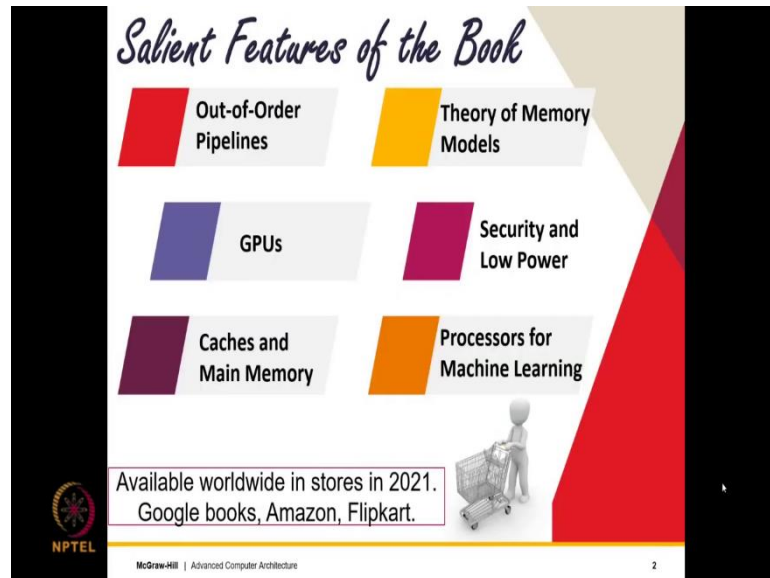


Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

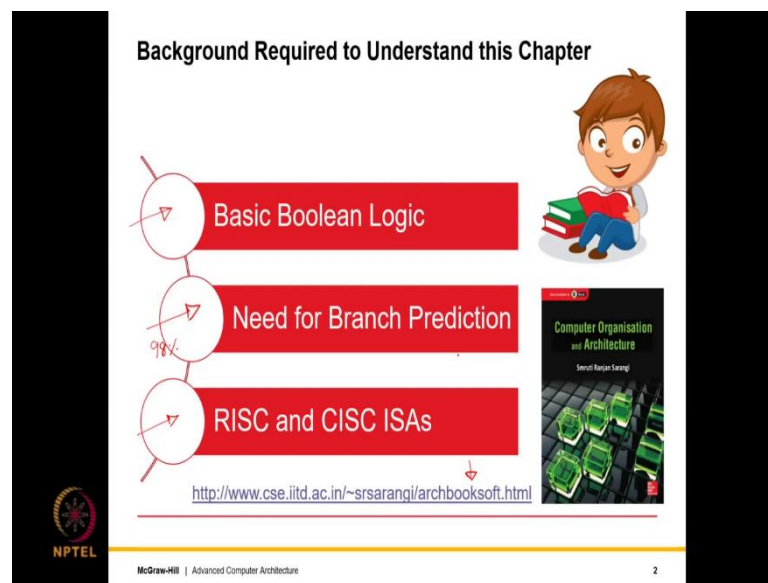
Lecture - 05
The Fetch and Decode Stages Part - I

(Refer Slide Time: 00:23)



Welcome to the chapter on the Fetch and Decode Stages. So, in this chapter, we will study all about the fetch logic in particular, the branch prediction logic and then, discuss some decode time optimizations, there are typically done mostly in CISC processors to substantially increase the amount of substantially increase the performance as well as make it easier to process the instructions in other stages of the pipeline.

(Refer Slide Time: 01:21)

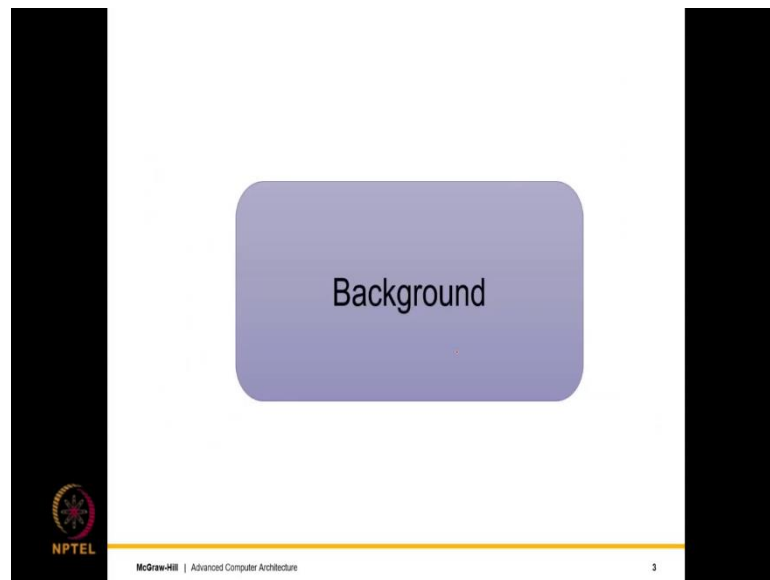


The basic background required to understand this chapter well we need three things, the 1st is the understanding of basic bits in Boolean logic, the 2nd is so, this is the 3rd chapter, the 2nd chapter was on out-of-order pipelines so, the motivation part at least.

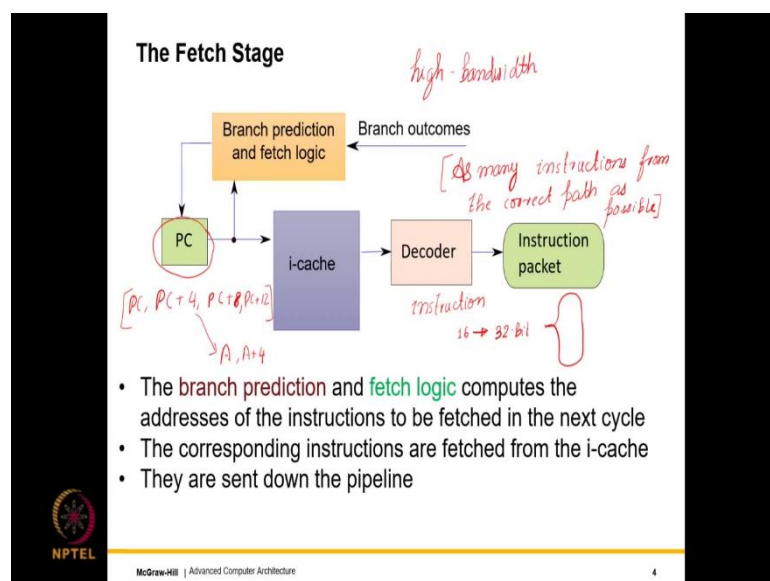
So, the need for branch prediction should be clear and why we need an accuracy which is more than 98% that should be clear and the viewers of this video should be reasonably aware of RISC and CISC ISAs, what are their tradeoffs, their advantages, the typical formats and so on.

