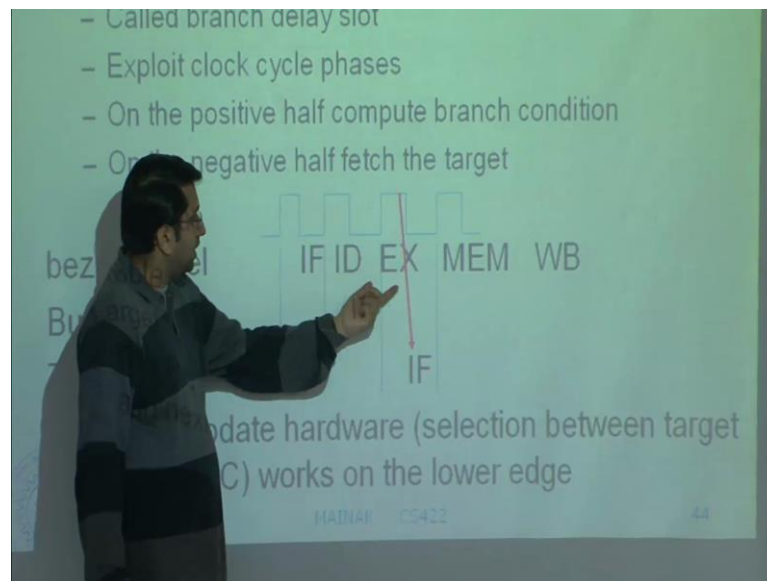


Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

Lecture - 13
Basic Pipelining, Branch Prediction

So, just to remind you what we were discussing; we were talking about control hazard.

(Refer Slide Time: 00:18)



And last time, we saw a typical pipe-stage pipe. So, this is what happens; you fetch a branch instruction here and you get to know the target at this point. So, essentially, you have two instruction delay before you know the target; but, MIPS has a phased execution; where, they make sure that, the branch target is rarely in the positive half of the cycle, so that the fetcher can fetch in the negative half of the cycle. So, that is how they nullify one of the bubbles. But, they still have one bubble left; and, they rely on a compiler to fill up this particular instruction. So, this is called a branch delay slot. And, if a compiler can fill up something; we discussed last time from where you can pick up an instruction to fill the slot; that will not be a wasted cycle; otherwise, it will probably fill up with a ((Refer Time: 01:24)) which essentially a bubble. And, is this clear to everybody? Any question on this?

(Refer Slide Time: 01:43)




And then, we also discussed something called the branch target buffer; where, essentially, the idea is that, in the fetch stage, you look up a particular cache. So, that is the branch target buffer. It is a cache for the branch targets. So, you look it up with the phase stage and the BTB gives you a target; which you can use in the next cycle to fetch. Essentially, for that MIPS is I look up the BTB right here. And, the problem was I did not know what to fetch here; but, now, I can use the BTB outcome to fetch an instruction here. And, in the BTB outcome, essentially, I have nullified all the loss without relying on the compiler.

(Refer Slide Time: 02:23)

Branch target buffer

- BTB is looked up with the PC of every instruction in parallel with fetching the instruction
 - On a hit, it provides two pieces of information: this instruction is a control transfer instruction and the target of this control transfer instruction seen last time
 - This target will be used to fetch in the next cycle
 - On a miss, the fetcher has no option but to fetch from the fall through path (PC+4) in the next cycle
 - A control transfer instruction is inserted in the BTB after the EX stage when its target is known
 - A lookup at this point may hit in the BTB; if the branch is not taken, the BTB entry is invalidated; otherwise the entry is updated with the taken target
 - If a lookup at this point misses in the BTB, a new entry is allocated provided the branch is taken



So, this slide summarizes nicely what a BTB does. The BTB is looked up with a program counter of every instruction in parallel with fetching the instruction. So, essentially, it has a program counter; you send a program counter to the memory or fetching the instruction. You send it to the BTB also or looking up for every instruction. On a BTB hit, it provides two pieces of information. What is that? The first one is that, this instruction is a control transfer instruction; which is why it is getting the BTB. And, the second thing is the target of this control transfer instruction in last time. The b t b always stores the last target that, this control transfer instruction ((Refer Time: 03:10)). So, this target will be used to fetch in the next cycle.

On the other hand, if you miss the BTB the fetcher really has no option, but you fetch from the fall through path – PC plus 4 in the next cycle. And, a control transfer instruction is inserted in the BTB after the execution stage when its target is known. So, we always insert the BTB; where, you know for sure where a particular control transfer instruction is going. We cannot insert any wrong in the BTB just to be all the time correct. So, essentially, what we do is once the branch is computed in the execution stage, you look up the BTB once more. And, at this time, the instruction may hit in the BTB, because the BTB might have seen this branch already before.

So, now, there are two options. If the branch is not taken in this particular execution, the BTB entry is invalidated. Why is that? Because we only want to store taken branches in

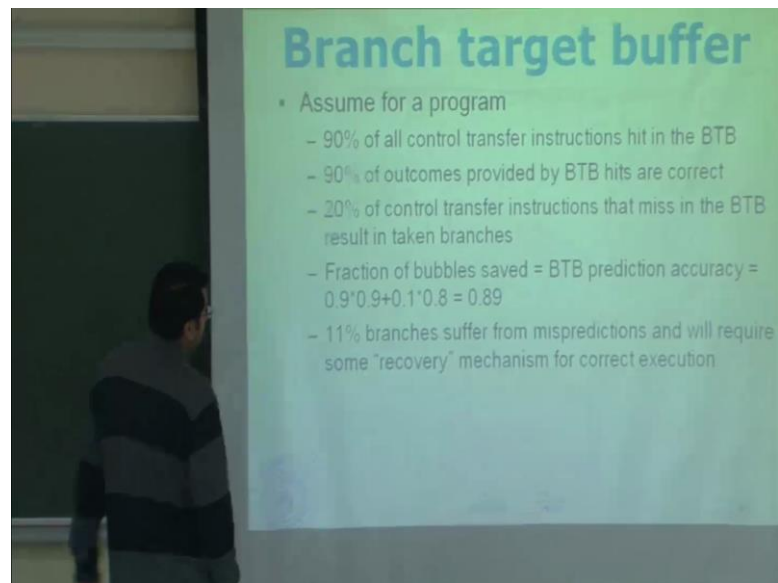
BTB, because remember that, if you miss in the BTB, in anyway, going to take a fall through path. So, it is better to save BTB space by not storing the not taken branches; otherwise, the entry is updated with the taken branch target. If the lookup at this point misses in the BTB; it is cut out here actually; a new entry is allocated ((Refer Time: 04:38)) branch is taken. If the branch is not there; of course, you do not allocate the ((Refer Time: 04:42)) Is this clear to everybody this particular protocol? You look up the BTB at the fetch stage; you update the BTB after you have done executing the branch instruction.

And of course, you can optimize this part a little bit to say BTB bandwidth. You can say that, you can carry forward the BTB outcome with you, because remember that, a branch instruction must have looked up the BTB at the fetch stage already once. So, you can carry forward its outcome to the execution stage and then match and decide whether to look up the BTB and insert anything or not. Any question on this? Is it clear?

Student: Sir, so, if there was a branch delay slot and we also have had a BTB. So, still the branch... It is the compilers responsibility to fill the branch delay.

Yes. And, the transfer has to obey that; it has to execute that. In that case, BTB is of no use; which is why MIPS R 3000 did not have a BTB. So, I am saying... So, BTB is going to be useful only when you say that, the compiler cannot fill up the details. So, what else can I do then? How to save the bubble? So, in many cases, BTB will probably out perform a compiler freely into the slot, because it can see the dynamic behavior of a branch and can learn; whereas, the compiler sees the static piece of code and may not know what will happen at run time. But, of course, the penalty you pay here will see gradually how it actually changes a pipeline hardware to input a BTB. So, there are extra pieces of hardware that you have to go here. And then, the danger of lengthening the cycle time also going to the BTB. So, let us take a simple example; just we evaluate the usefulness of the BTB.

