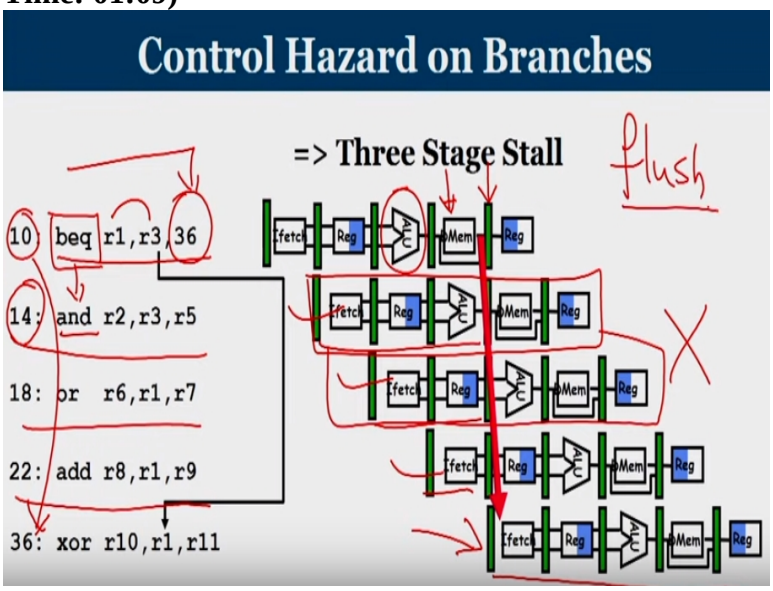
Advanced Computer Architecture
Dr. John Jose
Department of Computer Science & Engineering
Indian Institute of Technology-Guwahati

Lecture - 05
Control Hazards & Branch Prediction

Welcome to lecture number five of the course. In the last lecture the concept of pipeline hazards were introduced. We are continuing our discussion on the hazards itself with a special focus on how to handle control hazards and the technique of branch prediction, which has evolved as a popular field of research in computer architecture over the last few years.

Let us straightaway go into the topics of control hazard. A quick recap on control hazard, what we had discussed on the last lecture will be there initially to bridge the gap between the last lecture and this.

(Refer Slide Time: 01:09)



So, as we have seen in our last lecture regarding control hazard, consider a case that we have an instruction that has been given branch is equal to r1, r3, 36 the meaning of which is if the content of r1 and r3 are equal, then from 10 you are supposed to jump to line number 36. If the contents are not equal, then I will execute the follow-through instruction, the and instruction.

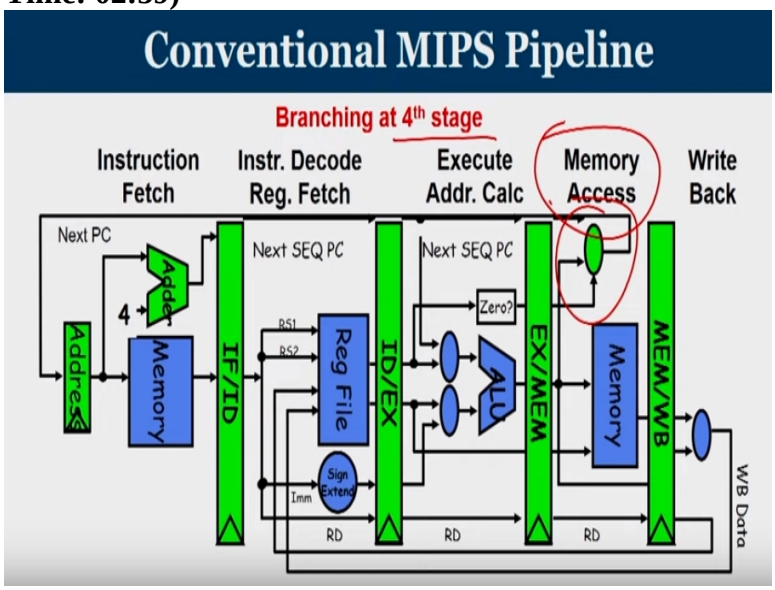
In a normal pipeline the comparison of r1 and r3 happens in a ALU. Only at the end of the MEM stage we will be able to decide whether control should be transferred to

36 or control should be transferred to 14. By the time you come to know about the outcome of the brands, already 3 instructions have already entered the pipeline. One has fetched and decoded and executed one has fetched and decoded and the third one is fetched.

Let us assume that the branch condition is taken, that means r1 value is equal to r2. So my execution now go to 36. So, here I am bringing the instruction that is stored in line number 36. We can see that already 3 instructions have entered the pipeline and these are nothing but the follow-through instruction from line number 14, 18 and 22 which are not needed at this point of time because they are not supposed to be the instruction that are to be fetched, because the instruction that is to be executed after this branch instruction is dependent on the outcome of branch.

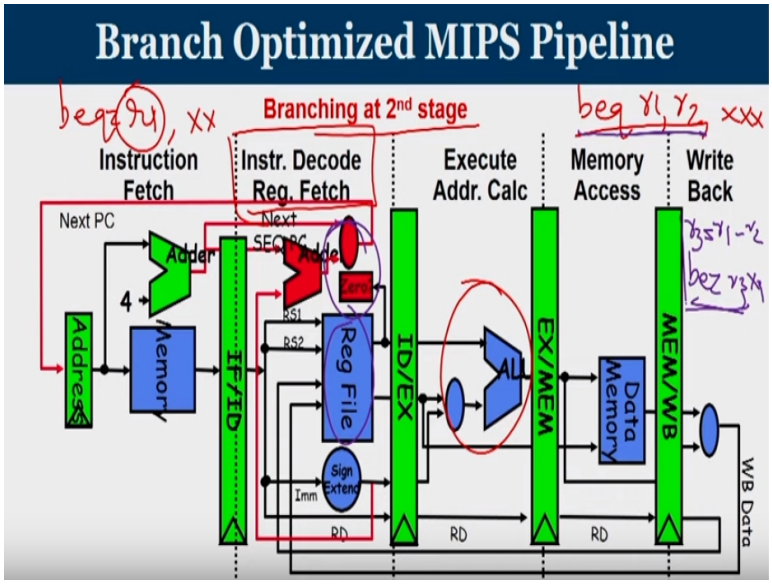
Since we are fetching 11 instruction each in the subsequent cycles the follow-through instructions are being brought which later we realize they are not needed. So in this case, we have to flush out the already brought instruction.

(Refer Slide Time: 02:59)



So, generally our branch instruction happens in the fourth stage. At this stage only we will come to know what are or where we have to jump. Should we jump into the target instruction or should we continue with the adjacent instruction.

(Refer Slide Time: 03:15)



But one optimization can be made such that branch can be resolved in the second stage itself. If branch condition is not in the second stage, then only one instruction has entered the pipeline and flushing of that one instruction is rather easy. So, if we go to the conventional branch statement, where we are dealing with branches only at the end of the fourth cycle, already three instructions are there in the pipeline.

