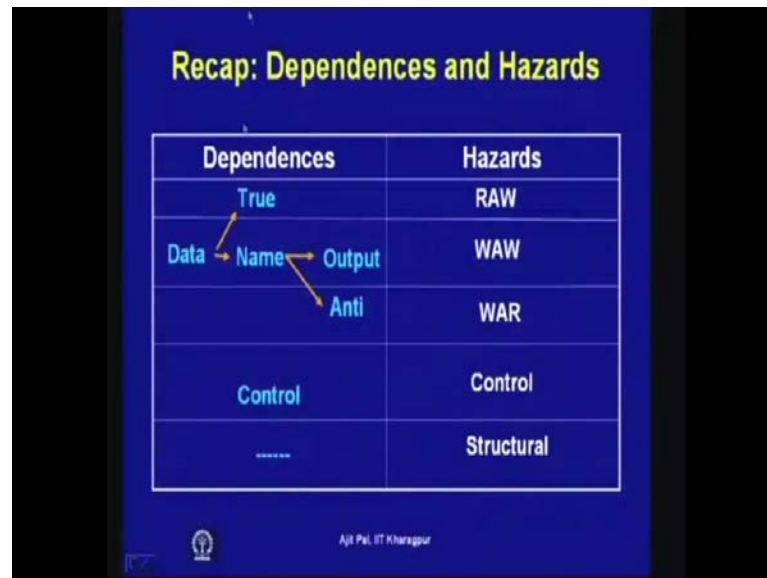


High Performance Computer Architecture
Prof. Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 15
Control Hazards

(Refer Slide Time: 01:00)



The slide shows a table titled "Recap: Dependences and Hazards". The table has two columns: "Dependences" and "Hazards". The rows are as follows:

Dependences	Hazards
True	RAW
Data	WAW
Name	WAR
Output	Control
Anti	Structural

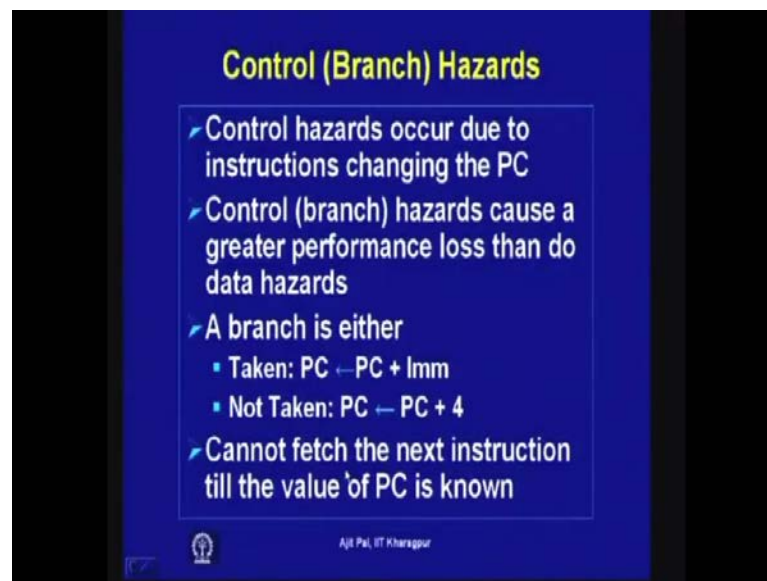
Arrows indicate relationships: an arrow from "Data" to "True", an arrow from "Name" to "Output", and an arrow from "Anti" to "Output".

Hello, and welcome to today's lecture on control hazards. Before I do that let us have a quick recap of various types of dependences and hazards which we have discussed the last couple of lectures. We have seen that the dependences can be broadly divided into 2 categories data dependences and control dependences. And the data dependences again can be divided into 2 broad categories. First one is a true data dependences and which leads to read data type of hazards. And we have discussed various techniques by which you can overcome these true data dependences by using hardware and software means we have seen we can use hot providing. You can use instruction scheduling, static instruction scheduling, dynamic instruction scheduling by hardware. And by that you can overcome read after write type of hazards arising out of true data dependence.

Similarly, we have discussed about name dependences having 2 different verities; one is known as output dependences. And second one is known as anti dependences and output dependences lead to read after write type of hazards. And anti dependences lead to write after read type of hazards and these 2 types of hazards can be overcome by using resistor

in a main. And we have seen how resistor in a main can be done by the compiler or by the hardware as it has been done in thamosilous algorithm. So, this is how the data dependences tackled and hazards are arising out of data dependences can be overcome by different techniques so far we have concentrated on data dependences and overcoming the hazards arising out of data dependences. Now, we shall focus on control dependences we have seen control dependences lead to control hazards and in simple terms.

(Refer Slide Time: 03:13)

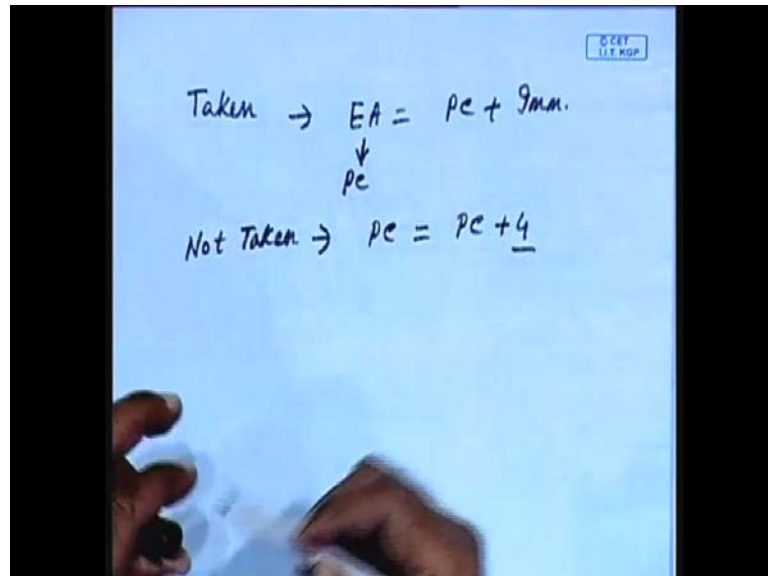


We can discuss about control dependence, we can tell about control dependences in this way control hazards also occurred due to instruction changing the program counter. We have seen the program counter keeps track of the instruction to the executed next that is program counter holds the adds up the next instruction. And this particularly when there are branches this program counter has to may not be known immediately. And I has been count that control hazards cause a better performance loss than do data hazards.

So, data hazards sometimes leads to some losses we have to introduce loss, but it has been count that control hazards are more and leads more performance loss. So, we have to focus attention to control hazards. And we have to see how they are impact can be minimized they are the loss can be reduced we know that a branch can be can have 2 outcomes. Number one is known as taken, another one is not taken that means whenever

you have got a branch instruction there are 2 possibilities in case of taken you have to generate a new address that means in case of taken branches.

