

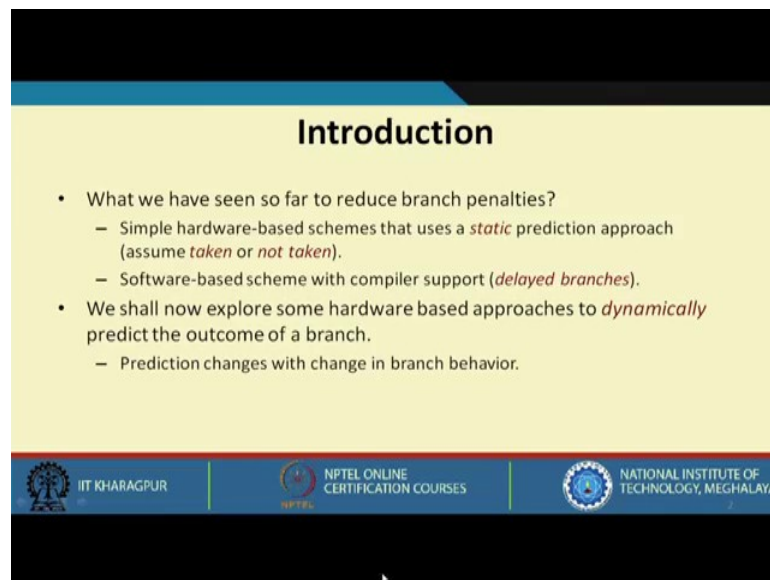
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 58
Pipeline Hazards (Part 4)

We continue with the discussion on control hazards. If you recall what we were discussing in the last lecture, we looked at various schemes with which we can reduce the branch penalty. We were making some static predictions, assuming beforehand branches taken or not taken, or we were relying on the compiler to move codes around and give the pipeline a better code to execute, which will generate less number of stalls. But both these approaches are static in some respect, which means whatever we are deciding or predicting or assuming that is statically done once.


Now, we look at some approaches where we try to exploit the dynamic behavior of the loop of a branch instruction to improve or enhance the performance with respect to control hazards even more.

(Refer Slide Time: 01:43)



Introduction

- What we have seen so far to reduce branch penalties?
 - Simple hardware-based schemes that uses a *static* prediction approach (assume *taken* or *not taken*).
 - Software-based scheme with compiler support (*delayed branches*).
- We shall now explore some hardware based approaches to *dynamically* predict the outcome of a branch.
 - Prediction changes with change in branch behavior.



In our earlier lectures we have seen that in order to reduce the branch penalties, we were using some kind of static approach. Either you are doing some prediction, or we are relying on the compiler to try and fill up the branch delay slots with some useful

instructions. We shall now discuss purely hardware based approaches to dynamically predict the outcome of a branch.

The prediction can change with time. Like you see in a program whenever there is a branch depending on the program some of the branches can be taken most of the time, some other branches can be not taken most of the time. So, you really cannot say statically beforehand that all branches are mostly taken or mostly not taken.

(Refer Slide Time: 03:17)

Branch Prediction Buffer (BPB)

- The BPB is a small high-speed memory (like cache) that is indexed by the lower few bits of the branch instruction address.
- 1-bit prediction scheme:
 - The memory contains the target address of branch, and also a bit that says whether the branch was recently taken or not.

