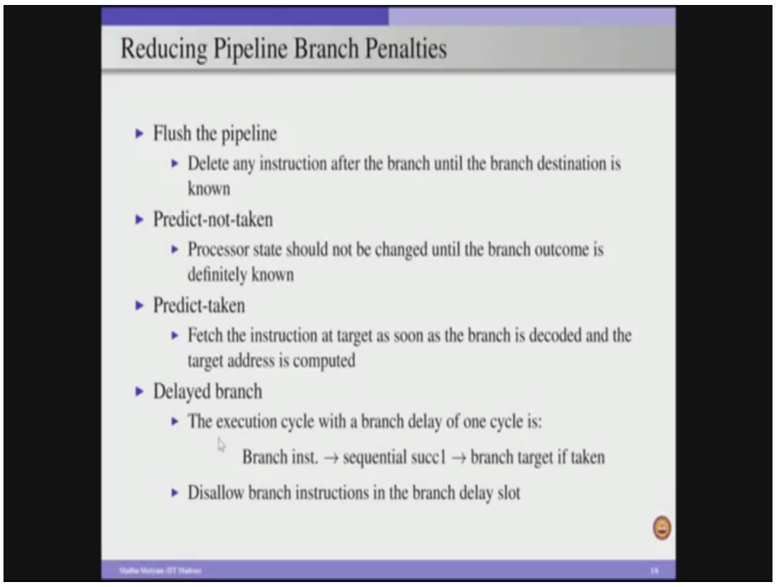
Computer Architecture

Prof. Madhu Mutyam
Department of Computer Science And Engineering
Indian Institute of Technology, Madras

Module – 05
Lecture – 16
Handling Pipeline Hazards (Part 2)

So in the last module, we discussed a various pipeline hazards. And we also discussed that among all the pipeline hazards, the control hazards are going to create significant performance penalty. So, as a result, we need to look at the techniques to minimize the penalty associated with this control hazards. So, we know that the control hazards happen because of the branch instructions, which alter the normal flow of execution of the program.

(Refer Slide Time: 00:46)



Reducing Pipeline Branch Penalties

- Flush the pipeline
  - Delete any instruction after the branch until the branch destination is known
- Predict-not-taken
  - Processor state should not be changed until the branch outcome is definitely known
- Predict-taken
  - Fetch the instruction at target as soon as the branch is decoded and the target address is computed
- Delayed branch
  - The execution cycle with a branch delay of one cycle is:

    Branch inst. → sequential succ1 → branch target if taken
  - Disallow branch instructions in the branch delay slot

So, simple technique one can use to deal with this control instructions is flushing the pipeline. Here the processor will treat the branch instructions also like any other instruction. And as soon as the target address is computed for a branch instruction then if target address is different from the, the next instruction of the branch instruction then we flush the pipeline. Otherwise we will continue executing the instructions as it is.

So, this is like a simple technique which will not add any extra hardware overhead to the processor, but it is going to create a significant penalty in terms of performance as long as, if our applications have large number of branches and most of the times the branches are taken.

So, as a result, like this flushing the pipeline is, though it is a simple technique, it may not been efficient from the performance point of view.

So, rather than considering this flushing the pipeline, so we can consider a different type of other techniques, where we can take the help of both the hardware as well as the software. So, our underline hardware can implement specific technique, but the software, that is a compiler, can exploit this underline feature, and then reorganize the code to minimize the penalties associated with this branches.

So, first technique is Predict Not Taken. So here our hardware, the processor always assumes that the branches are not going to take place. So, once we have this predict not taken, so automatically after the branch instruction the processor fetches the next instruction in the sequence, in the program order. And we know that after the branches fetch and after some time the branch target will be computed and if this particular branch instruction is going to take place. That means it is going to get a new target address and at that point of time our prediction is wrong and as a result we have to flush the pipeline.

But as long as if the branch is not going to take place, then we are going to get the benefit from this particular technique. So, in other words the simple technique where the hardware side it always assumes all the branches are not going to take place. Note that here it is independent of a branch instruction and so on. Application can have multiple branch instructions, but once our hardware or the processor is designed in such a way that it always considers this predict not taken, then it assumes that all the branches are not going to take place.