(Refer Slide Time: 06:31)



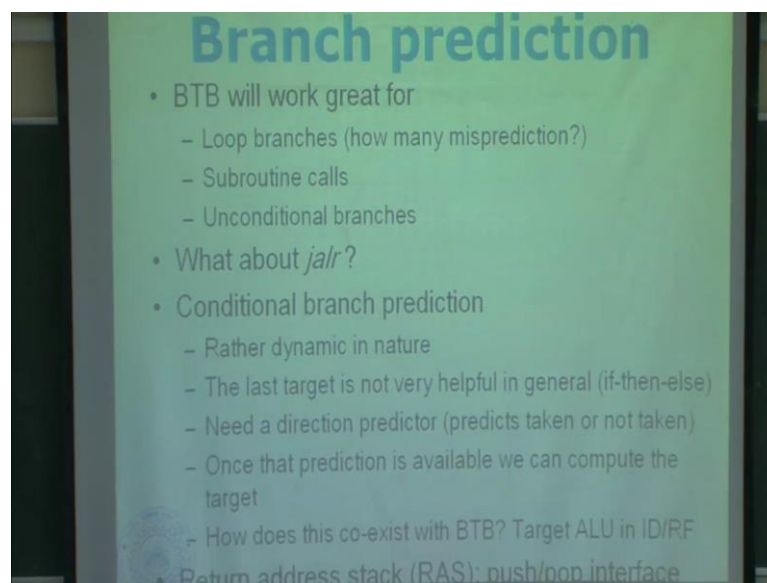
Branch target buffer

- Assume for a program
 - 90% of all control transfer instructions hit in the BTB
 - 90% of outcomes provided by BTB hits are correct
 - 20% of control transfer instructions that miss in the BTB result in taken branches
 - Fraction of bubbles saved = BTB prediction accuracy = $0.9 \cdot 0.9 + 0.1 \cdot 0.8 = 0.89$
 - 11% branches suffer from mispredictions and will require some "recovery" mechanism for correct execution

So, let us assume that, for a program, 90 percent of all control transfer instructions hit in the BTB; 90 percent of outcomes provided by BTB hits are correct. 20 percent of control transfer instructions that miss in the BTB result in taken branches. ((Refer Time: 06:50))

Want to know what fraction of bubbles are saved with this particular statistics. So, what is that? So, fraction of bubble saved the same as BTB prediction accuracy and which is... What is that? 0.9 fraction hit and 0.9 of that are correct. And, 10 percent of branches that miss; out of those, 20 percent are actually wrong, because these are taken branches. So, 80 percent is correct. So, 0.1 multiplied by 0.8; that is, to 0.08. So, we say 89 percent of bubbles we miss. But, we lose 11 percent. So, 11 percent branches suffer from mispredictions and will require some recovery mechanism for correct execution, because 11 percent of these cases – what will happen is that, it will take some BTB prediction and go along that path for the wrong; which you discover later. So, you have to fix up something in the pipeline to make sure that, the wrongly fetched instruction is removed from the pipeline. Is this example clear to everybody?

(Refer Slide Time: 08:09)



So, this one we had already discussed that, the BTB will work great for loop branches. So, essentially, we say that, we will mispredict the first time and the last time; otherwise, it will be the same target all the time, because the loop branch will always go back. Only last time, it will actually fall through. And, the first time, they will be a miss in BTB, because we have not seen the branch before. Subroutine calls – these are also great, because they always go to the same place. So, you call a subroutine from somewhere; it

is always going to the subroutine target; and, unconditional branches. So, in these cases, the BTB will be highly accurate. So, again in this case, the first time will take a misprediction; and, this case also, first time will be a misprediction. And, subsequent predictions will be correct provided that particular entry is not replaced from a ((Refer Time: 08:57)) So, they have enough capacity in the BTB – should be before this.

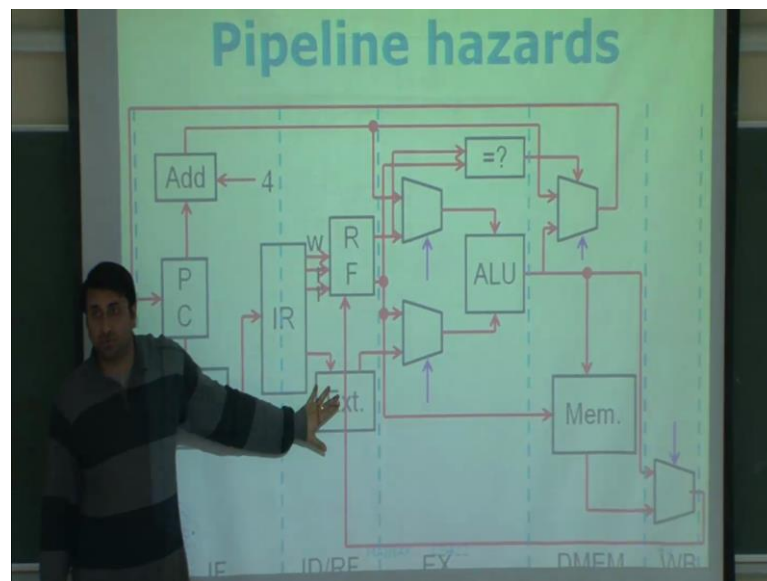
Now, we talked about this indirect procedure calls; which are jump and link register instructions in MIPS. What you think about this? How will they perform in the BTB? Indirect procedure calls – what are these by the way? How do they actually compile? What kind of instructions they are going to use? What does this instruction do actually – jalr? How is it different from direct procedure call? Can anybody remember? What kind of program constructs would need to a jalr. So, you will get this from function pointers when you have... Then, the function pointer may resolve to any procedure pretty much in a program at run time; so, which means the targets are going to vary for these instructions; they are not constant. So, the BTB is primarily highly accurate in this case depending on the program deviation. But, of course, if you have a ((Refer Time: 10:06)) of locality that is over a phase of a particular execution in a particular function point resolves to the same function all the time; then, the BTB buffers ((Refer Time: 10:14)) So, there is no guarantee that you can give in this case for the BTB like here. So, that is about these three categories of branches.

And then, of course, we have left out one major category; its conditional branches; where, like an if else kind of constructions lead to conditional branches; where, depending on the conditional outcome, you take a branch, which is very different from these two. Loop branches are conditional branches; but, they actually behave very regularly; it is not like if else type of branches, which may go either way. So, about conditional branches, usually, processors use a separate type of predictor, which are called direction predictors, because here all you want to know is which way am I going, because I know the target; I am seeing the instruction actually; that offset gives me an instruction and simply add the offset to the PC to get the target. All I need to know whether should I add that offset or not; that is, should I jump the target or should I just fall through. So, it is a binary prediction. However, it is very dynamic in nature.

The last target is not very helpful in general, because the last time, the way you went, you may not go next time the same way. So, maybe this time, you execute an if part of

the code. Next time, you might execute the else part depending on the conditional outcome. So, you need a direction predictor – predicts taken or not taken; which is a binary prediction. Once that prediction is available, we can compute the target. And, the question is how does this coexist with the BTB? We will talk about that very soon. One thing that will require to make good use for direction predictor is that, we need an ALU in the second stage of the pipeline to compute the target. So, this one was discussed last time.

(Refer Slide Time: 12:16)



Somebody raised this issue that, why cannot you compute the target right here; because part of the target is inside the instruction; it comes from the offset. So, the point was that, I need a naming here to read; we need an adder essentially. So, we will assume the existence of an adder. So, let us see how that actually works very soon. So, last thing that is left is a type of instructions that is written from a procedure. That is one type of control transfer instruction, which is not covered in any of these types. So, you end a procedure; you take a control transfer; which returns me back to the call insert. And, here also, the BTB would not be very useful. Why is that? Can somebody guess? Or, can somebody see – why is a BTB not used for return instructions?