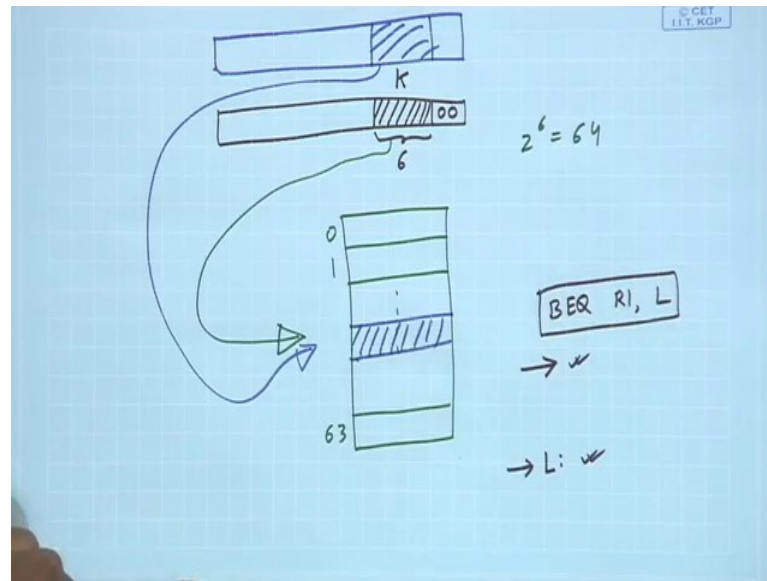
The diagram illustrates the BPB structure. A pink box labeled 'k-bit Branch address' is connected to a vertical stack of memory cells. Each cell contains a 'Prediction' bit (green) and a 'Predicted branch address' (blue). A blue arrow points from the 'k-bit Branch address' to the 'Prediction' bit of a selected cell. Another blue arrow points from the 'Predicted branch address' of the same cell to the right. A vertical ellipsis indicates multiple cells in the buffer.

Logos at the bottom: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, NATIONAL INSTITUTE OF TECHNOLOGY KHARAGPUR, and a portrait of a man.

Depending on your program, a branch can be either mostly taken or mostly not taken, or it can be 50-50. The first approach we talk about uses a hardware data structure called a Branch Prediction Buffer.

BPB is a small high-speed memory, and this is the address of the branch that is shown in pink. Now you recall in MIPS32, every instruction starts with an address that is a multiple of 4.

(Refer Slide Time: 04:06)



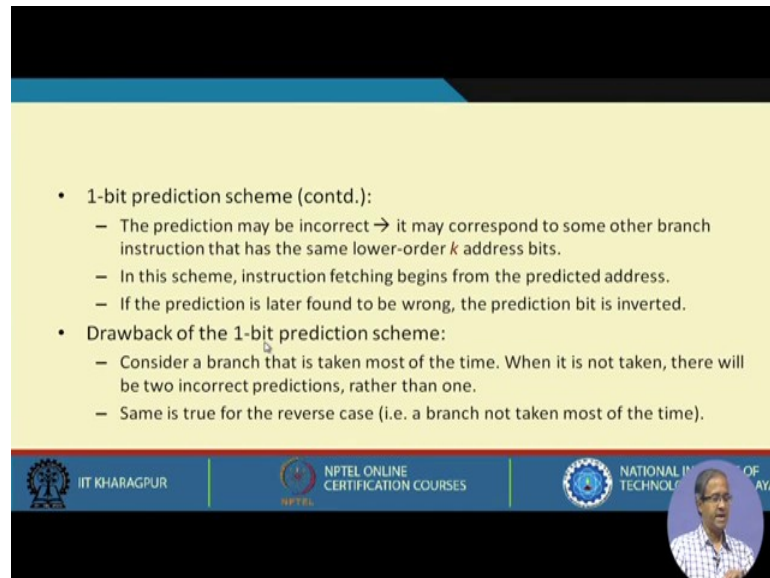
Thus whenever you have a 32-bit address the last 2 bits will always be 0; this means multiple of 4, then you have remaining 30 bits. Here you are selecting a few bits from the lower side, say k number of bits. Value of k is a design parameter, this can be 4 5 6 whatever, let us say 6 bits.

Our approach will be something like this. These k bits of the branch address are used to access this table BPB. BPB is a small high-speed memory that is indexed by the lower few bits of the instruction. If $k = 6$, $2^6 = 64$. I will be having a small memory with 64 entries. These few bits will be used as the address to access this memory. So, BPB is indexed by these k bits, and what is stored in this memory? Here the predicted branch address is stored. The predicted branch address can be either the address of the next instruction or the address of the target.

When this instruction is executed if it is not taken branch, next instruction executed will be the next one. If it is a taken branch next instruction executed will be the one from the label L. So, this is one possible branch address, this is the other possible branch address. The predicted branch address is stored here. The idea is that you check the last few k bits, you look up this table and see what is stored here. And whatever you stored here that is your predicted branch address from where you start fetching the next instruction. And this green one you have some additional bits stored. In the 1-bit prediction scheme

we have single bit with every entry, this will tell whether the last prediction was a taken branch or a not taken branch. Let us say 0 means taken and 1 means not taken.

