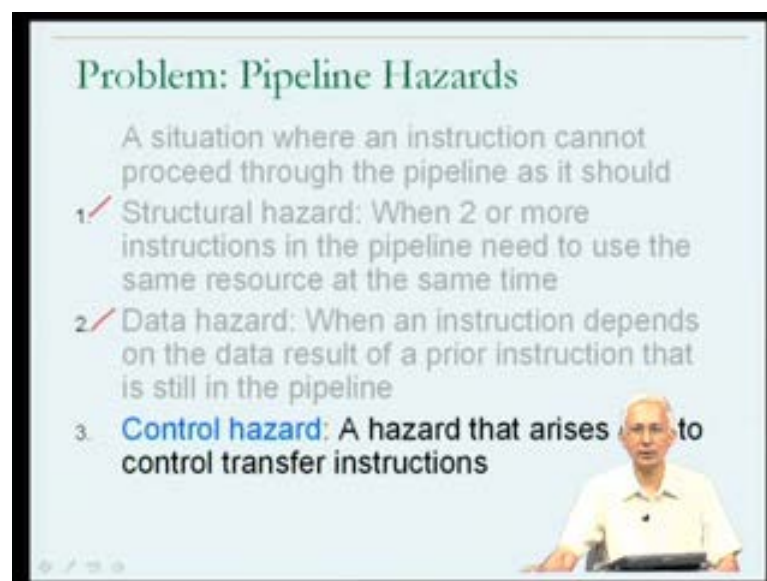
so the dynamic the hardware dynamically schedules instructions from the instruction window for execution on the functional units and hence the instructions could execute in an order that is different from that specified by the program in general this the reordering must be done safely

so the consequence of executing the instructions in this different order should not result in a different outcome of the program so that checking for dependence in independence of instructions is important but, we should note that the order in which the instructions execute could be different from that specified by the program and the result should be the same. Such processors are hence sometimes called out of order processors in that they execute the instructions in an order that is different from that specified by the program the order specified by the program is often refer to as the program order but, in the interest of eliminating data hazards or improving the performance of a particular program the hardware might scheduled the instructions to be executed in some other of order and such hardware such a processor would be called an out of order processor.

Now, I will just go back to the previous slide and point out that the instructions would in this particular scheme of things it is conceivable that the instructions would be fetched in program order which I will write as in program order the instructions would consequently be decoded in program order now they may be scheduled for execution out of program order but, they may be a requirement that they actually complete the write-

back in program order and this is a common characteristic of many of the aggressively pipeline processors today they are in order in program order in the early stages they finally, finish instructions in program order it is only in the intermediate stages that in order to eliminate data hazards or to improve performance that they handle instructions in carefully checked different order hence the term that you may see in some of the literature or sales literature that you read about in out of order processor so much for instruction scheduling ok.

(Refer Slide Time: 23:19)



Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

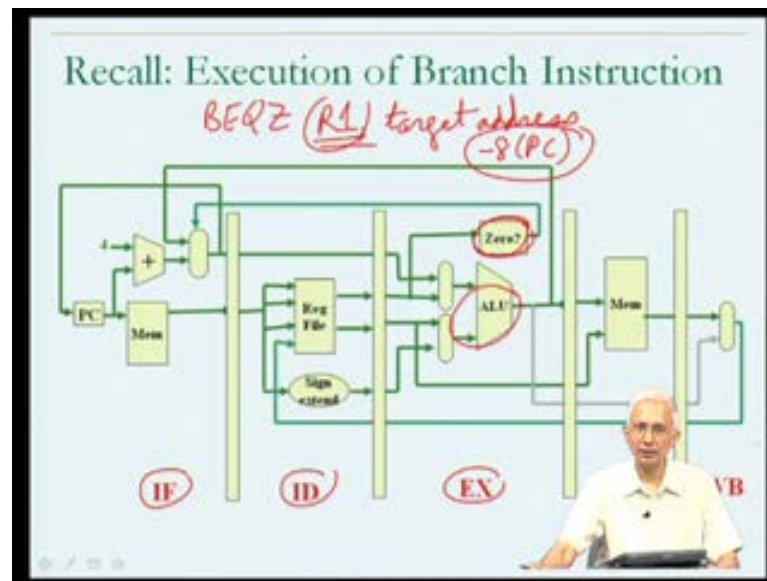
1. **Structural hazard:** When 2 or more instructions in the pipeline need to use the same resource at the same time
2. **Data hazard:** When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. **Control hazard:** A hazard that arises due to control transfer instructions

© / 7 9

Video inset: A man in a white shirt sitting at a desk.

Now, with this we have understood two kinds of pipeline problems one was the structural hazard one was the data hazard we will next look at the third class of hazards which I had called control hazards and in the introduction I had mention that a control hazard is a hazard that arises due to control transfer instructions let us try and understand a control hazard once again through an example.