So as a result, for any branch instruction in the program it, the processor always going to start fetching the next instruction after this branch, and the control will alter only when our prediction is wrong. And so as a result if the prediction is wrong then we flush the pipeline at that point of time, and then start fetching the instruction from the target address. In other words, in this process, in this Predict Not Taken technique processor state should not be changed until the branch outcome is definitely known.

So, our compiler can exploit this behavior or compiler can know that the processor is implementing all the branches in a Predict Not Taken fashion, and then it can reorganize the code, so that it favors this predict not taken scenario. If the applications have branch instructions, which are not going to take place most of the times then this technique will be efficient from the performance point of view.

And this next technique is Predict Taken. For any branch instructions, the processor is going to assume that the branch is going to take place and as a result, so in order to execute the instructions from the target address, first of all we have to know the target address. So effectively, once a branch instruction is fetched and after decoding we have to compute the target address and only then we can start fetching the instructions from the target address.

So for example, the five stage pipeline whatever we considered in the previous module. So, we cannot exploit any benefit from this Predict Taken technique and this Predict Taken technique will be helpful only when the target address is known before the actual branch outcome is released. So, as a result among these two, Predict Not Taken and Predict Taken, the Predict Not Taken will be beneficial for improving the performance using our compiler support.

And finally, we can also consider another technique called as the delayed branch. So, here what we can do is, we insert delay slots between the branch instruction and the target instruction or the next instruction following the branch. Effectively, we insert few delay slots and these delay slots will be filled with useful instructions and again, since we are going to take the help of compiler and compiler going to put useful instructions in these delay slots.

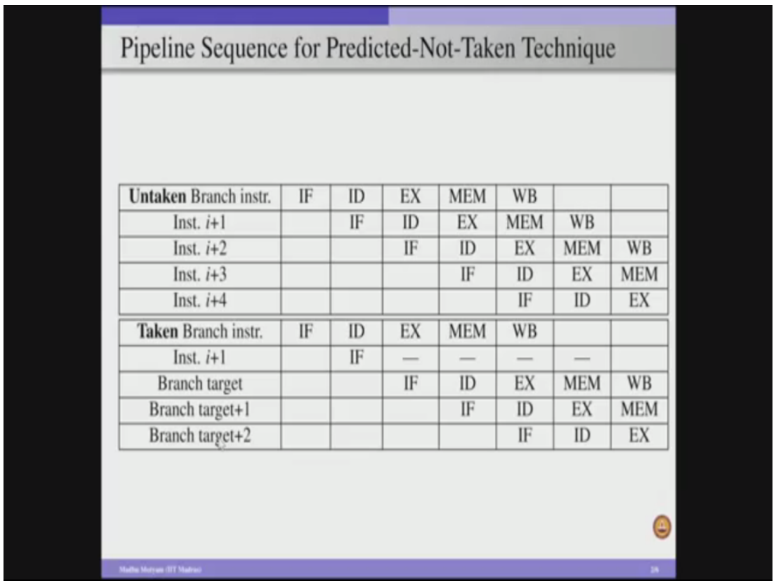
And we can consider any number of delay slots after the branch instruction, but to make the design simple, we can consider one delay slot after the branch instruction. So, this delay slot will immediately follow the branch instruction. So, as a result, once we have this Delayed Branch Technique, our sequence of instructions in our code will become like first branch instruction, then the sequential successor this is a delay slot and then the branch target if taken.

So once we have this delay slot and if we keep useful instruction in this delay slot, so that while the processor is busy executing these useful instruction that is there in the delay slot, we compute the target address for the branch, if the branch is taken. And by the next cycle, we already know the target address so if the branch is taken, we flush or we discard the whatever instruction that was there in the delay slot, and then we continue with the target address. And if the branch is not taken and if our delay slot is actually having the next instruction to the branch instruction, then we will, we do not have to do anything and we just continue executing the instructions in the program order.