(Refer Slide Time: 07:24)



The slide contains the following text:

- 1-bit prediction scheme (contd.):
 - The prediction may be incorrect → it may correspond to some other branch instruction that has the same lower-order k address bits.
 - In this scheme, instruction fetching begins from the predicted address.
 - If the prediction is later found to be wrong, the prediction bit is inverted.
- Drawback of the 1-bit prediction scheme:
 - Consider a branch that is taken most of the time. When it is not taken, there will be two incorrect predictions, rather than one.
 - Same is true for the reverse case (i.e. a branch not taken most of the time).

The slide footer includes logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY DELHI, along with a small video inset of a speaker.

In this 1-bit prediction scheme you see just one thing. Whenever you encounter a branch instruction you check these k bits, go to the index and also check the prediction because you will also come to know later whether the branch is actually taken or not taken. You can compare with the prediction whether this is matching. If it is matching then you know you are correct.

But there are some problems. This 1-bit prediction scheme means the prediction may be incorrect because the value you're getting from the BPB may correspond to some other branch instruction that has the same lower order k address bits.

There can be some other branch address for which accidentally these k bits may be identical. So, they will both point to the same location. You will be trying to get the same entry here in the BPB and say that this will be your target branch address; obviously, maybe this entry was for this one. Now by mistake for the other one also you map here. So, that will be a wrong prediction. This sometimes a prediction may be incorrect and here the instruction fetching will begin from the predicted address, and during execution later on well for MIPS32 as I said at the end of ID you will come to know whether your prediction is right or wrong.

At the end of ID you will be knowing whatever you have taken from BPB was a right prediction or a wrong prediction. If you see that your prediction is wrong, you can appropriately change the prediction bit in the BPB, and instruction fetching will begin from the predicted address. Now the drawback here is that suppose I have a branch instruction that is taken most of the time. When it is not taken there will be 2 incorrect predictions rather than 1, why? Let us look at a scenario. Suppose I have a loop that is executing 100 times. So, for 99 of the time it will be taken branch, but for the last time it will be coming out of the loop, that will be a wrong prediction.

So, out of this 100 the last time we will be getting one wrong prediction. So, the prediction bit will be the appropriately inverted. So, in BPB that entry was showing as taken next time when you again go back and enter that loop, maybe this is a nested loop. Next time when you enter that loop again the first iteration of that loop we will there will again be a miss prediction, because last time you had set the prediction bit to not taken because it has come out of the loop, but next time when you enter the first time there will be a loop again. So, there will be a wrong in prediction. So, there will be 2 wrong predictions for every execution of the loop.

(Refer Slide Time: 11:41)

Example 1

- Consider the following nested loop where the inner loop iterates for 20 times before exiting.

```
for (loop=0; loop<1000; loop++)  
{  
    .....  
    for (i=0; i<20; i++)  
        A[i] = A[i] + 5;  
}
```

- For every execution of the inner loop, there will be two mispredictions:
 - Last iteration of the current loop.
 - First iteration of the next loop.
- Though the branch is taken 95% of the time (19 out of 20), correct prediction occurs only 18 times.
 - 90% accuracy.

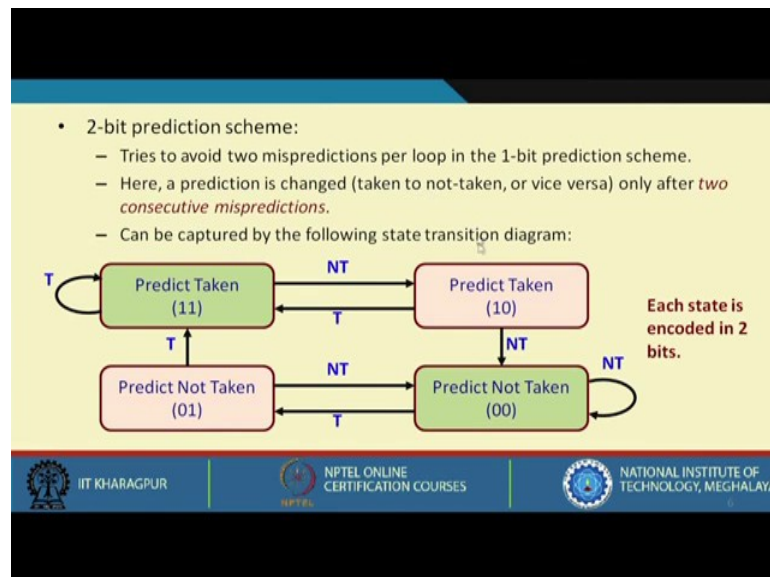
The slide footer includes logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY DELHI, along with a portrait of a man in a blue shirt.