(Refer Slide Time: 23:43)



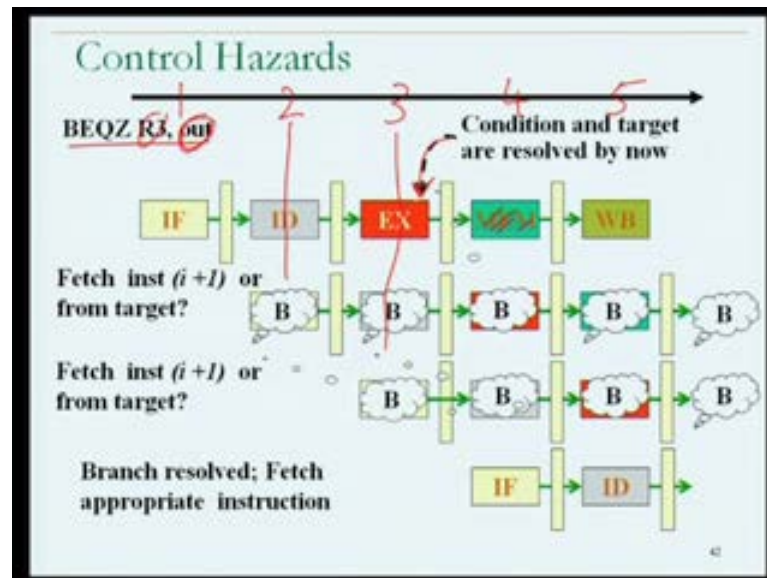
Now, let me just remind you from our basic hardware that we had derived how a branch conditional branch instruction gets handle through the pipeline remember that it gets fetched using the fetch hardware it gets decoded using the decode hardware we show in example of a of a branch instruction branch if equal to 0 R 1 target address the target address is specified as displacement from the program counter.

So I will write in that form so when this instruction is fetched subsequently when it is decoded during decode the instruction will be properly understood and the contents of its operand register in other words R 1 will be red from the register file in the execute stage two things have to be done one is that its target address has to be calculated within theALUand secondly the target address calculation within theALUamongst to the calculation of using the displacement from the program counter value and you will recall that in the case of the MIPS 1 instruction set the calculation was p c plus 4 minus 8 in addition to that the check of the condition must be done whether the contents of register R 1 which was red into one of the special purpose registers within the processor in the ID stage is in fact equal to 0.

So that is the condition checking which also we assume was done by hardware included in the e x stage of the pipeline subsequently nothing of interest happens in the m e m stage and in the write-back stage the program counter value could be updated so that is

what happens during the execution in the in our pipeline processor of a conditional branch instruction so let us look at that example more critically.

(Refer Slide Time: 25:31)



So, here was the a similar instruction branch if equal to 0 R 3 out so once again I am going to use the timeline the branch if equal to 0 instruction goes to the pipeline as we have just described the IF stage, ID stage, EX; nothing much of interesting happening the MEM stage and finally, updating of the program counter value possibly in the write-back stage. Now, from the discussion on the previous slide you will remember that the important part in the pipeline where the activity of the branch instruction was reasoned out happened in the EX stage. That is why the target address was calculated and also where the condition was checked whether the contents of R 3 were equal to 0.

Therefore, we know from that discussion that in our current scheme of the pipeline the condition and the target are resolved or determined as far as this branch instruction is concerned by the end of the EX stage let me just go back to the previous slide to make sure that is what we had reasoned by the end of the EX stage the condition would have been evaluated and the target address would have been calculated. So, that is all that we I marking in this slide by this point in time that the condition, in other words, is R 3 equal to 0 or not and the target. In other words, what is the actual address associated without in terms of with p c relative addressing mode now, resolved by now? Therefore, the problem which arises as far as the pipeline is concerned is, let us suppose that this branch

if equal to 0 instruction was fetched in what i will call cycle number 1 and its decode in cycle number 2 etcetera.

So, I am just numbering the cycles. Now, the question is after the branch instruction has finished using the IF stage we know that in the next cycle when the branch instruction is using the ID stage we want some other instruction to be fetched but, the problem that we have is we do not know which other instruction should be fetched. The question that arises is written here; should we be fetching the next instruction? What should be fetching from the target? In other words, will the branch be taken or not? And, since we do not know the answer to that question we do not really know and therefore, we suspect that what would have to happen is that the pipeline would have to stall for that cycle in other words the IF stage could not fetch any instruction because the branch instruction has actually not reached the stage of execution where we know which instruction has to be fetched. Therefore, we cannot answer the question of whether we should fetch instruction I plus one or the instruction from the target of the branch hence they need to stall this - the pipeline.

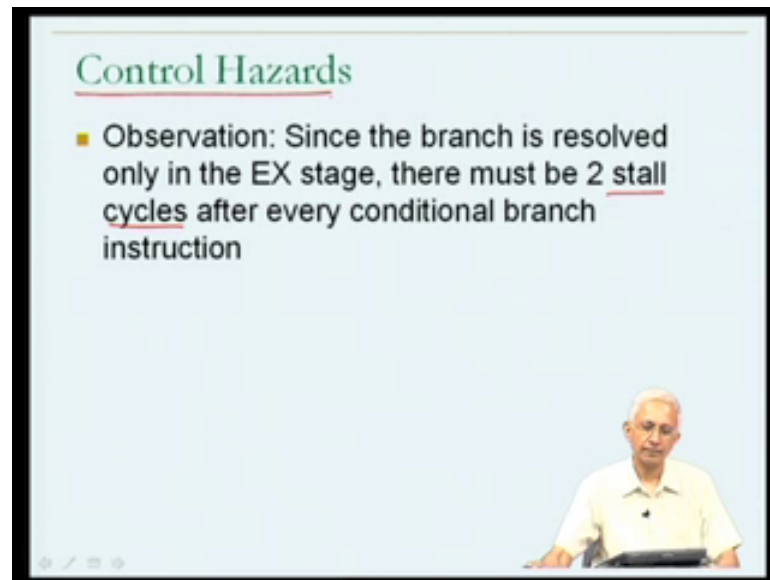
Now, the question is that is what happen in cycle number 2 but, in the next cycle cannot we fetch the correct instruction now unfortunately the activity of fetching in the cycle number 3 would have to start in the IF stage at the beginning of the cycle and from our understanding of what happens in the EX stage as far as the branch instruction is concerned.