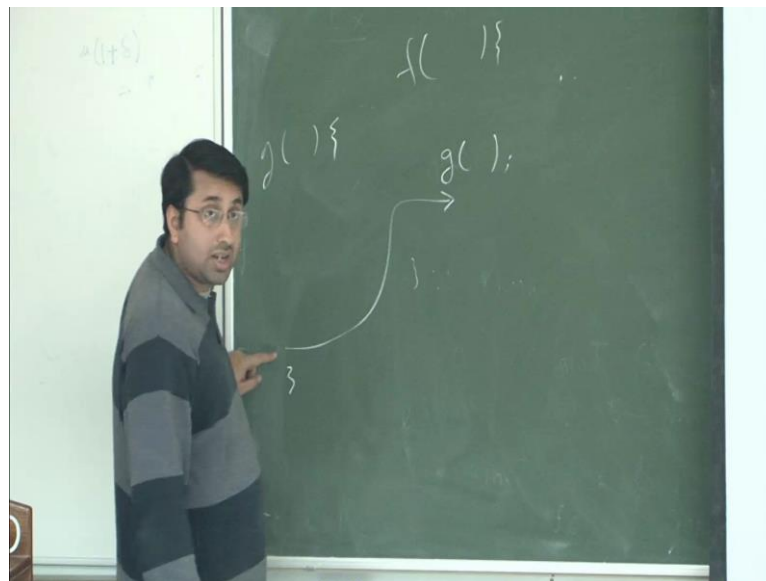
Student: Return address ((Refer Time: 13:11))

Why is that?

Student: Because this time we call ((Refer Time: 13:18))

No, no, no, no, no; this is not about return value. See jump instructions are concerned about the next instruction execute. The question is what is my next place that I return to? It is not the return value that we are talking about; has nothing to do with that. You understand what I am saying?

(Refer Slide Time: 13:45)



So, I have a function f. I am executing the function; I call g here. And, this is the g. At some point, g will return. And, when g returns, I will have to start executing here. The question is when I fetch this particular return instruction, do I know where to go next – that is a question. And, that is not covered in any of these types; it is a very special type of instruction. And, I am saying that, BTB is not going to be very helpful in this case. Why?

Student: It might be calling from different place.

Exactly. So, I might be calling g from many different places. So, this time, I call within f; next time, I might be calling g from h. So, I return... Target return address will be all different actually every time. Of course, it may not be so in some cases; but, anyway, it depends on the execution. So, BTB is not going to be very helpful for this case. So, what processors today include to tackle this matter is another hardware structure, which is called the return address stack. So, it has nothing to do with the execution stack; it is a hardware structure, which are the push-copy interface. And, you can guess what it does.

Whenever it fetches a jal or jalr instruction, which are procedure calls; it would actually push the return address, because it knows the return address at this time. When the jal executes, the return address is just the PC plus 4 or PC plus 8. And, whenever you encounter return instruction, it will just pop the stack. And, the top of the stack should be your return address. So, that is a prediction actually. That gives us a prediction. And, in most cases, this will be very accurate. So, provided you do not overflow the RAS; because if you have a very deep call stack, you might overflow the RAS; in which case, actually, the predictors will go wrong at some point. So, it is a hardware structure; it can also... So, that can happen.


Student: What is the size of it?

Depends; for example, the MIPS R 10K had a 4-entry return address stack. So, you can make it bigger; bigger stacks probably will be better. So, it depends on how much budget you have for hardware and all ((Refer Time: 16:04)) But, you can have a two smaller stack; then of course, you have to live with MIPS. At some point, you overflow and start giving out these predicted values.

(Refer Slide Time: 16:17)

Branch prediction

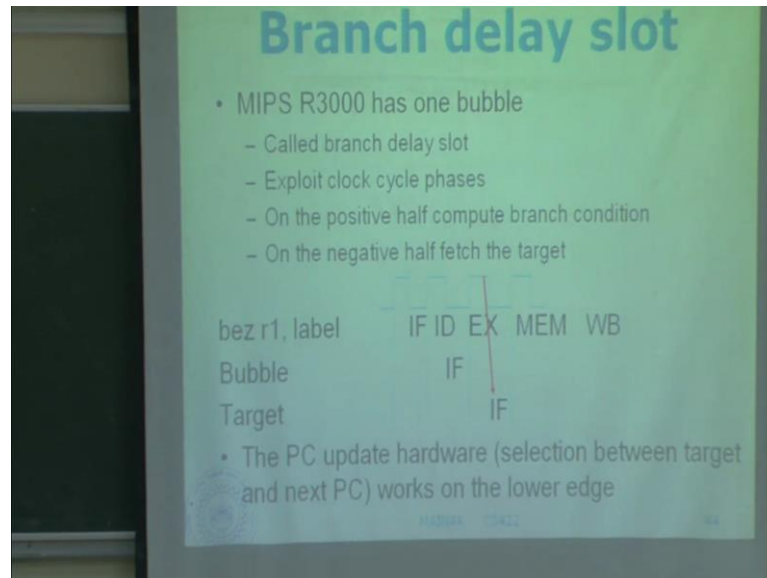
- Deciding the next PC
 - Observe that in five-stage MIPS with half-phase instruction fetch, a conditional branch prediction is of no use; same is true about a RAS
 - Every cycle, the fetcher has to select from three options: PC+4 (from IF/ID register), BTB output (from IF/ID register), and actual target (bypassed from EX stage; this is available early for jal, jalr, and j)
 - Assume that the BTB lookup returns a tuple: (hit/miss, BTB contents); on a miss the second entry is PC+4
 - If last to last instruction was a control flow instruction, compare BTB contents for that instruction with actual target; on a mismatch, select actual target and zero out ID/EX reg.; ID/EX inputs are ANDed with ~KILL
 - Otherwise select BTB contents



So, with all these things, we have now the BTB; we have a direction predictor; we have a return address stack. The question is what does my hardware look like? That decides the next PC. We have too many inputs now to be chosen from. So, first, I will assume that... First, we make an observation, that is, in a 5-stage MIPS with half phase instruction

fetch, a conditional branch prediction is of no use; same is true about RAS. So, let us try to understand why that is so.

(Refer Slide Time: 16:53)



So, let us go back to this ((Refer Time: 16:54)) So, we are saying that, we assume that, we have a phased execution just like MIPS R3K has. And, we are saying that, of course, BTB is useful. We already talked about that, because we know that, if we have a prediction of BTB here, we can use it here. If I do not have a delay slot filled in by the compiler. So, now, let us take up the branch that the ((Refer Time: 17:19)) prediction for direction predictor for conditional branches. So, first question that we have to answer is in which stage can I make a prediction; in which stage can I ask the direction predictor?

Student: Instruction decode.

So, I cannot be asking the direction predictor before decode; why is that?

Student: They have to be coded first; then we will know where there are the addresses.

No, why is the address important? I just want to know yes or no. Address is not yet important; I just want to know whether I should go down the fall through or the target; that is all the predictor tells me; it does not tell me the target. I will come to the target separately by using the offset; which I can do anytime; remember that; because I know that, my last 16 bits are going to be offset. So, I can just assume that, everything is a branch; even here I can actually take those 16 bits and add it to PC plus 4; maybe ((Refer

Time: 18:21)) if it is not a branch instruction; but, I can do it here actually. So, all are asking is that, the direction predictor will only tell me yes, no; it do not tell me the target. I am asking – where can I vary that predictor? What is the earliest possible stage in the pipeline? Your answer is correct; I cannot do it before the decode stage; but, why?

Student: We need to evaluate the condition first.

Condition evaluation will happen here. And, if I evaluate the condition, why we will have the prediction? Prediction is important only if I do not know what is going to happen. If I can evaluate ((Refer Time: 18:57)) then why should I have a predictor? ((Refer Time: 19:00)) Yes?

Student: After catching, instruction we will be known as its condition; then only we should...

You will know what?

Student: Whether it is conditional instruction or something else.

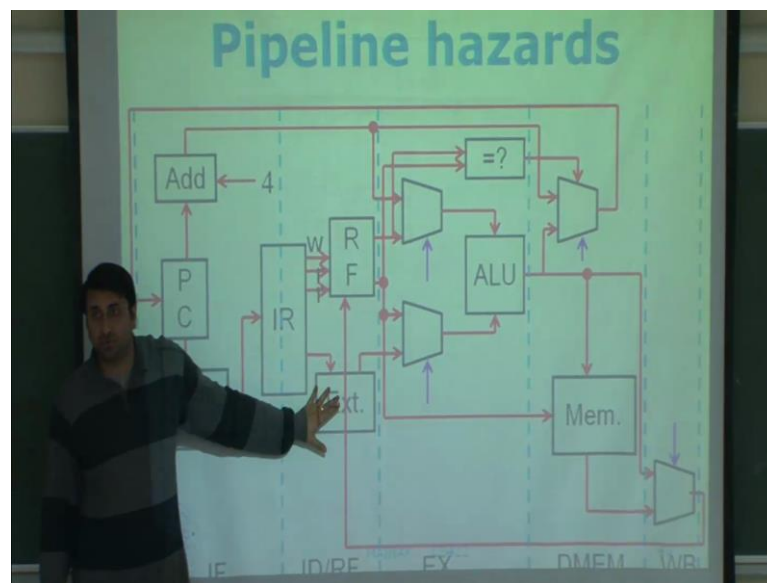
Right; exactly. So, I need to know if it is a conditional branch or not. So, that is the important point. And, that I will get to know only when I reach the decoder. I decode the branch and know that... I decode instruction; I know that, it is a conditional branch; then only, it make sense to look up the branch prediction. So, I cannot... There is no hope of using the direction predictor in the fetch stage. So, I have to wait till the decoder for sure. And, if I have to wait till the decoder; then, you can clearly see that, there is no point in using this predictor, because in the next cycle, anyway I will know the correct target actually. Is that clear to everybody? That in this particular setting, I do not need a direction predictor.