So, for the required background, we were scan look at an earlier book Computer Organization and Architecture published by McGraw Hill in 2015. So, a link is given over here. So, there are YouTube videos as well for all of these topics. So, I would urge the viewers to kindly take a look at these topics before proceeding forward because the material in this lecture set are heavily depends on these three topics. So, any viewer has to be very thorough with these.

(Refer Slide Time: 02:49)



(Refer Slide Time: 02:52)



So, first a brief background. Let us look at the relevant part of the pipeline that discusses the fetch logic. So, the crux of the key state element that captures the fetch logic is the program counter, the PC, this points to the next instruction that needs to be fetched from the I-cache. So, in a RISC instruction set, the size of each instruction is fixed, it is typically either 4 bytes or 8 bytes. So, for the sake of simplicity, let us assume it is 4 bytes.

So, then in this case, the addresses would be like this PC, PC + 4, PC + 8 and so on. So, if we have a fetch width that is more than 1. for example, if you fetch let us say 2 or 3 or 4

instructions per cycle, then what we can actually do is that we can take multiple addresses, which is essentially PC , $PC + 4$, $PC + 8$ and so on and fetch them from the i-cache, the instructions can then be sent to the decoder which will decode the instructions, then the decoded instructions will be put in the relevant instruction packets.

So, what exactly is decoding? Well, an instruction can be considered as a compressed bundle of information. so, a lot of fields are kind of compressed within the instruction. so, all of these need to be expanded such that it is much easier to process the instruction in subsequent stages of the pipeline. So, what we do is that we take instruction in a compressed format. so, it is not compressed in the typical compression sense, but it is just that we use as few bits as possible to represent information. So, lot of this is extended.

For example, instructions can embed an immediate and the immediate is a constant in computer architecture in parlance. So, the immediate can let us say these 16 bits. So, a 16-bit immediate is expanded to a 32-bit immediate. So, all of this process of expansion is done within a decoder.

So, the expanded instruction is called the instruction packet which of course, contains all its fields and all other pieces of information that are required to process the instruction in the subsequent stages of the pipeline, this is the instruction packet.

So, of course, if you are reading four instructions simultaneously, let us say from these four addresses, then we decode all four of them in parallel so, we have four parallel decoders and we create four instruction packets that are sent down the pipeline. So, this is somewhat a simplistic idealized view of the fetch process.

Branches actually can cause problems and a lot of serious problems in the sense that they make the control flow deviate. So, the addresses instead of being as well behaved as PC , $PC + 4$, $PC + 8$ and so on can actually if this is a branch, then it can branch to a separate address and we need to fetch from there. so, this issue will be there and so, if you are fetching four instructions at the same time well.

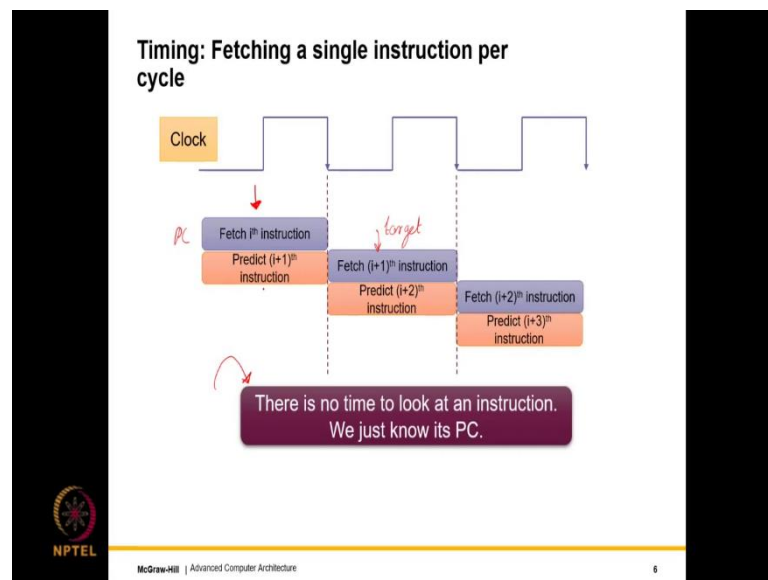
Any subset of them can be branches. so, it is the job of the branch prediction and fetch logic to essentially predict both the direction of the branches as well as the target and fetch instructions from there and kind of give us as many instructions from the correct path as possible. So, I said we can sustain a high fetch bandwidth.

So, the key summary from this slide is that we need as many instructions from the correct path as possible. So, recall that we had discussed what is the correct path and what is the wrong path in the previous slide set. So, the correct part is basically the correct branch path.

So, from the correct branch path, we need to this means predict branches correctly and fetch as many instructions as we can such that we can supply as many instructions as possible in parallel of course, to the rest of the pipeline because the fetch is like the head of the pipeline.

So, the fetch is slow the entire pipeline is slow. so, this fetch is a bottleneck and if let us say the fetch process is slow, the entire pipeline even if there is a high degree of high instruction level parallelism, it will not be able to leverage that mainly because we are not fetching as many instructions as we can execute in parallel. So, we need a high bandwidth fetch unit. So, the key over here we do not really care about latency, but what we really care about is high bandwidth.