Let us take an example. Here I have a nested loop. The inner loop executes 20 times and this is inside an outer loop. I am looking at the inner loop, which is executing 20 times. We are saying that for this inner loop there will be 2 mispredictions, last iteration of the

current loop. So, when $i = 19$ last time it will be coming out of the loop, but for i equal to 0 to 18, it will be taken, but for $i = 19$ it will be not taken. But when it comes back and enters the loop again for the next loop of the outer loop, again it will start with $i = 0$ and last time it was a not taken prediction.

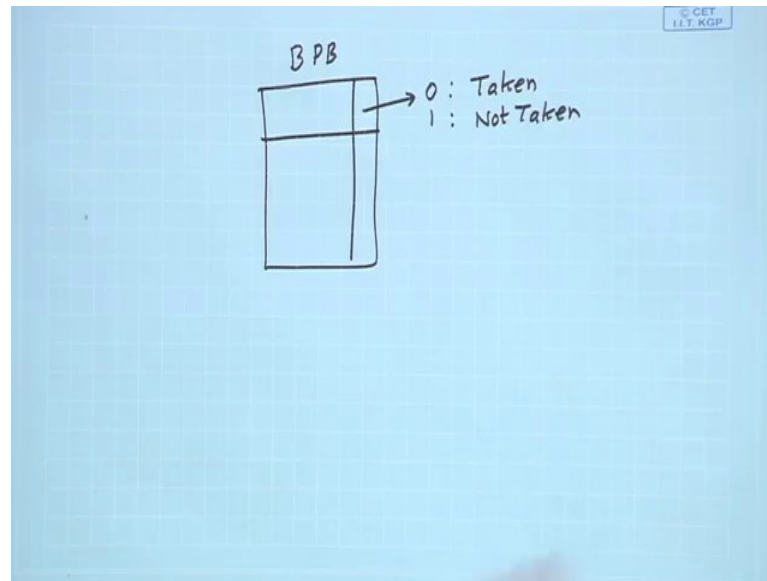
So, with not taken for $i = 0$ it will try to take the branch. It will be a misprediction again. So, actually though this branch is taken 95% of the time in reality, 19 times it is taken 1 time it is not taken, but this one bit prediction scheme provides correct prediction 18 times and twice it is mispredicting so, it is having 90% accuracy.

(Refer Slide Time: 13:19)



This is one drawback of this 1-bit prediction scheme. To avoid it you can have a 2-bit prediction scheme because the trouble with the earlier scheme was that every time there was a misprediction, I was flipping that bit in the branch prediction buffer BPB.

(Refer Slide Time: 13:33)