So, if the outcome of the branch is not in expected line, then 3 instructions has to be flushed out. Flushing of 3 instructions is a more complex task than flushing a one instruction which will generally occur if the branching is done in the second stage. So, how can we do a branching in the second stage? We cannot compare the instructions like branch, if equal to r1, r2 and then jumping into x.

So, this equality check of two registers is possible only if you can use the ALU. That means r1 and r2 values have been subtracted and then you look at the sign of the resultant and based upon that sign we take a call whether

r1 =r2,

r1 > r2 or

r1 <r2

Consider the case that let us say I wanted to check whether the r1 value is equal to zero or not. Branch is equal to zero.

If I simply want to check whether one register content is equal to zero, then I can do that using the zero tester that has been added as part of this optimized pipeline. So, we

have a register and we can easily check whether the register value is equal to zero or not. So right at the time of switching from the register file, the zero tester can be applied.

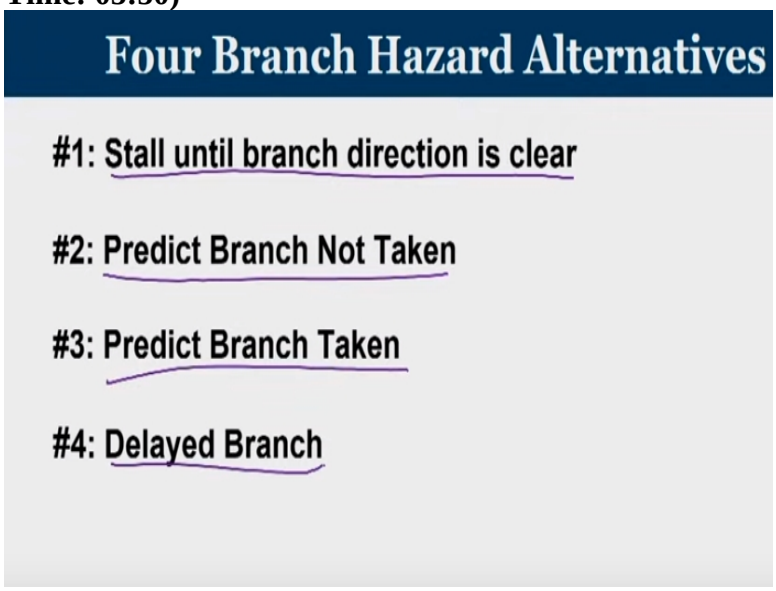
So basically in situations like this, these kinds of branch instructions, maybe we have to write one more instruction

$$r3 = r1 - r2$$

and then you check whether the value of r3 equal to zero or not. So maybe I have to add one more instruction, but we can use instruction branching by test for equality of zero.

So, what we have seen here is a conventional instruction pipeline where branches are resolved in the fourth stage can be replaced with an optimized pipeline where branches can be resolved at the second stage. If you wanted to write a branch statement which has to be resolved in the second stage, then the only possible way of doing a branch operation is to test for zero. I can check whether a register value is equal to zero or not equal to zero and based upon that, we can determine what is the branch condition.

(Refer Slide Time: 05:50)

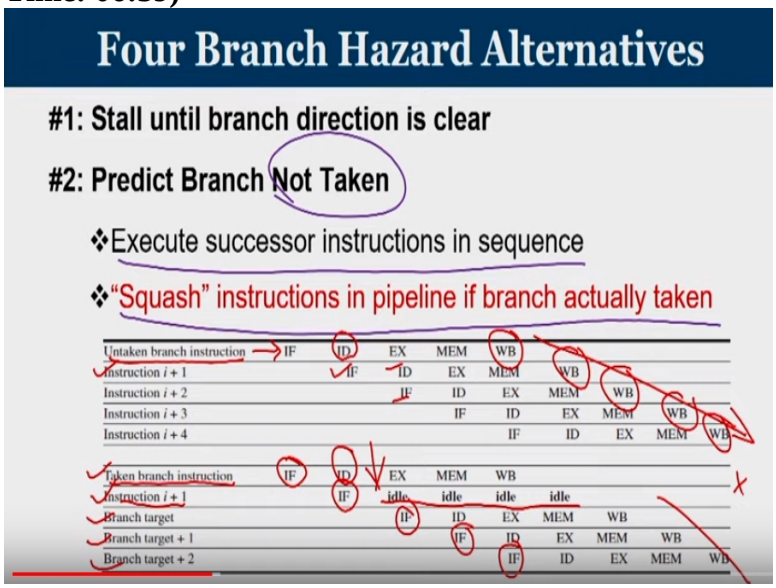


While dealing with branch, there are four alternatives. The very first one is the moment you come to know that an instruction that is being fetched is a branch, then we may have to stall until the branch direction is clear. That is a more simpler approach. So the moment you see it is a branch, we are stalling subsequent fetching of

instructions, until you come to know what is the outcome of the instruction that has been fixed.

The second approach is known as you are going to predict that the branch is not going to take. Third approach is called prediction that the branch will be taken. And the fourth one is called delayed branch approach. We will see one by one.

(Refer Slide Time: 06:39)



Now the first one is stall until branch direction is clear. This is the most conventional and orthodox approach. During the time of fetching by virtue of certain bits in the instruction, we will know that it is a branch instruction or not. The moment it is a branch instruction, further fetching in operations are been temporarily stalled. The moment the outcome of the currently fetched instruction is clear, then we fetch from the appropriate location.

The second approach is known as prediction. We are predicting that the branch will not happen. So what we will do, if you are going to predict that branch is not going to happen, we execute the successor instruction, whatever its successor instruction in sequence. Now, if the branch is actually taken, our prediction was branch will not be taken.

So then in that case, we bring the successor instruction, but unfortunately if the branch is going to happen, then you have the squash instruction that is already in the pipeline.

So look at this scenario, wherein you have given a sequence of code. So consider the case that you have a branch instruction here and assume that branch is not taken. So if the branch is not taken, and we assume that the branches are being resolved at the ID stage.

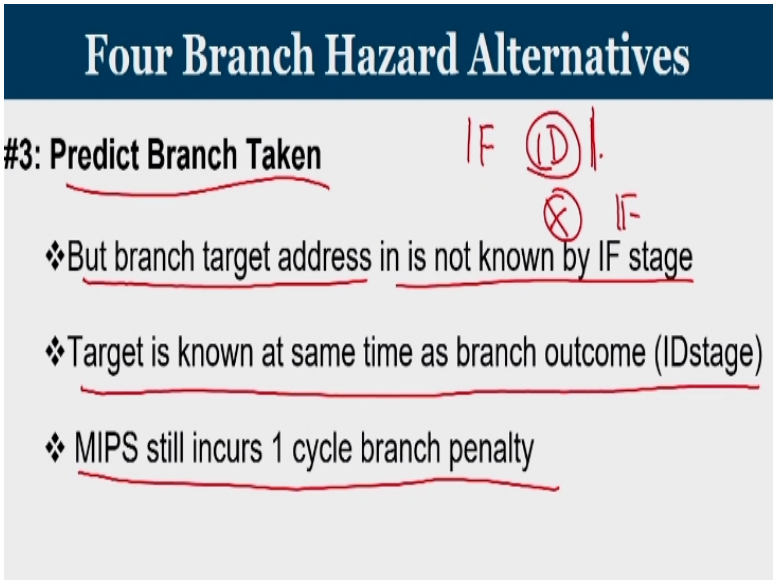
So when the first instruction, that is the branch instruction is in the ID stage I have to fetch a new instruction. Since my prediction is that branches will not happen I am going to fetch the subsequent instruction. So, I fetch the subsequent instruction and then I continue. So, in this way, since I am actually bringing in the instruction that are to be brought there is nothing wrong in the execution and instruction pipeline continue as it is.