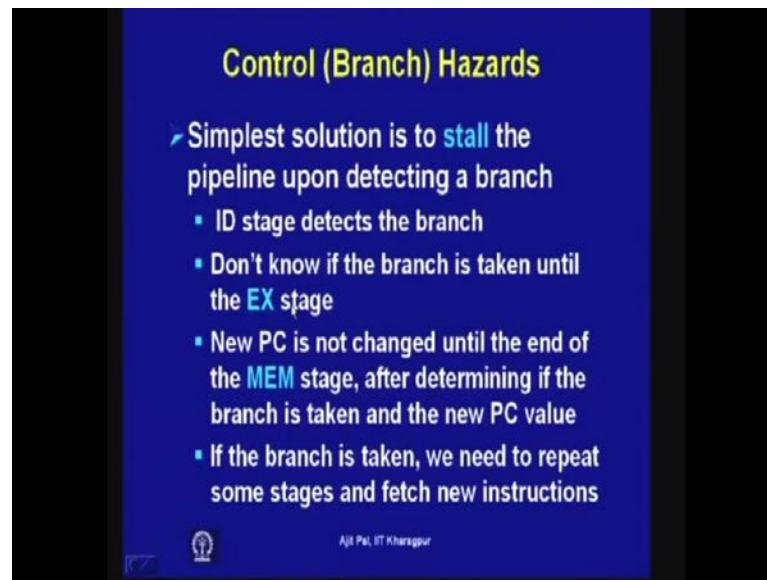
(Refer Slide Time: 04:50)



Effective address you check the address is equal to program counter plus that immediate data that is available as the part of the instruction added with the program counter and effective address is generated. And this is the address where it will be generate the instruction execution fruit starts this program counter has to be loaded by this effective address basically has to be loaded by this part. Another possibility is that not taken that means this the branch may not take the condition may not be satisfied in such case the program counter is essentially the address of the next instruction.

As we know it is equal to P C plus 4 because instructions are 4 bytes. So, the next address is present whole of the program counter plus 4. So, this is the next address of the extraction, but so these 2 I mean when we shall know the new address the branch is taken. And we can know the address when branch is not taken. So unless these 2 are known we cannot face the next instruction from the till the value of P C is known. That is unless the new value is depending on whether it is whether the branch is taken or not taken, we cannot really proceed cannot fetch the next instruction and start execution.

(Refer Slide Time: 06:25)



Control (Branch) Hazards

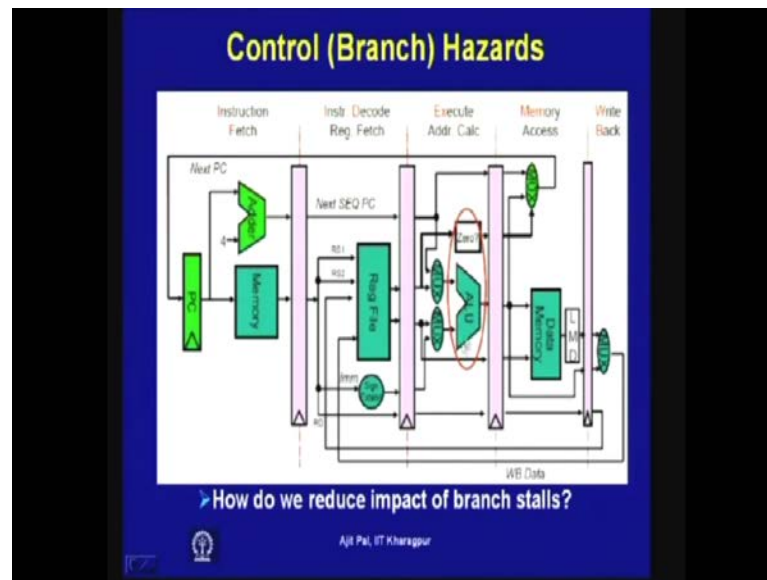
- Simplest solution is to **stall** the pipeline upon detecting a branch
 - ID stage detects the branch
 - Don't know if the branch is taken until the **EX** stage
 - New PC is not changed until the end of the **MEM** stage, after determining if the branch is taken and the new PC value
 - If the branch is taken, we need to repeat some stages and fetch new instructions

Ajit Pal, IIT Kharagpur

Now, let us see, what are the solutions? The first solution or the simplest solution is to install the pipeline upon detecting a branch. That means as soon as detected a branch what you can do it will install the pipeline and wait till the branch address is known. So, that is the simplest solution and the steps are given here the ID stage detects the branch. That is after the instruction is decoded it will be known whether it is a branch instruction or not and do not know if branch is taken until the execution stage. So, as we shall see in our pipeline we have to go to the execution stage, because there the condition for branch tested and it will be known whether branch will be taken or not.

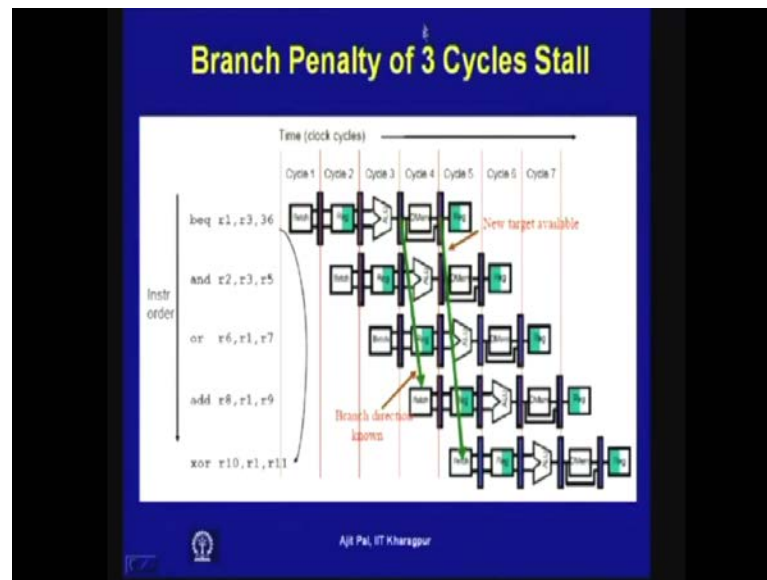
And then the new PC is not changed until the end of the memory stage is. That means until we go to the memory stage we do not know what will be the new address if branch is taken. It means after determining the branch is taken and new PC value is known in the, me memory state. And if the branch is taken we need to repeat some stages and fetch new instructions; that means that that fetching the new address. It was taken by place from consecutive addresses like PC plus 4, PC plus 8 those things are first out those instructions are first out. And you have to fetch it instructions from the new address and that is how it will continue.

(Refer Slide Time: 08:05)



As it is clear from this tag emergency the condition in this simple pipe line that we have discussed x 59 it checks whether a particular resistor content is 0 or not another mention it is done at the at the execution state. So, the condition is known only at the execution state and where the branch will taken place you can see the address is known in the memory access stage. So, in the memory access stage the address will be known and that content will be loaded in to the program counter. So, we have to wait till the memory accessed stage to know both the things condition whether satisfied or not. And the branch address, whether if the branch is taken so what will be the delay?

(Refer Slide Time: 08:56)



Obviously, the delay it leads to delay of 3 cycles so you have to call the pipeline by 3 cycles in the normal situation. So, you can see here whenever the branch is taken. And you can see the after the execution stage the branch condition is decided and after the memory stage the new target has been known. So, these are the instructions fetched this one; this one and this one that means after this `beq r1, r3, 36`. You know these this particular the instructions which are following these remaining these 3 instructions `and r2, r3, r5`, `or r6, r1, r7` and `add r8, r1, r9`. These instructions; obviously, are to first out, what do you really mean by that you can say fortunately none of these 3 instructions have done any permanent damage or permanently change in the status of processor.

And that will be take place only in the light back stage the content of the register will be modified only then the permanent change in state is done. So, you can see the before that happened the work the condition and the new end this is known. So, what you have to do all these 3 instructions are to be nullified by committing them into no operational instructions. And obviously, there will be no change, but we shall be losing the pre cycles. And the instructions fetch will take place if the branch is taken place where the branch is taking place. And I mean this when will be known that it will execute this or it will jump to this instruction I mean there is a address thirty 6 where it will jump. So, you can say the, this is how it will happen. So, we shall be losing 3 cycles. Now, the question will arises where that it is possible to reduce the number of stalls that means whenever

the branch instruction encountered. We are finding that if we do not use any complicated technique. If we simply introduce stalls then we shall lose 3 cycles for each encounter of a branch, so let us see.