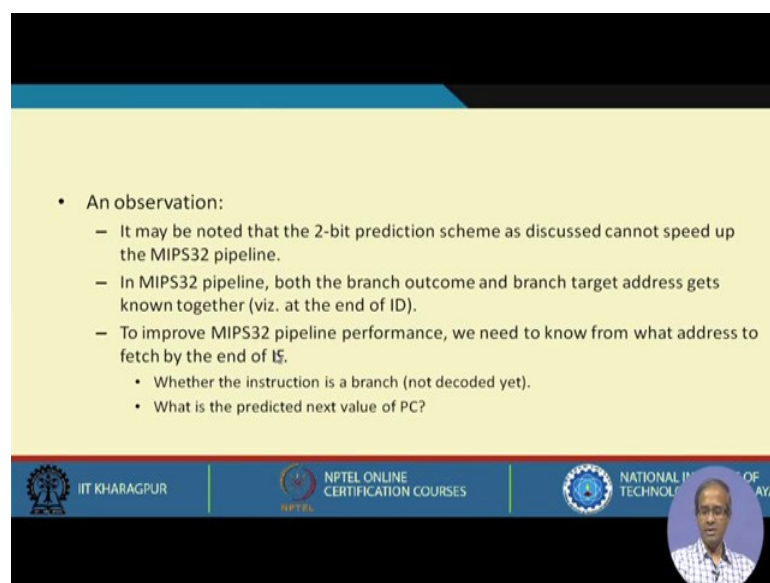


Here we were flipping this bit as soon as there is a misprediction, we are not taking this fact into account just like the earlier example showed, that most of the practical scenarios will be like this and if you follow this immediate flipping policy there will be 2 mispredictions in every loop. Our modified strategy will go like this. We are saying that, we will not change the prediction every time there is a misprediction we will be waiting 2 times. If there are 2 consecutive mispredictions then only we will be changing the prediction information. If we do it then if you think of the earlier case then 19 of the times we can have correct prediction. Here we try to avoid the 2 mispredictions for loop in the example that I have shown for the 1-bit prediction scheme, let us see here in the 2 bits scheme. I am showing a state transition diagram.

Four states are shown, the green states are the stable states, 0 0 indicates predict not taken, 1 1 indicates predict taken. While you are here if your loop you are seeing taken you remain here, and while you are here if it is not taken you remain here. But if it is predict taken, if you get a not taken misprediction, you do not straight away go here, but rather you come to an intermediate state, this is still predict taken, but maybe. So, here if you see that next time again there is a misprediction then only you move here. Which means that your behavior of the loop has possibly changed earlier it was mostly taken, now somehow it has become mostly not taken.

But if you see that the next time it is again taken we again come back here. Similar is the case here if it is not taken and if you find it is a taken a misprediction you temporarily come here; this is again predict not taken, but not green may be case. If it is again a taken; that means 2 consecutive mispredictions then you permanently move here and if it is again a not taken you again come back here. Since each state is encoded in 2 bits we call it a 2-bit prediction scheme. Here the prediction is changing only after 2 consecutive mispredictions. In general this will give better behavior as compared to the 1-bit prediction scheme.

(Refer Slide Time: 17:09)



• An observation:

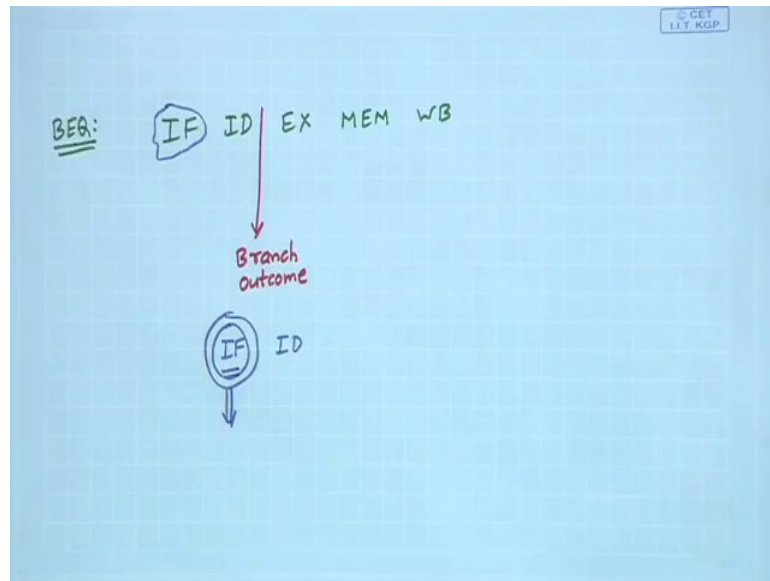
- It may be noted that the 2-bit prediction scheme as discussed cannot speed up the MIPS32 pipeline.
- In MIPS32 pipeline, both the branch outcome and branch target address gets known together (viz. at the end of ID).
- To improve MIPS32 pipeline performance, we need to know from what address to fetch by the end of IS.
 - Whether the instruction is a branch (not decoded yet).
 - What is the predicted next value of PC?

The slide footer contains logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY KANPUR, along with a small portrait of a man in the bottom right corner.