Every cycle, we are getting one-one instruction complete. In the second case, you assume that the branch is taken but my prediction is that the branch will not be taken. So, what I will do is when I am fetching the branch new instruction, the very next cycle, the adjacent instruction is being fetched. So I am decoding the branch. The moment I decode the branch at this point the outcome of branch is known but now it is a taken branch. The branch is actually taken.

So, whatever you have fetched, that is instruction i + 1 that is no longer needed. So, I am going to flush out that; that is called idle state. The remaining stages of the pipeline you are going to feed with the no operations. And then you fetch, the very next cycle you fetch the target instruction and then target plus one, target plus two. Like that, adjacent instructions are been brought.

So, the moment you understand that the branch is taken and whatever instruction you have brought in is wrong, then I may have to flush out one instruction that is the follow-through instruction. So, a prediction not taken will work really well, if majority of the instructions, majority of the branch instructions to be very specific are not taken.

(Refer Slide Time: 09:37)

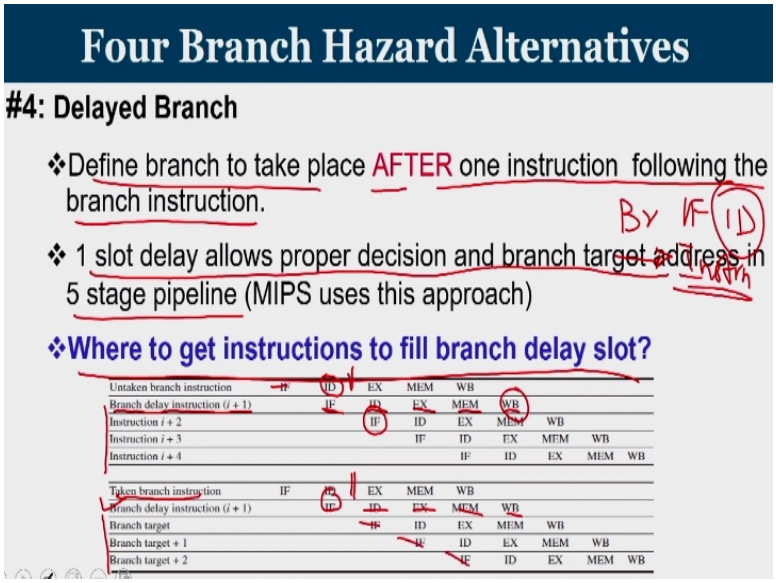


Let us now move into the third alternative where the prediction is taken. But, if you are predicting that the branch is being taken, then you may how to jump to an alternative address which is not the adjacent address but branch target address what it has been called is not known at the IF stage. So, target is known at same time as the branch outcome at the decode stage.

So, in the instruction fetch stage you are going to bring an instruction. In the instruction decode stage you try to understand why it is opcode and operand and probably at this time you will come to know it is a branch and then few bits in the instruction will tell you the offset for the branch means, what is the branch target address.

So, only at the decode stage, end of the decode stage we will come to know what is a target address. But if the prediction is been taken then we will incur 1 cycle branch penalty. That means, if the prediction is taken, there is nothing you cannot fetch anything at this point, you can start the new fetching that too from the target address only at this point. So, we may have to incur a stall if the prediction is to be taken.

(Refer Slide Time: 10:45)



Let us now move into the fourth alternative that is called the concept of delayed branch. So, you define branch to take place after one instruction following the branch instruction. So, let us say I have a branch instruction and the outcome of the branch instruction is known only at the ID stage. Now, can I run one special instruction immediately after the branch instruction, which is a must execute instruction.

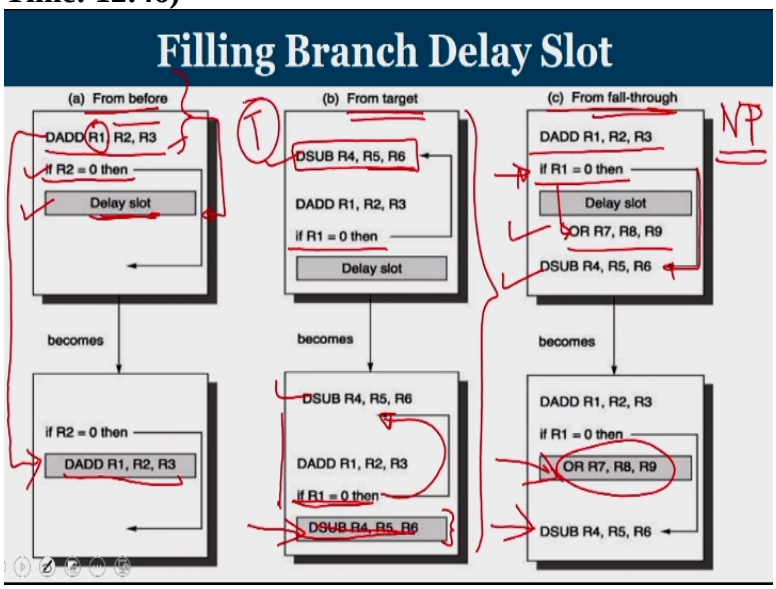
So, when I am in the decode stage another instruction is being fetched and by this time the branch outcome is known and then I can keep the appropriate instruction after this. So, 1 slot delay allows proper decision and branch target address in a 5 stage pipeline. MIPS basically use the branch delay slot. But the question is where to get instruction to fill in the branch delay slot.

So, this is basically what we are supposed to do. Consider the case that you have a branch instruction. So I am fetching an instruction and decoding an instruction. Only at the end of the decode stage you will come to know the outcome of the branch. In the meantime, what will I do when the decoding is progress which is the instruction that is to be fetched? I do not want to flush out an instruction.

I do not want to bring an unnecessary instruction and later flush it out. So, can I find an instruction which is a must execute, whether the branch is taken or not taken. That is called branch delay instruction. So I fetch that instruction and it continues its execution and reaches the write back stage. And then what I fetch? This is the instruction that I fetch, that is a proper instruction.