(Refer Slide Time: 08:32)



So, let us look at simple fetch unit that fetches the single instruction per cycle. So, this will be our first example and then, we will gradually complicate this. So, what happens is that when you are fetching the i^{th} instruction, we have what we have? Well, we only have its program counter and on the basis of that, we give the program counter to the instruction cache. The instruction cache sees the program counter and fetches the i^{th} instruction.

So, the i th instruction mind you we have not seen it. So, we are only fetching its, we know its address, we are fetching its contents, but we have not seen it and we have clearly not decoded it. So, we are not aware at this specific point of time which means in this cycle, we are not aware whether it is a branch or not.

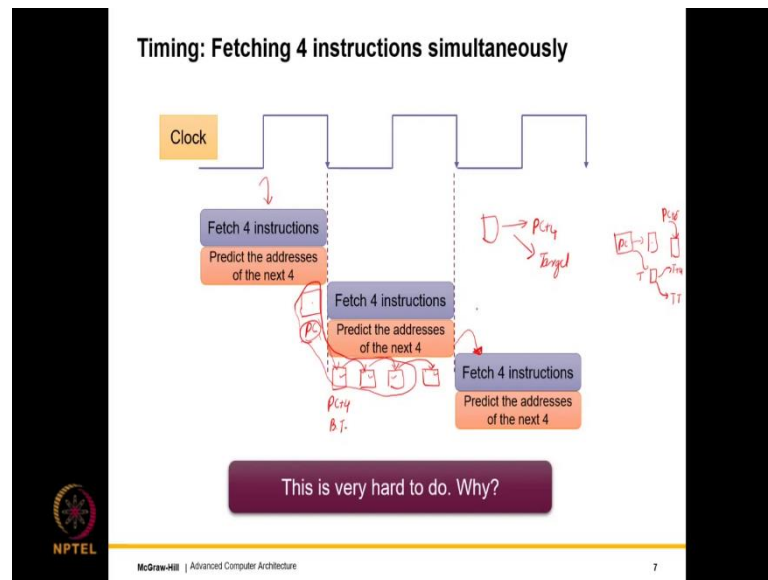
So, nevertheless, if we want to fetch the $i + 1$ th instruction in the next clock cycle. well, then we will have to predict many things, but we will definitely have to predict that after the i th instruction, what is $i + 1$ th instruction?

If it is a branch well, we need to predict its outcome and target and $i + 1$ th instruction will be the target otherwise, if it is not a branch, it is a simply $PC + 4$ that is the address. The same holds for the $i + 1$ th instruction also, we do not have enough time to see it. The moment we are fetching it, we need to calculate the address of the $i + 2$ th instruction.

So, we essentially need to predict what will it be? So, to predict $i + 2$ th instruction well, the $i + 1$ th instruction we will have to if it is a branch, we will have to predict its outcome and target and if it is not well in this case, it is again the PC of this instruction $+ 4$ so on and so forth we need to proceed.

So, the most important point that is kind of coming out from this slide in big bold letters is that there is absolutely no time to look at an instruction. So, we cannot look at an instruction means look at the instructions contents, there is no time to look at it, we simply know its program counter on the basis of that, we need to predict the next instruction and as a current instruction is a branch that will end up meaning that we need to do a branch prediction.

(Refer Slide Time: 11:16)



Let us now look at a more complicated version of the previous slide. So, in the previous slide, we are fetching 1 instruction per cycle, in this slide, we will fetch 4 instructions per cycle. So, let us consider the first cycle. So, assume that we are fetching 4 instructions and we know somehow, we will see how that these 4 instructions are on the correct path. So, let us say we know with high likelihood that they are on the correct path. Now, the question is that we need to predict the addresses of the next 4 instructions.

So, the operative part of this discussion is let us look at the last instruction in this bundle of 4 instructions. Let the program counter of the last instruction be PC. So, what we need to do is that in the current cycle, since we know the 4 instructions and their program counters, we need to predict the first instruction in the next bundle, this can either be $PC + 4$ or if this instruction is a branch, it can be its branch target. Then, we need to predict the next one, the next instruction and the next.

So, we essentially need to predict these 4 instructions on the basis of the program counter of the last instruction of the current bundle. There are numerous complexities in this, this is not easy. The reason it is not easy is because what we have essentially at with us is that we have one program counter, this can either be pointing to a regular instruction or a branch.

Regardless of whatever it points to the instruction that succeeds it on the correct path can again be a regular instruction or a branch, then that instruction can again be regular or a

branch so on and so forth. So, essentially, we are looking at so this so, let us see if I were to consider these 4 instructions, we can think of them as kind of a multi-way branch in the sense there are; there can be multiple branch instructions within them and so, this as far as we are concerned so which 4 instructions are these?

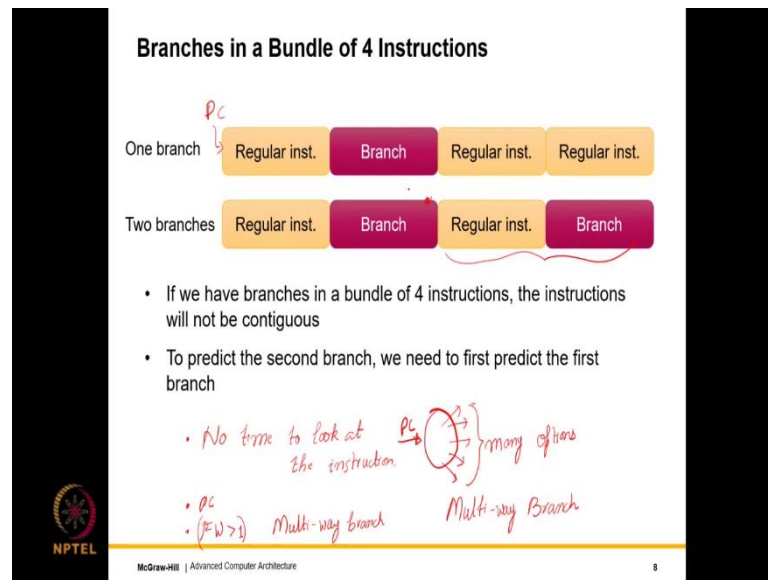
They are essentially these 4. So, these 4 instructions can have several branch statements within them, and thus numerous targets are possible, numerous correct paths are possible and instead of making one prediction in a cycle, we actually need to make four predictions. On a similar line, we using a similar logic, once we have 4 correct instructions in this cycle, we again need to predict the next 4 instructions on the correct path for the next cycle.