Now, the thing is that for a MIPS32 pipeline, really you do not gain much with this because in MIPS32 pipeline because of the simplicity of the instruction encoding, we will come to know everything at the end of ID anyway. We will know whether the branch will be taken or not taken, we will also know the address of the target at the end of ID itself.

So, whether we are using 1-bit prediction or 2-bit prediction really does not help much for MIPS32 because we have to wait till the end of ID to come to know that whether our prediction was correct or not. For MIPS32 pipeline we need to do something more.

(Refer Slide Time: 18:18)



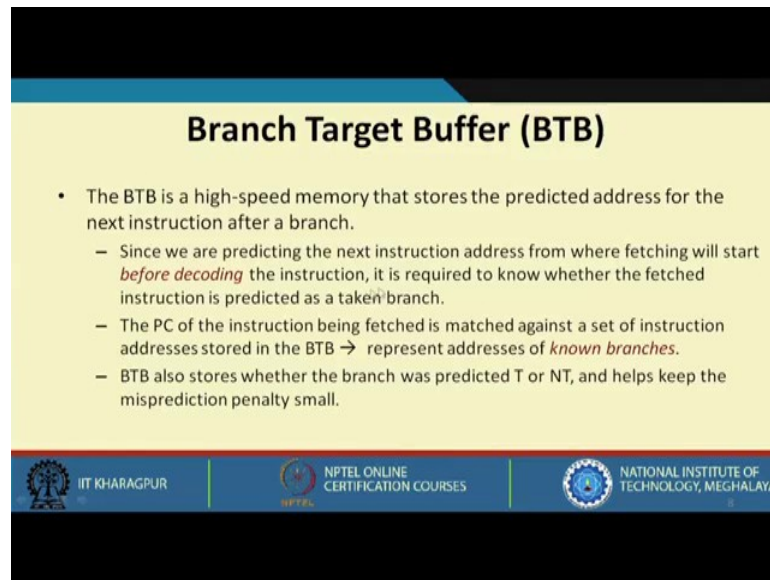
If you look at the pipeline stages again, it was IF, ID, EX, MEM and WB. Suppose this is a branch instruction, what I am saying is that it is only at the end of ID you come to know about the branch outcome precisely. So, all your decision will be here, but because you will be fetching the next instruction immediately here, but branch outcome is known later, maybe this fetching will be wrong.

What we are trying to do here is that: can we do something in the IF stage itself so that fetching can start in the immediately next cycle with some predictive accuracy? Because this is the branch instruction we cannot know until ID is over anyway, but here we are saying that we do not know this is the branch. You see there are 2 things. Firstly, I am saying that the instruction unless you decode it during the ID stage, you will not know that it is a branch instruction; and if you do not know it is a branch instruction, then all this things become meaningless -- it can mean add instruction also, but what you can do better is you apply a little intelligence. Suppose I maintain the memory address of the instructions I know that are branch.

So, whenever I am fetching an instruction I am comparing the value of the PC from where I am fetching (say 1000) with the address of a known branch instruction that I had seen earlier. There will be another table. I compare, if I see that 1000 is there, I will know that this is a branch instruction. So, I can start my manipulation during IF itself. I will not have to wait till ID. This is the philosophy we will be following now.

This last statement actually talks about to improve the MIPS32 performance we need to know from what address to fetch by the end of IF; whether the instruction is a branch and what is the predicted next PC. We want so many things earlier before the instruction is decoded.

(Refer Slide Time: 21:42)



Branch Target Buffer (BTB)