We realize that whether or not the branch is going to be taken and also what the target address is will be known only by the end of cycle number 3. Hence, even in the next cycle we are not in a position where we can answer the question whether we should fetch instruction I plus 1 or from the target and hence, we will actually have to stall this pipeline for one more cycle in which we have to push a bubble into the pipeline rather than a useful instruction starting through the pipeline.

Now, what this boils down to is that the earliest that we would be able to fetch, the earliest that we would know which instruction we have to fetch next would be in cycle number 4 we would know whether the condition was true or false and therefore, whether we should fetch from the target address or from the fall through address and hence this would be the situation as far as the control instruction like the branch if equal to 0 is

concerned notice that the consequences are very bad from the performance view point. In that, after every conditional branch instruction like this, they would be the need to have two stall cycles. In other words, two cycles in which no instruction would complete later on down the pipeline.

(Refer Slide Time: 29:37)



This is the actually, unfortunate observation that associated with every branch instruction. There will be these two cycles after the branch instruction when basically, a bubble would have to go through the pipeline or in a current terminology, we talk of this as two stall cycles. When we are uncertain in this case as to what instruction should be fetched and this is the control hazard problem.

The problem that after every conditional branch instruction of this kind there is a need to actually stall the pipeline for two cycles it is a hazardous situation. In that there is a performance loss; it is a hazardous situation. In that if the pipeline is not built to do these two stall cycles then incorrect program execution would result.

(Refer Slide Time: 30:21)

Reducing Impact of Branch Stall

- The execution of a conditional branch instruction involves 2 activities
 1. evaluating the branch condition (determine whether it is to be taken or not-taken) **EX**
 2. computing the branch target address
- To reduce branch stall effect we could
 - evaluate the condition earlier (in ID stage) ✓
 - compute the target address earlier (in ID stage) ✓
- The number of stall cycles would then be reduced to 1 cycle

Inset photo of a man in a white shirt sitting at a desk.

Now, immediately the reaction of the computer architect would be, we have to find some way to reduce that number of two stall cycles. That is too heavy a price to pay for the execution of branch instructions in a pipeline. Remember that, in the case of pipelining we want to get the maximum rate of execution of instructions. We want to avoid situations where there is a stall a cycle in which no instruction completes.

Now, in analyzing what is happening in the case of the branch instruction, we realize that the bottle neck. The problem is arising because the two pieces of functionality as far as the branch instruction are concerned, in other words, evaluating the branch condition and computing the branch target address. Both happening in the EX stage and that if you could find the way to do it earlier then we could reduce the impact of the branch instruction on performance.

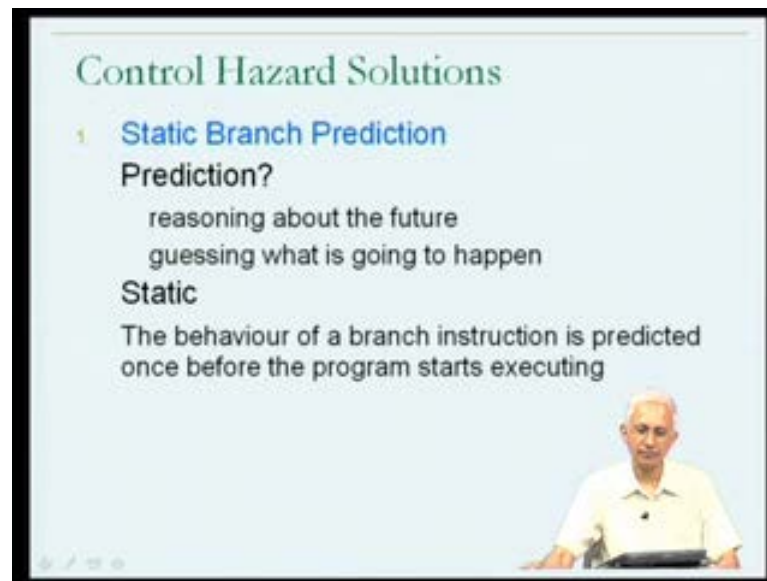
So, as a very aggressive piece of hardware design we might say we can try to reduce the branch stall effect by evaluating the branch condition earlier. In other words, in the ID stage and computing the target address earlier in fact in the same ID stage and we need to think about this a little bit because, it does not seem feasible. If you go back to the picture of the pipeline we know that the checking of the condition and the evaluation of the branch target are happening in the EX stage. The suggestion now is that they could be done in the ID stage but, you will notice that register R 1 is read only in the ID stage.

Now, if I did want to do the condition check and the target address calculation in the ID stage at the very least. I would need to have the hardware to check comparison with 0 inside the ID stage in addition I would need a simple ALU in the ID stage in order to do the target address calculation even this would not be adequate to solve the problem because we note that our assumption as far as the way the ID stage functions is that the reading of the source operand within the ID stage happens towards the end of the cycle and therefore, if I wanted to take the result from the register in other words, R 1 and compare it with 0. I would have to make the cycle time of the ID stage a little bit longer.

So, in addition to whatever work it used to do, **I now on to** add the work of comparison with 0 and calculating the target address which is going to mean making the cycle time a little bit longer. But, if I am willing to accept this these overheads the overheads being a piece of hardware should compare with 0 inside the ID stage and an additional ALU, a simple ALU included inside the ID stage along with a small time over head of the cycle time becoming a little bit longer than it turns out. This becomes feasible it is possible to reduce it is possible to identify whether the condition of the branch is true or false in the ID stage and it is possible to compute the target address inside the ID stage and what would be the impact of this on the branch hazard, the control hazard. The answer is the number of stall cycles would then reduce to one cycle.