So in this case, let us say the branch is not taken. So I am bringing instruction i +2, i + 3 and i + 4. And here we have the branch delay instruction. This is a taken branch. So by this time we come to know the branch is taken. But in the meantime, the delay slot instruction is been entering and then since I know the target, I bring instruction from the corresponding target.

(Refer Slide Time: 12:46)



Let us now see an illustrative example by which it is been familiar. How you fill up the branch delay slot. So consider this is the branch instruction, if R2 = 0 then you have to do something. When R2 = 0 then there is a delay slot. So think of a case you have an instruction let us say double add R1, R2, and R3. So R2 + R3 is being stored into R1. Now I am going to check whether R2 = 0.

So this instruction, the branch instruction is not dependent on the previous instruction. So what I will do is I will take that instruction and keep it after the branch statement. So if R2 = 0 then this and we are going to directly add R1, R2, and R3. So this is the process by which I identify an instruction from before the branch instruction and then put it in the branch delay slot.

So the instruction that is executed immediately after the branch is known as delay slot. I can fill up the delay slot by finding an appropriate instruction before the branch which is to be executed for sure whether the branch is taken or not taken. Now sometimes we may not always find instructions like that. So think of a case a branch is like if

R1 = 0

and then it is going to an instruction

DSUB R4, R5, R6.

So, this

DSUB R4, R5, R6

will be surely executed when the branch is been taken. So, can I find an instruction from the target of a branch and put up in the delay slot? So, this DSUB instruction I am going to put up after the branch instruction. So this is the delay slot. Now, the delay slot is filled with an instruction. So, first time when the loop execute, this will be executed.

Then you move to the loop or the condition statement. And when this branch is in progress, this instruction is being fetched which is surely executed and then you go and jump to the instruction immediately after DSUB. So this is being taken whenever there is high probability that a branch will be taken. When a branch is taken, when the probability of a branch to be taken is very high then instructions are brought from the target point of the branch.

Similarly, consider a case that you have a branch if

R1 = 0,

then we are going to jump into this location. And this is an instruction that is to be executed if the branch is not taken. So here the control flow is like this. You have something like this and then you are going to check the value of R1. If

R1 = 0,

then I am going to DSUB. If

R1 != zero,

then I am going to execute the or instruction.

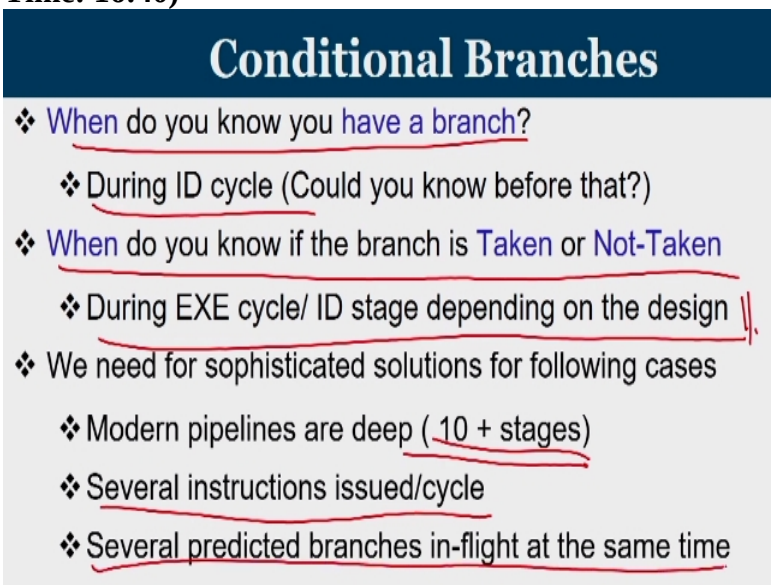
Now, if there is high possibility that this branch will not be taken, that means the control will reach to or can I put or in the delay slot, that means it will be always executed whenever this branch is been fetched. And so, it comes in the delay slot and then this is the target instruction. So bringing from follow-through happen, whenever

there is high possibility that the branch is not taken, NT. And this will be useful whenever there is high probability that the branch is been taken.

So what we have seen here is the concept of filling of delay slot. Delay slot in the MIPS 5 stage pipeline is defined as a one cycle window after a branch instruction. So, since we will come to know what is the target address of a branch only at the decode stage, what will be fetched when the decoding of the branch instruction is in progress?

We have to find out an appropriate instruction that has to be compulsorily executed, whether the branch is taken or not taken. It can be taken from before the branch, it can be taken from the target, it can be taken from the follow-through. These are the things that we have seen with the help of the illustrative example just described before.

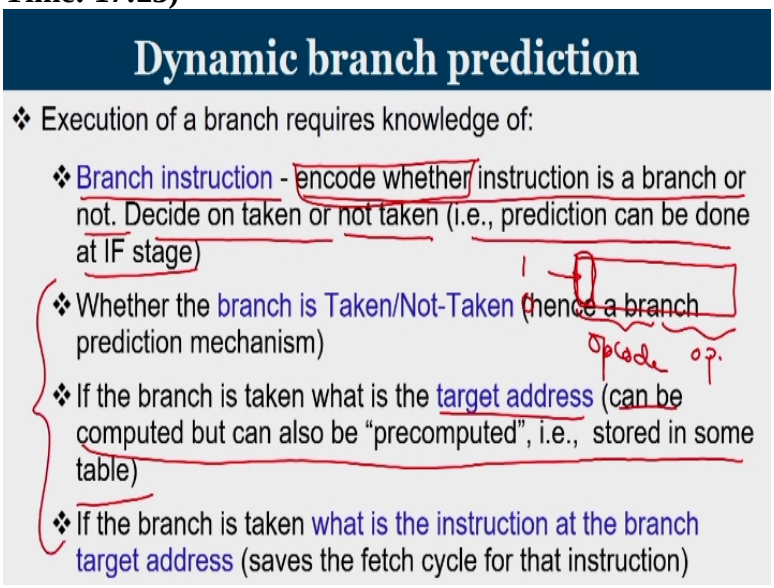
(Refer Slide Time: 16:40)



Moving further, let us try to understand what is the peculiarity or characteristic features of conditional branch. When do you know that you have a branch it is during the ID stage. When do you know that if the branch is taken or not taken, during the EXE stage or ID stage depending upon how the design is been implemented. So we need for sophisticated solution in the following cases.

Like modern pipelines are not 5 stage pipeline. Some of the latest microprocessor has more than 10 stage in its instruction pipeline. There are several instruction that are issued per slot cycle. We will see that down the course when you deal with super scalar processes. Several predicted branches are already there in the same timeline.

(Refer Slide Time: 17:23)



So execution of a branch requires the knowledge of the branch instruction. So we should know whether an instruction is branch or not. So we have to encode whether instruction is branch or not and decide on whether the branch will be taken or not taken. That is prediction can be done at the IF stage. So think of a case that you are going to define an instruction set. Let us say it has a portion called opcode.