Effectively so we can minimize the penalty associated with the branches by using these Predict Not Taken, Predict Taken and Delay Branch mechanisms. And for these three techniques, we take the help of compiler and compiler reorganizes the code based on the technique what we implement. Once we have the support of hardware and the software and we can minimize the penalties associated with the branches, and as a result we can improve the overall performance. And note that in most of the benchmark applications the branch instructions contribute to 22 to 25 percent of the instructions. And in a deeper pipelined processor, if we are not going to take care of these branches then we will get a significant performance penalty.

As a result, we need to look at efficient branch penalty minimization techniques to improve the overall performance. By the way, so in the delayed branch technique, we are not supposed to insert a branch instruction in the delay slot. The main reason here is, because if we keep another branch instruction in the delay slot and we do not know what is the outcome of the branch instruction? Whether the branch is going to take place or not take place. As a result this is going to complicate the overall things and that is the reason why typically, we disallow keeping a branch instruction in the delay slot. So, other than the branch instruction we can keep any instruction in the delay slot. Now, we will discuss the Predict Not Taken with an example.

(Refer Slide Time: 09:32)



Pipeline Sequence for Predicted-Not-Taken Technique

Untaken Branch instr.	IF	ID	EX	MEM	WB		
Inst. *i*+1		IF	ID	EX	MEM	WB	
Inst. *i*+2			IF	ID	EX	MEM	WB
Inst. *i*+3				IF	ID	EX	MEM
Inst. *i*+4					IF	ID	EX
Taken Branch instr.	IF	ID	EX	MEM	WB		
Inst. *i*+1		IF	—	—	—	—	
Branch target			IF	ID	EX	MEM	WB
Branch target+1				IF	ID	EX	MEM
Branch target+2					IF	ID	EX

So, consider a scenario where we have a non-taken branch instruction, we just following through the five stage pipeline instruction fetch, instruction decode, execute, memory access and the write back. And because this is Predict Not Taken technique, so that means our underline processor is implementing the Predict Not Taken. That means for any branch instruction in the program the processor always assumes that the branch is not going to take place.

And then it can start fetching the instructions immediately after the branch instruction. So, as a result, after the branch instruction is issued at time t. In time t+1 we fetch the next instruction following the branch instruction that is instruction i+1. And that will be in the instruction fetch stage, while the branch instruction is in the instruction decode. And then at the end of the instruction decode we know that the branch is not taken, and as a result we continue with instruction i+2, i+3, i+4 and so on.

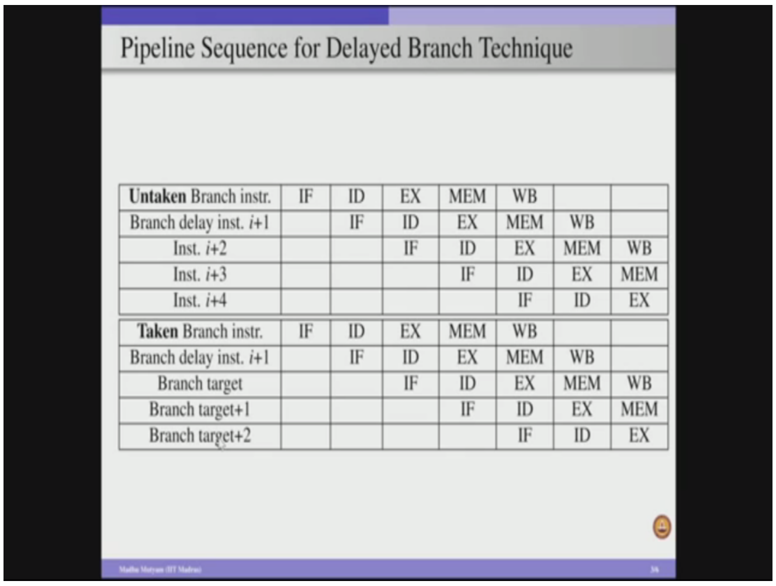
So, as a result, so there is no problem as long as if the branch is not taken and our underline processor is assuming all the branches are not going to take place. So effectively there is no performance penalty and we get very good performance with this code. But consider another scenario; still here also our processor is always assuming that the branches are not going to take place. And but unfortunately the branch instruction is taken, as a result the branch instruction has to go to a different target address rather than the next address in the sequence.