Notice that, if you go back to this diagram we are now assuming that the condition and the target will be resolved by the end of the ID stage; which means that while I would still need this stall cycle, I would not need this stall cycle. The correct instruction could correctly be fetched in cycle number 3 which means that there is only one stall cycle which is required. So, the number of stall cycles would then be reduced to one. We still have one stall cycle and ideally we would want to have a pipeline in which the control hazard is fully handled. In other words, where there is potentially no stall cycles; we need some more ideas to understand how such pipelines might be designed.

(Refer Slide Time: 34:11)



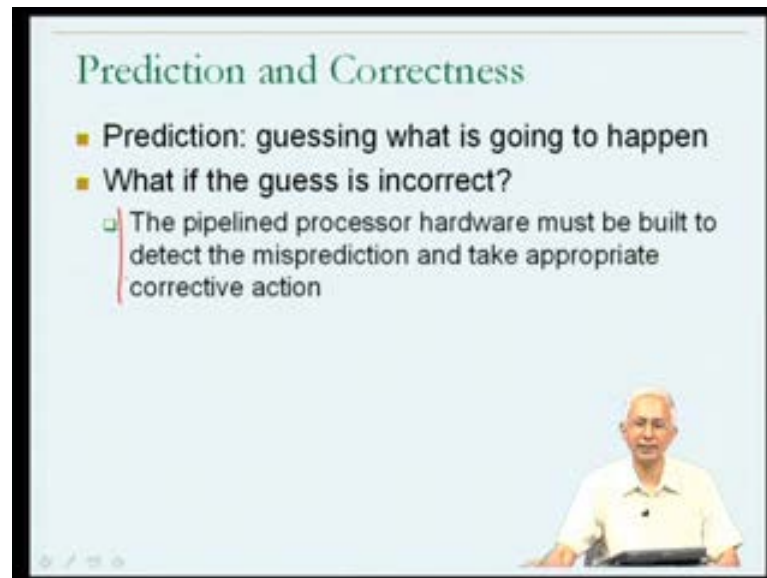
Now, I am going to talk about few ideas which are sometimes used in processors for handling control hazards; essentially to remove that one stall cycle which we were left with at the end of the previous slide. Now, one idea which is sometimes used is to do something called static branch prediction. We have come across the word static and the word branch before but, the word prediction comes as a bit of a surprise because when you hear the word prediction you assume that this means reasoning about the future. Predicting is to talk about something which has not yet happened.

So, trying to talk about guessing what is going to happen and that seems to be what is suggested here that, as far as the branches are concerned, try to guess what is going to happen. As far as the branch is concerned, the word static we have seen before we know that means that this is some kind of an activity which is going to primarily be done when the program is compiled statically once for all which typically means, when the program is compiled and that is essentially what we are talking about here.

Putting all those terms together static branch and prediction, this is what we come up with that this is an idea of the behavior of a branch instruction being predicted; once before the program even starts executing right and this is a rather curious notion. But, if one could predict what the branch is going to do before the program starts executing then conceivably one could set up the processor pipeline to use that information and

successfully handle without that need for the stall cycles just think about this in more detail ok.

(Refer Slide Time: 35:48)



Now, one problem with prediction or guessing whether the branch thinks about the branch is that the guess may be wrong and if the guess is wrong then the program would behave incorrectly. I am very clearly does not expectable what I mean by incorrectly is the program would not execute as specified by the instructions within the program.

So, very clearly since the guess about the branch could be wrong the prediction could be wrong, the pipeline processor hardware must be built to detect the misprediction and then take appropriate corrective action so that, the net result is that the program executes correctly. One cannot sacrifice correctness in the interest of performance. Just to remove the stall cycle - the control hazard stall cycle, one cannot build a processor that does not execute programs correctly. Therefore, this is the consequence; pipeline processor hardware must be built to detect mispredictions and take appropriate corrective action.

(Refer Slide Time: 36:47)

The slide is titled "Control Hazard Solutions". It features a list of solutions, with the first one being "Static Branch Prediction". Handwritten in red ink above this title is "BEQZ R3, (tgt)". Below the title, it says "Example: Static Not-Taken policy". A red arrow points to the text "The hardware is built to fetch next from PC + 4". Below this, there are two bullet points: one with a square icon stating "After ID stage, if it is found that the branch condition is false (i.e., not taken), continue with the fetched instruction (from PC + 4)", and another with a square icon stating "Else, squash the fetched instruction and re-fetch from the branch target address". A sub-bullet under the second point says "squash: cancel, annul the processing of that inst". In the bottom right corner of the slide, there is a small video inset of a man in a white shirt.

Let us look at a simple example of static branch prediction and understand this. now, the example I am going to talk about is a static branch prediction policy called static not-taken policy and as the name suggests, the idea has to assume that branches are not-taken and so basically, what does it mean to assume that a branch is not-taken? If I have a branch - branch if equal to BEQZ R 3 target and I assume that it is not-taken then, by default the instruction that is going to be executed next is going to be the next instruction in the program and therefore, what that boils down to is building the hardware to automatically fetch from the next instruction; not from the target but, from the fall through path.

So, that is the idea of the static not-taken policy; build the hardware to automatically fetch from the next instruction. In this slide I am denoting this as PC plus 4. We may correct that but, essentially what we are talking about is the next instruction now what kind of corrective action could be taken? Now, the corrective action that could be taken is we know that by the end of the ID stage, in other words, the branch instruction is fetched and in the IF stage is itself, is fetched. When it goes to the ID stage, some other instruction is going to be fetched that **which** instruction is answered by this line. By the end of the second cycle the ID stage contains the hardware to not only find out if the branch condition is true or false but, also what the target addresses. Therefore, by the end of the ID stage, we know both whether condition is false or not and what the target address is.