And then there is an operand. Now can I make my first bit of the opcode in such a way that if the first bit is 1 it indicate a branch. If the first bit is 0 it indicate a non-branch instruction. So during the fetching itself once I know that the most significant bit is 1, it shows that it is a branch instruction. So, the branch instruction can have an outcome which is either taken or not taken.

So can we have a meaningful encoding to represent branches and decide on whether the branch will be taken or not taken in the IF stage itself. So, whether a branch is taken or not taken, that we have to decide and then we have to make use of a prediction mechanism. If the branch is taken, then we have to find out what is a target address because it is not the adjacent address from which I am going to fetch.

We may how to fetch from a totally different address which is called the target address, which is computed from the current program counter and the displacement value that is mentioned as part of the branch instruction. So this target address can be computed but can also be precomputed or stored in some table. If the branch is taken,

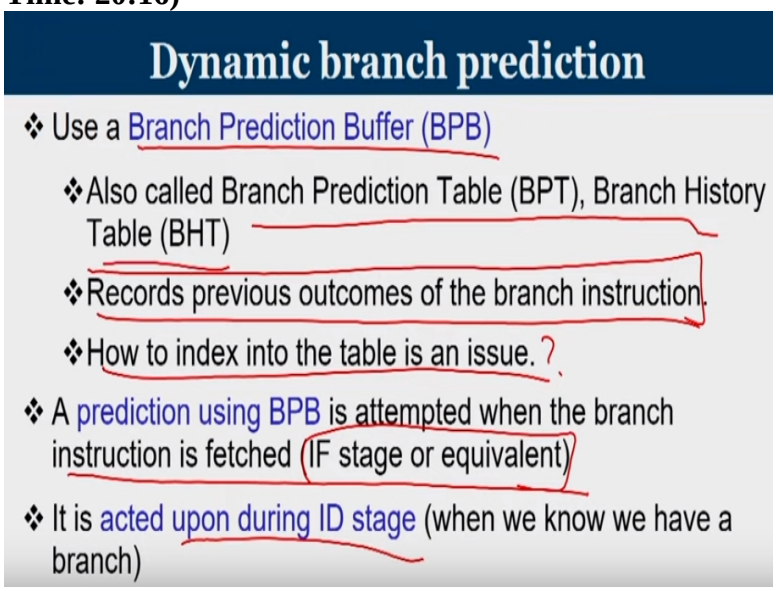
then what is the instruction at the branch target address? So these are the possibilities that we are going to explore.

So control hazard is a very important kind of an issue that instruction pipeline has to handle for improving its efficiency. And control hazard happens when the execution of the current instruction determine what is the next instruction to be fixed, should it be from a different address, or should it be from an adjacent address? And there are basically different kind of an approach in handling this.

But the most common approach is known as branch prediction. It is a small circuitry that is part of your processor, which is going to work whenever you are going to encounter with branches. And how are we going to work with this? When you come to know that an instruction that is going to be fetched is a branch, can I inevoke the corresponding branch prediction mechanism and get to know its output whether the branch will be taken or not taken.

If the branch is not taken, in the subsequent cycle go and fetch from the follow-through instruction. If the prediction is taken, then we have to fetch from an instruction that is called, that is kept in the target address. So we have to compute the target address and then we have to get it.

(Refer Slide Time: 20:16)



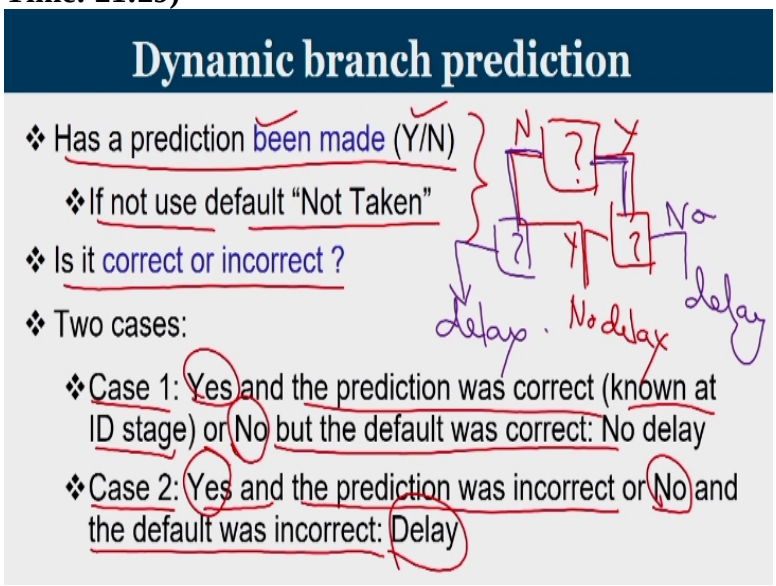
So, dynamic branch prediction, so that means on the fly during runtime, identify branches, and then we use a branch prediction buffer which is called BPB. It is also

known as branch prediction table or branch history table. It records the previous outcome of a branch instruction and how to index into the table is an issue.

So basically it is a table which will record this branch when it is executed in the previous time what happened whether it was taken or not taken prior to that previous to previous time, what is outcome of the branch. That is called it records the previous many outcomes of these branches. And a prediction using branch prediction buffer is attended when the branch instruction is also fetched.

So whenever you are going to fetch a branch instruction, which we will understand by a specific encoding pattern by certain bits in the instruction. And the moment you understand that it is a branch instruction, then we may have to make use of the entries in the branch prediction buffer. And based upon the outcome or the entries that is been given by the branch prediction buffer is acted upon during the ID stage when we know that we have a branch.

(Refer Slide Time: 21:29)



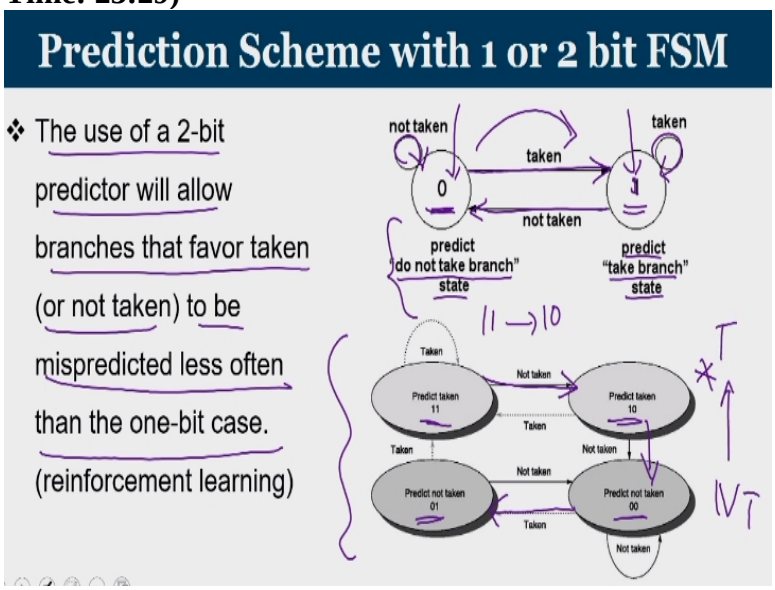
Now, has a prediction been made? Yes or no? If not use a default not taken. Is it correct or incorrect? So there are two things. First is are we predicting yes or no? And is the prediction correct. So, there are two cases that can come. Case 1 is yes. Means we are going to make a prediction and the prediction was correct. You will know whether it is correct only at the ID stage.