(Refer Slide Time: 12:34)

Impact of Branch Stalls

- If CPI = 1, 30% branches
 - Stall 3 cycles \Rightarrow new CPI = $(1 + 0.3 \times 3) = 1.9!$
 - If 50% of these branches taken \Rightarrow new CPI = $1 + 0.15 \times 3 + 0.15 \times 2 = 1.7$
- Penalty would be worse for current-day (longer) pipelines
 - IF and ID-like stages are each of multiple-cycle
- How do we reduce impact of branch stalls?
- Two part solution:
 - Determine branch taken or not **sooner**
 - Compute taken branch address **earlier**

Ajit Pal, IIT Kharagpur

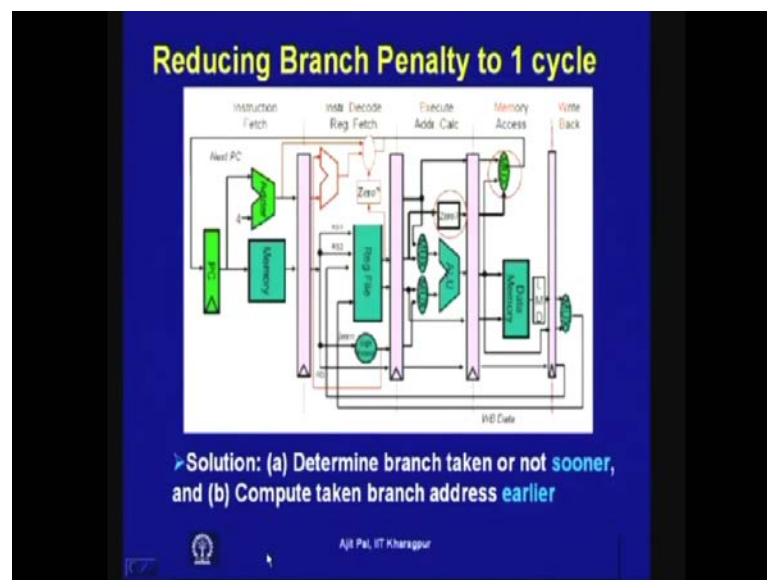
What will be our loss impact of branch stalls? So, let us assume your idea CPI is equal to 1. And let us assume that 30 percent of the instruction by branches; remaining 70 percent instructions are value operations. So, since there are 3 is a stall of 3 cycles. So, new CPI is equal to 1 plus 0.3 into 3 is 1.9. Of course, we have not considered this situation that you know all branches may not be taken here. We have assumed that as soon as a branch instruction is encountered 3 stalls will be introduced. But that is not really necessary even for the simple 5 stage that I have discussed. Because you see the branch can be whether the branch will be taken or not taken is known as the execution stage if it is not taken. Then obviously, it is not necessary to I mean wait for the next cycle, because branch cycle will not be taken.

So if branch is not taken then the loss will be of 2 cycles. For example, if 50 percent of this branches that taken then the new CPI will be 1 plus 1 is the 2 that is real situation then 15 percent of the cases branch is taken. So, in such a case the loss will be of 3 cycles will be because address will be known only at the end of the memory stage and so 0.15 into 3. And then whenever branch is not taken for the 15 percent of the cases the loss will be of 2 cycle. So, if you add up find that there in new CPI will be 1 percent not

1.9. Now, this penalty would be what is for current definition? If this will be the case for this simple pipeline that we have considered. But in the modern processors now a day's for example, even the instruction exchange or instruction decode stage is divided into several stages.

So, in such a case loss will be more the number of clock cycles or number of stalls that will be happening will be more but in our case it will be restricted to 3. So, how do we reduce the impact of branch stalls? Question is there, any way by which you can reduce the impact of branch stalls there are 2 part solution, first of all you have to determine branch taken by or not taken sooner. So, if you can find out that branch will be taken or not earlier in the pipeline stage. If you can make some arrangement for that then there is a possibility of some gain. Similarly, if we know the branch address earlier even then we can have some gain in performance or we can reduce the impact of this branch stalls. So, these 2 things had to be done determine branch taken or not sooner and compute taken branch address earlier. So, these are the things to be done and there can be a solution for this. So, let us see what is the hardware solution?

(Refer Slide Time: 15:07)

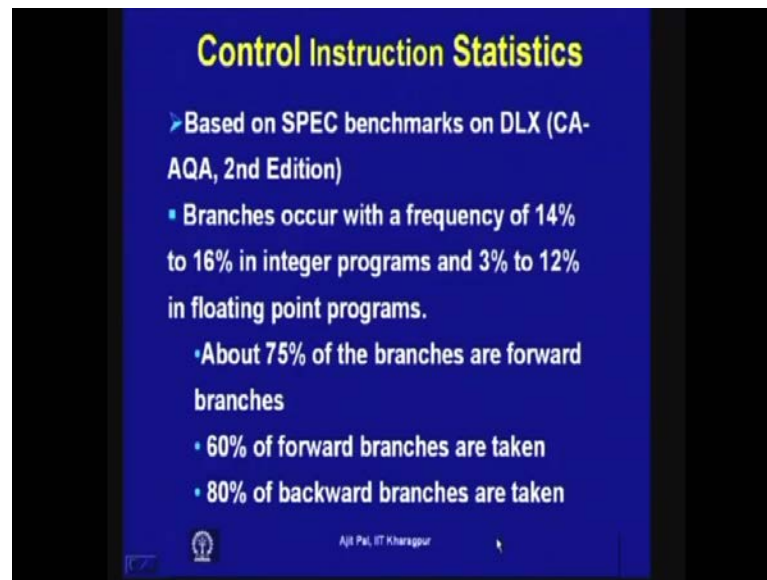


What can be done you can see that that detect at 0 degree detector which checks whether particular resistor is 0 or not is in the execution list, it can be moved to the instruction decode stage you can see it has been moved to the instruction decode stage. And not only that this multiplexer which was in the memory access stage can be allowed to be moved

to the instruction decoder stage. So, if we do that however if we wanted to do that you will require an additional hardware that is an adder. You will require an adder earlier this addition to generate an effective address you know you have to generate an effective address $PC + \text{immediate value}$. So, these values has to be calculated earlier this was done with the help of the ALU which is available in the processor. Now, if you want to move it to the to the V S stage then we will require an additional adder which actual perform the effective address calculation earlier in the instruction decode stage. So, you find that if you can add this; you can shift this hardware means this multiplexer along with an additional adder and this 0 detector to the instruction and decode stage.

Then we find that what will what is the outcome of this that means invoke the condition and the branch address are known in the second state itself we do not have to go to the fourth stage. So, the loss or the penalty is reduced only to 1 cycle, because in the instruction decode stage both will be known. And accordingly depending on the outcome either the next instruction can be fetched from $PC + 4$ or from the branch address which is not that means program counter will be loaded by $PC + 4$ or by the effective address which is calculated in the third cycle itself instead of waiting for the fifth cycle. So, you find that the, these 2 solutions can be easily accomplish with the help of additional hardware. And so this is how this can reduce the branch penalty to 1 cycle. So, after this we shall assume that the simple pipeline that we are discussing the branch penalty is 1 cycle. That means we shall assume that this change has been made in the hardware and our branch penalty now, 1 cycle.

