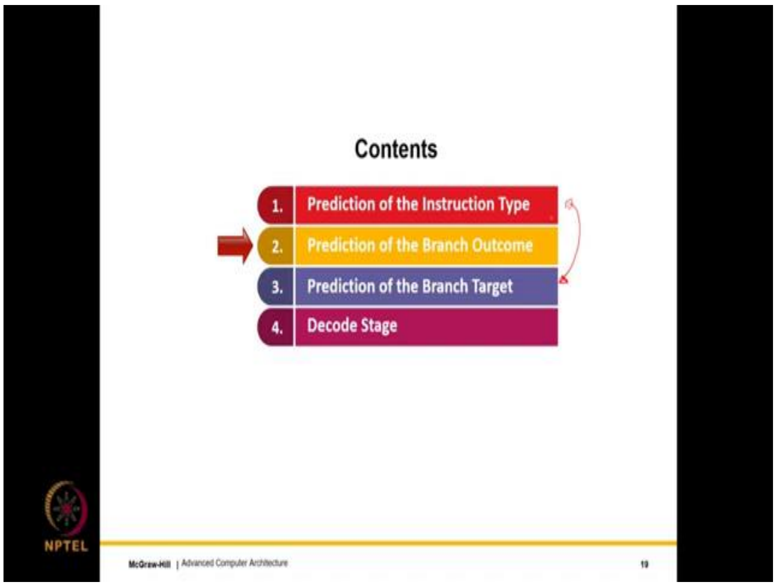
Advanced Computer Architecture

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

Lecture - 06
The Fetch and Decode Stages Part - II

(Refer Slide Time: 00:53)



Contents

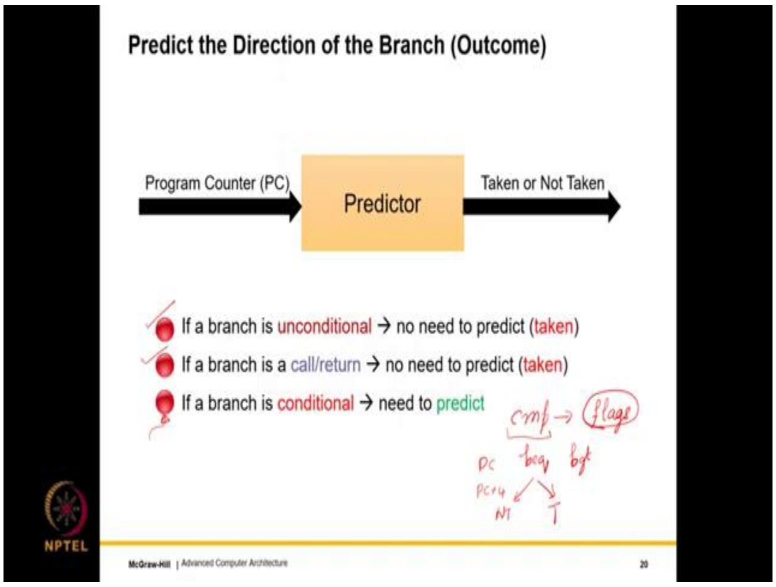
1. Prediction of the Instruction Type
2. Prediction of the Branch Outcome
3. Prediction of the Branch Target
4. Decode Stage

McGraw-Hill | Advanced Computer Architecture 19

Let us now look at the most important aspect of branch prediction, where arguably the maximum amount of research work has gone in, which is predicting the outcome of the branch. So, this is the most difficult problem, the rest of the problems are relatively easier to deal with, easier to manage and handle.

We already have created an ISD for predicting the instruction type; what we are further promised is that, we will somehow merge the ISD with the structure for predicting the branch target, but let us go sequentially step by step. So, once we know that an instruction is a branch, next we need to figure out whether it is taken or not taken.

(Refer Slide Time: 01:37)



Predict the Direction of the Branch (Outcome)

Program Counter (PC)
Predictor
Taken or Not Taken

- If a branch is unconditional → no need to predict (taken)
- If a branch is a call/return → no need to predict (taken)
- If a branch is conditional → need to predict

cmp → flags
PC
beq
bgt
PC+4
NT
T

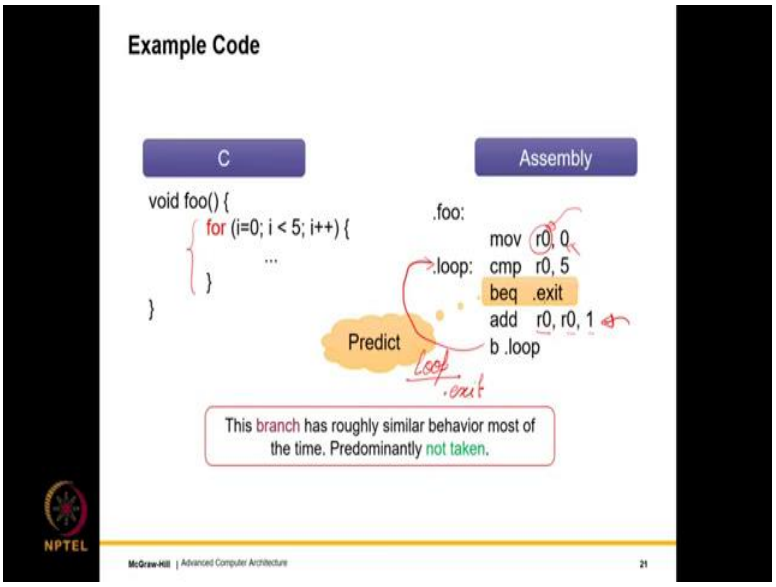
McGraw-Hill | Advanced Computer Architecture 20

So, here also we assume a black box, the black box a program counter comes in and what comes out is whether it is taken or not taken. Say the branch is unconditional, then it is always taken, so there is nothing to predict. If a branch is a call or return, it is also taken all the time, so here also there is nothing to predict.

The fun comes in conditional branches, like branch if equal, branch if not equal which are essentially dependent on the outcome of the last compare instruction in the program. So, the compare instruction essentially what it does; is that it sets a set of flags; these flags record the outcome of the compare instruction, whether it lead to an equality, inequality, greater than, less than etcetera and these flags are subsequently used by later branch instructions, like branch if equal or a branch if greater than and so on.

So, they see what the comparison resulted in and based on that they are either they go in one direction or the other. So, not taken as of course, the sequential direction PC, PC + 4 and so on that is not taken or they can go in another direction which is taken, where we jumped to the branch target.

(Refer Slide Time: 03:11)



Example Code

C

```
void foo() {
    for (i=0; i < 5; i++) {
        ...
    }
}
```

Assembly

```
.foo:
        mov  r0, 0
.loop:  cmp  r0, 5
        beq  .exit
        add  r0, r0, 1
        b .loop
```

Predict
Loop
.exit

This branch has roughly similar behavior most of the time. Predominantly not taken.

So, let us consider a piece of example code, we will show the C code and we will show the simple RISC assembly code. So, simple RISC assembly the simple RISC language assembly language is appendix A of the book and it has also been discussed in previous slide sets and it is definitely there in the videos of the previous book.

So, this is a two part book, in the first part of the book the simple RISC assembly language has been described in great detail. So, let us look at the for loop, which goes from 0 to 5. So, what we do is that, we treat r 0 as the loop variable, we initialize it to 0.

So, then we compare it with 5, if it is equal to 5, then of course we exit the loop, this is our termination condition; otherwise in every iteration of the loop, we increment the loop variable by 1 which we are doing over here and again we loop back. So, this is the loop, alright.

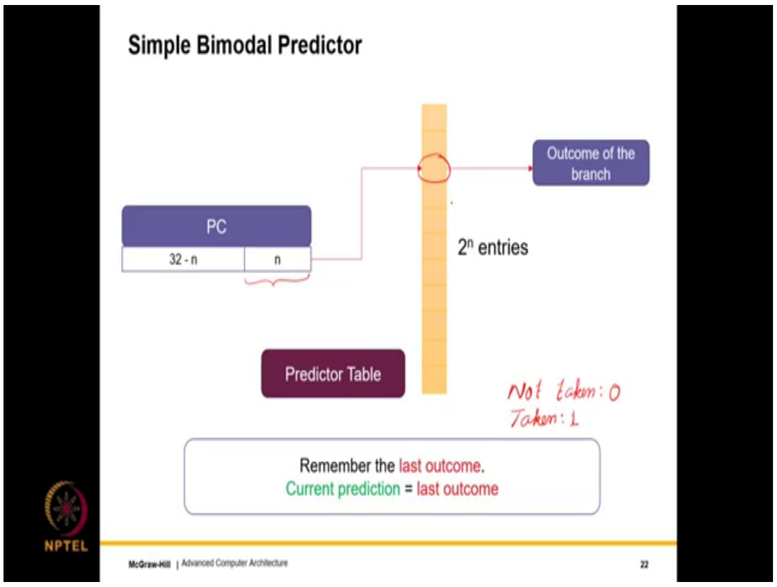
So, the jump is essentially a backward jump. So, this is the loop and as you can see we are implementing the for loop almost identically in assembly, where we initialize, we check the exit condition and we increment. So, normally whenever we implement a for loop, we follow these three steps of initializing the loop variable, checking for termination and incrementing.

So, we see that we have one conditional branch, which is beq and this needs to be predicted. So, the branch if we look at it, it has roughly similar behavior most of the time.

So, most of the time it is not taken, because the condition that makes it taken does not happen. So, that is only when r 0 = 5 and that will happen very rarely towards the end of the, at the end at the end of the for loop.

So, note that we are assuming a dot exit label which is the end of the program, so this is not shown. So, now, given that you have seen this pattern that, this branch by and large is not taken; it is only taken in the last iteration, can we use this pattern? Well yes.

(Refer Slide Time: 05:49)



Simple Bimodal Predictor

Remember the last outcome.
Current prediction = last outcome

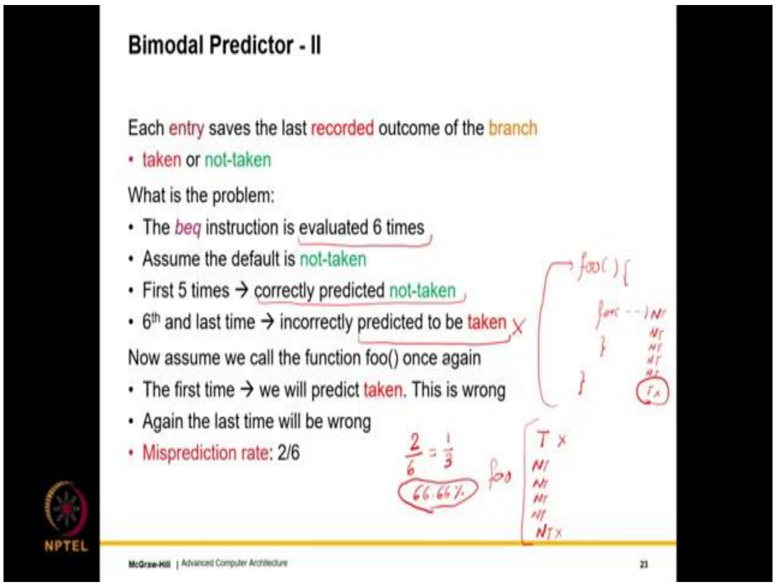
McGraw-Hill | Advanced Computer Architecture 22

So, what we do is, we create a predictor that is extremely similar to the ISD that we had designed in the previous lecture, where we predicted the type of the instruction. So, here what we do is, we take the n least significant bits over here and the n least significant bits that we have we use it to index address a 2^n entry table.

This was we were doing the same in ISD; this is exactly the same logic. So, what we do is, I came to the ISD, we remember the last outcome; so we can say that not taken is represented with 0, taken is represented with bit 1.

So, we simply store 1 bit over here, where we remember the last outcome and the outcome of the branch. So, whatever was the last outcome, we say that look the current predicted outcome is also the same. So, the last outcome was not taken; we say that look this time also we predict that the branch is not taken and likewise is the case for taken. This is a very very simple predictor, so let us evaluate and let us see if it works or not.

(Refer Slide Time: 07:17)



Bimodal Predictor - II

Each entry saves the last recorded outcome of the branch

- taken or not-taken

What is the problem:

- The *beq* instruction is evaluated 6 times
- Assume the default is not-taken
- First 5 times → correctly predicted not-taken
- 6th and last time → incorrectly predicted to be taken ✗

Now assume we call the function foo() once again

- The first time → we will predict taken. This is wrong
- Again the last time will be wrong
- Misprediction rate: 2/6

McGraw-Hill | Advanced Computer Architecture 23

So, what we see is that, the beq instruction is evaluated 6 times. So, let us go back. So, here if you see this iteration, it actually runs 5 times and the 6th time we actually find out that we have reached the end of the for loop. So, this statement is evaluated 6 times; 5 times for a genuine iteration and the 6th time to exit the for loop.