But because we issued branch instruction at time t, while the branch instruction is in the ID stage we fetch instruction, which is following the branch instruction in the program order that is instruction i+1. But at the end of ID stage for the branch instruction we came to know that the branch is taken. So, as a result, we already computed the target address for this branch instruction. Now we have to flush the pipeline, which is currently holding the instruction i+1.

In other words, we have to treat the instruction i+1 as no-op instruction and we re-fetch the instructions starting from the target address specified by our branch instruction the ID stage. As a result in the while the branch instruction in the execute stage, we are fetching a new instructions specified by the branch target address, and that is in the instruction fetch stage. And we will continue with the subsequent instructions with respect to this branch target address.

In other words, when we are considering always Predict Not Taken technique, and if the branches are taken then we are going to incur a penalty. And that penalty is, depends on what time the branch instruction issued, and at what time the branch outcome is realized and at what time the branch target address is computed. So, as a result we have to realize the target address and we have to realize the outcome of the branch as quickly as possible, with respect to the time at which the branch is issued so that the penalty can be minimized. Now we will consider another example which illustrates the concept of delayed branch technique. In this delayed branch technique,

(Refer Slide Time: 13:10)



Pipeline Sequence for Delayed Branch Technique

Untaken Branch instr.	IF	ID	EX	MEM	WB		
Branch delay inst. *i*+1		IF	ID	EX	MEM	WB	
Inst. *i*+2			IF	ID	EX	MEM	WB
Inst. *i*+3				IF	ID	EX	MEM
Inst. *i*+4					IF	ID	EX
Taken Branch instr.	IF	ID	EX	MEM	WB		
Branch delay inst. *i*+1		IF	ID	EX	MEM	WB	
Branch target			IF	ID	EX	MEM	WB
Branch target+1				IF	ID	EX	MEM
Branch target+2					IF	ID	EX

So, if the branch is not taken, let us assume that the branch instruction is issued at time t and this branch is not taken. So, when the branch instruction is in the ID stage, we are fetching an instruction, which is there in the delayed slot. We assume that the delay slot is actually having the instruction, which is following the branch instruction in the sequence. So, effectively delay slot is having instruction i+1 and we fetch that instruction and we will continue, but at the end of the ID stage for the branch instruction, we came to know that the branch is not taken.

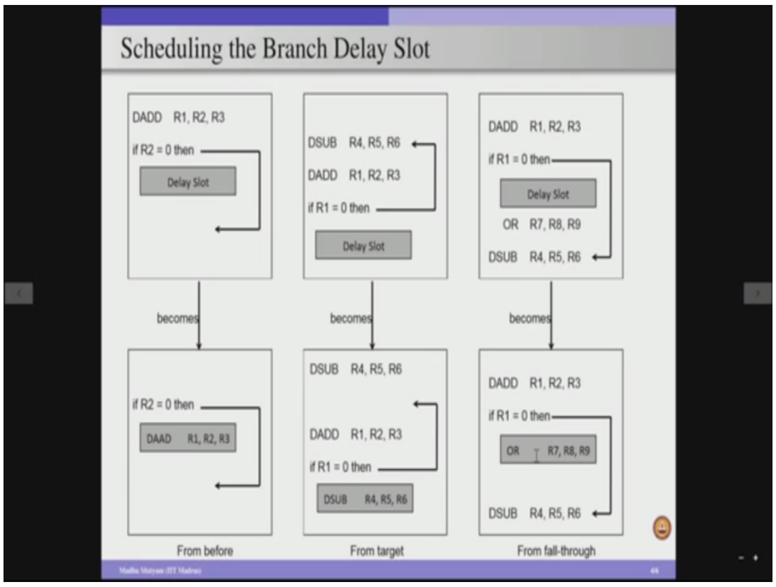
So, as a result we do not have to alter the program order and we just continue fetching the subsequent instructions and then these instructions will go through the remaining pipeline stages. So, there would not be any stalls and as a result no performance penalty we get. Now, consider another scenario, if the branch instruction is taken, so in this case. So, what we have done is in our the delay slot, we kept instruction which is following the branch instruction, and at the end of the ID stage of the branch instruction we came to know that the branch is taken.