(Refer Slide Time: 18:04)



Control Instruction Statistics

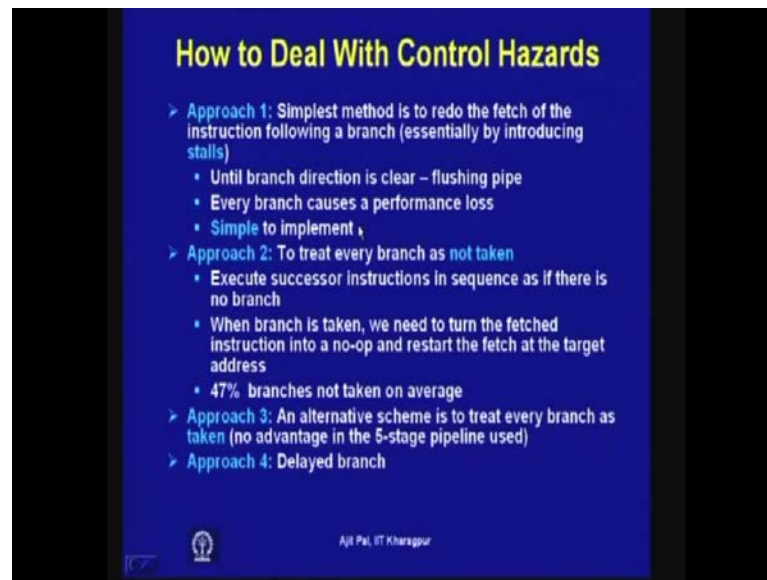
- Based on SPEC benchmarks on DLX (CA- AQA, 2nd Edition)
 - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
 - About 75% of the branches are forward branches
 - 60% of forward branches are taken
 - 80% of backward branches are taken

Ajit Pal, IIT Kharagpur

Now, here some statistics about the control instruction based on the SPEC benchmark on this DLX processor, it is taken from that computer architecture that is second addition book that and quality of approach computer architecture and quality approach from that particular book. And branches it has been found that this statistics is like this branches occur frequency of 14 to 16 percentage in integers programs and 3 to 3 percent to 12 percent in floating point programs So, this is the branch frequency the rate at which branch instructions countered in a program. And this is more in integer programs than in floating point terms and another statistics about whenever a branch is encountered. It has been found that about seventy 5 percent of the branches are forward branches.

So, the branch can take place in the forward direction in which it is increased or it can take place in the backward direction when the address is decreased. So, 75 percent of the branches are forward branches and more over 60 percent of the forward branches are taken. And 80 percent of the branches are taken actually you may be asking why 80 percent of the backward branches are taken. Why it more, the reason for that is you know this is because of loop in case of loop it goes back to a fetching instruction. So, because of that looping you know the percentage of backward branches taken is more so this is the statistics.

(Refer Slide Time: 19:59)



Now, let us consider techniques by which we can deal with control hazards what are the different techniques that we can adapt first techniques here I mean we have already discussed some hardware solution by which we can reduce the number of stalls Now, the number of approach available is 4 first step approach is simplest method is to redo the fetch of the instruction following the branch So, this is essentially by introducing stalls so what is been done in this case until the branch direction is known until the address is calculated So, you continue to ask the pipeline and so every branch causes a performance loss. So, whenever a branch occurs you simply introduce a stall in simple pipeline the fortunately number stalls has been reduced from 3 to 1.

So, whenever a branch instruction is encountered a stall is introduced so every branch leads to 1 instruction loss. So, this solution is very simple, simple to implement, because you do not have to check anything after the instruction is decoded. If it is a branch you introduce a stall then you proceed to the next cycle where both the condition is known whether the condition is satisfied or not is known. And also the target address has also be known whether it is $PC + 4$ or $PC + \text{some immediate value}$ which is the part of instruction. So, obviously this approach is not accepted if you are interested in improving the performance. So, what do you mean by the second approach? Second approach is to treat every branch as taken that means the compiler assumes that branch will be always taken. So, if the branch is always taken what will be done the address, the next instructions will be fetched.

I mean I am sorry in this case branch is not taken so treat every branch is not taken that means it always assumes that branch is not taken. So, when the branch is not taken obviously; the next instruction to be executed is $p + 4$. So, it proceeds in that direction so executes successor instructions in sequence as if there is no branch. And however this also I mean whenever this assumption is made it this is simply an assumption. It does not mean that branch will I mean all the branches will not be taken some branches will be taken. So, what will be done in such cases? So, when branch is taken, we need to turn the fetch instruction into a no op and restart the fetch at the target address. So, this is the thing we have to do whenever a prediction that is done by the compiler or the assumption is made by the compiler terms of 2 rounds. And it has been found that 47 percent of branches are not taken on an average.

So, in 47 percent of the cases you know there will be no need to any modification. So, perform they will be no performance loss for the 47 percent of the cases, but for the remaining 63 percent of the cases there will be some performance loss. Because we have up to we have to I mean we have to fetch an instruction I mean we have to convert the already fetched instruction into a no op. And we have to restart the fetch at the target address.

So, this is the situation when assumption has been made that branch is not taken now third approach is an alternative scheme is to treat every branch as taken. So, in such a case what is being done it is assumed that all the branches are taken that assumption is made. But unfortunately even for the simple pipeline we have seen that branch address is known only after only in the execution stage when the when already the weather branch will be taken or not taken is also known. So, as a consequence for the simple pipeline that we have discussed there is no gain no advantage.

So, this approach has been no advantage for the 5 stages pipelining discussing however there is a some performance gain whenever 2 whenever the branch address is not taken. So, as now there will be another approach which is known as delayed branch shall see how the instruction following the branch can be converted into a useful instruction normally. We have seen if the prediction is wrong then we lose one cycle that the instruction which was executed that has to be conveyed to known up. So, that we can overcome with this particular thing we can execute an instruction and it is not necessary to converted into a known off. So, that is known as delayed branch so we shall discussed

these zones techniques one after the other. Of course, the first technique nothing to discuss I have already mentioned that you have to simply processes introduce a stall after detecting a branch instruction in the instruction decode stage. So first approach has nothing we do not need further discussion to consider first step approach.

(Refer Slide Time: 26:41)

Approach 2: Predict-Not-Taken

		Clock Number								
Instr.	1	2	3	4	5	6	7	8	9	
i(T)	IF	ID	EX	MEM	WB					
i+1		IF	idle	idle	idle	idle				
T			IF	ID	EX	MEM	WB			
T+1				IF	ID	EX	MEM	WB		
T+2					IF	ID	EX	MEM	WB	

- Execute successor instructions in sequence
- PC+4 already calculated, so use it to get next instruction; chances are the branch is not taken
- "Squash" instructions in pipeline if branch actually taken
- Can do this because CPU state not updated till late in the pipeline

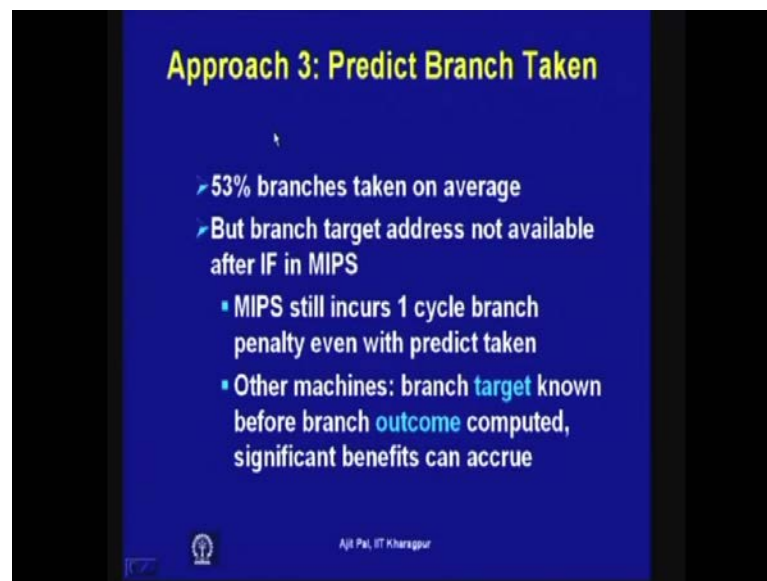
Ajit Pai, IIT Kharagpur

Let us now focus on approach 2 predict not taken. So, who is predicts things who is decides here; obviously, the prediction has been done by the compiler. So, compiler is assumed to be in that the branch is not taken. So, in such a case you can execute successor instructions keep on fetching plus 4 c p plus 8 is the one and one executing however in this p c plus 4 already calculated. So, use it to get the next instruction chances are the branch is not taken. So, whenever branch is not taken as we have seen we have to we have to modify the instruction. And that is why it is been done and that if branch is not taken the following instruction you have to squash instructions in the pipeline if branch is actually taken.