Or no, we are not going to predict, but the default was correct. See, there are two approaches. First is should I consult a branch predictor? So it can ask 2 outcome, yes or no. Now, if a consult a branch predictor, then the predictor was correct. Yes, the predictor was correct. Then in that case, there is no delay, whatever prediction I made the prediction was correct. And whatever instruction that was brought, that is also correct.

Now the second case is, no, I am not making use of a predictor, and whatever I am bringing normally that is also the correct instruction. In both these cases, there is no delay. That is what is meant here. Now, the second one, case 2, yes. And the prediction was incorrect. I am consulting a predictor that is called this Yes. And the outcome of the prediction was incorrect.

Or no, I am not going to make use of a predictor but a default instruction that I brought also was incorrect. In this case, I am going to incur a delay. So I consulted a predictor Yes, that is what is being shown in this group, in this flowchart. I am consulting a predictor, but outcome of predictor was not correct. So, in this case, I get a delay. Second one I am not consulting a predictor, but whatever instruction that I used, that is not the correct one. In that case also I will experience a delay.

(Refer Slide Time: 23:29)



So, prediction scheme with one or two bit finite state machine. So consider this finite state machine, wherein we have two states, state 0, which represent we are predicting

that the branch will not be taken and state 1 where we predict that the branch will be taken. So, for every branch is been represented by a one-bit finite state machine.

If the bit is 0, then it means that I am going to predict that branch will not be taken. If the bit is 1 then I am going to predict that the branch will be taken. Now let us try to understand what happens here. When I predict that the branch is not taken and if the actual outcome that is what this shows. If the actual outcome says that the branch was actually taken, then the state moves from 0 to 1. That is called a 0 to 1 transition.

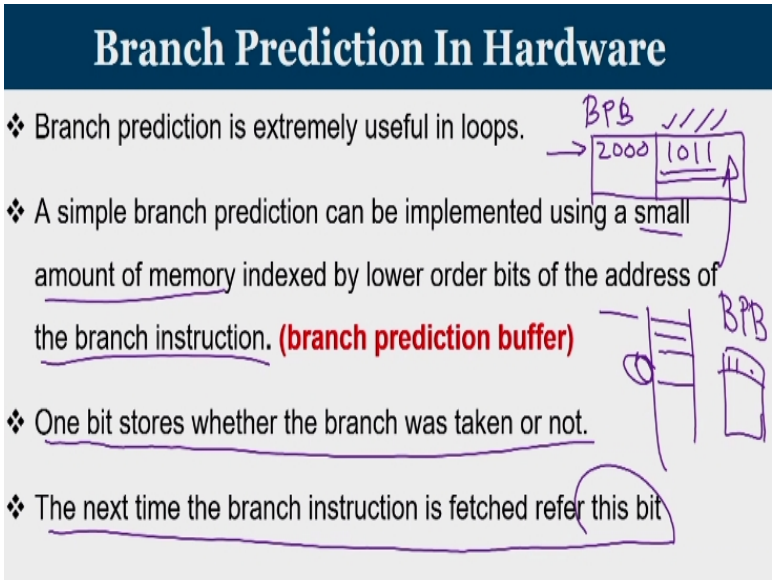
Now when I am in 0, that means the production is not taken, actually also the branch was not taken that means the self loop that self transition, I continue in the state 0. Similarly, when I am in state 1, I predict that the branch is taken and actually the actual outcome of the branch also was the branch was taken, I continue with the state 1.

So when I am in state 0 or state 1, if my prediction is not correct, then I switch the state, 0 moves to 1 and 1 moves to 0. Similarly, I can have 2-bit predictors as well. The use of a 2-bit predictor will allow branches that favor taken or not taken to be less mispredicted less often than the one-bit case.

So, in the case of a one-bit, every time there is a misprediction happens, you are changing the prediction, 0 moves to 1 and 1 moves to zero. Whereas, when you come to this 2-bit predictor scheme, the upper two states 11 and 10 the prediction is predict taken 01 and 00 the prediction is not taken. So when I am in state 11, I predict it as taken.

But if actually if it is not taken, that is what this head shows, I move my state from 11 to 10. Even at this stage, I predict it as taken. When again one more misprediction happens then I move to 00. Similarly, when you are in 00, the prediction is not taken. And if actually the branch is taken then you move to 01. So only for two consecutive mis-predictions I am moving from a not taken stage that is the lower side to a taken stage that is the upper side.

(Refer Slide Time: 26:11)



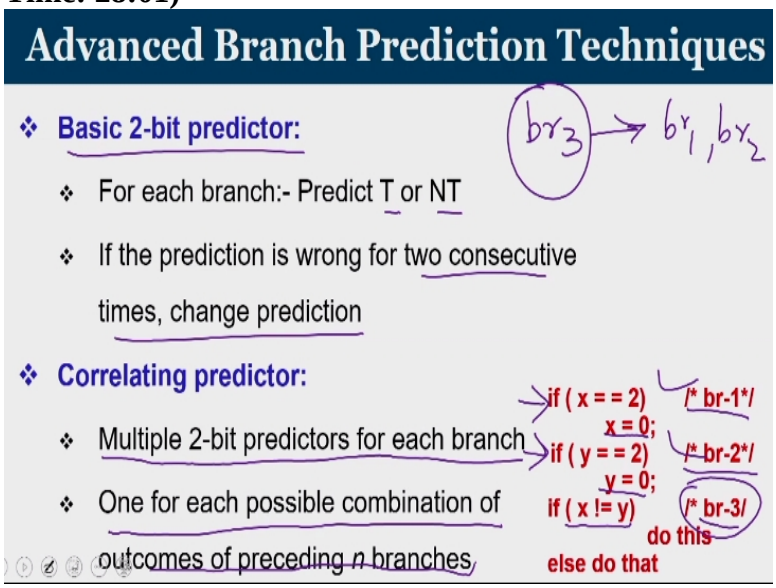
So branch prediction is extremely useful in loops. A simple branch prediction can be implemented using a small amount of memory indexed by the lower order bits of the address of the branch instruction. So whenever we have instructions that are stored in memory and then we have the addresses of these instructions and we are going to enter into this branch prediction table, branch prediction buffer by looking into the lower bits of the program counter when the branch is been encountered.

And we can have multiple bits that is stored in this branch prediction buffer. Generally it is 1 bit stores whether the branch is taken or not taken. So, consider the case that you have a branch prediction buffer where there is an entry in 2000. That means for program counter 2000 let us say the value is 1011.