Out of these 6 times, the first 5 times we correctly predict not taken, if we assume that we start from the not taken state; but the 6th and the last time we also predict not taken which is wrong, because in this case the branch is taken, this is what we should have predicted it to be.

So, now let us assume that the function foo contained this loop. So, if it contained this loop what will happen, we keep on calling the function foo is that. So, let us say we call it the first time. So, the predictions will be not taken 5 times. So, let us write that; the last time the branch is taken, so we will make a mistake.

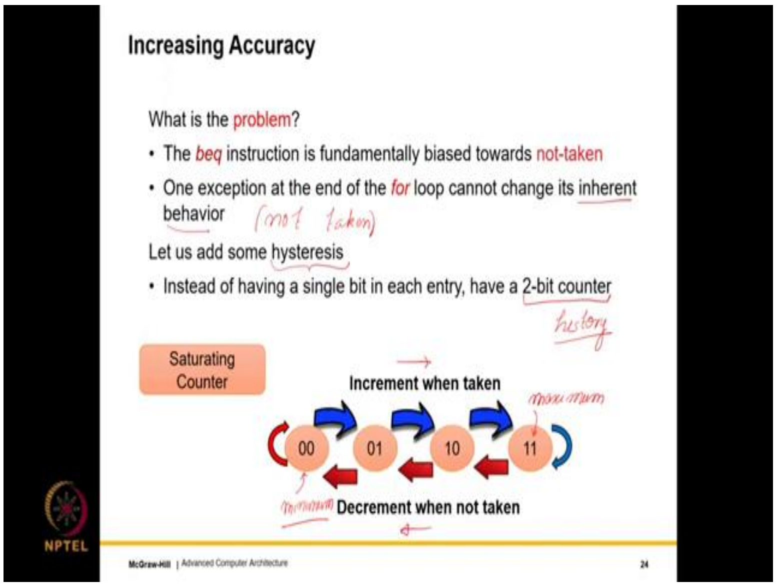
The next time we come to function foo; we will still predict that the branch is taken. So, next time that we come let me write it over here, we will predict that the branch is taken; why? Because the last time we encountered the branch, the branch was taken. So, we will make the same prediction; this will be a mistake, we will realize it was not taken.

So, next time we will predict it is not taken. We will keep on doing it for the next 4 times, again we will predict not taken which is the last iteration of the for loop, this is again a mistake. So, this is the function foo; if we call function foo once again, we will repeat the same pattern. So, what we will actually see is that, our misprediction rate is as high as $\frac{2}{6}$ or $\frac{1}{3}$.

So, recall our discussion in chapter 2 where we discussed that, the accuracy of a branch predictor should be very high; 96, 97, 98 % at least. But here we see that the accuracy is very low, in fact the accuracy over here is only 66.66 %, which is way lower than what we actually need.

But that is fine, this is the first predictor that we are seeing and this predictor as we can see has a built in flaw and the built in flaw is that it always predicts whatever was the outcome the last time, which in this case appears to be a bad idea. So, we make two mistakes; one while entering the loop and one while exiting the loop.

(Refer Slide Time: 10:19)



Increasing Accuracy

What is the *problem*?

- The *beq* instruction is fundamentally biased towards not-taken
- One exception at the end of the *for* loop cannot change its inherent behavior (not taken)

Let us add some hysteresis

- Instead of having a single bit in each entry, have a 2-bit counter history

Saturating Counter

Increment when taken
maximum
00
01
10
11
minimum
Decrement when not taken

McGraw-Hill | Advanced Computer Architecture 24

So, let us see where is the problem? The problem is that the beq branch if equal instruction is fundamentally biased towards the not taken direction. There is one exception at the end of the for loop; so but the point is that the single exception at the end of the for loop cannot change, should not change its inherent behavior.

So, what is the inherent behavior of the branch? The inherent behavior is that the branch is predominantly not taken, it is just in the last iteration of the for loop in which the branch is taken. However, this one exception should not change the way we perceive the branch.

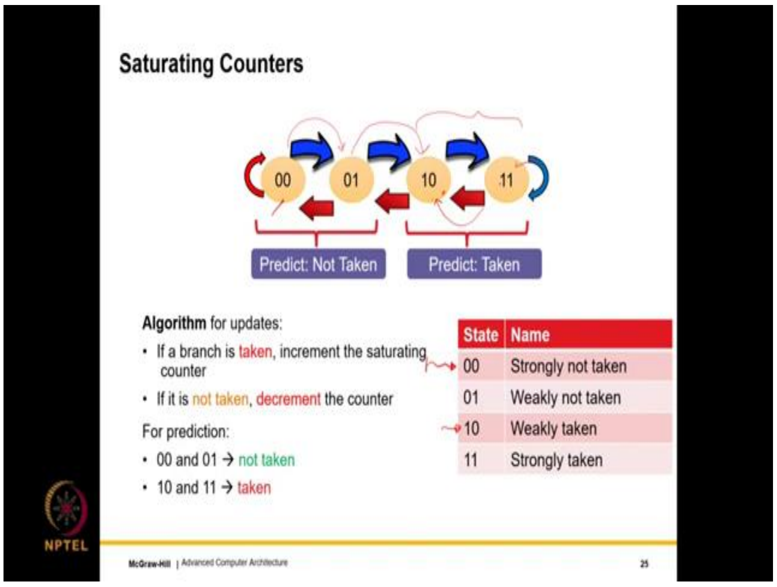
So, we need to add some degree of prior history or some degree of hysteresis to the branch. So, instead of having a single bit in each entry, let us have a 2 bit counter, which to a certain extent can have this history or some degree of hysteresis, such that the inherent behavior of the branch will get recorded, it will not change, it will get recorded.

So, let us add what is called a saturating counter. So, saturating counter is a 2 bit counter, where if a branch is taken we keep on incrementing it. The reason it is called saturating is because when it reaches it is maximum value which is 11; at that point of time it does not increase anymore, it is not incremented anymore and the value remains over there.

Similarly, when the branch is not taken, we decrement the value of the counter; when it reaches 00, it is not decremented further, it remains at 00. So, its value is not decremented further. So, this is why the counter is set to saturate at both the maximum value and the minimum value.

So, this is where the counter is set to saturate at both the maximum as well as the minimum values and whenever as I said when we increment, we go in this direction and when we decrement, we go in this direction.

(Refer Slide Time: 13:07)



Saturating Counters

00
01
10
11
Predict: Not Taken
Predict: Taken

Algorithm for updates:

- If a branch is taken, increment the saturating counter
- If it is not taken, decrement the counter

For prediction:

- 00 and 01 → not taken
- 10 and 11 → taken

State	Name
00	Strongly not taken
01	Weakly not taken
10	Weakly taken
11	Strongly taken

McGraw-Hill | Advanced Computer Architecture 25

So, how and when and why are saturating counters useful? Well when the state is 00 and 01, this is when we predict not taken. And if you can see whenever a branch is not taken, we keep decrementing the counters; that is why when the state reaches 00 and 01, which is close to. So, which you will reach when you have been decrementing the counter, so we predict not taken.

And similarly, for the two upper states 10 and 11 we predict taken. So, what is the algorithm? Again where is the branch is taken, we increment it. So, ultimately the inherent behavior of a branch is such that it is taken all the time, it will saturate at this stage and we will predict taken.

Just in case you make a single mistake once in a while, we will still move to this state. So, it allows a single anomaly. So, in this case we will still continue to predict taken; the same is the case with not taken as well, where if a branch is predominantly not taken, most of the time the state will remain at 00.

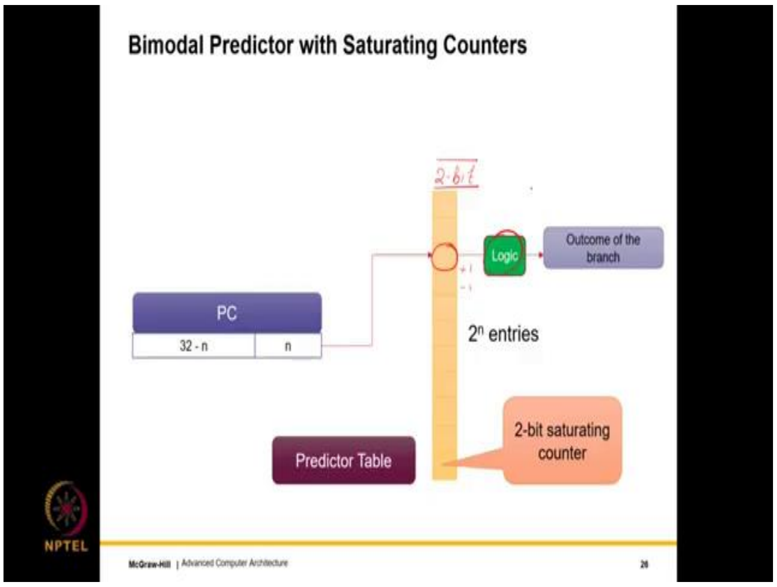
Let us assume that there is one odd anomaly, such as the one at the end of the for loop. So, in this case, the state will be 01, however we will still predict not taken; the main reason being that we need to make two such mistakes, I mean two such taken branches that two consecutively are required to pretty much change the behavior of the branch.

So, given that we have seen this pattern, let us give the names to the states. So, let 00 be strongly not taken, which is this state; 01 let it be weakly not taken, which basically means that, if let us say the branch is taken once, it will move to stage 10 and then we will start predicting taken from that point onwards.

Let 10 we called weakly taken, which for us which is named in a similar manner to weakly not taken. And finally, let the state 11 be strongly taken, which has the same connotation as strongly not taken. And once a state is strongly taken, if let us say it is not taken; just once it will still come to this stage, which is fine, we will still predict taken.

So, saturating counter is very good, it works very well when a branch is biased; in the sense when a branch predominantly takes one direction with one odd anomaly sometimes.

(Refer Slide Time: 16:02)

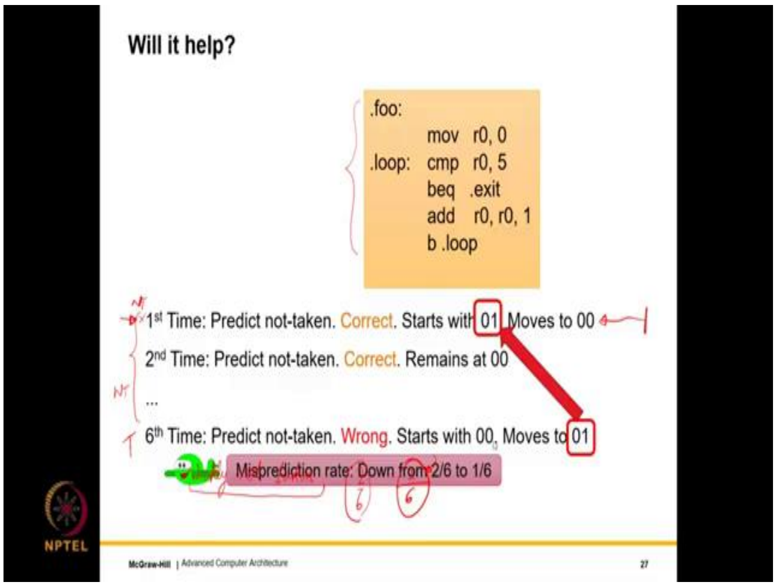


So, let us now add create a branch predictor with saturating counters. So, we shall take a regular bimodal predictor, everything remains the same; we use the bottom n bits to index it 2^n entry array. So, in this case instead of having a single bit in each entry, it has a 2 bit saturating counter. So, instead of having a single bit or 1 bit in each entry, it has a 2 bit saturating counter over here.

So, then if let us say we want to predict, we read the value of the saturating counter and we pass it through a logic block. What this does is, that it basically looks at the MSB of the saturating counter; if it is 0 we predict not taken, if it is 1 we predict taken. So, this is the predicted table as you can see over here which broadly remains the same; when we find the outcome of a branch, we again access this entry.

And if it is taken, of course we increment; if it is not taken, we decrement. But the logic is the same for the saturating counter; in the sense if the state is already 11, we do not increment it further, it remains there. So, the saturating counter is very useful, let us see why.

(Refer Slide Time: 17:37)



Will it help?

```
.foo:
        mov  r0, 0
.loop:  cmp  r0, 5
        beq  .exit
        add  r0, r0, 1
        b .loop
```

1st Time: Predict not-taken. Correct. Starts with 01. Moves to 00

2nd Time: Predict not-taken. Correct. Remains at 00

...

6th Time: Predict not-taken. Wrong. Starts with 00, Moves to 01

Misprediction rate: Down from 2/6 to 1/6

McGraw-Hill | Advanced Computer Architecture 27

So, let us take a look at the same piece of code again. So, in this case, the first 5 times we predict correctly which is not taken, the 6th time of course, we make a mistake. So, we predict taken. But the state after this is actually weakly not taken.