So, this clearly entails multiple having multiple branch predictors making multiple predictions per cycle and still ensuring that we are able to ensure a fetch bandwidth of 4 and those 4 instructions are on the correct path. So, on the correct path of course, that we will get to know later, but as far as we are concerned from a prediction point of view with a very high likelihood, they are on the correct path.

So, as you can see fetching one instruction at a time or fetching four at a time is conceptually similar, but the only difference is that if I fetching one at a time, then it is easy because you have two possible outcomes it is either $PC + 4$ or the target. But if you are let us say predicting two instructions at a time, then there are numerous options actually.

If this is PC, this can be $PC + 4$ and the final instruction after that can be $PC + 8$ or we can go to the branch target over here and again one option can be it can be either the target + 4 or it can this can be a branch and we can go to the branch target on that. So, what we see is that we have numerous options. so, this is why it is called a multi-way branch when we are considering predicting branches for a bundle of more than one instruction and this is clearly complicated.

(Refer Slide Time: 15:54)



This complexity is shown in this slide where if you see that if we have a bundle of 4 instructions with one branch, this is relatively easy to predict. But if we have two branches, it becomes much harder because the program counters of these instructions is dependent on the prediction of this. So, this kind of becomes a sequential process.

And if we can have even more branches. so, that will create more difficulty in terms of doing the prediction. So, basically as far as we are concerned in a bundle, if I were to look at the instruction that will just succeed this bundle, we do not have two options so, we in fact, we have many options.

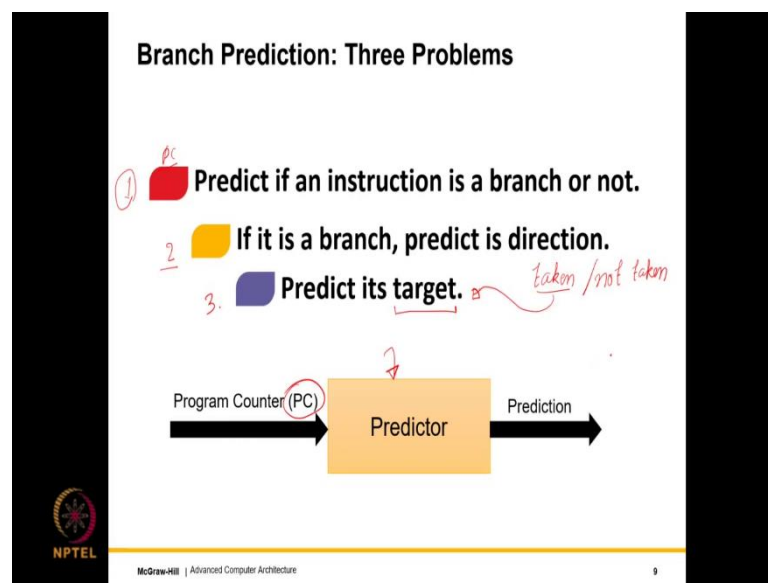
So, this even though we start at one point which is maybe the starting PC of this bundle. so, even though this is the starting PC, but the if I were to look at the instruction that will just succeed the bundle, we actually have a multiplicity of options, many options. Hence such a bundle with multiple branches is called a multi-way branch and we will be using this term multi-way branch reasonably frequently in our discussion in chapter 5, this is an important concept.

So, what is the summary up till now? Well, the summary up till now is if I were to kind of summarize the main points, the first is we do not have time, no time to, there is no time to look at the instruction, the only thing we can look at is its PC that is the only thing we know.

Furthermore, if you want to predict, if you want to fetch bandwidth, a fetch width. so, let us call it a fetch width if you want it to be > 1 , you are essentially looking at a multi-way branch where for the bundle as a whole, there can be multiple targets because the bundle can have multiple branches within it all of these branches have to be predicted correctly and we will have to find the final correct path which is hard. so, which single branch prediction as it is hard, but doing multiple branch prediction is even harder.

So, this is where the role of a smart compiler comes in. It should ideally generate code such that branches are placed relatively far apart such that if I were to fetch 4 contiguous instructions in the current on the correct path, there will be desirably at the most one branch. So, this will reduce the load on the branch predictor and allow us to design simpler systems as opposed to this multi-way branch scenario.

(Refer Slide Time: 19:32)

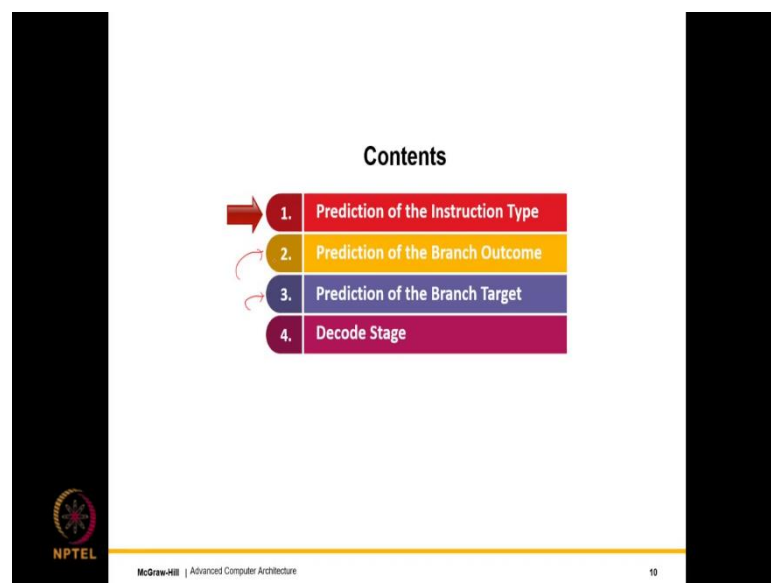


Nevertheless, what we will concern ourselves in this chapter is given a branch instruction, how do I predict several things? which are predict if an instruction is a branch or not is the first thing that we need to do. Because given an instruction as I have said we have not seen it, we have only seen its program counter address, we need to predict in the first place whether it is a branch or not. See if it is a regular instruction, we are fine, the next instruction will be $PC + 4$.