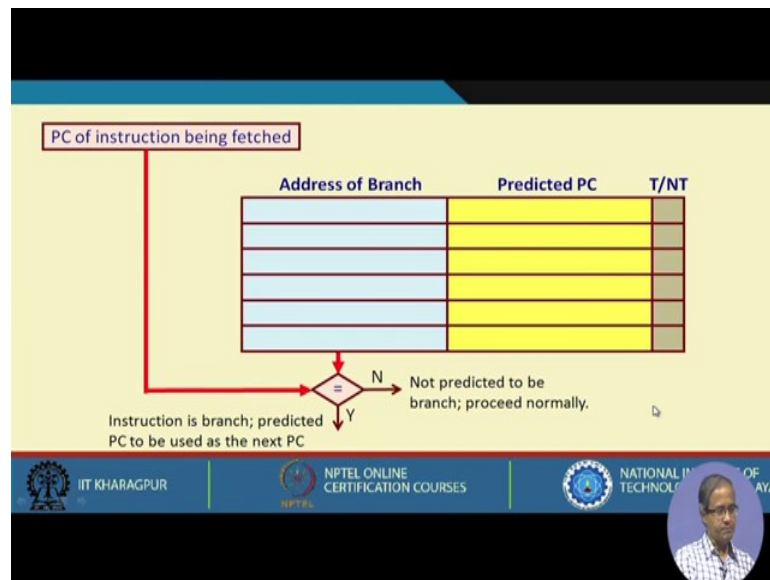
- The BTB is a high-speed memory that stores the predicted address for the next instruction after a branch.
 - Since we are predicting the next instruction address from where fetching will start *before decoding* the instruction, it is required to know whether the fetched instruction is predicted as a taken branch.
 - The PC of the instruction being fetched is matched against a set of instruction addresses stored in the BTB → represent addresses of *known branches*.
 - BTB also stores whether the branch was predicted T or NT, and helps keep the misprediction penalty small.

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

We use something called branch target buffer. BTB is a high-speed memory, which stores the predicted address for the next instruction after branch. Whenever there is a branch instruction from where you will be fetching the next instruction; is it the next instruction following or is it the target instruction that will be your predicted next instruction that will be stored there. Since we are predicting the next instruction from where we are fetching before the decoding, we will also have to know whether fetch instruction is predicted as a taken branch or not.

Just as I said the PC value from where you are fetching the instruction that needs to be stored in that memory. Because that will tell me whether that is a known address of a branch instruction.

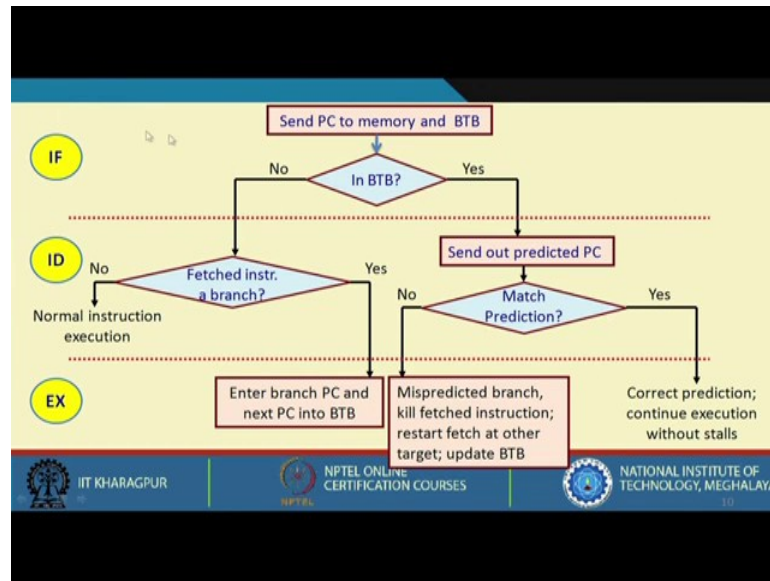
(Refer Slide Time: 23:16)



First you see this is how the table looks like. There are actually three parts; the first part is the address of the branch instruction this is the PC value, then the predicted next address. While you are fetching an instruction, this is actual an associative memory we will have to search this parallely. When an instruction is being fetched you parallely check whether this PC value is already present here or not. If we see that it is already present, it means that the instruction is a branch instruction.

And if it is a branch instruction, then the corresponding predicted PC value, that you will be using as your next PC from where to fetch the instruction. But if you see that it is not matching, then it is not predicted as a branch, you proceed normally. And also there is a third field, taken or not taken. You also keep this so that if there is a prediction mismatch, you can update this. So, the PC of the instruction being fetched is matched against the set of instruction addresses stored in the table.