So, as a result whatever the instructions we fetch, that is stored in the delay slot will be void now and then we can continue with the target address instruction. But, remember if our delay slot is actually going to store some instruction, which is independent of the branch condition and also is an useful instruction in the program, then we can complete the execution of that useful instruction without discarding it. As a result we do not have to waste this one cycle of the pipeline, and as a result we can improve the performance.

So, the catch is in the delayed branch technique if we want to get better performance improvement, then we have to fill the delay slot with some useful instructions. And again because we are going to take the help of compiler to fill the slots, these delay slots then if the compiler can identify independent instructions and those are useful then we can eliminate even the wastage of this one pipeline cycle stage and we will get significant performance improvement.

So, in other words among all the three techniques as long as if the compiler can find useful instructions from the program, then the delayed branch technique will improve the overall performance, compared to the other two techniques, Predict Not Taken and Predict Taken. Now as mentioned earlier, if we can keep useful instruction in the delay slot, then we can minimize the penalties associated with the control hazards significantly, but how do we select useful instruction? So we will consider three scenarios here the first scenario is -

(Refer Slide Time: 16:12)



Scheduling the Branch Delay Slot

We have an add instruction, which is going to read to register contents R2 and R3, and then perform the add operation and store the result in R1 and following this add instruction, we have a compare, branch instruction which is going to compare R2 content with 0. And if branch is equal to 0 then we are going to go to this target address, otherwise we are going to execute the delay slot instruction. So, how the compiler can reorganize this code such that the delay slot is filled with useful instructions.

So, if we see this code, we know that this ADD instruction is independent of this branch instruction, because in both the instructions R2 is used but R2 source operant in add operation and whereas the branch instruction is using R2 for comparing. So effectively, even if we move this ADD instruction to the delay slot, we are not going to get any functional incorrectness. So, as a result if we can rearrange the code like this, where our delay slot is actually having this ADD instruction.

Now we can see if R2 is equal to 0 if the condition is true, but that condition we know whether the condition branch is going to take place or not only in the second stage of the execution. And by the time we already fetch the instruction that is there in the delay slot, but now the delay slot is having useful instruction because we are supposed to perform this add operation according to our original code. So as a result our delay slot is not wasted and it is actually performing useful computation.

So, even if the branch is taken, here we are not wasting the delay slot and we are performing useful computation. And after the once cycle delay, then we will know the target address and then we start fetching the instruction form the target address. And the other case for example, if the branch is not taken here, in that case so we first fetch the ADD instruction that is there in the delay slot and after that we proceed with the remaining instructions after this branch instruction. In both the cases we are not going to lose any performance, and as a result if we can find useful instruction to keep in the delay slot we are going to gain significant performance improvement.

But again not always we will have that luxury of finding independent instructions. So compiler may not find useful instructions always, useful as well as independent instructions. So, if we cannot find useful independent instructions, then we have go to the next option, where we will consider another. Example here we have a subtract operation which performs a subtract on contents of R5 and R6 and store the result into R4. And then we will perform an add operation which performs add of contents of R2 and R3 and store the result in R1.

And there is a branch instruction which is going to compare whether R1 is equal to 0 or not. If it is 0 then we are going back to this subtract operation and we will repeat. And if the condition is false, then we will just follow the instructions that are there after this branch instruction. Now we have a delay slot after this branch instruction and we have to fill this delay slot with some useful instruction. So which instruction we can keep it here.

We can store this subtract operation in the delay slot, and if the condition is true that will be known only in the ID stage of this branch instruction. And by the time we already fetch the instruction that is there in the delay slot, which is nothing but the subtract instruction and after fetching this if we know that the branch is taken then we have to go to the target address, but now the previously the target address was pointing to the subtract instruction in the original code. But now we pointed this target address to the next instruction following the subtract operation. And we already executed the subtract operation.

So, effectively the functionality wise both are doing the same thing, we are not wasting the delay slot when the branch is taken, but now consider a scenario where the branch is not taken. So, according to the original code, if the branch is not taken, we are supposed to execute instruction that is following this branch instruction. And we are not supposed to execute this subtract operation, but after the code is transformed to this form then we know that if the branch is not taken. Even then the immediate slot associated with this branch instruction, we are fetching this subtract instruction. And so this is wasting one pipeline cycle.