What about the return address stack? Can I argue the same way? Because I push or call from the stack only after I know that, I am dealing with the procedure call and return. There is another – if I start popping the stack, I will be actually popping long things; I will be popping useful things at the wrong places actually. So, I should be popping from the stack only if I know that it is a return instruction. Or, I should be pushing on the stack only if I know that, it is a procedure call. Is it clear? So, for this particular pipeline, I am

only interested with dealing with the BTB. So, let us see how to do that. If something is not clear, you can ask questions; just to remind you.

So, what does it mean? That means every cycle, the fetcher has to select from 3 options. One is PC plus 4 from the fetch decode register. So, by the way, I will be using this particular notation to denote the pipeline register sitting in between fetch and decode stages.

(Refer Slide Time: 21:16)



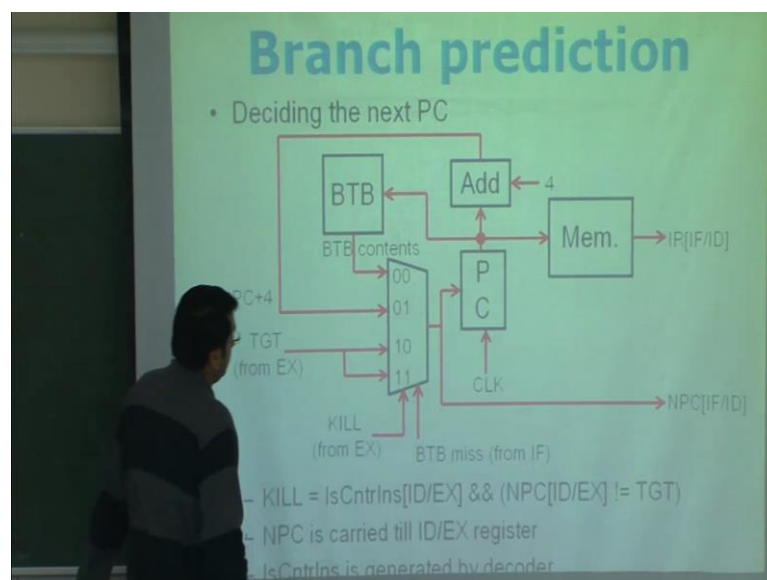
So, remember that, at every stage, it is... Every two consecutive stages are separated by time registers. So, this is the IF/ID register. This is the IDX register; this is the XD memory register and so on. I have a PC plus 4; which comes from this register, because remember that, I had the PC; I have incubated; and, I am going to latch it in the IF/ID register in the next cycle. So, from there, I can take PC plus 4. I have the BTB output also from the IF/ID register. So, whatever output the BTB has given be... And, the actual target, which is bypassed from the execution stage. This is available early for these three instructions, because these three instructions do not require any extra evaluation. I can evaluate target from the instruction itself if I am an adder in the decoder – decodes this, because here the target is inside the instruction. Here the target comes from the register value. And, here the target is also the instruction.

So, under what condition, do I have an input from the execution stage for the PC if a branch instruction is correctly executed there? So, which means two cycles earlier had

fetched the branch; which is now resolved here. So, I have an input coming from that stage telling me that, you might have to change the PC for you to make a misprediction. So, so let us assume that, the BTB lookup returns a tuple, which is... The first entry of the tuple is a hit/miss indication; it is a binary value. And, the second entry is the BTB contents. And, on a miss, the second entry is going to be PC plus 4. It tells me the fall through. That is not exactly coming from BTB; but, I will assume that, that is all the people will ((Refer Time: 23:18)) from the BTB hardware.

If last to last instruction was a control flow instruction, you compare BTB contents for that instruction with the actual target. So, if the last to last instruction was a control flow instruction, you have to compare the BTB contents for that instruction with the actual target. Is that clear to everybody? On a mismatch, select actual target and zero out the ID/EX register, because the problem is that, current instruction in a decoder is actually wrong. So, you cannot put the contents of that into that ID/EX pipeline register. You can... So, for which does is that it zeroes out the entire register? And, the good thing is that, the ((Refer Time: 24:07)) actually all zeroes. So, that will actually interpret as ((Refer Time: 24:12)) as it goes to the pipeline quantity. So, does not do anything. And, the ID/EX inputs are ANDed with tilde of KILL signal. So, we will look at...

(Refer Slide Time: 24:22)



Here it is ((Refer Time: 24:24)) So, this is the program counter. And, this is actually the pipeline register at the front of the fetch stage. So, here I show this in a dotted line; it is

actually the PC. At every clock, I will latch a new PC, which will actually go to the memory, start fetch and also do this addition lock. That is why I will show the clock get it fetched to the program counter register. So, what else? What is happening? So, in a particular clock cycle, I get a new PC; I fill it to the adder, which adds 4 to that. So, I get PC plus 4 here. And, this PC is also sent to the memory for fetching instruction, which will go into the instruction register in the fetch decode pipeline register. This PC is also sent to the BTB – the branch target buffer for looking up the BTB. And, what comes out from the BTB are two things: one is the BTB content; and, other one is an indication of pick miss.

And, the third thing that I get is a target from the execution stage. This is for an instruction, which is currently executing in the execute stage. And, this also comes accompanied with the KILL signal; and, this is enabled only if the currently executive instruction in the X pipeline stage is a control transfer instruction; and, its target does not match with the BTB outcome; which means I did something wrong in the past. So, I have to kill some of the instructions inside the pipeline. So, are the inputs okay? Before we look at this particular multiplex error? It does the choice of the next PC. Is it okay for everybody? So, what are my selections? So, when do I pick the BTB contents? When the KILL is 0; which means that, everything is okay; and, I have a hit in the BTB. So, that is 0 0. In that case, I will pick the BTB content as the next PC.

When do I pick PC plus 4? When KILL is 0 and I miss in ((Refer Time: 26:51)) when the selection is 01. And, if kill is 1; that overwrites everything else; which means I made a mistake in the past; I have to fix it now. So, if KILL is 1, the other bit is do not care. I will choose the target as the next PC that is coming from the execution stage. So, in the next clock, essentially, this PC will be latched here and I will be on the right path from then on. Is this hardware clear everybody? The PC selection? So, I essentially need a... It is actually a 3 2 1 marks; these two are actually in the same input. So, the only thing that may happen now is that, I have too many things on the critical path; that is, BTB in the critical path; I have multiplex on the critical path; which may actually lengthen the fetch cycle.

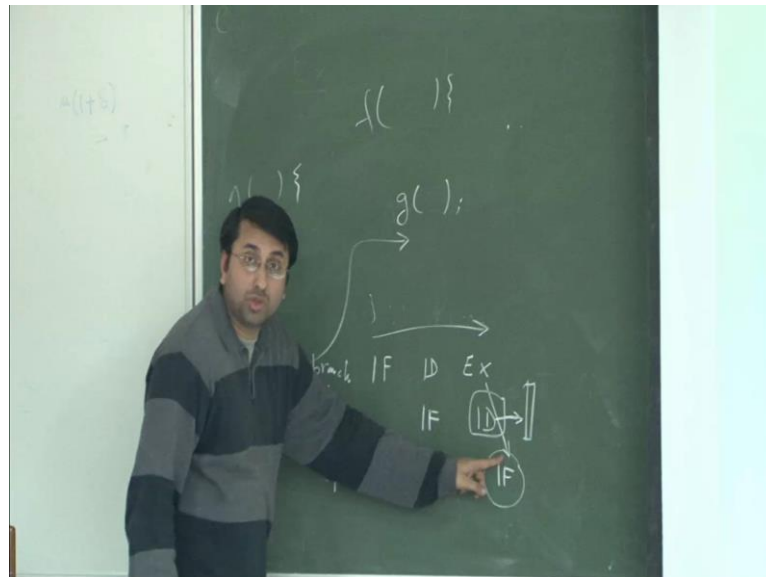
Student: What about the instruction in between that prediction?