And so, we will so next time that we enter the first branch, we will still predict not taken; primarily because the state is that weakly not taken, which means that from the saturating counter logic even if we make one mistake, the inherent behavior does not change.

So, in this case the inherent behavior still remains not taken and that is why we do not make the mistake over here; that we were making with the regular bimodal predictor. So, this mistake does not happen, that is why the mistake rate here is $\frac{1}{6}$, which if we compare it with $\frac{2}{6}$, it is roughly half, it is 50 % better.

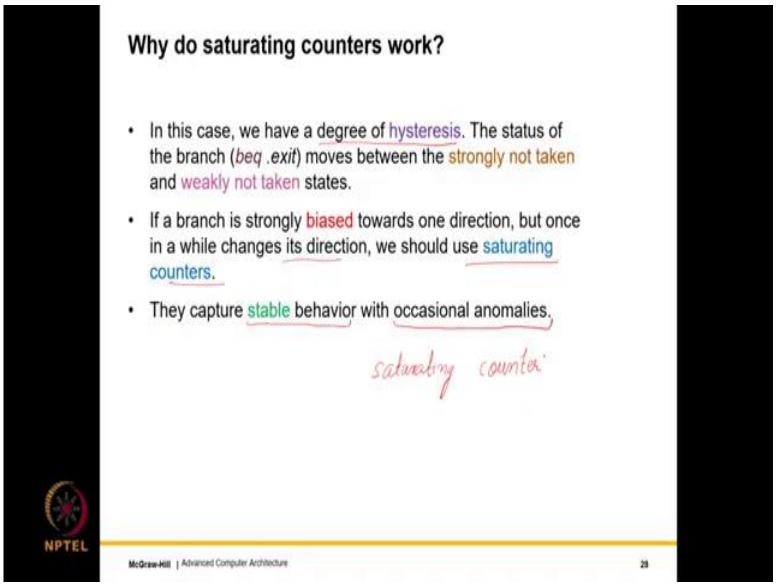
So, as you can see, we have been able to bring down the mistake rate substantially and now it is only $\frac{1}{6}$th, which is roughly 16 %. What was the key trick the key innovation that we used? Well, the key innovation that we use this that, the mistake that we were making over here.

Primarily because we were this the last exit iteration was taken and we continue to predict the same with a bimodal predictor. In this case, we do not do that, mainly because we created an additional state called weakly not taken. And this records the steady state behavior or the long term behavior or inherent behavior of the branch.

So, as you can see, we remain at 00, in this case because the status taken, we move to 01; the next time we start with 01, we move to 00. So, even if you were to execute these 6, these 5 iterations with 6 branches over and over and over again; we will still see that our error will saturate and stabilize that $\frac{1}{6}$.

So, the misprediction rate so let me erase the ink on the slide. So, the misprediction rate has gone down from $\frac{2}{6}$ to $\frac{1}{6}$. So, which in a certain sense represents a substantial improvement.

(Refer Slide Time: 20:39)



Why do saturating counters work?

- In this case, we have a degree of hysteresis. The status of the branch (*beq .exit*) moves between the strongly not taken and weakly not taken states.
- If a branch is strongly biased towards one direction, but once in a while changes its direction, we should use saturating counters.
- They capture stable behavior with occasional anomalies.

saturating counter

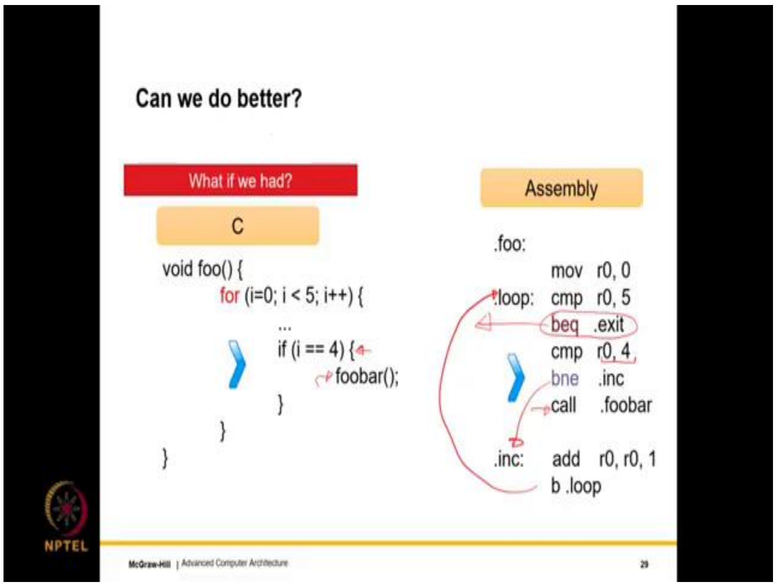
McGraw-Hill | Advanced Computer Architecture 28

Now, why do saturating counters work? Well, as we have discussed they introduce a degree of hysteresis; they introduce a degree of, in this sense, they incorporate the inherent behaviors of the branches.

So, in this case the weak you exit instruction moves between the strongly not taken and weakly not taken states. So, when does this work? Well, when a branch is strongly biased towards one direction; but once in a while changes its direction, that is when we use saturating counters.

They capture stable behavior with occasional anomalies. So, of course, a bi model predictor captures stable behavior; whereas if I were to include occasional anomalies, I would have a saturating counter based predictor.

(Refer Slide Time: 21:42)



Can we do better?

What if we had?

C

```
void foo() {
        for (i=0; i < 5; i++) {
                ...
                if (i == 4) {
                        foobar();
                }
        }
}
```

Assembly

```
.foo:
        mov   r0, 0
.loop:  cmp   r0, 5
        beq   .exit
        cmp   r0, 4
        bne   .inc
        call  .foobar

.inc:   add   r0, r0, 1
        b .loop
```

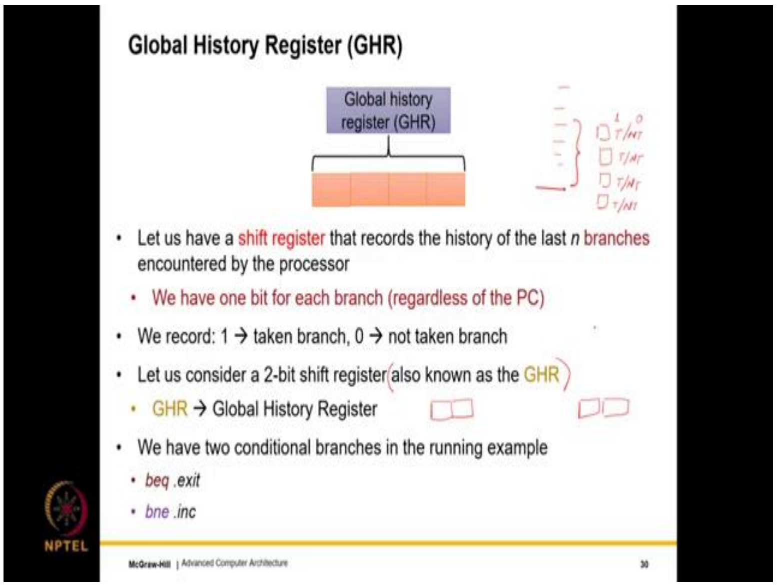
McGraw-Hill | Advanced Computer Architecture 29

Can we do better? Well, let us consider a slightly more complicated piece of code and let us look at, if we can do better or not. So, let us take a look at this piece of code which is 5 iterations; since the last iteration which is if i = 4, then we hear we call some other function which is foo bar. So, this function is called. Let us look at the assembly code over here.

So, in this case what we do is, where the first few lines are the same; we move 0 to r 0, they compare r 0 with 5. If the compression results in equality we exit, we go out of the loop; otherwise we execute this comparison if i = 4, they compare r 0 with 4. If the comparison is not successful, which will happen most of the time, we jump over here; we increment r 0 by 1 and we loop back. The interesting case happens in the last iteration when we compare r 0 with 4.

And the comparison results in equality, which means that we call the function foo bar and then after calling the function, again we return and we increment the loop index, which is the r 0 and go back to the loop. So, in this case, we still confine our attention to this branch and we try to reduce the error to 0 %. How do we do that? Well, we have a solution.

(Refer Slide Time: 23:31)



Global History Register (GHR)

- Let us have a shift register that records the history of the last *n* branches encountered by the processor
  - We have one bit for each branch (regardless of the PC)
- We record: 1 → taken branch, 0 → not taken branch
- Let us consider a 2-bit shift register (also known as the GHR)
  - GHR → Global History Register
- We have two conditional branches in the running example
  - *beq .exit*
  - *bne .inc*

McGraw-Hill | Advanced Computer Architecture 30

So, let us introduce a new structure called a global history register or a GHR. So, let this be a shift register that, records the history of the last n branches encountered by the processor.

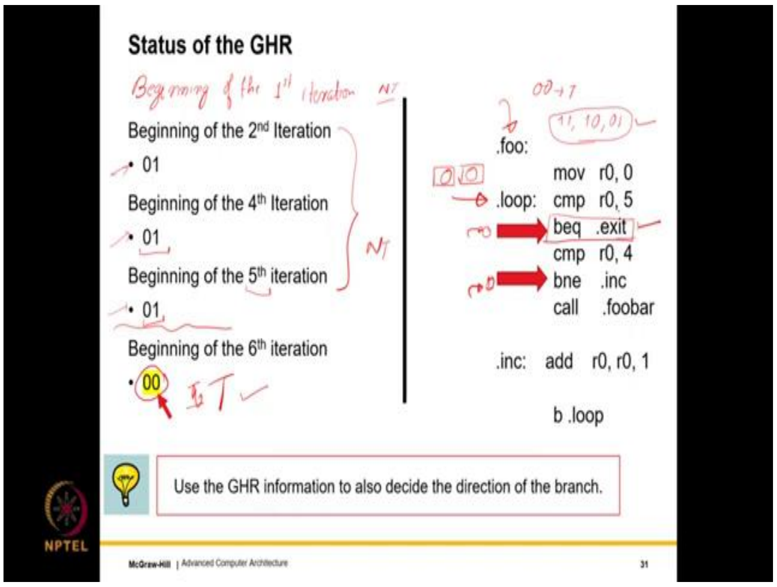
So, in a dynamic instruction sequence, if this is the current point; I look back at the last n branches regardless of the program counter. And for the last n branches, I store their outcome and the outcome can either be taken or not taken, taken or not taken.

So, if I have a single bit for each and I represent taken with 1 and I represent not taken with 0; then at any point I will have a n bit vector, where it will record the history of the last n branches in the dynamic instruction sequence.

So, if I consider a 2 bit shift register and call it the global history register, then. So, this, so shift register. So, shift register is also known as the GHR. So, the GHR in this case contains 2 bits in our example, which means it records the history of the last two branches.

So, recall that we have two conditional branches in the running example. So, of course, we can modify the GHR logic to only record the behavior of conditional branches. So, that will give us more accuracy as we shall see. So, let us assume we do that. So, we have two conditional branches in our running example and at every point of time, we have maintain the status of the last two branch instructions regardless of their program counter. So, let us call this the GHR.

(Refer Slide Time: 25:41)



Status of the GHR

Beginning of the 1st iteration NT

Beginning of the 2nd Iteration

- 01

Beginning of the 4th Iteration

- 01

Beginning of the 5th iteration

- 01

Beginning of the 6th iteration

- 00

NT

```
        00→T
        11, 10, 01
.foo:
        mov   r0, 0
.loop:  cmp   r0, 5
        beq   .exit
        cmp   r0, 4
        bne   .inc
        call  .foobar

.inc:   add   r0, r0, 1

        b .loop
```

Use the GHR information to also decide the direction of the branch.

McGraw-Hill | Advanced Computer Architecture 31

So, what is the status of the GHR? So, at the beginning of the 2nd iteration, the last branch over here would have been taken and the one before that would have been not taken.

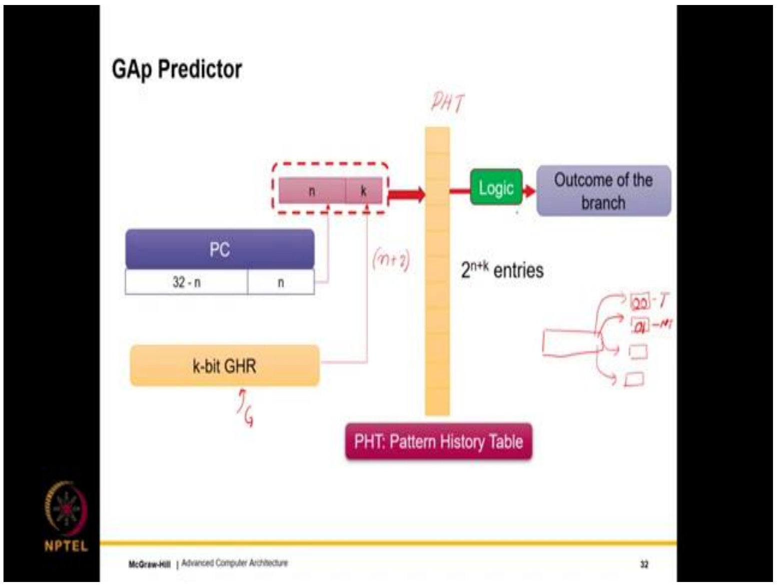
So, the status that we were record is 0 and 101. Even at the beginning of the 4th iteration it will be 01; even at the beginning of the 5th and last iteration also it will be 01; however, at the beginning of the 6th iteration the status will be different. Why? Because let us see what happens in the 5th iteration, this branch is not taken; there is an equality over here, hence this branch is also not taken.