So, effectively if the branch is not taken in this scenario, we are going to waste one pipeline cycle, but if the branch is taken then we are actually not wasting any cycles and then getting the useful computation. So, compared to the first technique, this is not going to give you that much performance improvement, but this will give you performance improvement as long as branches are going to take place. Effectively, if most of our branches are going to take place with high probability, then we can implement this type of technique in our code.

Consider an example here in the third case, we have an ADD instruction, which is performing add of R2 and R3 and store the result in R1. And there is a branch instruction which is comparing R1 content with 0, if it is true then we are going to execute the subtract instruction, otherwise we are going to execute the OR instruction. And now see here this branch instruction is using the operand, which is the destination operant for the add instruction.

And similarly, even in the previous case also, here the branch instruction is going to take the operand, which is the destination operand for add instruction. Effectively, in both the cases our branch instruction is dependent on immediate previous instruction. So, as a result, we cannot keep that previous instruction in the delay slot. And now in this third example, so what we can do is, we can keep the OR instruction in the delay slot. So, this example will be helpful for scenarios, where most of the time the branches are not going to take place. If the branches are not going to take place with the high probability, we can always fill the delay slot with the instruction which is following the branch instruction.

Now you can see clearly, if this branch is going to be false, that means it is not going to take place then the delay slot is actually having the instruction, which is following the else path of this condition. So, as a result we are performing the useful computation. And if the branch is going to take place that means, if it is going to take the true path then we are supposed to execute this instruction, but we are executing the OR instruction then we are going to get a performance penalty, a performance penalty of one cycle. So effectively this third technique will be helpful for the cases where branches are not going to take place with high probability. And the second technique is helpful in the cases where the branches are going to take place with high probability.

And the first case is actually helpful where if the compiler can identify, can find useful independent instructions in the program. So, effectively with the help of the underlying, the implementation for a handling the branches, the software or the compiler can re-arrange its code, the rearrange the program code in such a way that, it can fill the delay slot with useful instructions and then minimize the penalty associated with the branches.

And as I mentioned earlier, because the branch penalty is significant and it is critical for improving the overall performance so, we have to minimize the branch penalties, and once we do that then we can get a significant performance improvement. And we already discussed in the previous module that, the speed at that we achieve with the pipeline design is

$$Speedup_{Pipeline} = \frac{Pipeline\,Depth}{1 + (Branch\,frequency * Branch\ penalty)}$$

assuming that we do not have any hazards associated with the data and the structural hazards. And in that scenario, if we are going to minimize the branch penalty, we can get a significant performance improvement with the pipeline as compared to a non pipeline design.

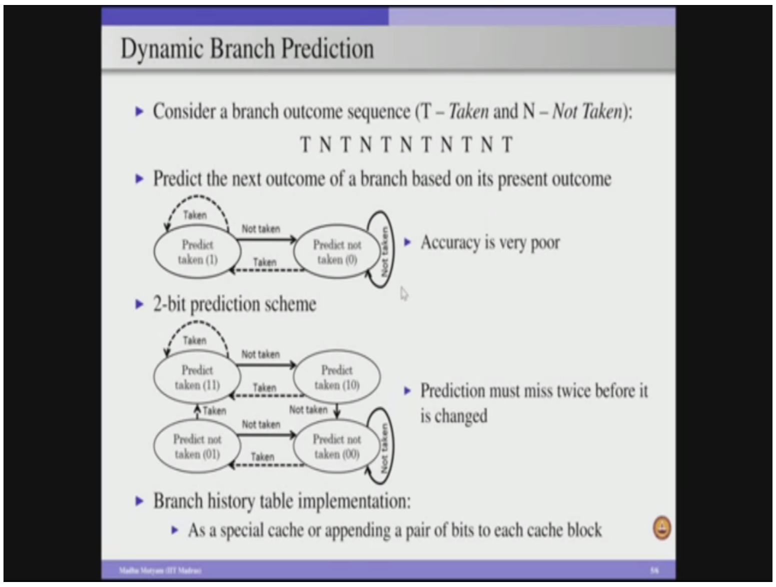
And so fore we discussed the techniques which are static in nature, that means when the processor is designed in such a way that it always can assume that all the branches are not going to take place or it can always assume that all branches are going to take place, or it can do nothing with the branches, or it can insert a delay slot and the compiler can insert a delay slot, useful instruction in the delay slot and so on. But here in all these techniques, we are not worried about the outcome of the branch, we are just treating all the branches equal, but in reality that is not a case.