Coming to that; yes.

Student: ... pipeline; we have ((Refer Time: 27:48))

So, that is exactly what is mentioned here. The ID/EX inputs are ANDed with tilde KILL. So, tilde is not killed. So, it kills 1; that means you are filling 0 in this particular register. So, if you go back to this diagram, the wrong instruction is currently here – currently being decoded, because I fetched an instruction; I made a prediction; which came here; and then, I used a prediction to fetch a new instruction; and, that instruction went into here. And, when the branch executes, the wrongly fetched instruction is here in the decode stage.

(Refer Slide Time: 28:32)



So, this is where my time goes. So, let suppose this is the branch instruction; this is the... Let us call it i naught that fetches here and that was fetched using the BTB outcome coming from here. And then, I have some i 1, which is fetched here. So, that is the phased execution; this is going to be correct all the time, because I will get the target from here. So, what we are talking about is how do you select the PC at this particular stage? I have three inputs. One is this target; one is coming from here, which is PC plus 4. The next instruction is this. And, one is that, I use this PC to look up the BTB. And the BTB told me something; there are three things.

Now, the point is that, if I made a wrong prediction for this branch for I looked up a BTB; that wrong instruction – now, where is it currently? This is currently the decode stage. When the branch is currently executing, that wrong instruction is here. So, what I

have to do is that, when this instruction – when the contents of this decode stage will be fed into register pipe; that has to be killed here; it should not be feeding these contents here, because these are wrong. So, what I am doing is that, I am ending all the ID/EX inputs with not killed. So, ID/EX 5. So, ID/EX 5 register is sitting here. So, KILL is 0; then, whatever you have computed will go in that register; otherwise, only zeroes will go into the register. Is the time line clear to everybody? What is happening when?

Student: i 1 is also because of that i 0; i 1 i f

No, i 1 is always correct, because we have a phased execution; i 1 will get the correct target.

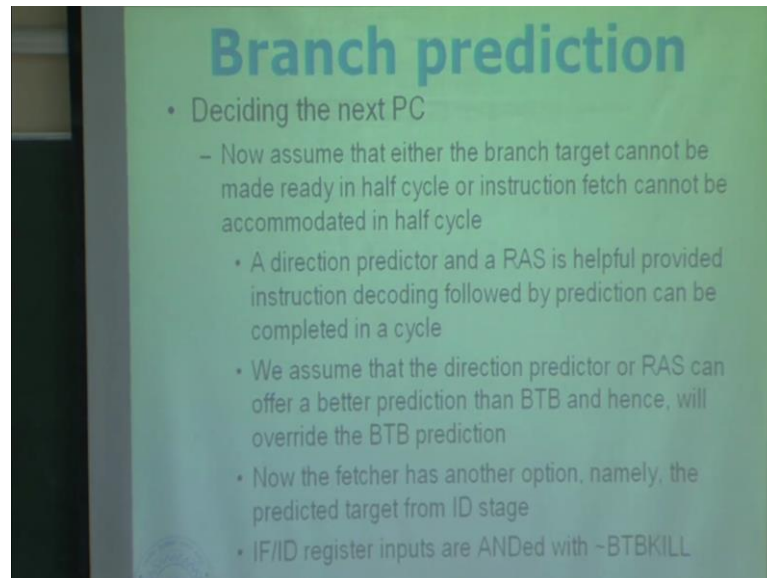
Student: That can because of dark prediction or that PC plus 4 also; that can be because of that.

No, no, no, no; i 1... So, in this diagram, I am actually selecting i 1's PC; that is exactly what is being done here. Should I pick... So, PC plus 4 is coming from here – this instruction. This instruction will also look up the BTB; the BTB outcome is coming from there. And, I have a target coming from here; which of these three should I pick? That is what I am asking actually. So, i 1 will actually proceed with the correct PC. And, remember that, it has to happen every cycle; this will be a continuous process; every cycle, it will be running this hardware; you should pick up one of these three things. Of course, in many cases, the target will not even be a legitimate phase, because there may not be branch instruction executing now in next stage; which is fine, because in those cases, the KILL will be 0 for sure; nothing to KILL actually. So, in that case, you will be selecting one of these two depending on whether you hit or miss in the BTB. Is this clear?

So, how that generator KILL signal? So, these are logic for KILL; it takes is control instructions. So, that is computed by the decoder, which is currently sitting in the decode execution pipeline register. It tells me whether the instruction currently execute in an execution stage is a branch or not. And, I add that with this particular clause; which says next PC ID/EX is not equal to target. So, notice that, next PC gets the PC whatever I calculate it. If there is a mismatch with this and the current instruction is a control instruction, I should kill; enable KILL. And, NPC is carried till ID/EX register, because beyond that point, this is not required anymore. And, is control instruction is generated

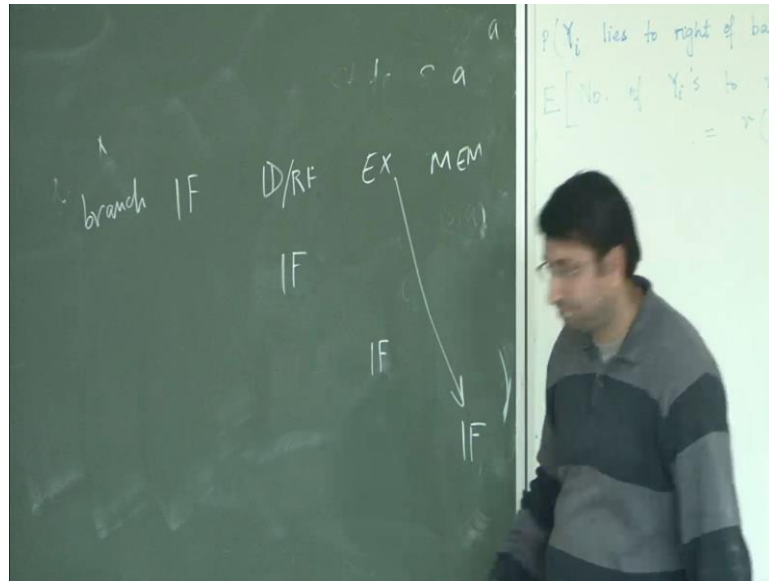
by the decoder. So, this particular variable will be high whenever the decoder comes across any control transfer instruction. Is this clear? If it is not clear, you are going to have a hard time in your next slide, because we are going to introduce the direction predictor; there will be a port outcome; there will be a port input in ((Refer Time: 33:03)) actually. So, ask now if it is not clear.

(Refer Slide Time: 33:10)



So, now, let us assume that, either the branch target cannot be made ready in half cycle or instruction fetch cannot be accommodated in half cycle. So, what does that mean? That means I have genuinely two bubbles that I have to nullify.