If it is a branch which is 2, we need to predict its direction, it is either taken or not taken, then we need to predict its target if it is a taken branch that is right if it is not taken no issue, but if it is a taken branch, we need to predict the target of the branch.

So, as we see we have three different separate problems over here; predict if an instruction is a branch or not problem 1. Problem 2, if it is a branch, predict if it is taken or not taken. Problem 3, if it is taken, predict its target. So, we create a black box predictor which given a program counter, this returns its prediction which can very well be the target as a branch.

(Refer Slide Time: 20:48)



So, now, let us come to the contents of this chapter. So, we will predict first the instruction type what kind of an instruction is it, is it a branch, a non-branch, if it is a branch, what kind of branch, then we will predict the branch outcome, then we will predict the branch target and finally, we will discuss some decode time optimizations.

(Refer Slide Time: 21:15)

Is an instruction a branch or not?

Insights

- Given a **PC**, the status of the instruction (branch or not) does not change.
- Can we use this information?

Approach

- The last time that we had seen a branch **remember** its PC
- Next time we see a **PC**, **check** if we have seen it before
- Also **remember** the type of the branch:
 - Unconditional branch
 - Conditional branch → depends on the result of a previous compare instruction
 - Function call
 - Return

Handwritten notes: **remember!!** (on a sticky note), **PC** (underlined), **Non-branch** (circled), **[log₂ 5] bits (3 bits)**

NPTEL

McGraw-Hill | Advanced Computer Architecture

11

So, now, let us come back to our discussion. Given a program counter, the status of the instruction whether it is a branch or not, this does not change. so, this is kind of hard wide in the program that because while we are running the program, the program is not changing. So, given a program counter, the status of an instruction never changes. Can we use this information? Well, yes.

So, what is the approach? The approach is that the last time that we had seen a branch; we remember its PC. So, whenever we see a branch, we remember its program counter, the next time we see a program counters, we essentially check to see if you have seen it before.

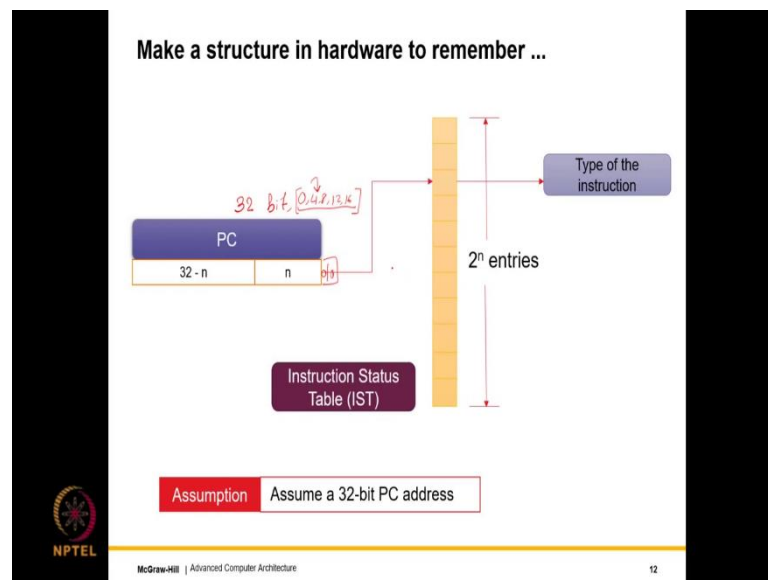
If we have seen it before. so, we essentially have some sort of a memory, some sort of a remembrance and we try to see whether it is an unconditional branch, whether it is a conditional branch which means it depends on the result of a previous compare instruction.

Whether it is a function call, whether it is a return statement and of course, the 4th, the 5th category which I am not mentioning which is rather obvious is whether it is a non-branch. So, essentially, we will have these five categories and for these five categories, we essentially need sealing of bits which is 3 bits to represent what is the type of the instruction.

So, what is the summary again? The summary is we do not have the time to actually look at the contents of the instruction; we simply have access to its program counter address

that is it nothing else. On the basis of that, we need to predict whether it is a branch or not. If it is a branch, what kind? If we have to do that, we will have to remember the last time, the last time we had seen this PC, what was the type of the instruction and use this memory for prediction.

(Refer Slide Time: 23:44)



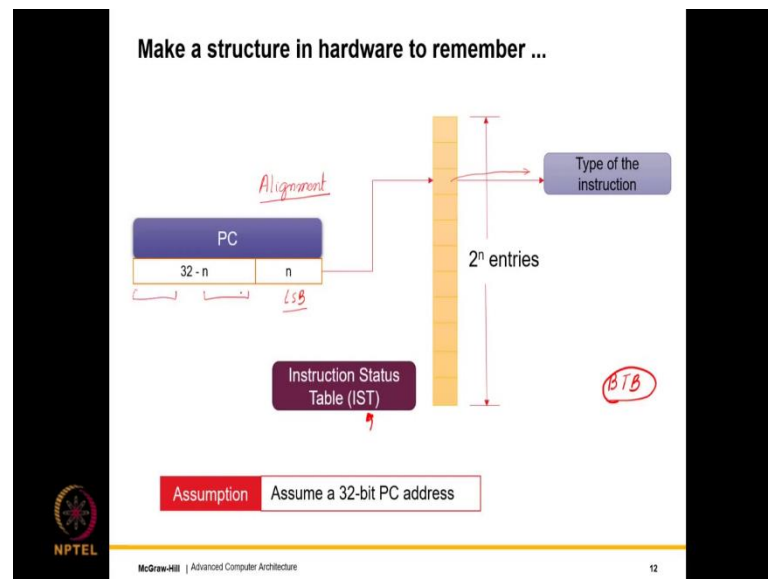
How do we make a hardware structure to remember? Well, it is very easy. So, kindly take a look at this structure with a lot of attention. We will be using this structure over and over and over again for all kinds of predictions throughout the book.