So, this if the ith instruction is a branch instruction and prediction was not taken and unfortunately it has turns over to be. So, when a branch is taken in such a case this i plus 1 is instruction we have to introduce stall here so one cycle is lost as you can see. And of course, the next instruction is taken since it is a second instruction it is from the branch target address the instructor is fetched and then execution continues. So, this is the approach to where predict not taken is done. And of course, this particular thing can be

easily done as I have already explained. Because CPU state is not updated till the locate in the pipeline. We have seen that CPU state is updated only in the later part of the cycle that is in the right the stage that is your modifying the register that means permanent change you are making. So, before that if the decision is known if your prediction is wrong there is no problem you can knob of course, there is a loss, but that has to be accepted.

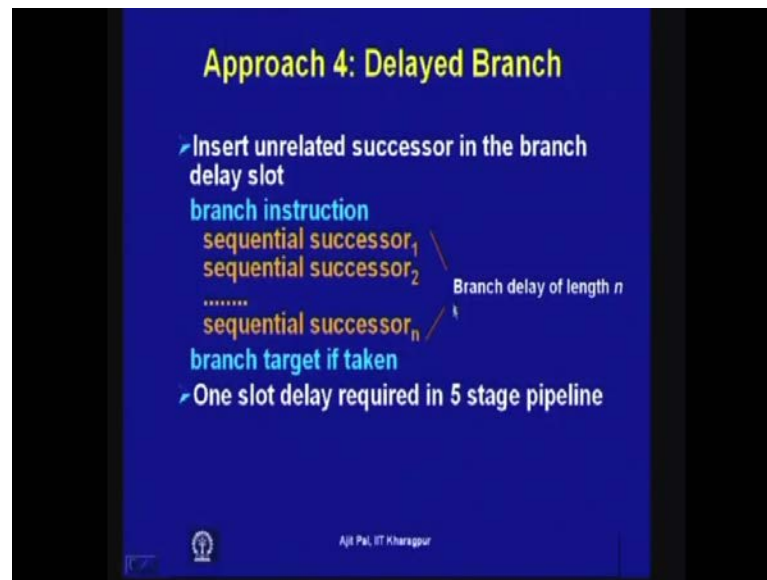
(Refer Slide Time: 28:58)



And let us take a considering the, what happens. So, 53 percent branches taken on an average, but branch target address is not available. So, here it is predict branch taken the second approach 53 percent branches are taken in on an average. But branch target address is not available after instruction fetch in MIPS so MIPS is still incurs 1 cycle branch penalty even with predict taken. So, as I have already mentioned for this simple pipeline there is no benefit for this prediction this assumption.

However there are machines where branch target is known before branch outcome is computed. So, there are processors where this thing this particular situation exist branch target is known before the branch outcome is computed. In such a case significant verification can accrue, because there may be processors where in the execution state both the target is known and the value is computed. So, in such cases you know there can be something, but not for the pipeline that we have discussed.

(Refer Slide Time: 30:20)



Now, we shall focus on the forth approach that is delayed branch we have seen we have a branch instruction.. Following that there are several sequential instructions and this is the target address where the, if the branch is taken it jumps to this address. So, in between the successive instruction there is a I mean and branch target address branch target if the instruction branch is taken you have got several instructions. So, these are known as sequential success of the branch instructor.

And these instructions are considered to be in the branch delay slot. So, here you have got branch delay of length n so there is a there are a instruction in this branch delay slot. However, for this simple pipeline, that have that we have already discussed there is a only one slot delay required in 5 stage pipeline. So, you have already seen that the in the delay slot the branch delay slot has got only one instruction. And so in general there they can gain instructions but in our simple case only one slot delay is required.

(Refer Slide Time: 31:53)

Delayed Branches

Machine code sequence:
Branch instruction
Delay slot instruction(s) _____ Branch is taken
Post-branch instructions _____ (if taken) at this point

Instr	Clock Number								
	1	2	3	4	5	6	7	8	9
i(T)	IF	ID	EX	MEM	WB				
D(i+1)		IF	ID	EX	MEM	WB			
T			IF	ID	EX	MEM	WB		
T+1				IF	ID	EX	MEM	WB	
T+2					IF	ID	EX	MEM	WB

✓ Instructions in the branch delay slot (s) get executed whether or not branch is taken

Ajit Pal, IIT Kharagpur

Now, we are interested in filling up that particular detail in the slot. And so this is the branch instruction; this is delay slot instructions and this is the post branch instructions. So, here this is the target now this d I plus 1; this is the delay slot and this instruction we have up to we have to fill up with the some instruction so later is useful. So, instructions in the branch delay slot get executed whether or not branch is taken. So, the point that you have to understand is that whether the branch is taken or not taken the instruction following the branch instruction is also follows the executed but if your prediction is wrong. I mean if the branch is taken then you have to nullify it but that instruction will always get executed. So, based on this observation, we can think about some solution which will help in proving the performance of the processor.

(Refer Slide Time: 33:11)

Delayed Branch

- **Simple idea:** Put an instruction that would be executed anyway right after a branch

Branch

Delay slot instruction

Branch target OR successor

- **Question:** What instruction do we put in the delay slot?
- **Answer:** one that can safely be executed no matter what the branch does.
 - The compiler decides this.

Ajit Pal, IIT Kharagpur

So, the simple idea is put an instruction that would be executed anyway right after a branch. So, this is the delay slot this is the branch instruction; this is the delay slot and this is the branch target or the successor of the instruction. Now, question is what instruction do we input in the delay slot? So, we have to put an instruction in the delay slot with some objective what is objective one that can safely be executed. No matter what the branch does that means whether the branch is taken or not taken that instruction can be executed? And it will not lead to any I mean you do not have to be convert into a knob that is the basic objective. And the compiler decides this compiler has to decide which instruction to put in this delay slot and there are several approaches.

(Refer Slide Time: 34:14)

Delayed Branch

- One possibility: An instruction from before
- Example:

- The DADD instruction is executed no matter what happens in the branch:
 - Because it is executed before the branch!
 - Therefore, it can be moved

Ajit Pal, IIT Kharagpur

One possibility is an instruction from before an instruction can be taken from before these branches. So, here is a delay slot DADD then see there are several solutions.