(Refer Slide Time: 33:27)

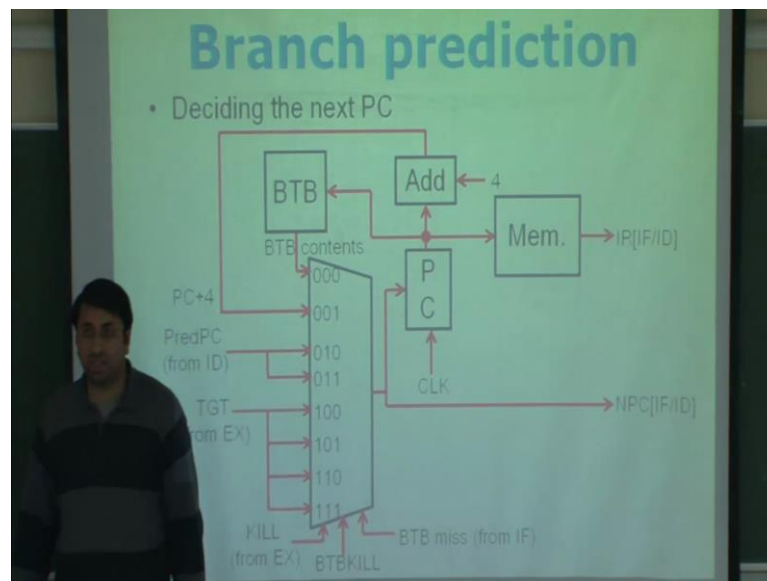


So, now the pipe timeline looks like this. So, this is the branch instruction. I will know the target only here. So, we have to fix up something here; maybe in these two cycles. And now, actually the direction predictor makes very much sense, because I can make a prediction here; which I can use in this cycle to overwrite the BTB prediction. So, a direction predictor and a return address stack is helpful provided instruction decoding followed by prediction can be completed in a cycle. So, now, you have to make sure that, you decode the instruction, look up the predictor; and, this whole thing can be computed in a single cycle; otherwise, of course, there is no use. We have to complete decoding and prediction in this one cycle. And, they have to be going serially; they cannot go concurrently. You first decode and then look up the predictor. You first decode; then, you will pop or push from the RAS. And, we assume that, the direction predictor or RAS can offer a better prediction than BTB. And hence, we will overwrite the BTB prediction. So, we will always do that. So, in some cases of course, it may be opposite that, the BTB may actually give you a better quality prediction; that is possible. But, in this particular discussion, we are going to overwrite the BTB prediction, which whatever the direction predictor tells you or the RAS tells you.

And now, the fetcher has another option namely, the predicted target from the decode stage, because the decode stage is normal to give another input into the fetcher; which is the outcome coming from the RAS or the direction predictor. And now, I have to generate one more signal, that is, BTB kill. So, let us try to understand what is really

happening. So, here I use this PC to look up the BTB. The BTB will tell me something, which I am going to use here. And, in this cycle, I may figure out that, I get a prediction from here from the direction predictor. So, I have to overwrite this one. So, that is this signal, which kills the instruction fetched according to the BTB's outcome. And finally, when the instruction executes, I may get another indication saying that, you made a mistake here also. Then, I will have to generate a KILL signal. So, I will have two KILL signals now: one – the BTB KILL; another is a traditional kill that we had last time.

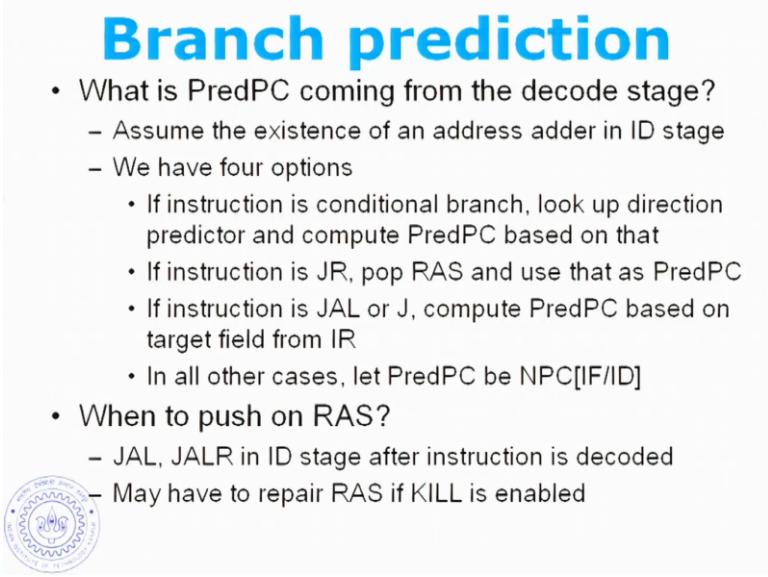
(Refer Slide Time: 36:06)



So, let us see what it looks like. I will have bigger marks for sure. So, I have a PC as usual. I compute PC plus 4; I look up the BTB; BTB gives me some contents. And now, to the multiplexer, I have ((Refer Time: 26:23)) the inputs; I have the BTB contents; I have PC plus 4; I have predicted PC coming from the decoder; and, I have a target coming from the execution stage. And, what I will assume here is that, whatever is coming in as predicted PC can be either from the direction predictor or from return address stack. I will now open up this particular thing here. So, now, what is the logic? You can easily figure out. So, when do I pick the BTB contents? When KILL is 0; when BTB KILL is 0; and, I hit in the BTB. In that case, I pick up the BTB contents. When do I pick up PC plus 4? When KILL is 0; when BTB KILL is 0; and, I miss in the BTB. When do I pick up the predicted PC? So, the predicted pc is picked up when KILL is 0. And, I have a BTB KILL signal coming in; and, I do not really care what is next. And, when do I pick the target? When the KILL is 1; everything else is gone, because KILL is

the golden rule. It tells me that, here is the correct target; it should go along this direction. Everything else here is a prediction; but, this is always correct. So, whenever KILL is on, I have to do everything else; I have to pick up this target and set the PC to the ((Refer Time: 37:49)) Is it clear?

(Refer Slide Time: 37:54)



Branch prediction

- What is PredPC coming from the decode stage?
 - Assume the existence of an address adder in ID stage
 - We have four options
 - If instruction is conditional branch, look up direction predictor and compute PredPC based on that
 - If instruction is JR, pop RAS and use that as PredPC
 - If instruction is JAL or J, compute PredPC based on target field from IR
 - In all other cases, let PredPC be NPC[IF/ID]
- When to push on RAS?
 - JAL, JALR in ID stage after instruction is decoded
 - May have to repair RAS if KILL is enabled

So, what is predicted? PC coming from the decode stage. So, let us assume the existence of an address adder in the decoder. So, we have four options here in the decoder. What are the options? If instruction is conditional branch, you look up direction predictor and compute PredPC based on that. The direction predictor will tell you either take fall through or go to the target. So, whatever it tells me to do, I will set PredPC to that. If the instruction is a return statement, that is, JR, you pop from the return address stack and use that as PredPC. If instruction is a jump and link or jump; so, these are direct procedure calls and unconditional jumps. Compute PredPC based on target field from IR. So, by the way, all though I call it PredPC, this is going to be correct all the time. This is actually correct; there is no prediction going on here. And, in all other cases, PredPC is NPC(IF/ID) whatever is carried forward from the fetch stage. So, remember that, what is NPC. So, NPC is this one; whatever PC I am using to fetch, I carry forward.

So, essentially, what I am saying is that, in the decode stage, if it is not a control transfer instruction, the PredPC is just the PC that carry forward; which means even in the decoder, PredPC will be selected by a multiplexer. I have a multiplexer, which I am not

showing that in the decode stage. When to push on the RAS? So, you push on the RAS, when you encounter a procedure call instruction in the decode stage after the instruction is decoded. And, you may have to repair the RAS if KILL is enabled. Can somebody say what this is actually? KILL is enabled when I made a misprediction; which means I fetched at least two instructions; I fetched two instructions along the wrong path; exactly two instructions. Then, I got a KILL signal telling me that, these two instructions are wrong; you should redirect yourself along the right path. Why do I have to repair the RAS in that case?

Student: You have to remind me to pop out ((Refer Time: 40:11))

Why is that?

Student: If we have to push something, some wrong address we have.

And, how can that happen?

Student: Maybe some kind of a function call is there and we have pushed that...

Which all are the function call here? You are right; just tell me which one of these are function call? Why should I push something on the RAS? Only when I see a function call. And, when do I see a function call here? Suppose this branch is mispredicted. So, I can KILL signal here finally, when I try to fetch.

Student: Execution; at the time execution – at the time of execution we have to...

When I push on the RAS in the decode stage I said. So, when I get the KILL signal here, why do I have to repair the RAS? What might have corrupted the RAS? Yes?

Student: New instructions, which we have just fetched – they might have ((Refer Time: 41:11))

Exactly. So, if one of these is a function call, you might have pushed something on the RAS, which is wrong; you should have pushed actually. What if one of this is a return instruction? You have popped something out of the RAS. That is a major headache actually. There is no way to repair it now; we have popped something out. So, I am not going to give details of this; there are ways. Can somebody guess what might be of way to fix this?