So this means that you have a branch whose address is 2000 and last time it was taken, prior to that it was not taken, then prior to that it was taken, prior to that it was taken. So last four outcomes of this branch which is located at the address 2000 is being recorded here. Similarly, there will be different program counter values and the corresponding bits which will reflect what was the outcome of this branch when it was executed last time.

Next time when the branch instruction is fixed, you refer into this entry. So, when the as long as the program counter matches, or the last few bits of the program counter matches this table will help you to retrieve what was the outcome of the branches when it was executed last.

(Refer Slide Time: 28:01)



Now, there are different advanced branch predictor techniques. The basic one is a 2-bit predictor that we have seen. For each branch, we will predict whether it is taken or not taken. If the prediction is wrong for two consecutive times, then you change the prediction. That is what the graph that we have seen. There is another category of predictor which is called correlating predictor. So we have multiple 2-bit predictors for each branch.

One for each possible combination of the outcome of preceding n branches. So consider the case you have if

$$x = = 2,$$

I make

$$x = 0$$

If

$$y = = 2,$$

I make

$$y = 0$$

So the first one is branch 1, second one is branch 2. And the third one is if x not equal to y then you do certain steps. So the outcome of branch 3 is dependent on the outcome of branch 1 branch 2.

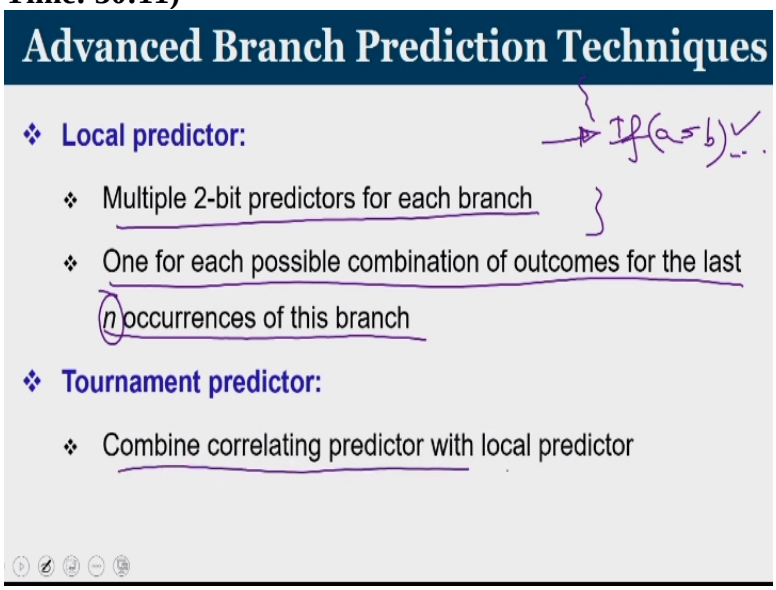
So if the branch 1 is taken or not taken and the branch 2 is taken or not taken that has a significant say in knowing whether branch 3 is happening or not. So we can tell that

branch 3 is actually dependent on the outcome of branch 1 and outcome of branch 2. So there is no point in trying to explore what happened to branch 3 last time when it was executed.

So, there are certain branches like this, where the outcome of this branch is not dependent on the outcome of the same branch over the previous iterations. In fact, the outcome of this branch is dependent only on few other branches. So in this case, branch 3 is dependent on outcome of branch 1 and branch 2. So we require predictors which will take care of this dependency, this correlation.

They are called correlating predictors. Now see the definition of correlating predictor once again. We have one branch predictor for each possible combination of the outcome of preceding n branches. So one branch is dependent on preceding n branches.

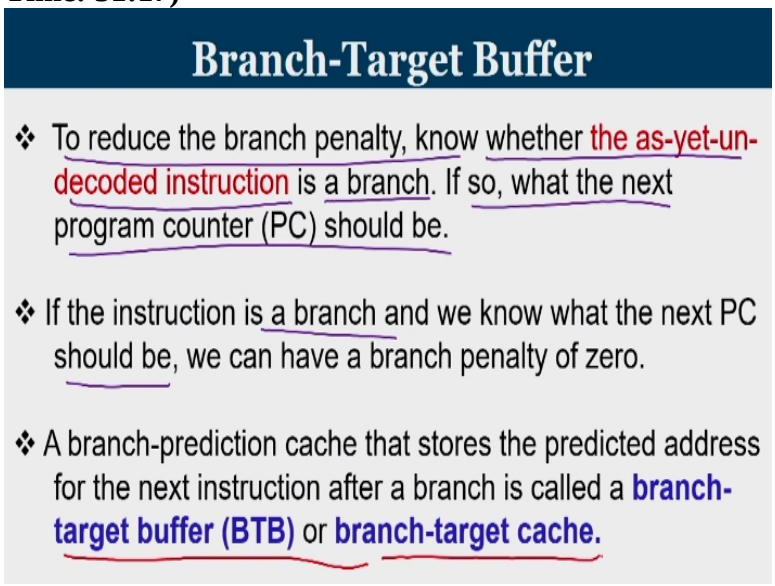
(Refer Slide Time: 30:11)



And then we have a local predictor. So we have multiple 2-bit predictors is there for each branch, one for each possible combination of the outcome of the last n occurrences of this same branch. So for example, think of a case that we have, an if a = b statement, so in the same line, I am going to run inside a loop. So whenever I encounter this line, I know it is a branch instruction, I look into what happened to this branch in its previous n occurrences.

And I find that there is a high correlation between the outcome of this branch and the outcome of the same branch during its previous iteration. That is what is called the local predictor. So the local predictor looks into the last n occurrences of same branch and correlation predictor looks into last n branches. And we know tournament predictors. What we do is you combine correlating predictor with the local predictor and pick one of them depending on the requirement.

(Refer Slide Time: 31:17)



Now, we will see what is called branch target buffer. To reduce the branch penalty can I know the as-yet-undecoded instruction is a branch or not. If so, what is the program counter should be? So can I store the target of a branch early? If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero.

So what we will do is generally in the branch prediction buffer, you have a program counter value and then we have few bits which will tell you whether the branch is taken or not taken. And then if it is a branch that is going to be taken and here I store the instruction that is stored from the target. For example, let us say this is branch is in location 2000.

Now let us say the location 2000 we have a branch which will tell if a = b then jump to 6000. Now somewhere in 6000 I am telling

R3 = R4 + R6

So what I do is in location 2000, so this is my branch target buffer wherein location 2000 I am telling the outcome of the previous branches and all and then I am writing

R3 = R4 + R6.