Now, if it is found out that the condition had been wrongly predicted, in other words, that the branch is supposed to be taken that the static prediction of not-taken was wrong then, what will have to be done is the instruction will have to be negated and the term which we use for that is to talk about it being squashed.

So, the technical term squash basically means to cancel or annul the processing of that instruction. So, this is the hardware activity; the hardware takes, makes this kind of a guess if it finds out that the guess is wrong then it has to undo the work that it has done by squashing the instruction. Let us just look at this with an example.

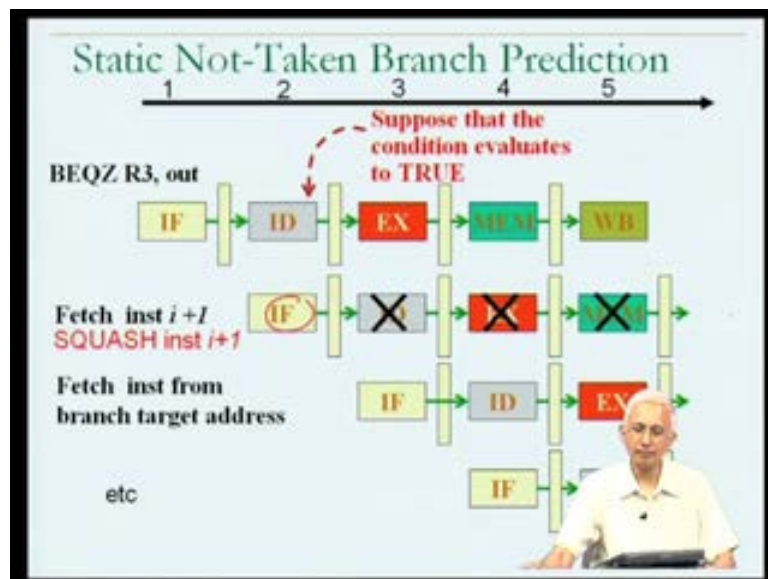
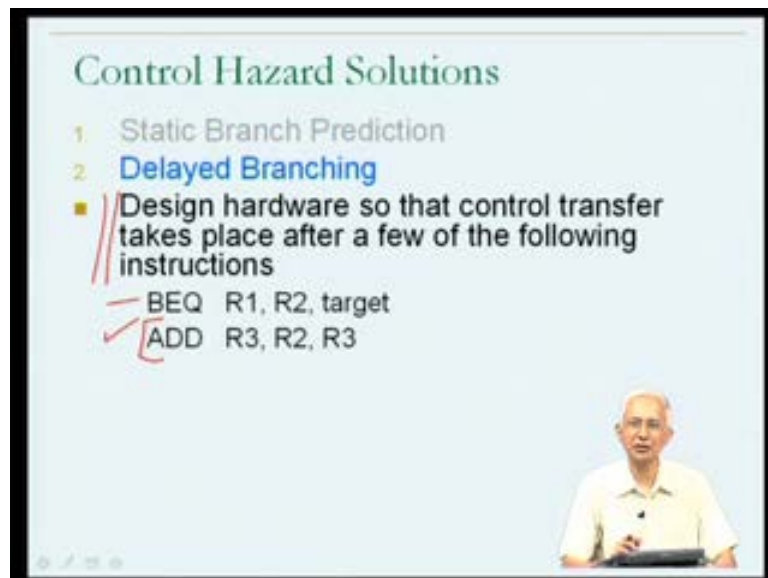
(Refer Slide Time: 39:07)

Control Hazard Solutions

1. Static Branch Prediction
2. Delayed Branching

- Design hardware so that control transfer takes place after a few of the following instructions

- BEQ R1, R2, target
- ✓ ADD R3, R2, R3



Now, here is a branch if equal to BEQZ R 3 out instruction; now, in cycle number 1 this instruction is fetched and we know that in cycle number 2 it is decoded but, until cycle number 2 is over we do not know what the next instruction that should be fetched is. Therefore, using the static not-taken branch prediction policy then in cycle number 2 while the branch instruction is being decoded. We automatically fetch instruction $i + 1$; the assumption here is that the branch if equal to 0 instruction is instruction i . so, we automatically fetch instruction $i + 1$ in other words the next instruction in the program now by the end of the ID stage we know whether our guess of not-taken was true or was correct or not.

Now, let suppose that the guess was correct and that the condition evaluates to false. In other words, the branch is not-taken then the fetch which we have done is correct and the rest of the sequence of instructions goes to the pipeline normally. Therefore, if it turns out that I guess was correct then, we have actually avoided the stall cycle. So, there are no branch stall cycles; but, this is not guaranteed to happen all the time the condition always actually evaluating to falls.

So, we need to look at the other possibility. In the other possibility as before, the instruction is fetched in cycle 1; in cycle 2 the instruction is decoded and once again I am assuming that the branch is not-taken; I am building the hardware to assume that the branch is not-taken. However, by the end of that cycle, I determine that the condition is in fact true, unfortunately the condition is true.

So, the hardware as soon as it detects this has to squash this instruction. It has to squash instruction $i + 1$ and subsequently, in cycle 3 it has to fetch a start fetching from the correct target address in other words, in cycle number 3 it has to fetch from whatever target address had been correctly computed in cycle number 2 ID stage and subsequently the instruction which had been incorrectly fetched in the cycle number 2 will have to be ignored or annulled. Its activity should be canceled which are indicated by an x mark suggesting that the instruction which activity is happening in the ID stage of that point in time should be ignored and any effects that it tries to produce should be negated by the hardware. Similarly, the instructions proceeds through the pipeline as canceled. So, we have this additional notation of an annulled or a canceled instruction going through the pipeline if there is a canceled instruction going through the pipeline it will not be allowed to modify the state of the processor such as, updating registers and so on. It merely

unfortunately occupies a cycle so it amounts to one cycle in which no instruction is completed but, at least to correctness of the program is not compromised ok.