(Refer Slide Time: 34:38)

An Instruction from before

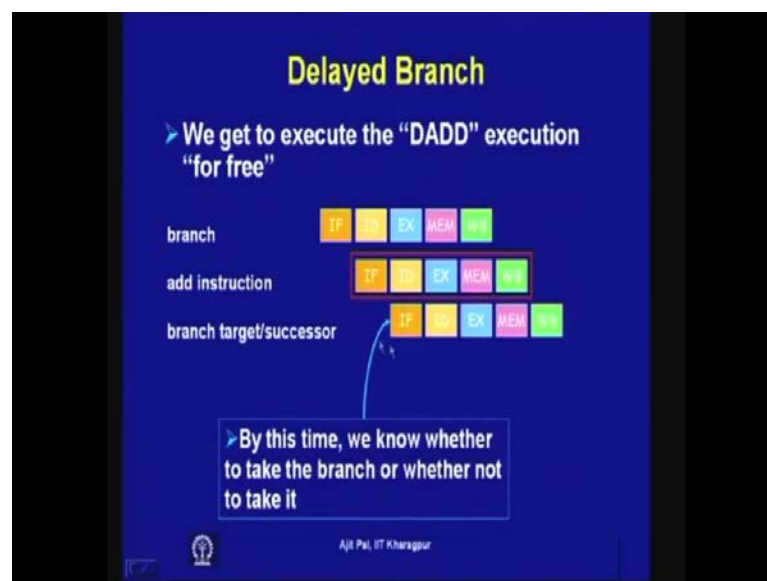
SECRET
IIT KGP

First solution is an instruction from before by this from before we mean you have your branch instruction. And so let us consider DADD R 1, R 2, R 3. This is an instruction before this branch instruction if R 2 equal to 0 then it will jump to this; this is targeted address. And this is the targeted delay slot what we are doing this instruction you can see this; this is the normal instruction execution floor. So, this instruction is executed after

that this branch instruction encountered. So, this instruction will be executed whether this branch instruction is taken or not. Now, if we move this to this slot that means we are converting it in to in this way if r 2 is equal to 0. Then and we are filling up this slot with DADD R 1 R 2 R 3 and then it is it will go to this.

So, this delay slot is filled up with an instruction from before it was here now you have moved it to here. So, we find that as we know this instruction will be executed irrespective of whether of branch instruction taken or not taken. And this instruction was supposed to be executed before this branch instruction. So, the here also this instruction is executed so the branch is whether a branch is taken or not taken there is no loss. So, we are able to put an instruction which is useful you do not have to convert into an knob if prediction is wrong. So; obviously so this can be moved to this delay slot and this will this is the possibility solution.

(Refer Slide Time: 36:55)



So, you get to execute the DADD execution for free. So, as if we are getting this instruction executed free of cost. Free of cost means some instructions are supposed to be executed here we are taking an instruction for the branch. So, as we go to the next instruction by that time will know where the branch is taken or not taken and also the target address. So this instruction can be either it will can be the p c plus 4 or it can be that p c plus that immediate value which is available in the as the part of the instruction. In case there is a I mean the branch is taken so we find that which is the best solution.

(Refer Slide Time: 37:49)

Delayed Branch

- Another possibility: An instruction from target
- Example:

```
DSUB R4, R5, R6
...
DADD R1, R2, R3
if R1 == 0 then
    DSUB R4, R5, R6
```

➤ The DSUB instruction can be replicated into the delay slot, and the branch target can be changed

Ajit Pal, IIT Kharagpur


This is the best solution that we can have and this is the preferred approach we can follow in our, for filling up instructions in the branch delay slot. Now, what is the second possibility? The second possibility is that an instruction from target means where it is jumping so it is if R 1 is equal to 0 then it is jumping to these d sub R 4 this instruction. Now, what can be done? This instruction can be replicated here it can be replicated here in this delay slot. And then we can change this branch target to get address that means you can you can loading. So, that it will be pointing to the in extension, because I is getting executed here. This particular thing can be done whenever in most of the situations your prediction is that branch will be taken then this is very advantageous. So, for filling up the delay performance by an instruction from the target and by doing this you are able to improve the performance. But improvement of performance takes place when branch is taken.

(Refer Slide Time: 39:15)

Delayed Branch

- Yet another possibility: An instruction from inside the taken path
- Example:

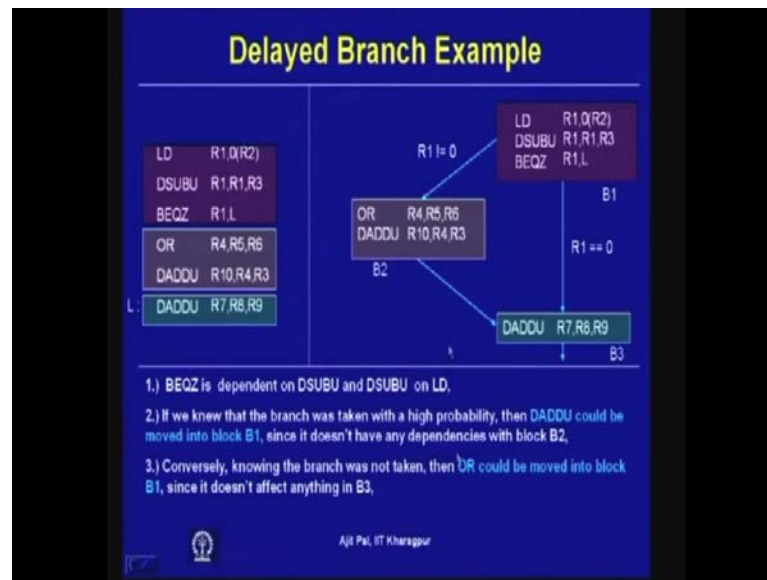
```
DADD R1, R2, R3
if R1 == 0 then
    delay slot
    OR R7, R8, R9
DSUB R4, R5, R6
```


- The OR instruction can be moved into the delay slot **ONLY IF** its execution doesn't disrupt the program execution (e.g., R7 is overwritten later)

Ajit Pal, IIT Kharagpur

So, yet another possibility is an instruction from inside the taken path. So, here you are taking it from inside the taken path means you can see here is the delay slot. And this particular instruction or can be moved into that delay slot only if its execution does not disrupt the program execution. So, from the taken path you, you have taken, you have to filled up the delay slot by using his instruction. So, this is an another approach you can follow.

(Refer Slide Time: 40:00)

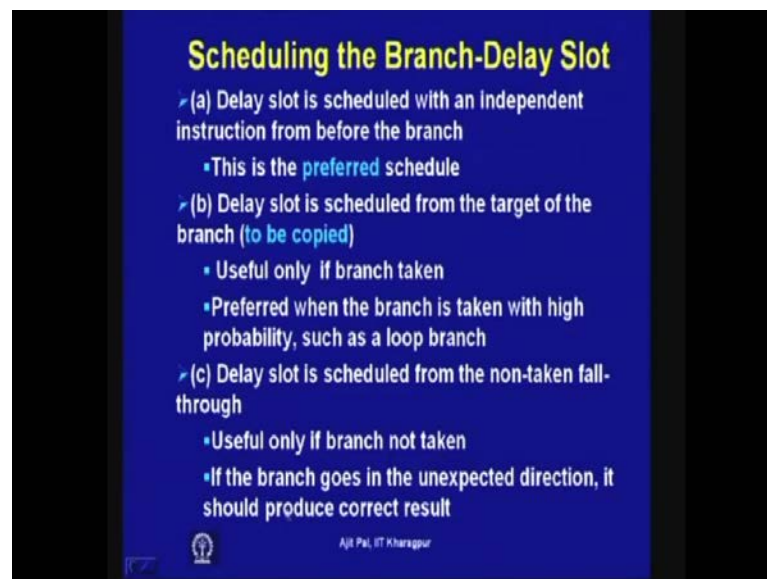


Now, let us see an example here we have got 3 parts so this is a delay loop load R 1 comma 0 R 2 DSUBU R 1 comma R 1 comma R 3 BEQ R1 comma 1. Now, you can see and 1 is the target address is this. And this is the essentially the delay slot I mean where you have to put your instructions that means following your weeks. We have to fill up the there is a delay slot where you have to put these instruction in this instruction this particular code sequence. First thing that you can do is to take this instruction DSUBU R 1 comma R 1 comma R 3 after this unfortunately this cannot be done. Because BEQZ is dependent on this, because of this dependency we cannot move this instruction to after this branch instruction. So, this first approach which is the first approach cannot be followed here.