So, in reality what happens is, same branch instruction when it is occurring several times in the program execution can behave differently. One time it can take the branch next time it may not take the branch and so on. So, as a result branch outcome can be one instance can be true and in the other case it may be false. So, once you have that then you have to exploit that behavior and then come out with efficient mechanisms to deal with these branch penalties.

In other words, these static techniques may not be helpful always, as we change the input to the program then branch outcomes can change significantly, and as a result we need to come up with dynamic mechanisms, which will adopt based on the run time conditions. And for that there are several techniques proposed in the literature and all these belong to the class of dynamic branch predictions, we have to predict the outcome of a branch at run time. And for that we can take the help of hardware and we provide extra hardware components, one for predicting the outcome of the branch and one for storing the target address.

Effectively, in the dynamic branch prediction technique, we are going to exploit the outcomes of the branches, which happen at different times in the program execution and use this previous outcome history, and predict the next outcome for the branch and accordingly we just fetch the instruction either from the target address, or from the next instruction following the branch.

(Refer Slide Time: 27:42)



Dynamic Branch Prediction

- Consider a branch outcome sequence (T – *Taken* and N – *Not Taken*):

  T N T N T N T N T N T

- Predict the next outcome of a branch based on its present outcome

Taken
Not taken
Predict taken (1)
Taken
Predict not taken (0)
Not taken

  - Accuracy is very poor

- 2-bit prediction scheme

Taken
Not taken
Predict taken (11)
Taken
Predict taken (10)
Taken
Not taken
Not taken
Predict not taken (01)
Taken
Predict not taken (00)
Not taken

  - Prediction must miss twice before it is changed

- Branch history table implementation:
  - As a special cache or appending a pair of bits to each cache block

Madhu Mutyam (IIT Madras) 58

So, consider sequence of branch outcomes for a particular branch instruction. Here T indicates branch is taken, and N indicates branch is not taken. So, lets us assume that there is a branch instruction which happened in the past 11 times, and then the outcomes for each of the times is like a first time it is taken, the next it is not taken, next time taken and so on. Once you have this history and if we can remember this history and then use this history to predict the next outcome, if the same branch is going to happen next time. That is what dynamic branch predictors are going to do.

Now, how much history we have to remember? Whether we take the decision based on the immediate past decision, or the immediate past outcome, or are we going to consider the last few outcomes and use that information? So, if you are going to remember large history of outcomes for given specific branch instruction, then we are going to incur significant hardware overhead. And because this branch predictor logic is going to be placed in the processor, so it is going to occupy chip area and as a result so it is not good idea to increase the overall overhead of the chip.

So, as a result we have to come up with efficient branch predictors, when I say efficient, it has to provide with high accuracy, whether the branch is going to take place or not take place and at the same time it should consume significant area overhead. Now, consider a simple case, let us assume that we are going to remember only the last outcome of the branch, and based on that we are going to take a decision. So that is nothing but predict the next outcome of the branch based on its present outcome. In this particular scenario, if we use this logic.

For example, when we are here, previous the present outcome is not taken and this next time again the same branch is going to come, based on this outcome if we are going to say next time also the branch is not going to take place. But actually the branch is taken so effectively there is a misprediction, and after that again the branch is not going to place, again there is a misprediction and so on. So, effectively, this just based on the present outcome if you are going to predict the next outcome, we may not get high accuracy branch predictor. And this actually is called as a one bit branch predictor and the state diagram for this one bit branch predictor will be something like this.