So, the bottom line is that under the static not-taken branch prediction policy the hardware was built to automatically fetch the next instruction assuming that the branch was not-taken and in situations where the hardware was wrong some corrective action has to be taken squashing the instruction and moving an annulled canceled instruction through the pipeline has a consequence of which there is the equivalent of one branch stall cycle so we see that there these two cases in the first case.

(Refer Slide Time: 42:25)

Control Hazard Solutions

- 1. **Static Branch Prediction**
 - Example: Static Not-Taken policy
 - The hardware is built to fetch next from PC + 4
 - After ID stage, if it is found that the branch condition is false (i.e., not taken), continue with the fetched instruction (from PC + 4) **0 stall cycles**
 - Else, **squash** the fetched instruction and re-fetch from the branch target address **1 st cycle**
 - Thus, average branch penalty < 1 cycle

where the guess was correct. There were 0 stall cycles in the second case where the guess was wrong they would be one stall cycle. Therefore, if 75 percent of the time are being more optimistic if 90 percent of the time I was correct in my guess, the 90 percent of the time the 0 stall cycles and only 1 percent or 10 percent of the time is there a stall cycle and therefore, the average control hazard problem will amount to point one stall cycles rather than one stall cycles.

If I look at the effect over a entire program, we have substantially reduce the impact if the rate of success of the guess is high, right. But, in general we could say that the average branch penalty will be less than one cycle and therefore, this could be, should be viewed as an improvement over the alternative where there was guarantee to be one stall

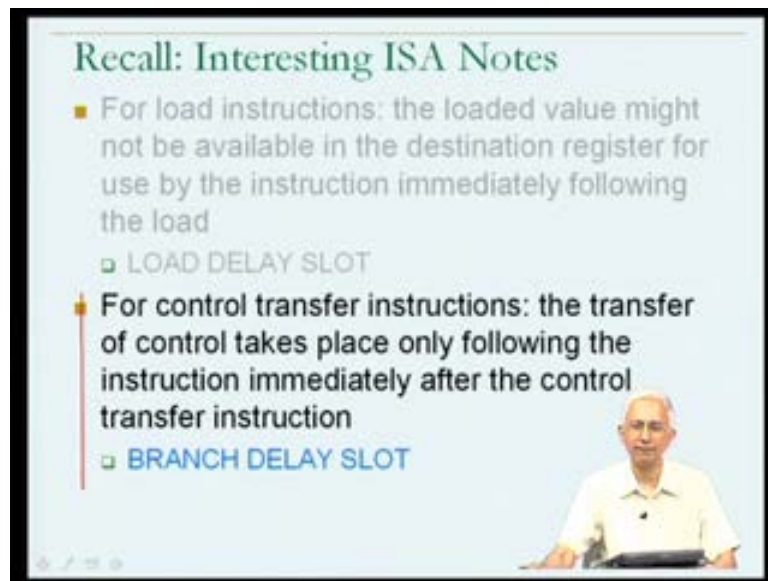
cycle where we refuse to make a guess and just waited for one stall cycle without taking any action.

Therefore, the idea of static branch prediction can lead to improved performance. As far as the pipeline is concerned, it will lead to a little bit more complication in the pipeline. In that, they must be built into the pipeline the capability of detecting that the condition was wrongly predicted and that therefore, an annulled instruction must be passed through the pipeline without updating the processor state.

So, some complication in the design of the processor pipeline but, this is an idea which seems to make some sense in the light of reducing the performance on average, I am sorry, improving the performance on average; reducing the number of stall cycles on average for a program and they could be other static policies but, this is one example which is seems to make good sense (Refer Slide Time: 44:03).

Now, an additional idea which is often talked about in connection with control hazards is the idea of something called delayed branching. And, the idea of delayed branching is that we actually design the hardware so that the control transfer takes place may be after a few of the following instructions. As a simple example, let us suppose that I design the process of pipeline so that after the branch if equal to 0 instruction the branch if equal to 0 instruction, is going to transfer control to target if R_1 is equal to R_2 . But, if I design the hardware so that there was a property that the control transfer even if R_1 is equal to R_2 , the control transfer to the target address will take place only after the add instruction which follows the branch instruction is executed. Then, by doing this I am actually providing that one cycle in which I need not make a guess about whether the branch instruction is taken or not because, the add instruction has to be executed by definition and therefore, I avoid the stall cycle entirely. There is no need to make a guess; there is no need to stall the process of a one pipeline after the conditional branch instruction; we automatically execute the next instruction as per the definition of what a branch instruction means and we have seen this idea before when we were talking about interesting notes in the MIPS 1 instruction set architecture manual; this was the branch delay slot idea that we saw.