As a result, when I reached the beginning of the 6th iteration which is this point and I take a look at my GHR for the status of the last two branches, which would essentially be this branch and this branch in the previous iteration, the status would be 00. So, in this case, we can think of the GHR as somehow encoding the context of a branch in terms of the outcomes of the n previous branches in the dynamic instruction sequence.

So, in this case as we see, at any point we can of course inspect the GHR. So, the beginning of the iterations 2nd to 5th, the GHRs are all 01, 01, 01, 01; but at the beginning of the 6th iteration, if we take a look at the outcomes of the last two branches, which would be basically the outcomes of the branches in the 5th iteration, their status is 0 1 0. So, the GHR would contain 00.

So, since it contains 00, recall that the GHR values at other points was different; it was 01, 01, so in this case, it is 00. Can we use this information to predict the beq dot exit branch better? Well, yes we can and we will see how. So, you can clearly see a difference, you can see that in the other iterations the state of the GHR was 01. But in the crucial 6th iteration when we exit the loop, the state of the GHR is 00 and this information is different, it is unique and it can be used.

(Refer Slide Time: 28:11)



So, what do we do? Well, we make a gap predictor. So, we will mention why it is called a gap predictor in some time; but essentially what we do, what our idea is, is that we take n bits from the PC and we take and we consider a k bit GHR. So, in this case of course, in our example k = 2. So, what we do is, we concatenate n and 2.

So, in this case we create a n + 2 bit field. This is used to address a table of 2^{n+k} entries. So, now, this table does not have 2^{n} entries, but it has 2^{n+k} entries.

So in this case, what are we doing, we are essentially combining the information of the GHR and the PC; which means that for a single PC, of course not considering aliasing for, different GHR values will have different saturating counters.

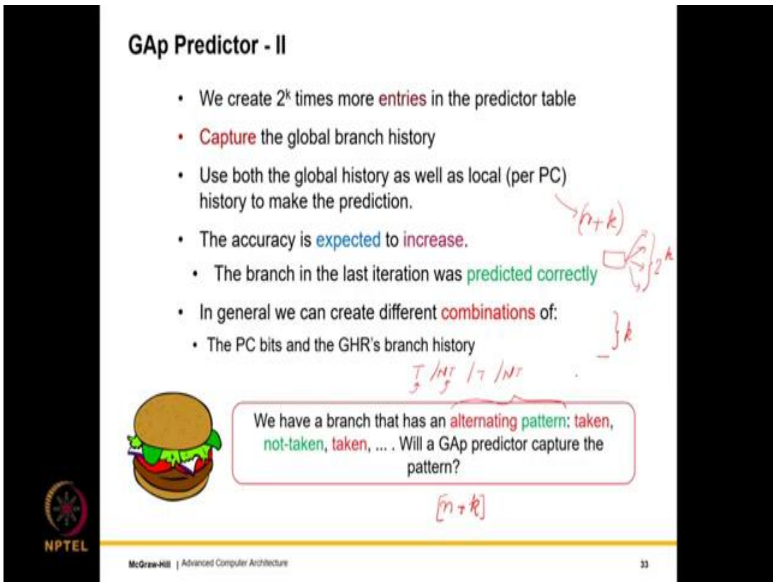
If it is a 2 bit GHR value, we will actually have four saturating counters instead of one associated with the PC and for a given GHR value, of course a saturating counter will record the behavior of the branch, the steady state behavior of the branch; not the steady state, but in a saturating counter sense, it will record the behavior of the branch.

So, what are the two values that we see most common; 0 1 and 0 0. So, most of the time it will be 01, with 01 we can store not taken. So, in any case in the process of training, this is what the saturating counters will get trained to. But if you considered 00, what we see over here is that, in no other iteration is the value at this point for beq = 00.

And furthermore, when the GHR is 0 0, the branch is always taken. Hence what we can record over here, which automatically the saturating counters will get trained 2 is actually 00 and at this point the branch is taken.

So, what we see from this is, there is a nice way to differentiate the cases, the previous cases when the branch is not taken versus the case in which the branch is taken and this can be used for better prediction as we see right here.

(Refer Slide Time: 30:47)



GAp Predictor - II

- We create 2^k times more entries in the predictor table
- Capture the global branch history
- Use both the global history as well as local (per PC) history to make the prediction.
- The accuracy is expected to increase.
  - The branch in the last iteration was predicted correctly
- In general we can create different combinations of:
  - The PC bits and the GHR's branch history

We have a branch that has an alternating pattern: taken, not-taken, taken, ... Will a GAp predictor capture the pattern?

$(n+k)$ $\}2^k$ $\}k$ T/NT /T /NT $[n+k]$

McGraw-Hill | Advanced Computer Architecture 33

So, what did we do? Well, we created 2^k times more entries in the predictor table; we had to do that, because we consider a GHR, we capture the global branch history.

We use both the global history as well as the local per PC history to make the prediction. So, we actually use both, both the global history as well as the local history. So, how do we do that? Well, because we are combining both pieces of information; we are taking n bits from the program counter and we are taking k bits from the global history.

So, this is a total of 2^{n+k} entries. So, which basically means that per PC, we have 2^{k}, per PC we have 2^{k} saturating counters. See each one of them is determining the behavior of the PC with respect to a given context.

So, the accuracy in this case is expected to increase and as we can see in this case, the branch in the last iteration will be predicted correctly. So, in general we can create different combinations, so the PC bits and the GHRs branch history. So, it is important for me to look at this issue once again.

So, as you can see in the last iteration, the branch over here is different, the GHR is different; it is going to be, hence and in no other iteration, do we see such a pattern, hence the branch will be predicted correctly. There is one point that we need to mention which we have not written over here, but let me complete this; this is the beginning of the first iteration.

So, in the beginning of the first iteration, essentially we are entering this function. So, wherever we are entering this function from, essentially the context of the last two branches will come from there. So, if it is 00, of course we will predict taken which is a mistake; but the odds are that it will not be 00, I mean if we assume it varies uniformly. Even if it does not, it will not be 00 all the time, I mean statistically well we need to take our chances.

So, if it is any other value 01, 10, and 11 say in these cases of course, the context is different; if it is 01 we will predict not taken; if it is 10 and 11, then the saturating counter will automatically train itself to predict not taken. The only difficult scenario can arise, if let us say when we enter the function foo, the existing context is 0 0; then automatically we are already trained ourselves to predict taken over here. So, will predict taken and we will make a mistake here.

But as I said there are four states, so the odds are more likely that we would have 10, 11, 10, or 01 as the starting state of the GHR. So, in this case, we can the saturating counter will automatically be trained to predict not taken. So, but we will revisit this issue later, because the point is that the. So, if the first iteration the state is one of these, then it is clear that we are not making any mistake for beq dot exit. So, the error rate is 0 %.

So, effectively we have 100 % accurate branch predictor; but in the beginning of the first iteration, if the context the state of the GHR is 0 0, then of course there is a problem. Because for the same PC and GHR combination, sometimes is predict taken which is this case; but when we enter the loop, we want to predict not taken.

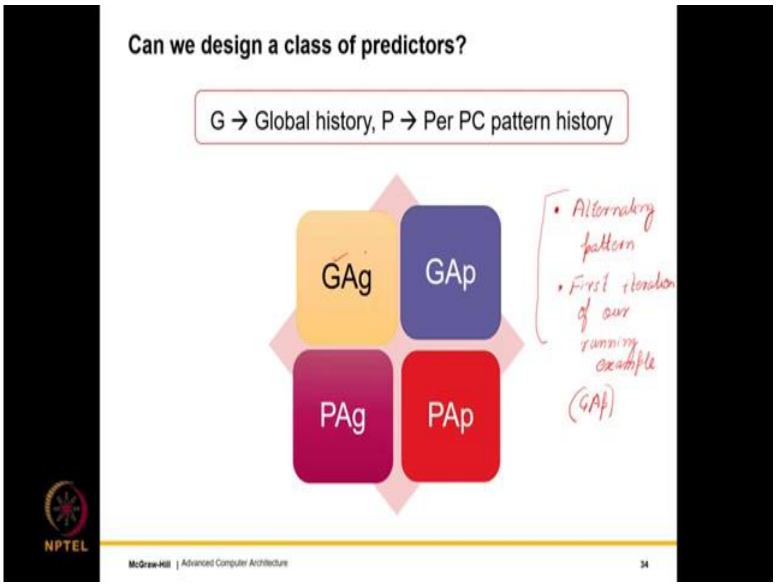
So, how do we exactly capture this context? and how do we exactly address this issue? this will be discussed a few slides later, but it is important to keep this thing in mind. So, let us proceed. So, what we have seen is that, we have created a new kind of predictor and the predictor is not just limited to predicting the 6th iteration correctly which we just did; but let us assume we have a branch that has an alternating pattern, taken, not taken, taken, not taken.

So, it is clear that a saturating counter will not capture this pattern; because saturating counters are designed to capture steady state behavior, but this behavior needs to be captured. So, the question is, will a gap predictor capture this pattern? Well, the answer has to be seen by looking at the design of the gap predictor. So, the gap predictor is essentially combining n bits of PC information and it is concatenating this with k bits of the GHR.

So, the GHR is kind of unique, it has a unique signature for taken, it has a unique signature for non-taken; then of course, a gap predictor will capture the pattern, but this need not be the case all the time. So, there need not be a very unique signature for this; because after all the GHR is not taking the PCs into account. So, essentially it is the last k branches in the dynamic instruction sequence, which can be any set of program counters.

So, in this case having a unique signature for taken and not taken might not be possible all the time. So, we need to design some other predictor that can capture such alternating patterns and also solve the problem that we discussed regarding the first iteration of the loop. So, both of these have to be captured somehow. So, we need a better way of capturing them.

(Refer Slide Time: 37:32)



Can we design a class of predictors?

G → Global history, P → Per PC pattern history

GAg
GAp
PAg
PAp
• Alternating pattern
• First iteration of our running example
(GAp)
NPTEL
McGraw-Hill | Advanced Computer Architecture
34

So, to design a better way of capturing such patterns. So, what are the two patterns that we were still not being able to capture? Well, the first is an alternating pattern, which is taken, not taken.

So this can happen, particularly if you have random numbers in the code, this can happen; the other is the first iteration of a running example. So, then of course, the context is coming from instructions that are outside the function and we pretty much do not know and the context is not under our control.

So, basically in that case also, if the context as we have discussed, if it is not taken; then of course it is possible to predict the branch with 100% accuracy. But if the context is taken and taken, then there is a difficulty. So, if I were to look at both of these problems; I would need as branch predictor, which is more complicated than gap.

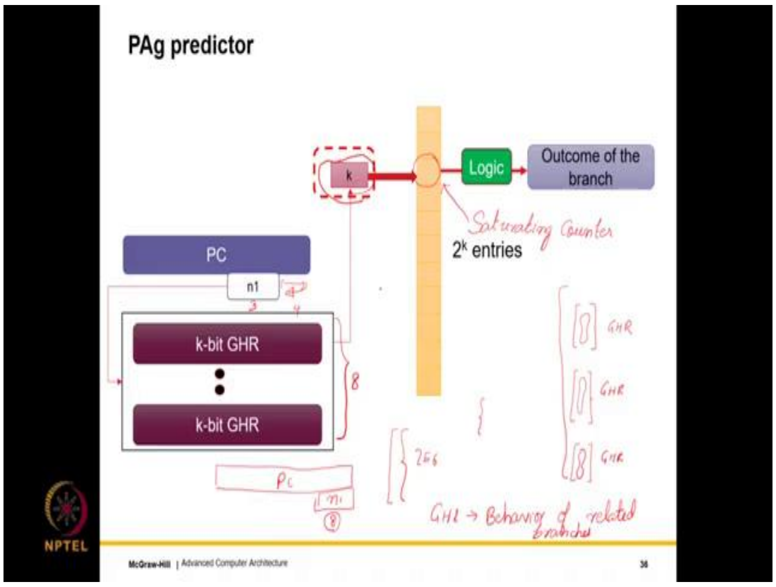
So, what I do is, I start with creating a design space of predictors, where I call them GAg, GAp, PAg, and PAp and we will see that they have, they have they captured different patterns in the code. And these patterns are commonly visible patterns in the code and only if they are captured, when we get a high accuracy.