Now, what are the other alternatives, if we know that branch was all over taken within a high probability. Then DADDU could be moved into the block B1 that means if the, you see this is this is target address. So, we can take this instruction from the target and put it immediately after this instruction. I mean immediately after this BEQZ R 1 1 that means the branch instruction and if we know that the branch was taken with high probability. So, if you if you have high probability of branch taken then this particular approach is followed since it does not have any dependencies on block 2. Since there is no dependency it can have I mean this instruction I mean there is no dependence of the instruction on this. So, it can be moved without any without any problem, but this solution is will be good whenever branch is taken with high probability.

Now, what are the third possibility? Third possibility is that knowing the branch does not was not taken then or could be moved into block B. So, this instruction that is the whole through that that approach that I told is the whole through. So, this instruction can be moved I mean can we move to the block immediately after a following this and or could remove this block B 1 since I does not affect anything in B 3. So, we can see we have got 3 possibilities whenever we can fill out by depending on different situations we can do that and this particular example, illustrate various alternatives possible.

(Refer Slide Time: 43:16)



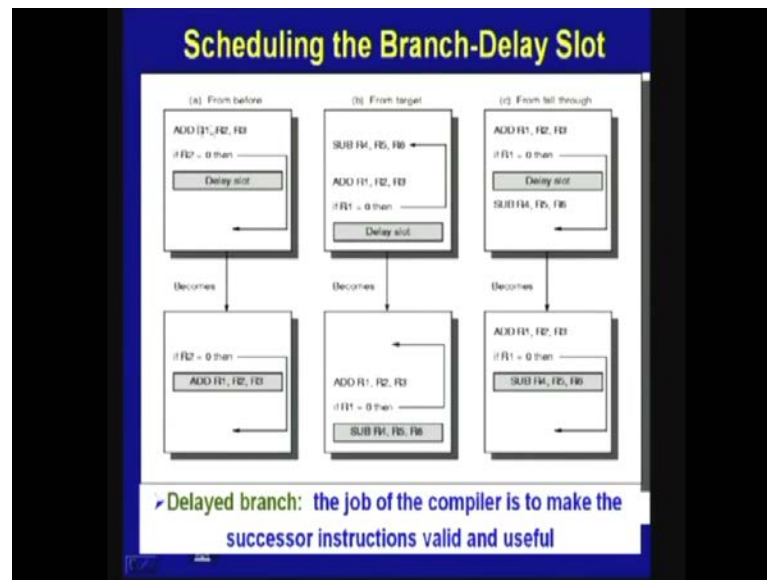
Scheduling the Branch-Delay Slot

- (a) Delay slot is scheduled with an independent instruction from before the branch
 - This is the preferred schedule
- (b) Delay slot is scheduled from the target of the branch (to be copied)
 - Useful only if branch taken
 - Preferred when the branch is taken with high probability, such as a loop branch
- (c) Delay slot is scheduled from the non-taken fall-through
 - Useful only if branch not taken
 - If the branch goes in the unexpected direction, it should produce correct result

Ajit Pal, IIT Kharagpur

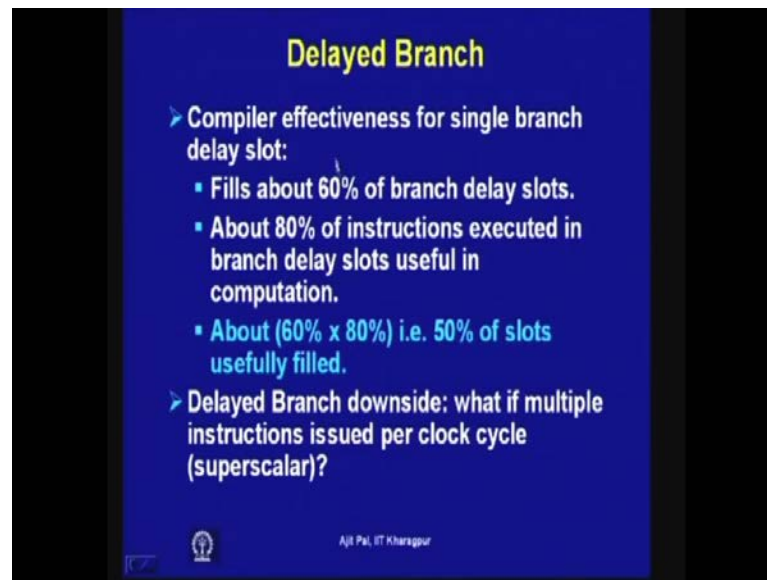
So, we can summarize the scheduling of branch delay slot possibility first one is delay slot schedule with an independent instruction from before the branch. So, this is the preferred schedule I have already mentioned second is delay slot is scheduled from the target of the branch you have to copy and an instruction. And this is useful only branch is taken and this is preferred when the branch is taken with high probability such as a loop branch. So, in case of loop branch, we have seen branch is taken and probability is larger so in such a situation this is this is this is double. And third possibility is have already told delay slot from the non taken fall through so non taken non through this is useful if branch is not taken. And if the branch goes in the another un expected direction it should produce correct result. So, these are the 3 possibilities by which we can fill up the branch delay slot. And we have seen this is how the performance can be improved.

(Refer Slide Time: 44:31)



And this particular diagram summarizes the 3 possibilities; first one is from before that means this instruction which is before this instruction can be filled up here. And this is a second approach from the target; this is the target address; this is the, from target. And we will copy this in the slot and change the direction I mean this pointer value where this branch target address is there. That address has to be modified from the fall through this is filled up with this instruction `SUB R 4 R 5 R 6`. And as we have seen this particular approach is suitable when there is probability of branches not taken. So, the basic objective is of this compiler, the job of the compiler is make the successor instructions valid and useful allayed branch slot. And this is the philosophy that has been used to fill up the.

(Refer Slide Time: 45:37)



Delayed Branch

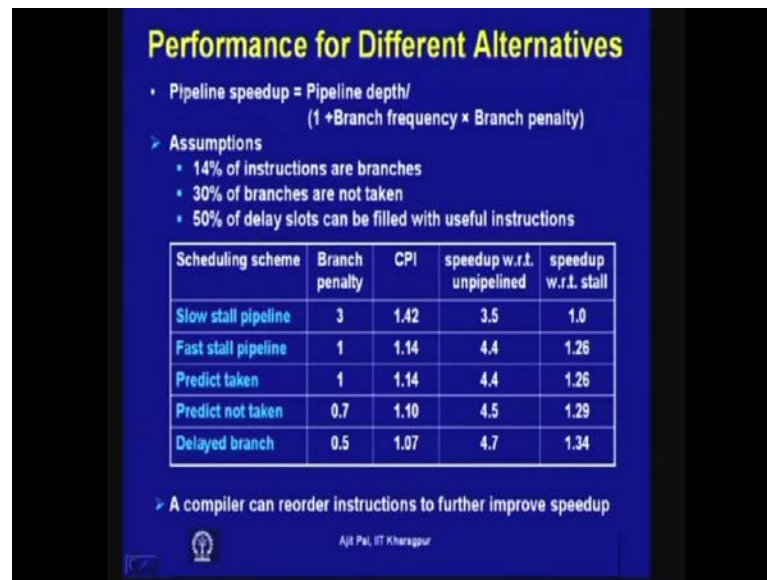
- Compiler effectiveness for single branch delay slot:
 - Fills about 60% of branch delay slots.
 - About 80% of instructions executed in branch delay slots useful in computation.
 - About (60% x 80%) i.e. 50% of slots usefully filled.
- Delayed Branch downside: what if multiple instructions issued per clock cycle (superscalar)?