(Refer Slide Time: 45:21)



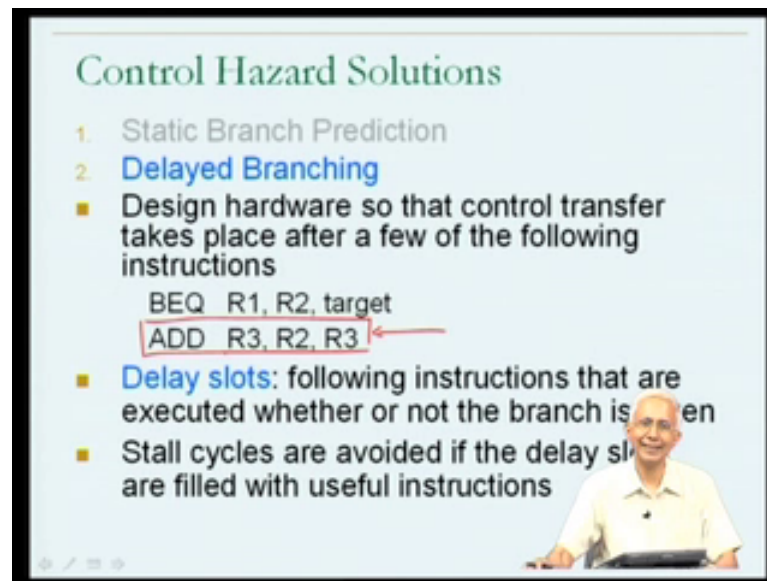
Recall: Interesting ISA Notes

- For load instructions: the loaded value might not be available in the destination register for use by the instruction immediately following the load
 - ▣ LOAD DELAY SLOT
- For control transfer instructions: the transfer of control takes place only following the instruction immediately after the control transfer instruction
 - ▣ BRANCH DELAY SLOT

The warning that we saw in the process of instruction set architecture manual was for control transfer instructions. The transfer of control takes place only following the instruction which immediately follows the control transfer instruction which is the situation that we were seeing over here. The control transfer follows; it happens not after the branch instruction but, after the instruction following the branch instruction hence the name branch delay slot.

So, this idea we had seen something that is used inside the MIPS 1 instruction set architecture and we now understand why it is being used to allow for the MIPS 1 processors to be designed using pipelines of the kind that we have seen in which its difficult to avoid one stall cycle **ok.**

(Refer Slide Time: 46:14)



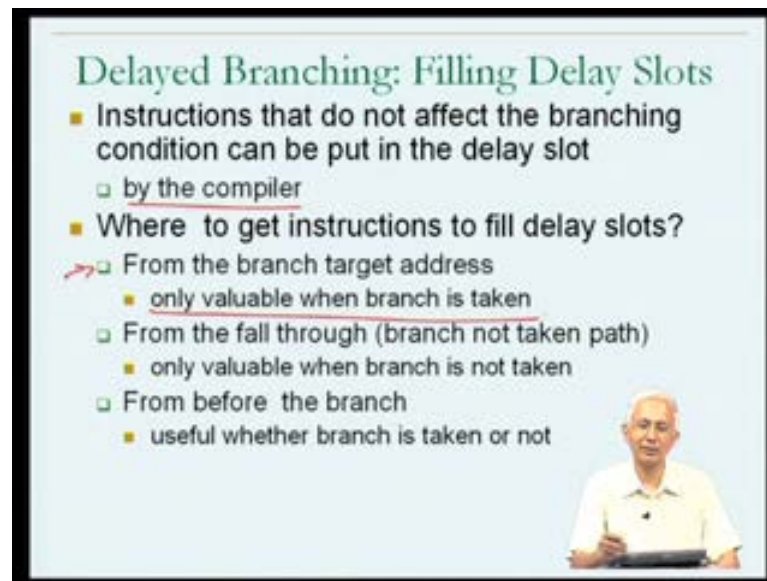
Control Hazard Solutions

1. Static Branch Prediction
2. **Delayed Branching**
 - Design hardware so that control transfer takes place after a few of the following instructions
 - BEQ R1, R2, target
 - ADD R3, R2, R3 ←
 - **Delay slots:** following instructions that are executed whether or not the branch is taken
 - Stall cycles are avoided if the delay slots are filled with useful instructions

The slide includes a small inset image of a man in a white shirt at the bottom right corner.

So, with this delayed branching then this is the idea and we will use the terminology of delay slot to refer to the instruction. In this case, one instruction after the branch instruction that will be executed whether or not the branch is taken. Now, the question that arises now is, its well and good to talk about this property of the branch instruction. But, from the perspective of the programmer or the compiler, this becomes a little bit of a problem. We have to actually write programs of the compiler; compiler has to compile programs taking this into account and the question then becomes a question of how does one fill the delay slot. There is a need to put something into the instruction slot following the branch instruction. How does one find the useful instruction to put there? That is a question which arises **ok.**

(Refer Slide Time: 46:50)



Delayed Branching: Filling Delay Slots

- Instructions that do not affect the branching condition can be put in the delay slot
 - by the compiler
- Where to get instructions to fill delay slots?
 - ➔ □ From the branch target address
 - only valuable when branch is taken
 - From the fall through (branch not taken path)
 - only valuable when branch is not taken
 - From before the branch
 - useful whether branch is taken or not

Video inset showing a speaker in a white shirt.

Now, very clearly the possible kinds of instructions which could be put into a delay slot - our instructions, they do not affect the branching condition and therefore, in some sense one has to be careful in deciding what instruction could go into the delay slot. Now, this could be done by the compiler or it might be something that the programmer does but, in either case it is viewed from the perspective of the task being done by the compiler. The question then boils down to, from where will the compiler get instructions to put into the delay slots? There are 3 possibilities; a few possibilities. One possibility is that the compiler could take the first instruction from the branch target address; so, we had the branch instruction and its going to transfer control to some target, if the condition is true. Therefore, the idea here in the first suggestion is that the first instruction from the branch target could be brought and put into the delay slot.

Another possibility is that the first instruction from the not-taken path could be put into the delay slot and the third possibility is that instruction could be taken from somewhere before the branch instruction and put into the branch delay slot. One of the instructions preceding the branch instruction could be moved after the branch instruction and in all cases it could have to be carefully determined that the meaning of the program does not change.

Now, the first idea moving the first instruction from the branch target address into the branch delay slot clearly will be a good thing to do for all cases where the branch is

taken because, we know that if the branch is taken then the instruction which is going to be executed after the branch is going to be the instruction at the branch target address and therefore, moving that instruction into the branch delay slot where it will be executed anyway is beneficial by the same token.