So, at this point it is important to mention, what is G? and what is P? Well, what is G? If I were to come back to this slide. So, G is the global history register. So, G is the global information of the behavior of the last k branches in a dynamic instruction sequence, this is the G and essentially the pattern is something that takes PC bits. So, it takes these PC bits.

So, in this case, the reason we call it GAp is because we have the global history of course, and then we take these PC bits concatenate them. And we address a table. So, this table is also known as a PHT or the pattern history table and from this table, we get a set of we will get single saturating counter as an output and with this we can determine the outcome of the branch.

Furthermore, we to reduce aliasing as we have discussed, we can store a part of the tag within each entry. So, that is independent to this discussion, but the reason as you can see, it is G A p is because this is the G part and the PC this is the p part. So, similarly we can have a hierarchy a design space of other kinds of predictors, which hopefully will capture these two patterns, the alternating pattern and predicting the first iteration of a running example with a greater accuracy.

(Refer Slide Time: 40:37)



So, let us consider the most simple GAg predictor, which of course is not practical, it is not used; but it is a bonafide member of our branch predictor design space. So, in this case, we do not use PC bits at all. So, there is no P as you can see.

So, we have a single k bit GHR, we only use the global history to access a table with 2^kentries, where each entry contains the saturating counter and this determines the outcome of the branch.

So, this of course, is not a good idea, because we are not using the PC information at all. So, there is no per branch information that is stored, which of course is not the right thing to do; to solve many of our problems, we will use the PAg predictor, PAg predictor.

So, this predictor is designed differently. So, it is meant to solve both the problems that we have been discussing. So, instead of one GHR, we have multiple k bit GHRs. So, what we do is, we extract n 1 bits from the PC address. So, I have deliberately offset n 1, I have not kept them as a LSP bits; but they have.

But I have kept them somewhere in the middle. So, what you need to do is that, if n 1 = 3, we will have 2^3, 8 of these GHRs, each GHR will store k bits.

So, what we do is that, we use these 3 bits of the PC address, the corresponding GHR extract k bits from it furthermore these k bits are used to index and entry the pht of 2^k entries, the history table with 2^k entries and each of these entries contains a saturating counter.

So, what is the advantage of having multiple GHRs as opposed to a single GHR? Well, the advantage of having multiple GHRs is like this that, consider a very large program. So, typically what happens is that, we operate in the vicinity of one set of functions. So, maybe let us say we are initializing the program.

So, this would be one piece of code; then let us say that we are executing some logic, it will be another piece of code. So, consider this PowerPoint presentation. So, when I started PowerPoint, the initialization code was there, then I was making the presentation so another piece of code was being used and then I am giving the presentations, another piece of code is being used.

So, all of these pieces of code are in a sense co located in terms of addresses, they are located in similar addresses. So, we should ideally have separate GHRs for these; because we do not want the GHRs of, we do not want the single GHR, it does not capture the context very well, but for each code region we would like to have a separate GHR, such as the context in that code region which is more relevant is captured.

So, what we do is, that is the reason we do not use the LSP bits; but we also do not use MSP bits, because they might remain constant. But we extract n 1 bits from somewhere of the in the middle of the PC. So, this of course, where and which bits we extract, this is dependent on many simulation studies, so it is hard to predict; but ideally it should not be the least significant bits, because this is defeating the purpose of having multiple GHRs.

So, we typically have multiple GHRs, when we want each GHR to capture the behavior of a certain region of the code. So, that is typically the reason, why we have multiple GHRs. So, we need to bear in mind that, if n 1 if let us say we extract an n 1 bits as the least significant bits of the PC and let us assume that n 1 is let us say 8. So, what will actually happen is that, we will pretty much be tracking 256 contiguous locations and each GHR will capture the context of the instructions whose lowest 8 bits match.

So, in general there is no meaning for the context of instructions whose lowest 8 bits match, because there is no relationship between them typically; but if let us say the program is kind of constrained within these 256 instructions.

Then the GHR can pretty much provide the context of the single instruction of let us say one instruction it can kind of provide with past history. But this any way we do not require, because we have other ways of recording as we have seen histories of a single instruction. So, GHR pretty much captures the behavior of other related branches. So, the philosophy of having a GHR is to capture the behavior of related branches.

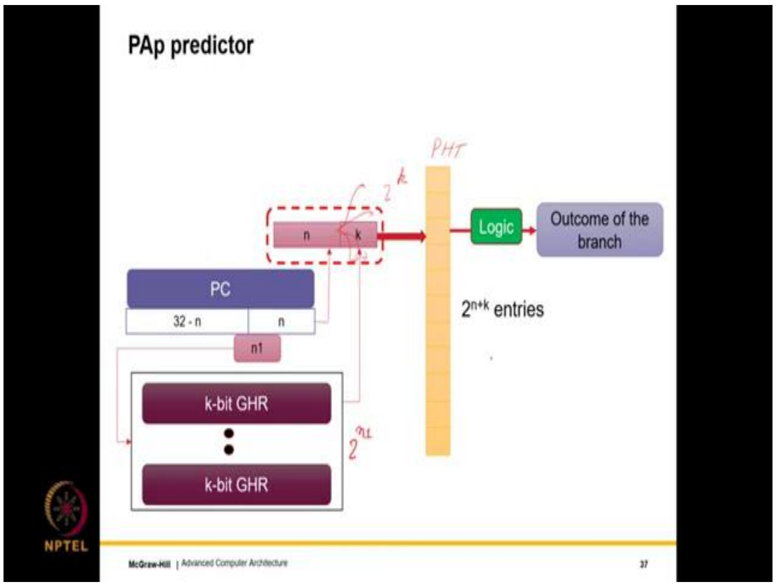
So, what makes a branch related? Well, one way that we can actually define it is that, we can say that look all the branches that are within a certain address range are kind of related to each other; because they are pretty much maybe part of the same function or part of the same super function that calls the sub functions.

So, in that case, it does make sense to offset n 1 a little bit, such that if let us say n 1 is offset by 4 bits; so then pretty much we are capturing a region of 16 bytes and any instruction within that region it will, they are pretty much sharing the GHR entry.

So, there are numerous tradeoffs in designing this, but pretty much the idea of a GHR is to capture the context of the current branch and co located branch and other branches that are in the same address region. Once we have done that, the context is used to address the table of saturating counters.

So, as you see this is not, I mean all that beneficial; well the reason being that we are only capturing global history; I will read global history of a given range, but we are not using the PC bits directly.

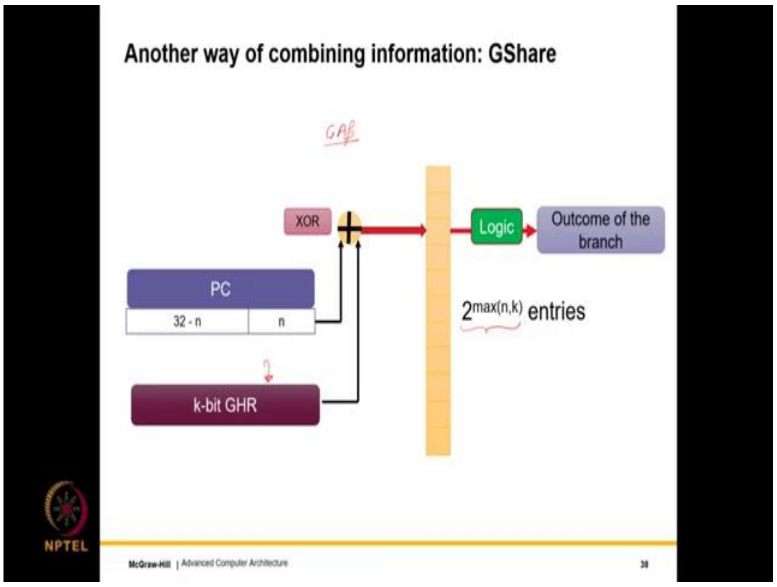
(Refer Slide Time: 48:05)



So, the pap predictor pretty much combines everything that we have seen up till now. So, it uses n PC bits directly in addition it has 2^{n1} GHRs, where essentially we capture the behavior of other branches right in the region and we also taken the, take into account the last n bits of the PC. So, what we do is, we concatenate them.

So, concatenating them basically means that, for a given PC, we have 2^k entries in a pattern history table. And so, then we can pretty much capture the, capture different kinds of global history patterns; for different kinds of global history patterns, we can store different saturating counters. So, the PAp predictor is clearly the largest of all four and it is also the most expressive predictor.

(Refer Slide Time: 49:10)


Another way of combining information: GShare
GAp
XOR
Logic
Outcome of the branch
PC
32 - n
n
k-bit GHR
$2^{max(n,k)}$ entries
NPTEL
McGraw-Hill | Advanced Computer Architecture
38

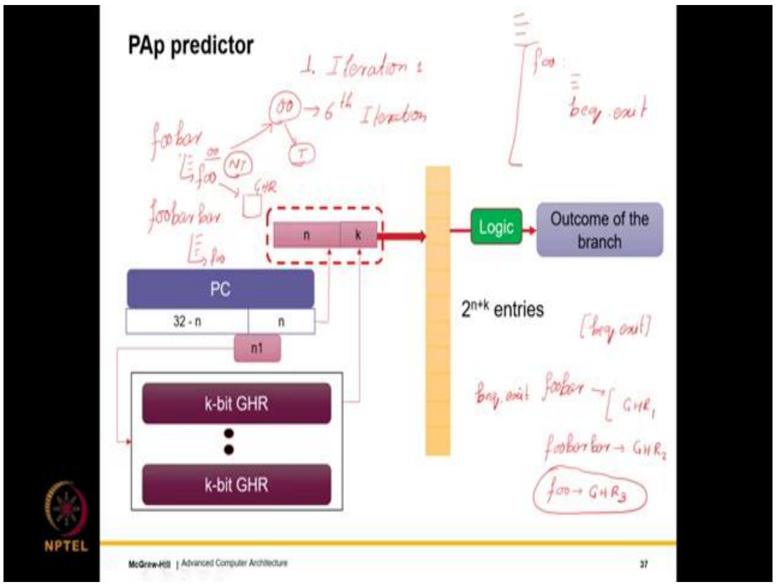
There is another way of combining the information. So, as you can see in the PAp predictor we have 2^{n+k} entries; there is another way of combining this information.

What we can do? Is that, we can take a k bit GHR, of course it is a single GHR and we can take n bits of the PC and we can XOR them. So, xoring them is basically one way of kind of mangling the bits. So, in that what happens is, this somehow mixes, so this is actually an extension of the GAp idea or the GAp predictor.

Where we the G A p predictor would do the same, but it would concatenate. In a g share predictor what we do is instead of concatenating, we compute an XOR, exclusive OR of the bits, which whose conceptual aim is to pretty much achieve the same of combining the PC and GHR information albeit using much less space.

So, that is the reason the number of entries that we have here is $2max^{(n,k)}$ entries as a total number of entries that we have here. And each of these entries has a saturating counter, which of course we have seen a lot of saturating counters by now; all of them operate in the same fashion. So, let us go back to the PAp predictor over here and first clean the ink on the slide.

(Refer Slide Time: 50:45)



So, let us see if it solves our problems or not. So, let us first consider the first issue that we are highlighted, which was to take care of iteration 1. So, in iteration 1 our line of argument was that look the foo function starts; we have some instructions and then we have beq dot exit.

So, we do not know the context of the foo function, in the sense what were the branches before we entered the foo function; because they determine the global history. And since they determine the global history, we were not sure; because you know there are different functions that could be calling foo and given the fact that different functions are calling it, they could have different behaviors.

So, for example, it is possible that one function foo bar would be calling foo, that would have a different context and another function foo would be calling foo, that would have a different context. So, capturing the context in this sense is in a sense important. And also the other thing is that, maybe you see if they have different contexts, then it is fine.

That will get captured by a single GHR; but it is possible that they would have the same GHR patterns and then there would be some degree of aliasing. In the sense what we have seen is that, if let us say the last two branches are not taken and not taken essentially 0 0, we could predict the 6th iteration correctly.

Now, it is possible that the context from which the foo function was being called, we are having a similar GHR with 0 and 0. So, in this case, we know that the first beq dot exit that is not taken; but we see or in a similar context, it is actually taken. So, to solve such kind of issues, it is important to actually have multiple GHRs, which is exactly what a PAp predictor has.

It has multiple GHRs. So, in this case, if foo bar is calling foo, we would use one GHR for all the branches that are within the foo bar function, but once it enters the foo function we would use. So, we would enter a different region and this region would have a separate GHR of its own.