The meaning of this is if there is a branch instruction in 2000 and these are the outcome of these branches and if the branch is taken then it is jumping into 6000 whose address is already saved. So a fetch from 2000 will decode an instruction like

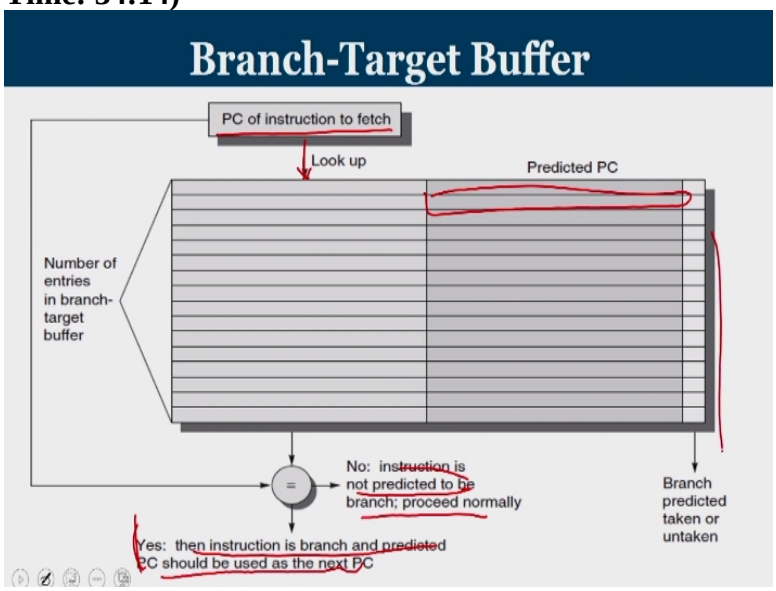
R3 = R4 + R6

In this scenario, even though I am trying to fetch something from 2000 the decoding happens for the instruction 6000 with a prediction taken.

So that makes a branch prediction cache that stores the predicted address for the next instruction after a branch is called a branch target buffer. So the branch target buffer is also known as branch target cache. It stores the instruction which is there in the target of a branch. So if I have a branch A, let us say branch a is diverted to and in a location L and if L has another instruction B then the instruction A is there in the buffer and instruction which is there in the target location L that is also stored in the buffer.

So in this way I can have a 0 penalty branch instruction. It is also known as branch folding.

(Refer Slide Time: 34:14)

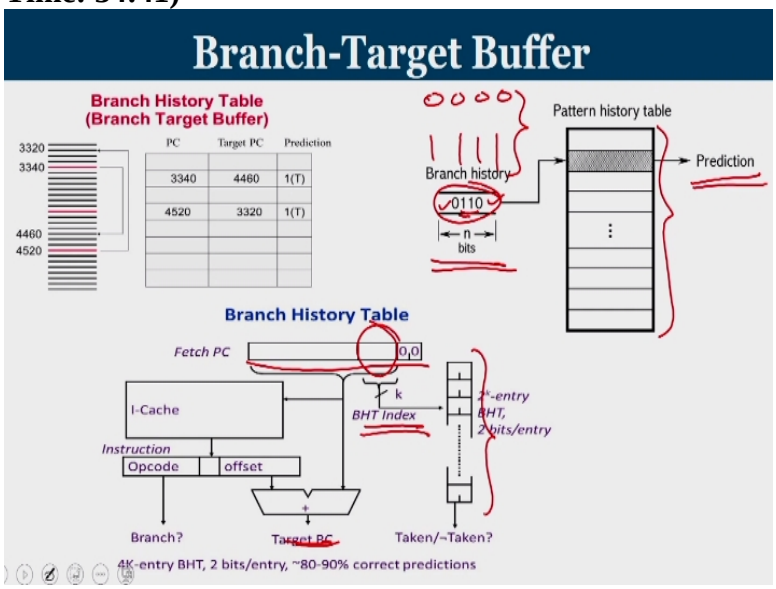


So this is where we have the branch target buffer. So the PC instruction to fetch and based upon that I am going to look it up. If I am not able to find the hit that means the instruction is not predicted to be a branch, you proceed normally. If it is, then the

instruction is a branch and the predicted should be used as the next PC. So here I have the predicted PC.

This PC value will tell you where you should go and then this will tell you whether it is taken or not taken.

(Refer Slide Time: 34:41)



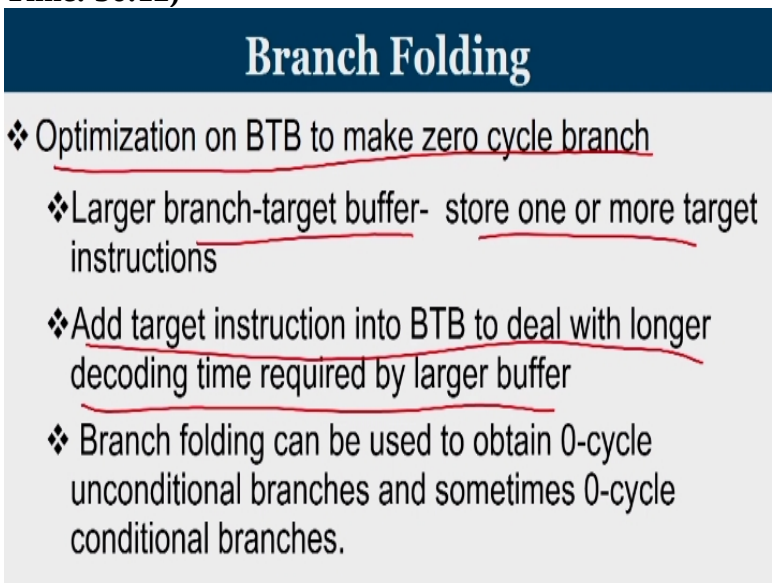
So these are all different organizational structure of branch target buffer. So, branch history is represented using n bits. The last four branches are being represented as 0110. Means the last occurrence was not taken. Prior to that two occurrence were taken and previous to that it was not taken. So this pattern will help us to index into this table. So if you look at the last four occurrences, then you can have variations from 0000 up to 1111.

These four bits will give you 16 combinations and based upon that you index into the table and that will predict whether the branch is taken or not taken. So when you fetch a program counter value, few bits in it is being used to compute the target PC. So this value is being added with whatever is the current instruction.

And then you find out the target PC and few bits, the few least significant bits will help us to index into the branch history table and the history table will have the entries of a state machine and then it proceeds. So we are going to have a general idea of what is a branch predictor and how these branch predictors are helpful in execution of this program.

There is a tutorial also that comes along with this week where at the end of the tutorial a problem is been worked out how can you approach dynamic branch prediction and how the table entries are being manipulated.

(Refer Slide Time: 36:12)



So the optimization on branch target buffer is to make a zero cycle branch. When I can have a large branch target buffer you store one or more target instruction. Also, rather than storing what is a target PC, I can store the instruction of the target PC. So add target instruction into branch target buffer to deal with longer decoding time required by the large buffer.

So branch folding can be used to obtain 0-cycle unconditional branch and sometimes 0-cycle conditional branches as well. So with this we come to the end of this lecture number five. In this lecture, we were trying to understand what is a control hazard and what are the different approaches in handling control hazard and specifically our attention was on how a branch predictor works. Thank you.