The second option of moving the first instruction from the fall through path, in other words, the branch not-taken path would be valuable only if the branch is not-taken. The last option is, one which would be useful whether or not the branch is taken but, one must, the compiler must identify in instruction **which can be careful** which can carefully be moved into the branch delay slot by checking the independents of instructions, it changes in meaning of the program, etcetera.

But, let me just take one of the other two if I think what I will do is, I will take an instruction from the branch target address as an example just to show that, that may be generally useful despite the fact that it is a value only when the branch is taken ok.

(Refer Slide Time: 49:41)

The slide is titled "Delayed Branching... Compiler's Role" and contains the following bullet points:

- When filled from branch target or fall-through, patch-up code may be needed
- It may still be beneficial, depending on branching frequency
- The more the number of delay slots, the harder it is to fill them usefully
 - A processor might require 2 or more delay slots after every branch

Handwritten notes in red ink include "BEQZ" with a checkmark and a circled "2" next to the last bullet point. A small video inset in the bottom right corner shows a man in a white shirt.

Now, let us think about the situation where we are going to fill the branch delay slot with the first instruction from the branch target and alternative would have been to filled with the first instruction from the fall through path. Now, clearly what the compiler is going to have to do is, it can move the instruction from the branch target to the branch delay slot.

But, in so doing it has to bear in mind that the instruction in the branch target in the branch delay slot is going to be executed whether or not the branch is taken.

Therefore, if the branch is not-taken and the first instruction from the branch target is executed then some corrective action has to be taken and therefore, the compiler will have to do the corrective action by including some patch-up code or corrective code which will correct the meaning of the resultant program to be correct to be as per the original program; just look at a very simple example of this.

So, the situation is I have a branch if equal to 0 R 1 target instruction and if I look at the target I am showing you a label called target somewhere later in the program where the instruction at the target address is add I R7 R 7 1 and it is followed by some instructions in x 1 is apparently load word instruction. I am not showing you all the instructions in between; hence, the dot dot dot. Now, the idea that we are going to look into is moving the add I instruction into the branch delay slot now the branch delay slot is the one immediately following the branch equal to 0 instruction. Now, if I am to move the add I instruction into the branch delay slot then I do have to worry about the fall through path. Therefore, I am going to show you a label which is the fall through label.

So, there are some instructions at the fall through path. Now, if I move the add I instruction into the branch delay slot then you will notice I am moving it. Therefore, the add I instruction is going to move up and the target address is going to get associated with the next instruction in the program. So, I show the add I instruction moved up and I show the target address is now associated with the a word instruction.

Now, we have to worry about the situation where the branch if equal to 0 instruction is executed and the condition is found to be false. If the condition is going to be found to be false then, we are actually going to transfer control to the fall through path which is the next instruction in the program and unfortunately, in the fall through path the add I R 7 R 7 1 instruction should not have been executed

I had artificially caused to be executed by moving the add I instruction from the target address into the branch delay slot. Therefore, the corrective action which I could take at this point is to undo the effect of that instruction; may be by subtracting one from the register that had been incremented by the branch instruction which was moved into the

branch delay slot. So, this is an example of patch-up code; a very simple example but, in general the compiler would have to do the filling of the branch delay slot and the appropriate patching up of the program to ensure that the resultant program runs correctly regardless of whether branches are taken or not-taken.

The net result will be of course, that in this particular example, every time the branch is taken we get the benefit of the branch delay slot filled with an instruction that would have been executed anyway however in situations where the branch is not-taken we have two additional instructions that are executed that could have been avoided and we have the equivalent of two cycles in which no useful instruction is being executed **ok.**

Now, we should therefore, note that the idea of the filling in the branch delay slot with an instruction from the target of in the fall through path may involve some patch-up code but, depending on the programs behavior in other words depending on the frequency of the branch being taken or not-taken this idea may still be beneficial for the improved execution of the program and therefore, it should not be ruled out - out of hand.

Now, it should also be born in mind that the more the number of delay slots the harder it will be to fill them usefully and the reason that I point this out is that after now we were assuming that associated with each branch instruction there was one branch delay slot that had to be filled but, we could have pipelines in which because of the nature of the pipeline there is a need to have two branch delay slots or 3 branch delay slots

In other words, after the branch when the branch instruction is executed it might not be the case that the condition and the target address can be resolved in the ID stage it may be that it was not feasible to do that in a particular process of pipeline and this the result may be that there is a need for 2 or 3 or may be from 4 branch delay slots in which case that would be a warning to the programmer and the compiler in the instruction set architecture manual.

For example, that there is that... there are two branch delay slots and it then becomes the problem of the compiler. Every time I come across a branch if equal to 0 instruction to find two instructions they can be filled in delay slots after the branch instruction and that could, that the problem becomes harder and harder. In that, it becomes necessary to find

preferably two useful instructions and if there are 3 branch delay slots, 3 useful instructions and so on, we will continue from this point in the lecture that follows.

Let me just remind you that in this lecture we have looked in some detail at the hazardous problem of control hazards which arises from the execution of branch instructions - conditional branch instructions and while many techniques are possible to reduce the impact of the branch instruction by complicating the earlier stages of the pipeline and make the resolution of the branch as early in the pipeline as possible, it may be the need. It may end up being necessary for the designers of the pipeline to pass the problem on to the compiler in the form of warnings in the instruction set architecture manual regarding when the effect of the branch will take place and the compiler then has to worry about filling branch delay slots. We will look a little bit more about this issue in the lecture - in the next lecture; thank you.