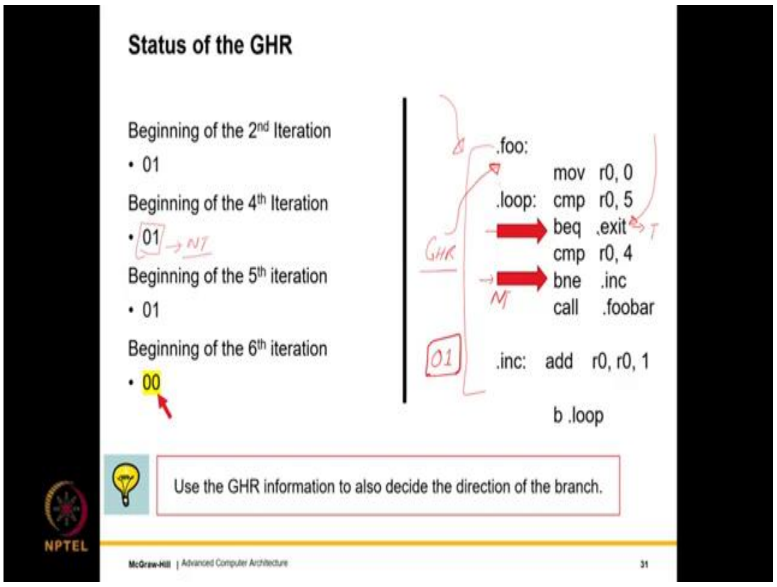
So, this separate GHR of its own would capture the context of all the branches that are only within the foo function and adjoining addresses. So, what we are saying is that, if we have multiple GHRs; then all the branches within the foo bar function which will be in the same region will have 1 GHR for it, which will be a separate GHR.

All the branches in the foo bar function, given that they will be co-located will have a separate GHR for them and all the branches within the foo function, which are all co-located in terms of addresses, will have a separate GHR. For predicting the first branch, the branch that we are interested in beq dot exit.

We will essentially use GHR 3, which is the GHR of the foo function and this would kind of be unaffected by their behavior in other regions of the code. So, what would this be populated with? This would be populated with the information of the branches, the last time the foo function ran and managing this is far easier. So, the last time that foo function ran.

We deterministically know what were the values of the branches; the outcomes of the values of the outcomes of the branches and that would accurately uniquely precisely determine the context and this context can be used to predict. So, let us give one example, let us go back to our loop example, where we essentially had two branch instructions beq dot exit. So, I will take you there for reminding you.

(Refer Slide Time: 55:28)



Status of the GHR

Beginning of the 2nd Iteration
- 01

Beginning of the 4th Iteration
- 01 → NT

Beginning of the 5th iteration
- 01

Beginning of the 6th iteration
- 00

```
.foo:
        mov  r0, 0
.loop:  cmp  r0, 5
        beq  .exit
        cmp  r0, 4
        bne  .inc
        call .foobar

.inc:   add  r0, r0, 1

        b .loop
```

Use the GHR information to also decide the direction of the branch.

McGraw-Hill | Advanced Computer Architecture 31

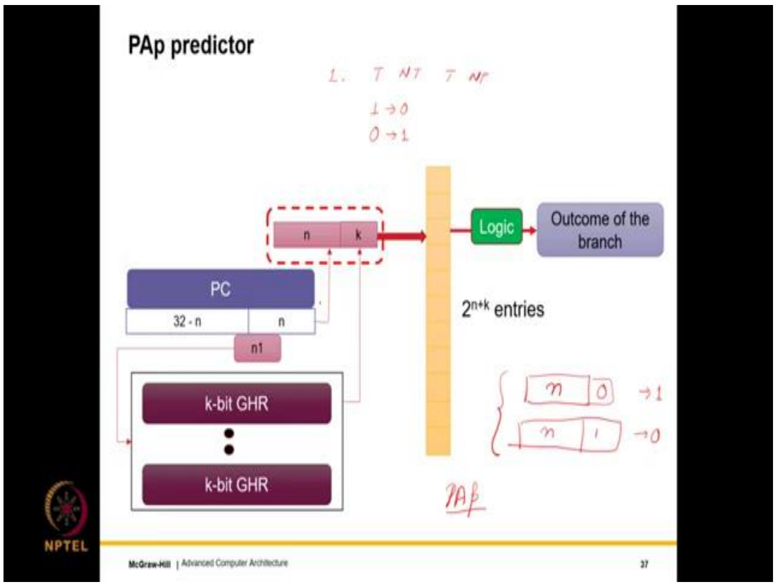
So, we had two branch instructions beq dot exit and bne dot inc. So, what happened is that, in the first in this branch of the 6th iteration exited. So, what were the two branches before it? Well, so this was one, so this one was taken and the one before it which was bne dot inc, so this one was not taken.

So, basically given the specific convention that we are following, what the GHR would contain in this case would be the contents of this one first, which is 0 and the contents of this one which is 0 1. So, which means that if I am able to dedicate a GHR for the foo function.

The next time that we come to this branch, we are going to see the value of the GHR as 0 1 and given the fact that 0 1 always corresponds to not taken, we will predict correctly. So, what are we effectively done with the PAp predictor? What we have done is that, for every region of code, we have broadly associated a GHR with it; this ensures that this GHR is not polluted with the GHRs by the behavior of branches in other regions.

And this further means that, let us say we exit function foo go to other functions do whatever we want; the moment we come back in, we will pretty much use the GHR associated with this region, which will capture the behavior of what happened last time. And so, this at least in this case is a more accurate predictor of the branches. So, let us now consider the second pattern, which was an alternating pattern.

(Refer Slide Time: 57:40)



So, the alternating pattern let us say for a given branch. So, let us say the pattern is taken, not taken, taken, not taken and in between an arbitrary amount of code can be executed. So, what we really want in this case is, we want to in this case we are interested in the last outcome of the current branch and based on the last outcome of the current branch, we make a prediction.

So, the last outcome was 1, we say it was 0; if the last outcome was 0, we say it is 1. So, this is ideally not possible in a saturating counter based system; the reason being that the saturating counter will always predict. So, saturating counter is more for a very biased branch; but in this case, it is an absolutely fifty-fifty alternating unbiased branch.

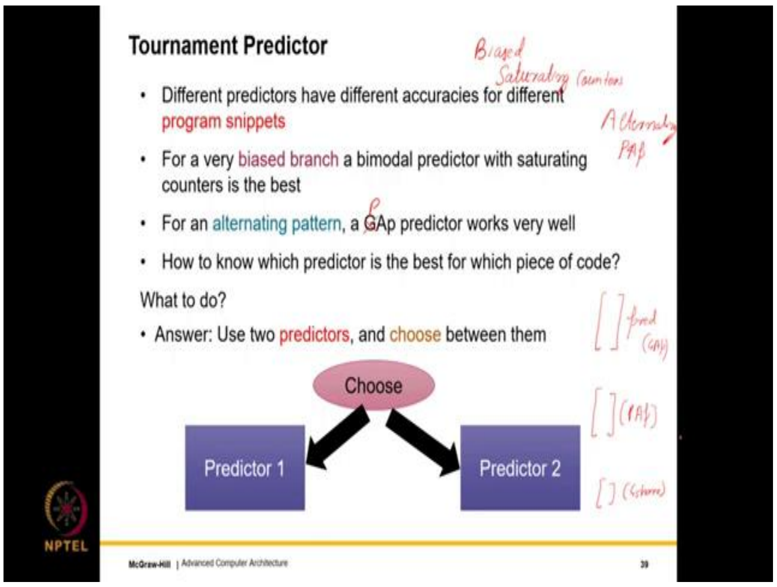
So, saturating counter will not really work, because the moment it sees 0, it will try to predict 0; but you are trying to do the reverse. So, you would need a global history in this case and the global history should be confined to this PC; it should not be polluted with other global histories. That is why we need multiple GHR registers.

Now, if you want to predict it; well all that we will do is, we will take n bits from the PC. And let us consider a single bit GHR, which is the last outcome. So, we will essentially have two entries n 0 and n 1. So, n 0 even if you have a saturating counter it does not matter, n 0 = 1; because of course 0 is not taken, 1 is taken and n 1 is always 0.

So, these two values will ultimately saturate and the PAp predictor will be able to learn this alternating pattern perfectly. So, what we see is that some of the major issues that we had with a traditional bimodal saturating counter or GAp predictor; the PAp predictor is able to fix most of them rather well and one reason for this is that, it is able to capture certain patterns rather very well, which other predictors could not.

One of them is this alternating pattern, the other one is the first iteration of the foo function, which should ideally depend upon; what is it, should ideally depend upon only the context of the foo function and nothing else. So, having multiple GHRs helps us avoid that destructive interference.

(Refer Slide Time: 60:30)



Tournament Predictor

- Different predictors have different accuracies for different program snippets
- For a very biased branch a bimodal predictor with saturating counters is the best
- For an alternating pattern, a GAp predictor works very well
- How to know which predictor is the best for which piece of code?

What to do?

- Answer: Use two predictors, and choose between them

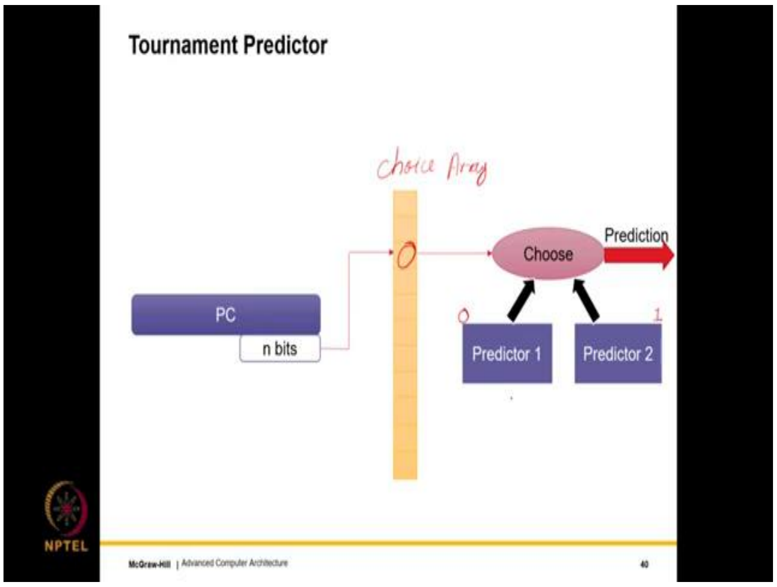
Now, given that we are seen so many predictors; what we have seen is that, for a biased branch, for a highly biased branch, we will need a predictor based on saturating counters. For something that is heavily dependent on the context, like an alternating pattern; we would need a PAp predictor, a PAp like predictor that has multiple GHRs and that has a lot of context.

So, here I actually stand corrected, for the alternating pattern actually a GAp, instead of a GAp, a PAp predictor works very well. So, the question is that, for a given piece of code, how do we know which predictor to use? Because for a highly biased region, we should ideally use a saturating counter base prediction, because it will work best; because we can always use a PAp predictor, but larger is the predictor, the longer it will take to train and more are the resources, more are the overheads.

So, in this case what we should ideally do is that, for different regions of the code depending upon their behavior, we should use different predictors. So, maybe for one region of the code, the GAp predictor is the best; for another region of the code is possible that the PAp predictor is the best, maybe for another region of the code the g share predictor is the best.

Because they have different behaviors and for these different behaviors, different kinds of predictors actually work well. So, what we need is that for a given code region, which means for a given PC; we need to choose which predictor to use. So, we will have multiple predictors and choose one of them.

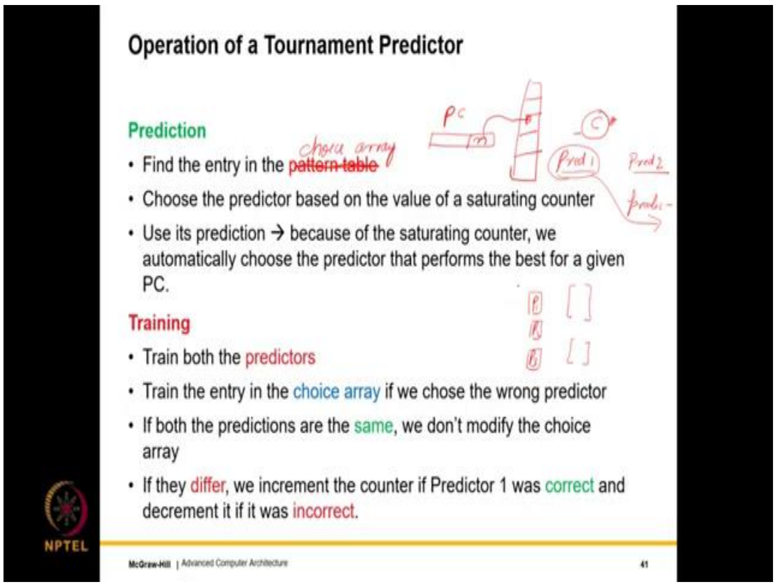
(Refer Slide Time: 62:23)



So, what we do is that, we define a tournament predictor which given in the PC we have a choice array. So, in the choice array, again we have a saturating counter. So, the saturating counter essentially tells us which predictor to use; if it is 0, we use this predictor; if it is 1, we use this predictor and based on the predictor to use, we use its prediction and that is the final prediction.