(Refer Slide Time: 25:28)



So, this is the data structure and this is how you are comparing. How you will be using it now let us see. This is the flowchart, which tells you what happens during the IF stage, what happens during ID, what happens during EX. Now in the during the IF stage as you can see your sending the PC to memory for fetching the instruction, parallelly you are also sending it to BTB for searching. While it is being fetched, you parallelly check whether it is in the BTB or not. Suppose you find yes, which means it is a branch instruction; you send out the predicted PC. At the end of the ID the instruction that has been fetched we will also have completed the decoding process. So, then you also know whether your prediction and the decoding of the instruction is matching or not. If you see that the predictions match you continue execution without any stalls.

But if you see that prediction is wrong because your table says that it is taken, but after decoding you see that it is not taken; that means, it is a mispredicted branch. Then you will have to kill the instruction you have fetched. You need a stall here, and you will have to restart fetch at the other target, and once you do this you will also have to update BTB because now your prediction has changed.

So, you will have to modify this T and NT, and also update the predicted PC, and here if it is not in the BTB it can mean 2 things, that it is either not a branch instruction or it is a branch instruction, but you are seeing it for the first time. So, whether or not it is a branch instruction, that again you will come to know during ID. If you see that it is a

branch instruction then you will have to enter this information into BTB. Because you are seeing the branch instruction for the first time, but if it is not it is a non-branch instruction you proceed normally to the execution.

(Refer Slide Time: 28:10)

• There will not be any branch penalty if an entry is found in the BTB and is correct.

- Otherwise, there will be penalty of at least one clock cycle.
- Since the BTB may also need to be updated, the penalty can be two clock cycles.

Found in BTB	Prediction	Actual Branch	Penalty Cycles
Y	T	T	0
Y	T	NT	2
Y	NT	NT	0
Y	NT	T	2
N	-	T	2
N	-	NT	1

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

This table shows the various penalties, you see if an entries found in the BTB and the prediction is correct prediction as true and the actual branch was true then this no penalty or it is not taken and not taken then also there is no penalty.

These are the assumed values and there is also justification.

(Refer Slide Time: 29:43)

Example 2

- Consider the following parameters of a MIPS pipeline with BTB:
 - 90% of the branches are found in BTB.
 - 8% of the predictions are incorrect.
 - 75% of the branches are taken.

Compute the branch penalty.

Delayed branch requires penalty of 0.5 clock cycles on the average

$$\begin{aligned}
 \text{Branch penalty} &= (\% \text{ Branches found in BTB} \times \% \text{ Mispredictions} \times 2) \\
 &+ (\% \text{ Branches not found in BTB} \times \% \text{ Taken branches} \times 2) \\
 &+ (\% \text{ Branches not found in BTB} \times \% \text{ Not-taken branches} \times 1) \\
 &= 0.90 \times 0.08 \times 2 + 0.10 \times 0.75 \times 2 + 0.10 \times 0.25 \times 1 = 0.32 \text{ clock cycles}
 \end{aligned}$$

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

There is a small example where you can work out based on these values.

There is a BTB, and 90% of the branches are actually found in BTB, and out of that 8% of the predictions are incorrect and the remaining 92% have correct. And 75% of the branches are actually taken. So, what will be the branch penalty?

The calculation of the branch penalty is shown, which is a weighted sum of the various possibilities.

(Refer Slide Time: 31:33)

Conditional Instructions

- Conditional instructions can help eliminate some branch instructions and hence also the corresponding branch penalties.
 - MIPS32 has a number of conditional instructions (e.g. conditional move).

An example code:
`if (X == 0) A = B;`
Assume:

- R1 holds X
- R2 holds A
- R3 holds B

Using branch
`BNEZ R1, L`
`ADDU R2, R3, R0`
`L:`

Conditional move
`CMOVZ R2, R3, R1`

Essentially, we converted the control dependence into data dependence.

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

And lastly you will look at some conditional instructions that also help in reducing branch penalties. In fact, MIPS has a number of such conditional instructions. So, what are condition instructions? There is something like conditional move, let us look at a justification why we need this. Consider a C code like this, this a conditional move.

So, with this we come to the end of this lecture. Over the last few lectures we discussed the various hazard scenarios in the MIPS pipeline and looked at many of the techniques that are used to detect and mitigate the effects of hazards. You shall see some other advanced issues later in some lectures, but for this lecture I think we are done and we can stop here.

Thank you.