Student: Do not do RAS operation ((Refer Time: 41:44))

What is the point? I will have to push a pop. Is it clear? If I have a return instruction, it is a ((Refer Time: 41:55)) There is no way to get the RAS back to the correct stage. So, what can you do? How to fix it?

Student: Can we use secondary RAS?

Secondary RAS; you can copy the RAS; exactly. So, one possibility is that... So, you probably... We probably encounter this particular idea over and over as we go forward and do all these predictions. This is called check pointing. So, before you pop the RAS, you can check point the RAS. You can copy the RAS entries somewhere, so that whenever you say a KILL signal, you copy the RAS and it is back. So, you are getting the right cycle. So, it gradually gets complicated; we will see why. Now, we are just dealing with maybe one or two batch instructions. As the pipe gets longer and longer, you may make too many wrong predictions; and, too many wrong things may get to the pop; too many to the RAS, and too many pops may have out of the RAS. The question now here is – how many check points can I make? So, there are many issues coming up. So, anyway; so, this is a difficult problem.

Student: Sir.

Yes.

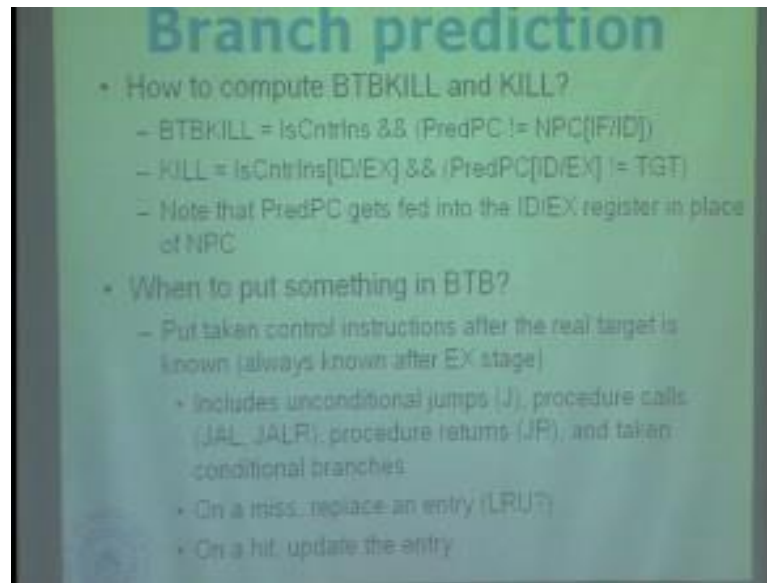
Student: Sir, here only the second instruction can corrupt the RAS – the second...

This one? Yes

Student: That one cannot...

Yes, this one cannot; you are right; yes. Yes, because this one only get a chance to go to decode; this one will be at KILL even before that – this instruction.

(Refer Slide Time: 43:19)

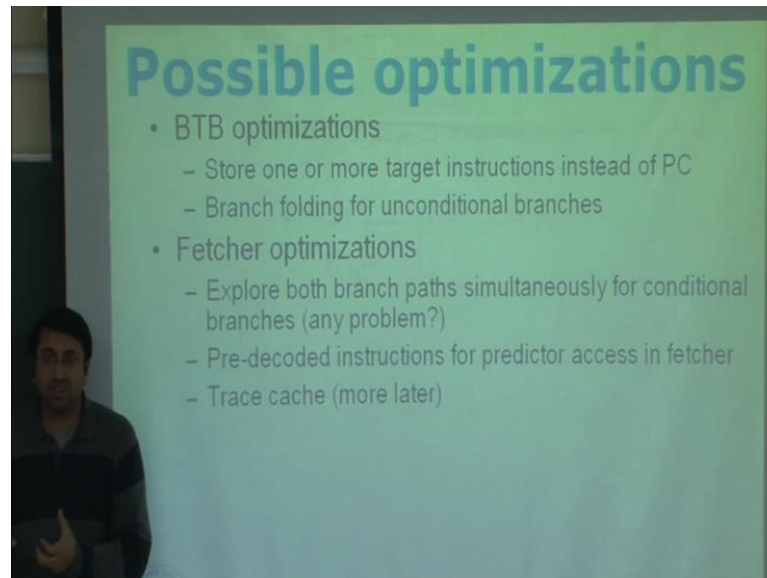


How to compute BTB KILL and KILL? So, BTB KILL is control instruction; and, PredPC is not equal to NPC. So, whatever you have predicted in the decode stage does not match with whatever PC I carried forward from the fetch stage. So, that means I made a mistake in the BTB. And, KILL is same as before. Is control instruction and PredPC not equal to target? So, last time, we had NPC. Now, NPC will be staged to PredPC, because the PredPC is the one that overwrites NPC now. And, that is what is carried forward. So, note that, PredPC gets fed into the ID/EX register in place of NPC. Is it clear – this particular hardware?

So, these are very generic hardware now. We have incorporated pretty much every possible prediction that you can do related to control transfer instructions. So, this is very generic. This is what is found in the fetcher of every modern processor. Only thing is that, as the pipeline gets longer, you may have to kill more instructions. I have this particular selection of PC hardware; it is more or less same. And, when to put something in the BTB I want to tell; we have already discussed this actually. Put taken control instructions after real target is known. And, it includes all control transfer instructions – conditional, unconditional jumps, procedure calls, procedure returns and taken conditional branches. All control transfer instructions will go into BTB even though we have a separate RAS for handling these instructions, because BTB is providing you an ((Refer Time: 45:01)) prediction for everything pretty much. On a miss, replace an entry;

you can have a replacement policy like LRU or anything on a hit updated entry. So, we have talked about this topic.

(Refer Slide Time: 45:13)



So, possible optimizations; you can optimize a BTB to store one or more target instructions instead of actually PC. So, instead of storing that target PC, you can actually store the instruction itself, so that you can nullify one fetch operation. So, is it clear to everybody what I am talking about here? So, currently, BTB stores the program counter and you have to take the program counter and fetch the instruction, instead of saying... Do not even fetch; put the instruction itself in the BTB. So, for unconditional branches, this is called branch folding. Essentially, you are nullifying one instruction there.

Fetcher optimizations – you could explore both branch paths simultaneously for conditional branches. So, do you see problem with this? So, I am saying that, I do not really have a predictor. So, I will go along both the paths and start fetching from both the sides. And eventually, I will figure out which one is correct. So, I will nullify one of the paths. So, what is the problem with this? So, no processor actually does this. So, that was some ((Refer Time: 46:29)) What is that?

Student: More than two ((Refer Time: 46:32))

They will actually try to...

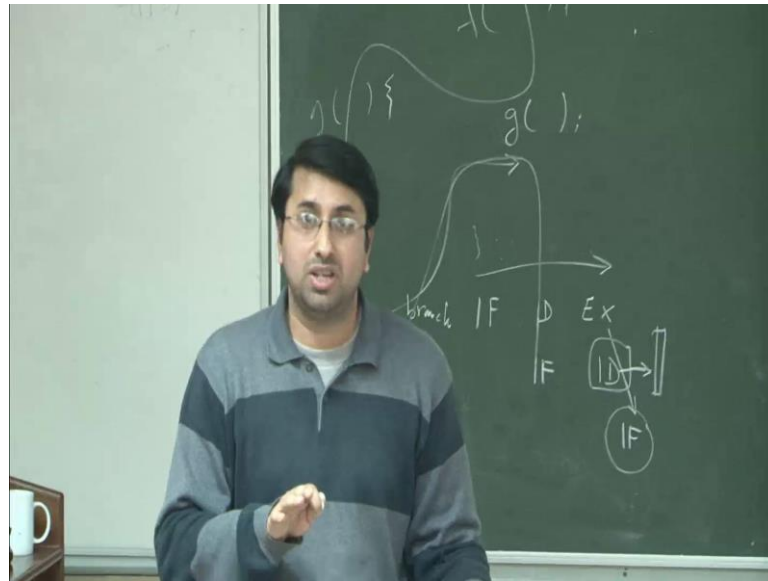
Student: ((Refer Time: 46:41))

So, let us assume for now that, whatever you compute, you do not store them actually in the persistent memory; let us suppose that we have some BTB buffers; we can buffer ((Refer Time: 47:00)) So, we will get to that point later. So, that is not a problem actually. Is there any other issue? So, what she has pointed out is interesting; she is saying that, while we are fetching these wrong instructions, what if they actually modify some registers or anything? So, till now, that problem is not there, because these instructions do not execute; even before executing, they get killed. So, will get to that point very soon. What is the problem here? Exploring both the paths simultaneously; yes.