So, this the advantage of this is that for different program counters, we get to know what is the best predictor that we can use or what works best and this also gets trained in exactly the same way as a regular branch predictor and this also captures the history information to a large extent of given a range of code what to use.

(Refer Slide Time: 63:20)



Operation of a Tournament Predictor

Prediction

- Find the entry in the ~~pattern table~~ choice array
- Choose the predictor based on the value of a saturating counter
- Use its prediction → because of the saturating counter, we automatically choose the predictor that performs the best for a given PC.

Training

- Train both the predictors
- Train the entry in the choice array if we chose the wrong predictor
- If both the predictions are the same, we don't modify the choice array
- If they differ, we increment the counter if Predictor 1 was correct and decrement it if it was incorrect.

McGraw-Hill | Advanced Computer Architecture 41

So, the operation of a tournament predictor is as follows, we find the entry in the pattern table first; then we choose the predictor based on the value of the saturating counter. So, what do we have? let us not call it the pattern table, let us call it the choice array right; the pattern table is for an intra-predictor table.

So, let us call it the choice array. So, what happens is that, if we have the program counter; we extract n bits out of it and choose an entry in the choice array. So, here we will find a saturating counter c and there are two sub predictors pred 1 and pred 2; say pred 2 is correct what we do is, we increment it. And of course, this is incremented in a saturating counter fashion.

Similarly, pred 1 is correct what we do is, we decrement it. So, ultimately what will happen is that, for a given program counter, one of the predictors will most likely be found to be suitable. So, the saturating counter will go and saturate over there, we use it. Of course, later on in the behavioral changes, then the saturating counter will also embody that fact.

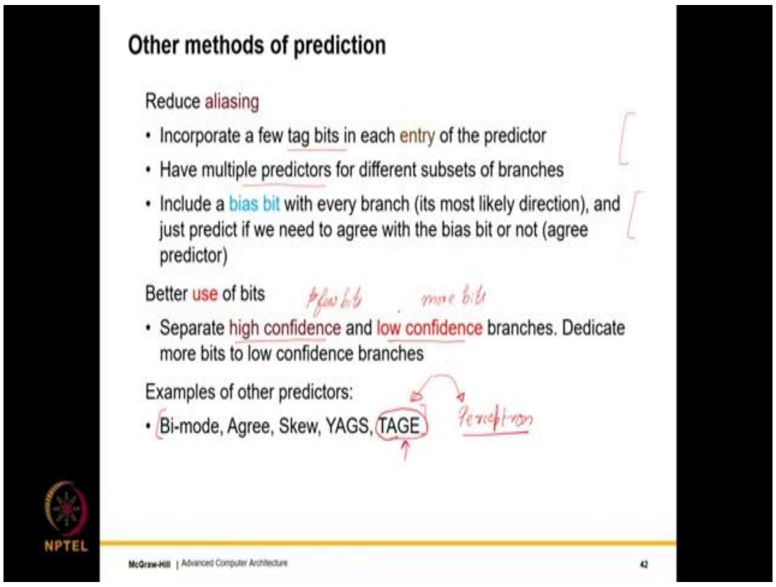
So, based on the value of this counter we choose the predictor; then the predictor we give it the PC and all information that it needs, it provides a prediction and that is the final prediction of the tournament predictor. So, for training what do we do? Well, for training we train both the predictors with the PC and the ideal outcome; we train the entry in the choice array, if we chose the wrong predictor.

So, let us say that we chose pred 1, but actually pred 2 was correct; then of course we train the entry in the choice array. If both the predictions are the same, then of course nothing needs to be done; if they differ, what we do is the if predictor 1 was correct, then we change the saturating counter.

So, let us say we increment the counter and if it was incorrect, we decrement the counter. So, so it does not matter, you know as long as we stick with one convention; that let us say if predictor 1 is correct, we keep on incrementing; predictor 2 is correct and predictor 1 is wrong, we keep on decrementing or vice versa it does not matter, as long as the same convention is followed.

So, what we see over here is that for different code regions, different predictors are known to work well and the tournament predictor essentially has all of these predictors as sub predictors and for a given code region is just historically; based on historical knowledge, it chooses the most accurate predictor.

(Refer Slide Time: 66:25)



Other methods of prediction

Reduce aliasing

- Incorporate a few tag bits in each entry of the predictor
- Have multiple predictors for different subsets of branches
- Include a bias bit with every branch (its most likely direction), and just predict if we need to agree with the bias bit or not (agree predictor)

Better use of bits few bits more bits

- Separate high confidence and low confidence branches. Dedicate more bits to low confidence branches

Examples of other predictors:

- Bi-mode, Agree, Skew, YAGS, TAGE Perceptron

McGraw-Hill | Advanced Computer Architecture 42

What are some other methods of prediction? Well, to reduce aliasing, we can incorporate a few tag bits. For different kinds of branches, subsets of branches, we can have multiple predictors; this can either be decided dynamically as in a tournament predictor or it can be decided statically.

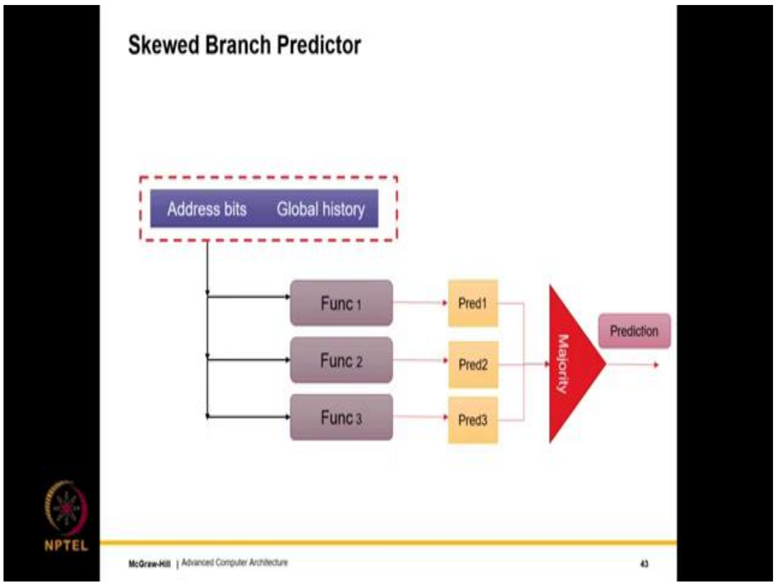
Which means the compiler can say that look for this region use this predictor, for this region use this predictor. We can include a bias bit with every branch, which maybe the compiler can find out, which is this most likely direction and just predict if we agree with the bias bit or not.

What many other modern branch predictors do is that, they separate branches into high confidence and low confidence. So, we use different bits for different kinds of predictors. So, for high confidence branches, we use predictors with few bits; but with low confidence branches, for low confidence branches, we use predictors of more bits, because of course a bit budget is important.

We have a finite amount of area on the chip and we do not want to increase it substantially. The examples a few more predictors are Bi-mode, Agree, Skew, YAGS, TAGE. So, TAGE of course, is known to be a very accurate predictor. And nowadays many predictors are kind of inspired by very early stage neural networks with perceptron's.

So, you can also look at perceptron based predictors. So, these predictors are known to have very extremely high accuracy and the way that they work is that, they divide branches into different subsets and they use different methods to handle each subset.

(Refer Slide Time: 68:15)

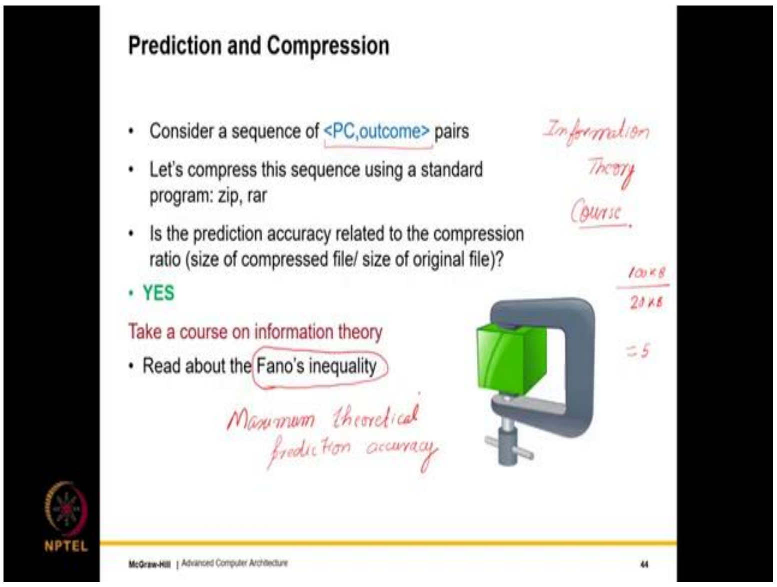


Skewed Branch Predictor

So, one of the famous predictor is a skewed branch predictor, where it takes all the information address bits global history and it has different functions to kind of mangle the bits and means compute different combinations of the bits and for each of these combinations, we use one predictor.

So, this is kind of a conceptual extension of g shape and let us say if we use three predictors, we use the majority decision; if 2 say not taken, 1 says taken, we use taken as a prediction, the majority decision.

(Refer Slide Time: 68:54)



Prediction and Compression

- Consider a sequence of <PC,outcome> pairs
- Let's compress this sequence using a standard program: zip, rar
- Is the prediction accuracy related to the compression ratio (size of compressed file/ size of original file)?
- YES

Take a course on information theory

- Read about the Fano's inequality

Maximum theoretical prediction accuracy

Information Theory Course

100 KB

20 KB

= 5

McGraw-Hill | Advanced Computer Architecture 44

Is there a theoretical analysis of branch prediction? Well, yes. So, if you take an information theory course, what you will find is that, prediction and compressibility of the sequence, they are very related. So, let us do one thing, consider a sequence of PC and outcome pairs. Let us compress this sequence using a standard program, such as zip or rar.

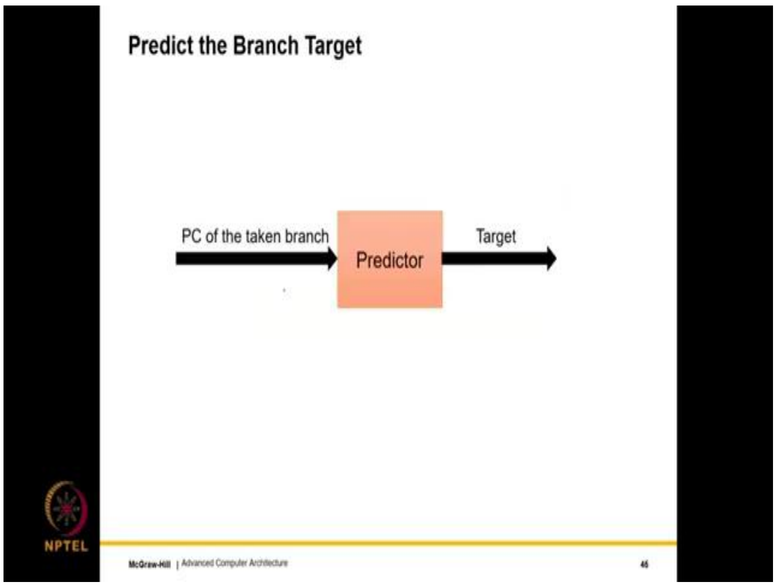
So, let us say the size of the original file, we just had the PC and outcome pairs was 100 KB and after compression it becomes 20 KB. So, the compression ratio is 5. Is the prediction accuracy related to this ratio? Well, yes; how? There is something called the Fano's inequality that relates both.

So, what the Fano's inequality basically says is that, for a given trace of branches, which means for a given program that has a certain behavior and the behavior can basically be in a sense quantified with just creating a trace of the PC and branch outcome pairs.

If we have this, there is a theoretically maximum prediction accuracy. So, there is a maximum theoretical prediction accuracy, that is related to the compressibility of the sequence and the mathematical result that stands for this relation is the Fano's inequality. So, this is the maximum theoretical prediction accuracy, this can be given by the Fano's inequality.