So, let us consider a program counter address. So, we are we have made our life easy, we are considering a 32-bit program counter. In most modern processors, it is actually 64 bits, but we just want to make our life easy, you can very well replace 32 with 64 it will not cause a different. Also, one more thing in modern processors is that program counters are typically aligned two 4 byte boundaries in the sense that they cannot take an arbitrary value, but they typically take value 0, 4, 8, 12, 16 and so on.

So, this is the case, but we will not use this assumption well, why in the interest of simplicity, we will not use this assumption even though you should know it does happen. So, we will make a somewhat simple assumption just for the sake of easier explanation which will assume that the program counter addresses can be rather arbitrary in the sense that they will not be aligned to 4-byte boundaries.

Even though in most actual systems, practical systems they often are aligned and in our case, if you were to consider that, then anything aligned to a 4-byte boundary in the last 2 bits will be 0 and 0. But let us as I said for the sake of simplicity, avoid that, but when whenever a branch predictor is practically being implemented, this should always be kept in mind that alignment of the PC's to either a 4; 4 byte or 8-byte boundary is important.

(Refer Slide Time: 25:41)



Nevertheless, let us proceed with somewhat simplistic assumption. So, what we do is we take a look at the n least significant bits; I will discuss why it is the least significant bits in one second. So, then if I take these n bits well, then there are 2^n combinations. so, we have a small table with 2^n entries. so, this n bit will uniquely map to one of the locations within this table.

This table as we discussed will contain the status, the type of the instruction, we will simply re return them. Let us call this table the instructions status table the IST which of course, is the temporary placeholder because later we will add more information to this and we will actually call it the BTB, but till we introduce the term BTB, kindly hang on to the term IST and once you are introduce the term BTB, you can discard this term.

So, what is the crux? Well, the crux is we take the n least significant bits, use it to address a table which has 2^n entries. So, now, let us see what are the potential problems with this design? What is good with this design is that we can quickly map the program counter to

one entry within a table, find out the last time we had encountered this program counter what was the status that the type of the instruction and read it.

(Refer Slide Time: 27:28)


Basic Method of Operation

When we see a branch instruction (*Execute/Decode*)


- Access its corresponding entry in the table
- Record the branch type

When we see an instruction:

- Check the corresponding entry of the table
- Read the type of the instruction



Why do we choose the n least significant bits?



McGraw-Hill | Advanced Computer Architecture 13

So, this is the basic method of operation where essentially we see a branch instruction, we see its PC, we access the entry in the table when we are seeing it, we record the branch type and when we see an instruction at fetch time.

So, this will of course happen at execute time when we know that an instruction is a branch or maybe at decode time also does not matter, but when we see an instruction and we do not know it is a branch or not, we check the corresponding entry of the table and we read the type. So, the question now arises.

So, you will have many of these food for thought burgers throughout the discussion. So, why do we choose the n least significant bits and why not let us say the n most significant bits or may be n bits in the middle? So, this has something to do with what is called aliasing which we will discuss in the next few slides.

(Refer Slide Time: 28:32)

Instruction Status Table (IST)

If we choose the least significant 10 bits of the PC address, we will have 1024 entries in the IST

Why 10 bits?

For a 32-bit PC, we cannot have a 2^{32} entry table.

- Too big
- Too slow
- Too much of power
- Too much of area

We need to manage with a smaller table

4 billion

7-8

NPTEL

McGraw-Hill | Advanced Computer Architecture

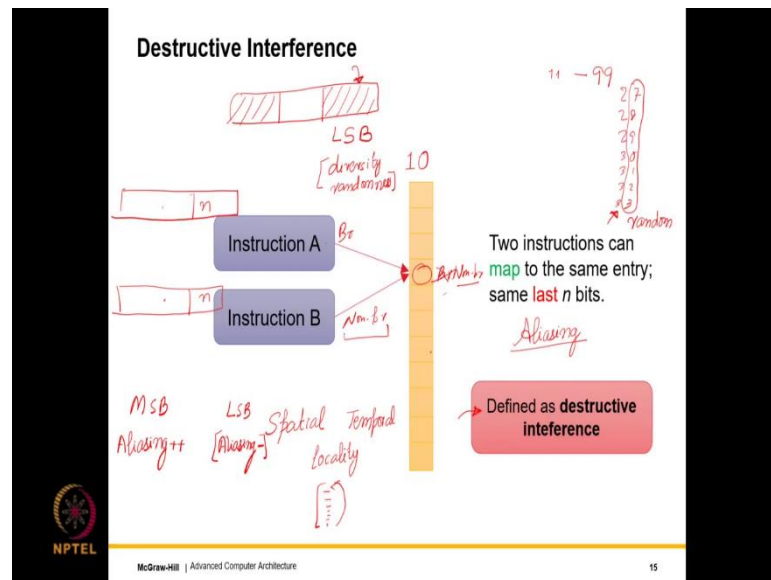
14

So, if we choose the least significant 10 bits of the PC address, we will have 1024 entries in IST. So, the first question is that why not use the entire 32 bits, why just the 10 least significant bits right where $n = 10$, why just these bits, why not all of it? Well, if we have a 2^{32} entry table, it will be very large.

So, how much is 2^{32} exactly? It is 4 billion roughly ok, it is roughly 4 billion entries and 4 billion is too higher number, too larger number and we simply do not have the space within the chip to store that many entries. So, it is too big, too slow, too much of power, too much of area.

So, from all of these resource considerations, we need to manage with a significantly smaller table. Hence, we choose a table with 1024 entries as the IST which is something that we can afford.

(Refer Slide Time: 29:43)



So, the primary problem that will happen with this table which also takes us back to why we choose the least significant bits and why not any other bits is basically that it is possible for two instructions. if this is a PC of 1, it is possible for two instructions to actually share the n least significant bits.