Ajit Pal, IIT Kharagpur

Now, let us have some statistics compiler effectiveness for single branch delay slot. So, it has been found that it fills about 60 percent of the branch delay slot about 80 percent of the instruction executed in the branch delay slot useful in computation. So, that means out of 60 percent of instructions which are I mean branch delay slots are filled up 80 percent is the, is useful in computation. So, that means that in 50 percent of the cases this slots are usually filled usually fully filled. And so in another words we can tell that there is a 50 percent improvement you know improvement compared to whenever we assume you consider the first step follow the first step approach where you introduce a slot.

Now, we have not consider a very important aspect of delayed branch downside is what is multiple instructions issued per clock cycle. That means whenever it is supers tailor process we have considered the simple situation it is a pipeline processor so branch delay slot. We are issuing one instruction at a time and that delay slot has to be filled up with one instruction that is it is a superscalar processor. Then it is necessary to issue 2 or 3 or 4 depending on the degree of the superscalar processor. So, many instructions we should and so many instruction has to be filled up in the delay slot. So, the task of the compiler are becomes I mean it is difficult in such situation for a superscalar composition.

(Refer Slide Time: 47:37)



Performance for Different Alternatives

- Pipeline speedup = Pipeline depth / (1 + Branch frequency × Branch penalty)
- Assumptions
 - 14% of instructions are branches
 - 30% of branches are not taken
 - 50% of delay slots can be filled with useful instructions

Scheduling scheme	Branch penalty	CPI	speedup w.r.t. unpipelined	speedup w.r.t. stall
Slow stall pipeline	3	1.42	3.5	1.0
Fast stall pipeline	1	1.14	4.4	1.26
Predict taken	1	1.14	4.4	1.26
Predict not taken	0.7	1.10	4.5	1.29
Delayed branch	0.5	1.07	4.7	1.34

- A compiler can reorder instructions to further improve speedup

Ajit Pal, IIT Kharagpur

Now, here the performance for different alternatives are given here. So, pipeline speedup is equal to pipeline depth by 1 plus branch frequency. And branch penalty you see we have that performance is dependent on 2 things number 1 is branch frequency. And second is branch penalty and it has been found that this branch frequency varies from instruction from program to program. And since the depending on the branch frequency your improvement that will take place is depend not only upon branch frequency. But also on the branch penalty that branch penalty that will take place is dependence not dependence the approach that we are following that means technique that we are adopting to improve the performance means. So, that branch penalty will be dependent on that so first part that branch frequency is dependent on the program. And second part branch penalty is dependent on the approach that you are following.

So, based on these assumption that fourteen percent of the instructions are branches. So, that means branch frequency is 14 percent and 30 percent of the branches are not taken. And assuming that 50 percent of the delay slots can be filled with useful instructions it useful instructions based on these assumptions. This is the result for different situation first is slow stall pipeline where we assume that the branch penalty is 3 cycles. So, whenever you have got a branch penalty of 3 cycles you get a CPI of 1.41. And speed up with respect to un pipelined is 3.5 ideally it should be 5, what you get 3.5 So, that is the speed up with respect to un pipelined whenever your, you have to it incurs number of stalls is 3 and speed up with respect to stall that means whenever of course, in

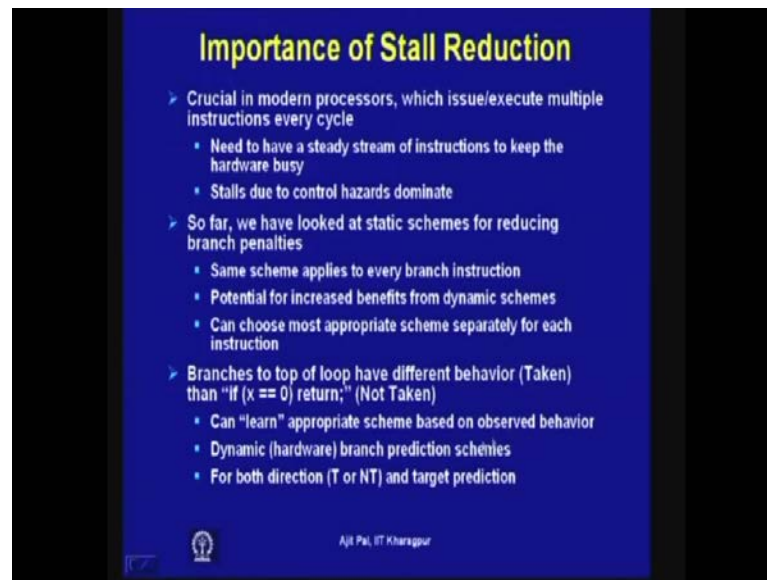
this case it will be 1. So, whenever we are adopting the technique of introducing stall with respect to stall performance speed up is 1.

So, with these let us compare the other techniques first one is fast stall pipe line. First stall pipe line means we have used additional hardware to reduce the number of stalls. And as we have discussed the branch penalty can be reduced to 1 from 3 as we do that this CPI series. CPI improves from 1.42 to 1.14. So, there is significant improvement in CPI and we find that speed up is 4.45 4.4 it is there is a significant increase from 3.5 to 4 point. And speed up with respect to stall approach define approach is 1.26 then if we consider the third case the prediction is branch taken. And we have send when for our simple pipeline there will be always loss of one cycle. So, in this case if it is effectively same as the second approach. So, there is no performance gain as a, which quite obvious. So, the first stall pipeline and predict taken is given the same performance that means if it remains 1.14.

So, in this case also there is loss of 1 cycle. And in this case also there is always a loss of 1 cycle so CPI remains 1.14 but this speed up to un pipelined also remains same. And speed up with respect to stall is 1.26 that also remains the same and predict not taken in this case the branch penalty is 0.7. We have seen there can be 50 percent of the cases it can be filled up with full instructions. So, branch penalty is 1.7 and CPI is 1.10. So, we find that CPI is improving compared to the previous case and there is consequent improvement in speed up with respect to un-pipelined.

And also there is consequence speed up with respect to stalls whenever we take up the approach of stall is 1.29. And last but not least is that delayed branch approach in which case the branch penalty is 0.5 and CPI is 1.07. So, it is very close to 1 1.07. And speedup is also very good 4.7 with respect to the ideal un-pipelined and this speedup with respect to stall is 1.34. So, here we call we call it static branch prediction the prediction is done with the help of a compiler. And a compiler can be lot of instruction to further improve speedup that we have already discussed.

(Refer Slide Time: 53:10)



And later on we shall consider another approach particularly important, because of the importance of stall reduction the crucial in modern processors which issue and execute multiple instructions per every cycle. So, need to have a steady stream of instructions to keep hardware busy stalls due to control hazards dominate. So, this importance of stall reduction is very important. And so far we are looked at static schemes for reducing branch penalties and same scheme applies to every branch instruction. That means what do you mean by steady static? Static means if there are hundred branches, for all the hundred branches you are adopting the same policy because it is done by the compiler steady static.

However there is potential for increased benefits from dynamics schemes, what do you mean by dynamic scheme here you know dynamically at when the while instruction execution is progress for a particular branch prediction can be not taken for another. Branch prediction can be taken so it will dynamically keep on changing as you as the instruction execution takes place. And that is done at execution time with the help of a hardware. So, it can choose appropriate scheme separately from the each instruction.

So, the branches to top of loop have different behavior taken or not taken. And can learn appropriate scheme based on observer behavior and dynamic branch prediction scheme can be used for both direction taken or not taken and target prediction. So, in my next lecture I shall discuss in detail this dynamic technique that means dynamic prediction

schemes can be used. And we shall see how the performance is be improved by adopting to dynamic technique.

Thank you.