Student: We always need extra ((Refer Time: 47:34))

Yes, exactly. So, we are just wasting resources here by doing this. Essentially, half of our instructions will be killed anyway. So, why do that? Rather have a predictor, which will give you more than 50 percent probability of incorrect, because here essentially, I am assuming the time correct with probability half; that is all I am saying. And, I know that, other half will get killed. But, the predictors may be smarter. And, they can give you very high accuracy – much more than 0.5 probability of incorrect. You can pre-decode instructions for predictor access in the fetcher, because we have a problem here that, we cannot access the direction predictor until we decode the instruction. So, what I am saying is that, you could have a single bit in the instruction, which says oh, this is a conditional branch. So, the fetcher can immediately look up that bit and can access the predictor right in the fetch stage. So, then, you have a much better prediction to start with actually; you do not have to rely on the BTB, which tells you what happened in last time. And, there is something called trace cache. So, we will talk about that more later. It is essentially, roughly speaking, it is a cache, which stores the dynamic sequence of instructions that we encountered last time we executed along this path.

(Refer Slide Time: 48:56)



So, essentially, what happens is that, if you are executing this particular function f , you would actually store instructions in this sequence, instead of storing f somewhere and g somewhere. You will actually store it like this. So, you will store the trace that you executed. So, next time, when you look up the trace cache, you know which way to go. But, again the problem is that, it tells you what happened last time. So, traces may change depending on your control paths. So, of course, there are ways to ((Refer Time: 49:23)) the problem; and, we will talk about that later in the trace cache.

So, before I go to direction predictor, which probably I will be going to start today or maybe I can introduce that. Any problem or any question in this till now? So, if we have... We have not yet opened up the direction predictor model. We are saying that, there is something sitting there – a black box, which gives you and gets to answer. You ask... Tell me what to do with this conditional branch? It either says taken or says ((Refer Time: 49:56)) So, there has to be something going on inside. We will open that up now; but, before that, any question?

Student: ((Refer Time: 50:04))

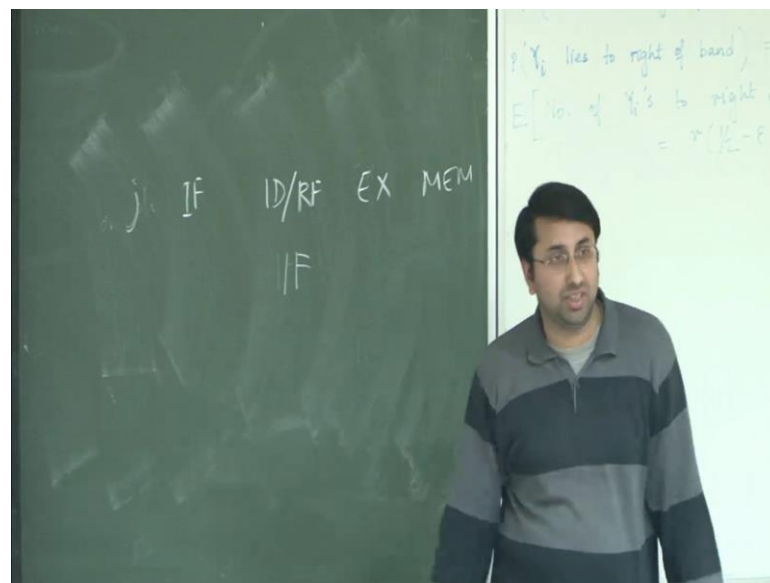
Yeah; which one? Second one? So, what here I am saying is that, see here we could not access the direction predictor until we decode the instruction, because I know that, at this point only, I know that, it is a conditional branch. I am saying that, suppose you pre-decode the instruction; and, you put a single bit in the instruction, which tells me if it is a

conditional branch or not. So, fetcher can look up that bit and then access the predictor right here actually.

Student: BTB optimization – even if we say instruction BTB, we have to spend entire cycle in instruction fetch. What do we achieve? I mean the cycle time will be...

No, no, no; you saved one instruction right? So, let us see what is happening in a bubble. So, you have an unconditional jump instruction.

(Refer Slide Time: 50:59)




So, traditionally, what would have happened? This instruction would go through the pipeline. In this stage, it gets fetched; in this stage, it actually computes the target or maybe here depending on where you have the address adder. And then, if you have nothing to do with these two stages ((Refer Time: 51:24)) Here what will happen is that, you look up the PC; you take the PC; you look up the BTB. BTB tells you the target instruction, which would have got fetched here. But, it actually gets fetched here itself and gets to the decoder here. So, you saved the cycle effect. Any other question?

(Refer Slide Time: 51:49)

Direction predictors

- How about static prediction?
 - Always not-taken (NT) or always taken (T): penalty?
 - Forward not-taken and backward taken (rationale?)
- Deeper pipelines
 - Define branch penalty
 - Example: MIPS R4000 takes 3 pipe stages to compute the target and to evaluate the condition after fetch; assume 4% unconditional jump, 6% NT conditional branch, 10% T conditional branch; evaluate CPI increase for three schemes: unconditional flush, predicted always T, predicted always NT
 - **Big problem: deeper pipelines increase branch penalty**
 - Must have better branch predictors for deeper pipeline



MAINAK CS422 57

So, I will just introduce the basic thing here. So, people started with... So, is the problem clear to everybody that, I have an additional branch and I want to build a function. The input to which is this particular branch instruction; and, the outcome is binary – zero output taken or ((Refer Time: 52:11)). So, to begin with, people looked at static prediction; that is, you take a conditional branch and you predict always not taken or always taken. So, what is the penalty associated with it? Depends right in the branch ((Refer Time: 52:31)) If you say always not taken, maybe you are always correct; that depends on a branch. However, the point is that, it is very simple. And, this can be actually done in compile time. So, we can... So, always it is not taken. So, for example, if you take a conditional branch – the loop conditional branch; and, if you predict always not taken, you will be wrong most of the time, because the loop ((Refer Time: 52:59)) branches are actually taken most of the time; it is not taken only once last time.

So, once people made this observation that loop branches are special types; so, they said, well, let us improve this static prediction. We will say forward not taken and backward taken. So, whenever you have a forward branch, that is, here is a branch instruction and its target is actually in front of it. Then, I will say that, it is not taken. But, the target is before it; then, I will say taken. So, essentially, what happens is that, the loop branches will follow in this category in the backward branches. And, I will say always taken for those branches. So, now, I am correct, except the last time. And, usually people would use forward not taken for ((Refer Time: 53:47)) type of branches, where your target is actually forward in front of you. So...

(Refer Slide Time: 53:56)



Essentially, you have an if condition and then we have else. So, this branch's target will be this one actually. it is a forward branch. And so, you say forward not taken means that, you are actually saying that, for if else type constructs, the if part will be executed most often. And actually ((Refer Time: 54:20)) That is very interesting. So, it maybe because the way we think is that, we put the true part first before we put the false part. So, it tells what the forward not taken is; backward taken is a very good ((Refer Time: 54:34)) actually in most cases.

Now, what happened is that, this was pretty good actually; but, of course, you can improve over that. If you observe the dynamic behavior of the latches. So, just to give you the problem, give you some idea about the problem. Assume that, you have a pretty deep pipeline, not like a pipe-stage pipe; we have more stages actually. So, we talked about branch penalty at some point. So, branch penalty is essentially number of cycles that you lose because of a wrong prediction. So, it is essentially the time from when the branch is fetched to the time the branch executes. So, how many cycles we have in between is the maximum branch penalty that you pay.

So, here is an example; suppose a processor – MIPS R4000 takes three pipe stages to compute the target and to evaluate the condition after fetch. So, after fetch, you have three more pipe stages before you get to the execute stage as opposed to one. Assume 4 percent unconditional jump, 6 percent not taken conditional branch, 10 percent taken conditional branch. Evaluate the CPI increase for three schemes: unconditional flush, predicted always taken, predicted always not taken. So, I will talk about this problem

next time maybe because we do not have time. So, we will see what exactly the issue is when you evaluate this ((Refer Time: 55:56)). So, we will start from here.