So, this is very well possible, one of the bits let us say over here would differ, but the n least significant bits will be common. If these bits are common, they will actually map to the same entry in this table. So, they map to the same entry, what will happen is it is possible that one of them is a branch, one of them is a non-branch.

So, then what will happen is we will have destructive interference in the sense that if let us say the non-branch instruction had come the previous time and now, we fetch a branch, we will still predict it to be a non-branch because this is what this entry will contain, and we will make a mistake.

So, then when we see the status of this instruction will again converted back to branch, but next time, a non-branch may come, it will read the status as branch, it will again make a mistake, it will again change the status for non-branch.

If let us say the non-branch instruction keeps getting fetch, there is no problem, but the moment we fetch the branch instruction that maps to the same location, we will again make a mistake. So, this is known as aliasing, or destructive interference. So, with destructive

interference what happens? So, this is a problem primarily because we have a small finite size table, we are not using all 32 bits instead we are only using 10 bits. This is exactly why we have destructive interference because of this.

Now, let us answer that question why are we using the least significant bits and not the most significant bits? So, the idea behind the least significant bits is that you consider a set of numbers.

So you consider let us say you want to count from 1 to 100 and let us say consider some point in the middle 27, 28, 29, 30, 31, 32 which bit changes more, the one at the units place or one at the tens place? Clearly, the bit at the unit's place is something that changes more which is some and it is something that I would say is more random if of course, I am accessing.

So, is assume that I am accessing numbers between 11 and 99 at random, then what will happen just extend the logic from this counting example, you will find that the digit at the units' place changes far more frequently and there is much more diversity here as compared to the digit at the tens place which means that if I were to just consider a least significant bits.

I would see much more diversity, I would see much more randomness as compared to the most significant bits. As compared to bits over here, if I consider the least significant bits, I will see a much more uniform spread. So, I will see much more randomness.

Now, let us come to the next important properties of two properties of a program, spatial and temporal locality where spatial locality is that if I have accessed one address, I will access some other address nearby very soon and temporal locality is I tend to access the same data or same instruction over and over again.

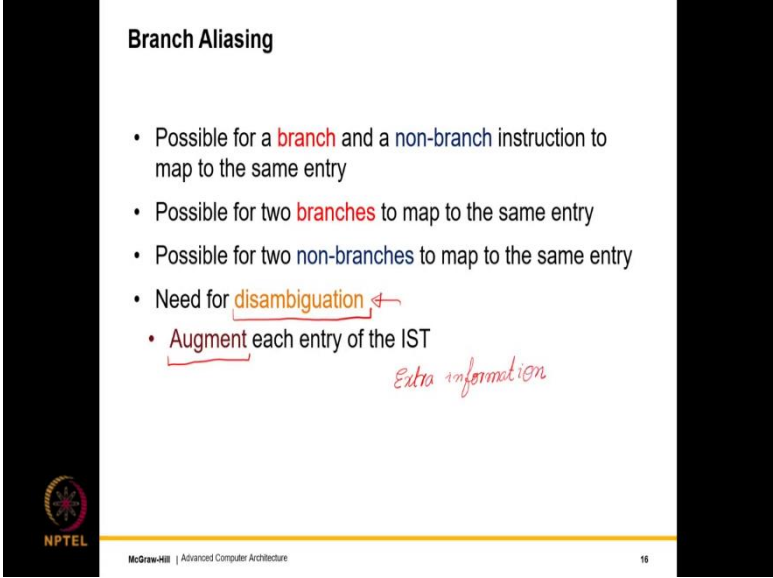
So, if I were to combine both of these and I would look at programs that let us say use loops or something and since I am looking at instructions. well, the typical control flow would be I go down the loop, down, down, down, down, again come up, down, down, down, again come up and a program spends most of its time roughly 90% of the time in loops. So, that is the reason there is a lot of locality.

So, within a small region, you will see most of the diversity in the least significant bits and the most significant bits for a small region will roughly remain the same.

See if I use the most significant bits to address this table, it is possible that most of the instructions that will actually map to the same entry and we will have a massive amount of destructive interference, massive amount of aliasing. So, if I use the MSB's, the aliasing that I will actually see which is entries instructions mapping to the same entry will be very hard.

As compare to this, if I use least significant bits, combining this with the fact that is programs have spatial temporal locality and most of the randomness is seen towards the lower bits, we will see a significantly reduced aliasing. so, let us say it is reduced. So, this will this is good for us, it will increase the accuracy of this process.

(Refer Slide Time: 35:51)



Branch Aliasing

- Possible for a **branch** and a **non-branch** instruction to map to the same entry
- Possible for two **branches** to map to the same entry
- Possible for two **non-branches** to map to the same entry
- Need for **disambiguation** ↗
- Augment each entry of the IST

Extra information

NPTEL

McGraw-Hill | Advanced Computer Architecture

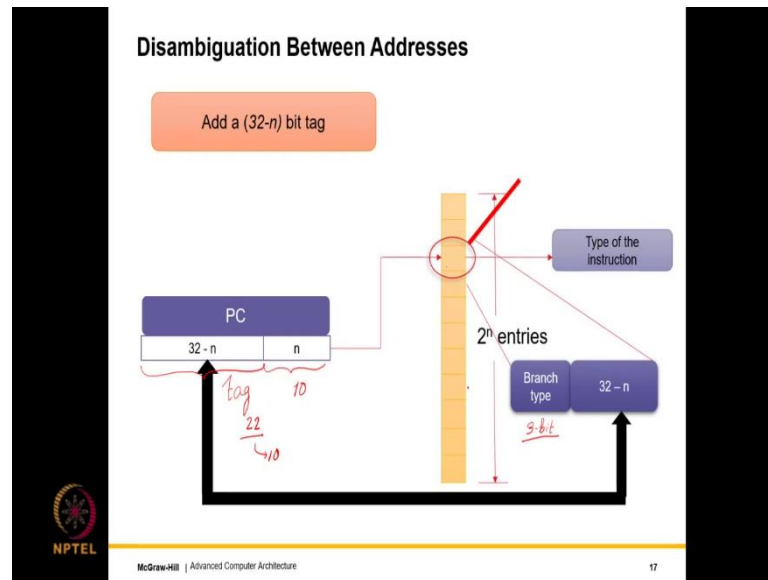
15

So, as we have discussed, branch aliasing is a major issue it is possible that two branches map to the same entry, two non-branches map to the same entry or a branch and a non-branch map to the same entry. So, given the fact that we are only remembering what is it that we had seen last time and based on that, we are predicting what it is most likely this time, aliasing will cause an issue.