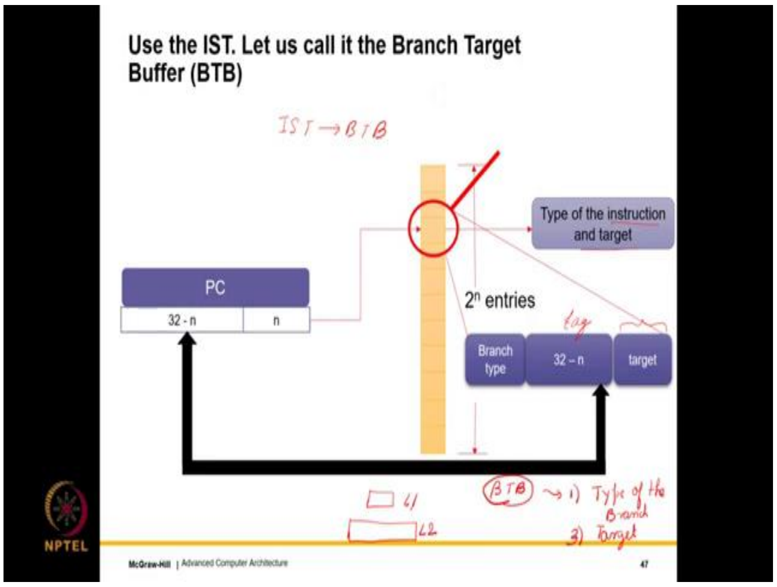
So, information theory has a concept called entropy, which is very closely related to the compressibility of a sequence. As a Fano's inequality uses that to put a bound on the maximum prediction accuracy; which means that regardless of whatever you do, we simply do not have enough information in the sequence to have a higher prediction accuracy. So, we have looked at predicting the branch outcome, which is arguably the most complex part of this design.

(Refer Slide Time: 71:43)



Predict the Branch Target

(Refer Slide Time: 71:52)


Use the IST. Let us call it the Branch Target Buffer (BTB)

IST → BTB
Type of the instruction and target
PC
32 - n
n
2^n entries
tag
Branch type
32 – n
target
BTB → 1) Type of the Branch
3) Target
L1
L2
NPTEL
McGraw-Hill | Advanced Computer Architecture
47

The next is prediction of the branch target, this is easy. So, we have the PC of the taken branch, we need to find the target; well, all that we do is that we augment the IST, which is something that we had promised in the last video that we are going to do.

So, we augment the IST; recall that the IST was a single 2 to the power n entry table, where we use the last n bits of the PC and it gives us the type of the instruction. In this case, we store the target over there. So, we store the type of the instruction and the target. And for additional we can also store a tag for anti, for taking care of aliasing related issues and of course, we can also store the target.

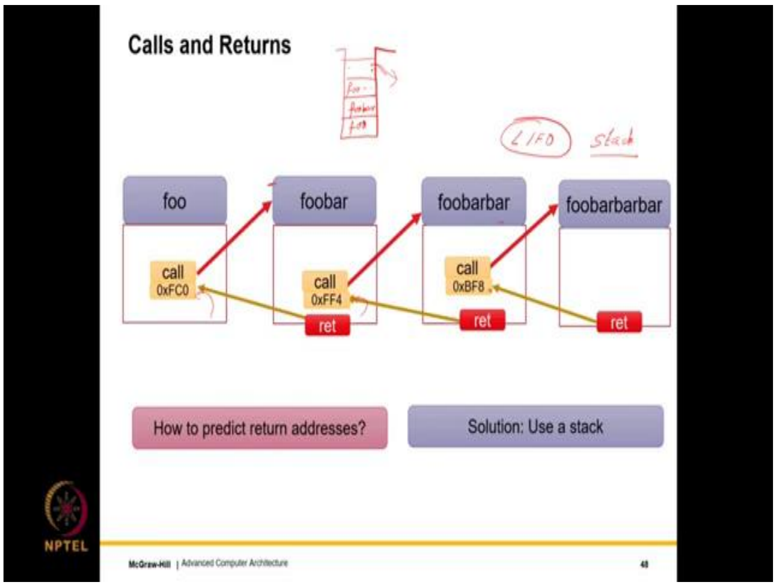
So, the BTB which is essentially extending IST solves two purposes, which is it solves problem 1; problem 1 was the type of the branch and it solves problem 3 which was finding the target. So, along with the type of the branch, we also stored its target and anytime we see a branch, we record its target over here.

Such that we access the BTB and we get both the type of the instruction; if it is a branch, what kind of branch and also its target, because for unconditional branches, the outcome need not be predicted, but the target has to be predicted.

So, the BTB in that sense is a very important structure and modern processors have actually have a two level BTB; they have a level 1, BTB which stores the most frequent program counters and then it has a level 2, BTB, which stores slightly in frequently observed program counters. So, together, so the L 1 structure is small and fast and the L 2 structure is slow.

So, given that now we have solved all three of our problems, which is finding the type of the branch, predicting the outcome taken or not taken and finding the target, let us look at the other branch types.

(Refer Slide Time: 74:08)



Calls and Returns

How to predict return addresses?

Solution: Use a stack

So, unconditional branches of course, nothing much needs to be done; we just need to find the target and that’s it, but calls and returns are interesting. So, consider a function foo. So, the function foo calls foobar. So, this is the return address; foobar calls foobar bar, this is the return address and foobar bar calls foo bar bar bar. So, then again we return. So, when once we return, we return to this return address once again, once again.

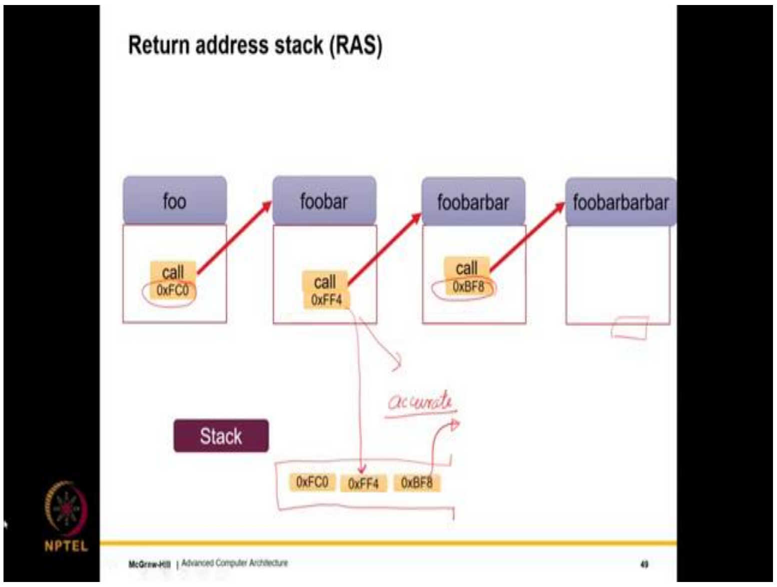
So, how do we predict these return addresses? Well, we can use a regular BTB mechanism, that can be done; because after all to predict. So, call is also an unconditional branch and to predict the starting address of this function, which is essentially the branch target of the call, we will use a BTB.

So, why not do it for ret? Well, we have something; we have a far more accurate solution. So, just take a look at this pattern. So, if we look at this pattern, it is a last in first out kind of pattern; in the sense that foo bar bar bar was last in, but it was first out, this was second last in and second last out.

So, anytime we see a last in first out pattern, we use the stack data structure; the stack data structure is nothing, but just like a stack of books, they stack the function. So, first we stack foo, then we stack foo bar, then we stack foo bar bar, then we stack foo bar bar bar; whenever we return, all that we need to do is we need to pop out.

So, we stack means what? We stack the return address. So, whenever we return from let us say foo bar bar, we just pop out the function and we basically that is the return address and the return address is supplied to the pipeline. So, how would the stack look like? Well, we have a description in the next slide.

(Refer Slide Time: 76:26)



Return address stack (RAS)

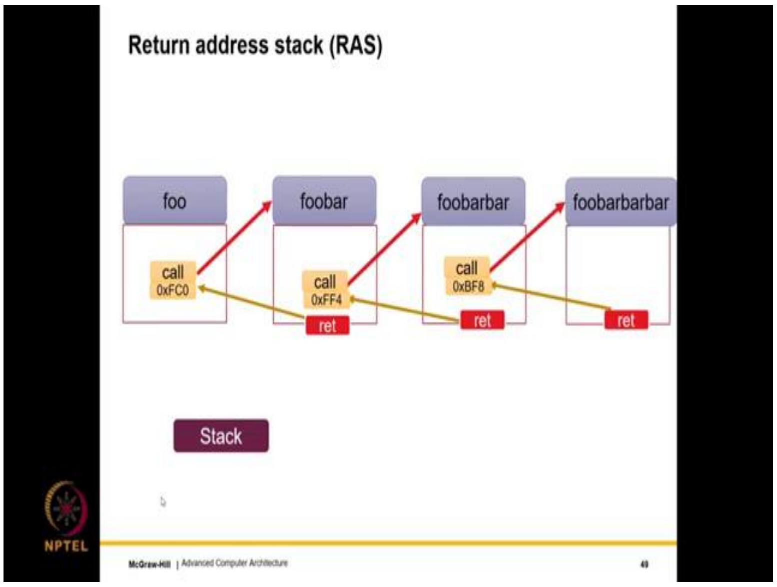
So, let us see that foo calls foo bar and the return address is 0xFC0. Say FC0 is a horizontal stack. Now, let us say foo bar calls foo bar bar. So, in this case 0xFF4 will be put in the stack, which is the return address for the foo bar bar function; then again let us say this is one more function call. So, then the return addresses again put in the stack.

Now, once we return what we do. So, take a look at that again. Once we return, all that we need to do is, we need to pop the stack; the top entry of the stack corresponds to where we need to return to which is essentially this address, you pop it. Let us say we want to return again, then we pop it once again.

And now, let us say from here we call some other function; well no problem, you pushing the return address once again and that is not an issue. But let us say that this function returns again, then essentially the last return address is popped and now our return address stack is empty.

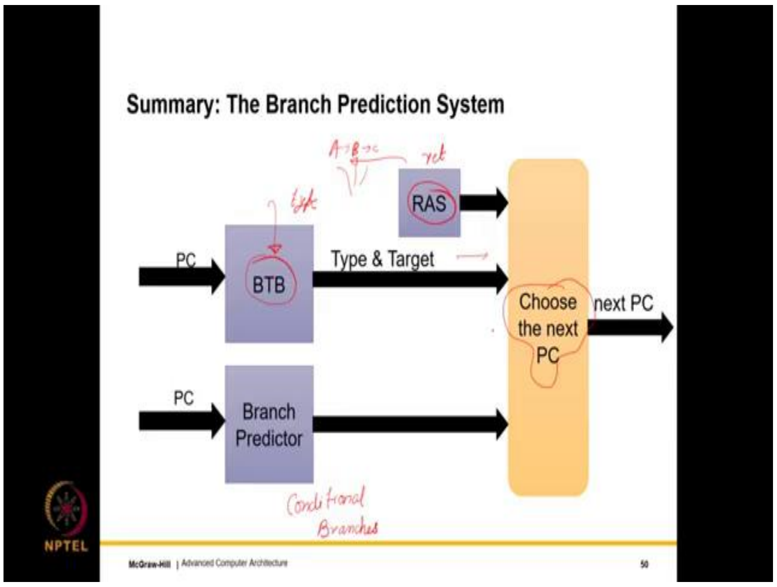
So, in this way instead of using a BTB which is kind of inaccurate, there is far more accurate solution of actually operating the function calls and returns; all that we need to do is, we need to have a small stack in hardware that simply just stacks the return addresses as we just saw. So, let me replay the animation for you.

(Refer Slide Time: 78:36)


Return address stack (RAS)

So, foo calls foo bar, we store the return address once again, once again and whenever we return, we just pop the stack and the top entry of the stack is where we need to return to once again, once again. So, this is as simple as maintaining a stack in hardware.

(Refer Slide Time: 78:55)



So, now let us summarize the entire branch prediction system. So, the branch prediction system, the BTB is a key structure in the entire system; it has an it has a rather it has an extremely important role to play. So, what the BTB does is that, for every PC, for certain bits of the PC, it stores the type of the instruction; if it is a branch, what kind of branch and so on.

Additionally, it also stores the branch target. So, the type and target are given. So, the BTB is also used by unconditional instructions like call instructions; then we have a return address stack, which is used by ret instructions. So, the moment we call a function, say if let us say A calls a function, A calls function B and B Calls C and so on.

Their addresses are put into a stack. When we return, since we will be traversing the reverse route; we keep on popping entries from the stack. Dealing with conditional branches is the most difficult, so that is the reason why our second problem was to find the direction of a branch or rather the outcome of a branch.

So, predicting taken or not taken is hard; but we looked at a lot of designs and we were able to capture many patterns with our designs. So, the basic fetch logic would consider consists of these three modules and they will help us choose the next program counter. Of course, now there are augmentations to this to perform many branch predictions in parallel, such that we can choose the next PC or the next two PCs or next three PCs or next four PCs.

Now, we will come to the last part of this lecture, which is describing the decode states. So, this will be a rather short lecture.