So, it has two states, the Predict Not Taken which is represented with 0 and Predict Taken which is represented with 1. So, initially assume that we are in the 0 state and so because this saying predict not taken, so we assume that the branch is not going to take place. And then we will fetch the instruction accordingly, but after the branch outcome is realized if it is actually not taken, then we will be in the same state and without changing the state. But if the branch outcome says that the branch is taken, so the taken is represented with dotted arrow. So, there is a state change from predict not taken to predict taken the state 1.

And after that next time if the same branch happens, we will look at the state diagram and its state is 1. So, then we will assume that the branch is going to take place and we will fetch instructions from the predicted target address, and if the prediction is wrong, then we are again going back to this state and so on. Effectively we will oscillate between predict not taken and predict taken if you are going to consider 1 bit history, and if the branch outcome is something like this. So, effectively this 1 bit branch predictor is not providing high accuracy and as a result we cannot use this.

So, in order to come up with a reasonable accuracy branch predictor we can actually go for the 2 bit branch predictor, where we will consider four states and the state change happens only when we predict wrongly or we predict correctly, two times consecutively then only we change the state. So for example, here we can see our four states are 00, 01, 11 and 10. Assume that we are in 00 state and 00 indicates that predict not taken, so we will be assuming that the branch is not going to take place and we will fetch the instruction accordingly.

But if the branch is taken actually after we perform the branch computation, then we will go to predict not taken with the state 01 using this dotted arrow. And still in this 01 state also because it says predict is not taken, next time also if the same branch occurs then we are going to assume that the branch is not going to take place, and we proceed further. But again if here also, for example, the branch is taken, that means we actually mispredicted two times then we will go to state where, the value is 11 and it is treated as predict taken.

So, effectively from this state predict not taken if we have two times misprediction happen, then we will go to predict taken. And in this, we assume that the branch is going to take place and the next time, when the branch instruction comes then using this state we treat the branch is going to place, and we fetch the instruction from the predicted target address and proceed further. For example, if the next time the branch is not taken then we will move from this state to predict taken, but state number is 10 using this, the bold, the arrow not taken, and we will continue in that.

Next, again if the next time the same branch happens and if it is not taken, then only we will go to this predict not taken. Effectively we can see here from this state with a two mispredictions, we are going to change the decision. Similarly, from this when there are two mispredictions then we are going to the other state. So state change happens, state change means here prediction decision changes, when there are two mispredictions happen consecutively for our given specific branch instruction.

And we can actually come up with n bit branch predictor, where rather than considering two bit we can have n bits, so that we can get high accuracy. But experimental results show that, the two bit predictor gives a reasonable accuracy, and the extra hardware overhead we incur because of this n bit predictor, where n is greater than 2 is not good compared to the benefits we get in terms of accuracy. So, as a result we can, we can go with the two bit branch predictor in our processor designs.

And to implement this branch predictors, so we can either associate a specific cache, a small cache type of thing in the processor design, where this cache will store the branch history information, branch history target. So, it remembers the previous outcomes of the branches and next time when the same branch happens we index into the cache and then based on the outcome we can take the decision. It remembers the state of the outcome, or we can also implement this branch predictions using extra bits associated with the cache block, because the branch instruction is part of a cache block in the instruction cache, and that entire block will be associated with the state of this the state diagram. And using that state we can identify, whether we can predict whether the branch is going to take place or not and accordingly we can proceed further.

So, effectively the branch history table can be implemented either by using a special cache, or by appending a set of bits to the cache block which is holding the branch instruction. In the case of special cache, we are going to store this entire history information and whenever there is a new branch instruction comes we index into the cache and then we will get the predicted outcome and based on that we will proceed further.

This is going to tell you about the whether the branch is going to take place or not, but once a branch is predicted to be taken, we have to know the target address. For that we also associate an extra table in our processor that is called as a branch target address buffer. So, BTB branch target buffer, which actually stores the predicted target address. Previously if the when the same branch occurred, what was the target address it took? And that address will be stored in this BTB, and next time if the same branch is going to take place, then we index into the BTB and use that predicted address and fetch the instruction from the predicted address.

So, effectively in order to implement this dynamic branch prediction, we need hardware components associated with the prediction logic, as well as the branch target buffer which stores the target addresses. So, with that, so, I am concluding this module and the next module, we are going to discuss simple pipeline implementation of MIPS ISA.

Thank you