So, there is thus a need for disambiguation. So, disambiguation is one of the solutions to fight aliasing. So, ambiguate is to complicate, disambiguate complicate confuse,

disambiguate is to kind of uncomplicated and unconfused if there is such a word. So, if I were to disambiguate what do, I augment each entry of IST with additional information, extra information.

(Refer Slide Time: 36:56)



So, what we need to do is that the primary confusion that actually arises over here is because of the fact that we are using the lower n bits and the upper 32 - n bits they are not being used. So, it is possible that two addresses have the lower n bits in common, but the upper 32 - n bits are not common. So, we follow the same procedure which is that we use the lower n bits to actually address this table of 2^n entries.

However, in each entry along with storing the branch type which is as small as I said a 3-bit field, what we do is that we additionally store these 32 - n bits. so, these 32 - n bits are stored here. So, then whenever we are trying to predict for a given program counter, we access the entry, check if the remaining bits match or not so, these remaining bits are known as a tag.

So, we check if the 32 - n bits match or not. So, if they match, we are sure that this entry is actually not the entry of some other branch; it is the entry for this branch. So, we are sure that these the entire PC matches. So, n bits of course, match because we come here and the 32 - n bits match because we store them over here. So, once all of them match, we have of course, taken destructive interference out of the picture and then, we simply read out the branch type.

So, instead of a thin table this kind of becomes a fatter table because per entry we are storing more bits, but the advantage is that we are avoiding aliasing to a large extent. So, of course, here a lot of innovations are possible.

If let us say this is n is 10 bits, then the tag will be 22 bits, but we can maybe instead of saving 22, we can maybe let us just save just 10 bits. So, what will then happen is that the possibility of aliasing will still be there. however, it will get substantially reduced and what will we gain? Well, we will gain storage space over here at this table will be smaller.

(Refer Slide Time: 39:43)

The slide is titled "Disambiguation" and contains the following text and handwritten notes:

- We can keep the status of only **branches** in the IST
- However, for every **instruction**, we need to check the IST
- If there is no entry, then we need to predict that it is not a branch
- Can be **wrong** (with a handwritten checkmark)
- What to do ☹️

Handwritten notes include: "tag \rightarrow PC" and "temporal locality (Pops)".

Why does an IST work?

- Mainly because most pieces of code exhibit **temporal locality** (with a handwritten checkmark)
- In a given window of time **instructions** tend to repeat themselves

The slide footer includes the NPTEL logo, "McGraw-Hill | Advanced Computer Architecture", and the number "18".

So, then what does this give us? Well, it gives us a larger table at the cost of better accuracy. It also tells us that whether there is an entry or not.

So, if there is no entry so, what can happen is that if there is no entry in this table, then so, in this case, we can actually check, in the previous case where we are not storing the tag, we were not sure which PC it exactly corresponded to, but in this case, since we are storing a tag, we know exactly which PC it kind of corresponds to.

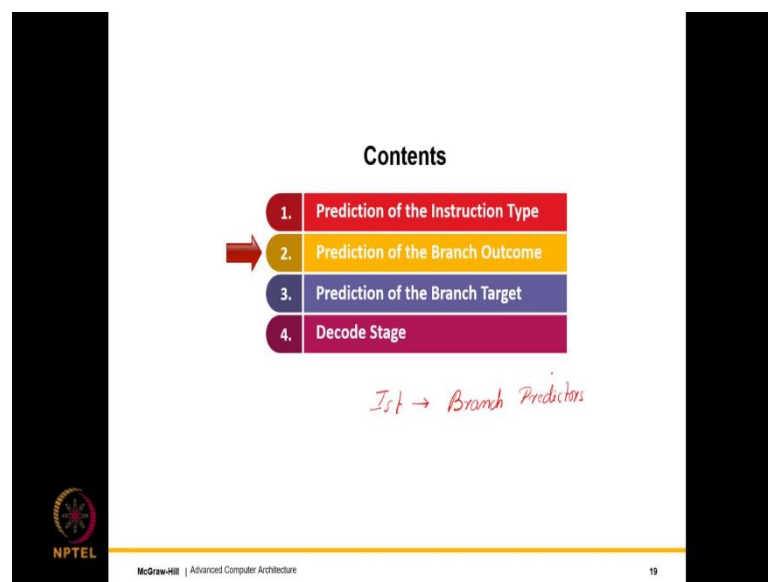
So, if let us say we do not find an entry for a PC, what do we do? Well, we look at what is more probable so, since only 20% of the instructions are branches and 80% are not, we predict that it is not a branch this of course, can be wrong nothing much can be done.

Why does an IST work? Well, the reason it works is mainly because most pieces of code exhibit temporal locality that is that they keep on executing the same instruction over and

over again and when do we see that? When we see them in loops and as I said a program spends most of its time roughly 90% of its time in loops, different kinds of loops for loops, while loops all kinds of loops, but the same instructions keep getting executed over and over again.

And because of temporal locality, we capture a small region of the program and if that is stored well in IST, we using least significant bits, we can these predict with reasonably good accuracy for we can predict the branch type with the reasonably good accuracy. So this is just the second statement is just repeating the first part that we have a lot of loops in the program and this is why our IST using the LSB bits works.

(Refer Slide Time: 41:59)



Now, let us move to the next problem which is predicting the branch outcome. So, predicting the branch outcome will of course, this is built on similar principles, we will essentially extend the IST. so, we will extend the IST to create similar predictors essentially, similar principles built on similar principles.

And these branch predictors will incorporate many more effects along with temporal locality, they will incorporate many more artifacts aspects of the way that programs are typically written. So, this will be covered in the next lecture of this lecture series.