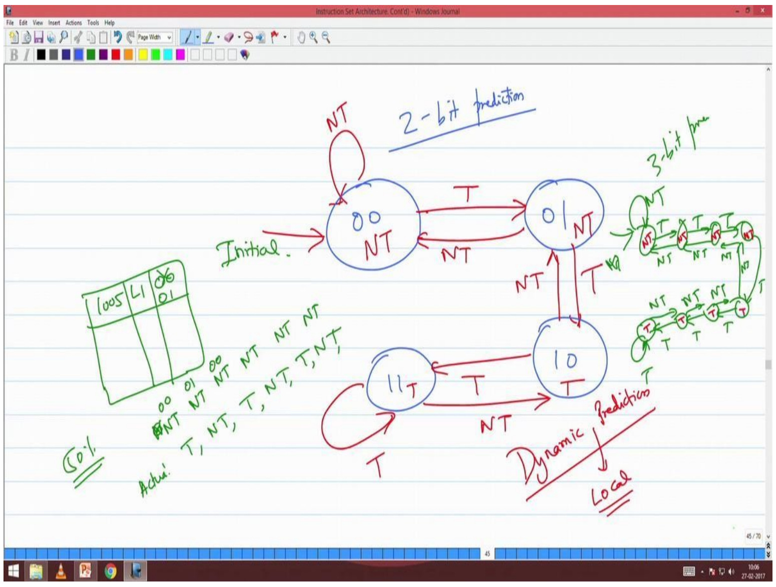
Computer Organization and Architecture
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

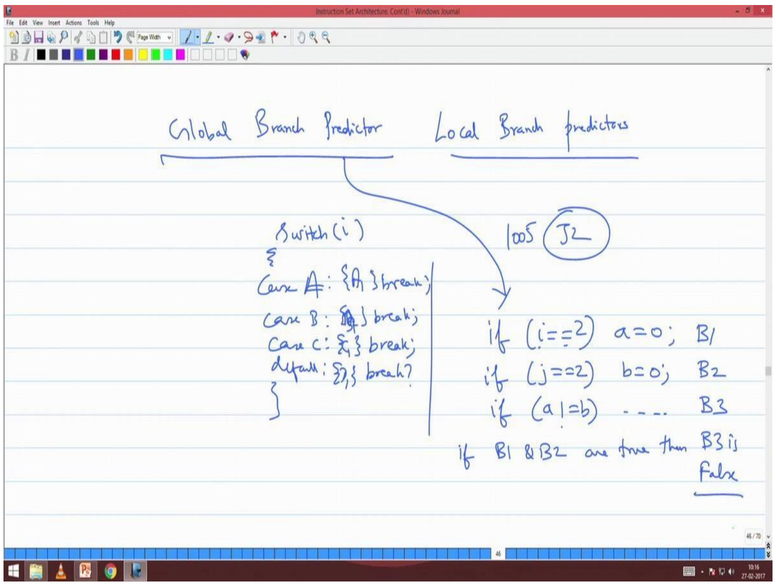
Lecture - 24
Global Branch Prediction

(Refer Slide Time: 00:24)



So, we will continue our discussion on branch prediction. So, last class we saw at two bit branch predictor the way it works and I also extended it into three bit and n bit. And we also stopped with one statement which was empirically proved statement saying that a two bit predictor works as good as a n bit predictor.

(Refer Slide Time: 00:47)



Global Branch Predictor

Local Branch predictors

```
switch(i)
{
Case A: {A1} break;
case B: {B1} break;
Case C: {C1} break;
default: {D1} break;
}
```

1005 J2

```
if (i==2) a=0;   B1
if (j==2) b=0;   B2
if (a!=b) ....   B3
```

if B1 & B2 are true then B3 is False

Now, all the branch predictions that we have seen are basically are local branch predictors. They are local in the sense the decision that we make whether it is a one bit predictor or a two bit predictor, the decision that I make for a particular branch whether it is taken or not taken is based on the performance of the that branch in the previous instances of execution. I will not consider any other branch, so if am looking at a branch which is at 1005 now the decision I am going to take at some execution of this 1005 branch is based on it is behaviour in the previous instances there is nothing you know I am not considering something other than this. Now, so is there something else that I can do and is there some sense in doing that that is what we will see with what we call as the global branch predictor.

So, should we do something better than this, we just saw some static things like not taken, always not taken, always taken then we saw a dynamic one bit prediction and a two bit prediction, but can I do something globally. So, is there some you know is there some compiling which and all is their motivation for doing it. Can you come out with some justification of why should I look into something called global branch prediction. What is global branch prediction? The decision that I need to take of a particular branch depends upon other branches that are recently executed. Can you get some examples in favour of this? So, can you get me a simple c code?

Student: nested.

What is else (Refer Time: 02:44)

Student: nested for loops.

Nested for loops, no; nested for loops would have worked even which you are not taken or not taken.

Student: if else statement.

If else statement what if else will be only one jump know. The question here that the decision for 1005 is based on some branches that are closed to it. Can you give me a motivational example for this very simple example. So, I am looking at global branch prediction; that means, I am looking at more than one branch; decision I want to take for a particular branch say v depends upon some other branch that has been that has some other branch situated at some other address, I identify a branch based on a address correct. So, then what will I decide, for example, I have switch long since I used this i correct and they have case A case B which are all conditions case C default. And each I will have a body followed by break this is switch case.

Now can you tell me anything about this, can you give me very simple example obviousness is anyway of correctness, can you give me an obvious example? Say suppose I have a branch for A, suppose I have A branch for B etcetera. Suppose A is not taken do you mean to say B will be surely taken, that anyway we have that is why we have prediction. What are we gaining there, see I should that is not a chance is not a motivation, chance has been the motivation already. So, chance cannot be further motivation for us; surely that should see that motivation is something like there is case where it is going to absolutely happen, chance is everywhere.

So, let me write this if i equals to two let me say a equal to 0; if j equal to 2 then b equal to 0. Let me say this is branch 1; this is branch 2. So, if sorry B 1 and B 2 meaning this condition let me call it are true then B 2 is false, correct. So, this is a very common code we can see, very simple code correct. So, now, whether B 3 will be taken or not can be inferred from whether B 1 was taken or suppose let us say if the condition is true I will take it, let us assume if the condition is true. So, if this condition is true and this condition is true, surely I will know that this is not going to this is going to be false.

So, I can infer something about this third branch depending upon the behaviour of branch 1 and branch 2 correct, fair enough. So, this is what we mean by a global branch prediction. There is a motivation for this and this motivation will not come from your for, while, and other structures it actually comes from internal statements. So, if I have a ten million times executing loop in which I have these three constructs then it is worthwhile for me to now take this type of a correlation there. It is actually the logic that is inside the body of a loop which results in lot of conditional jump statements which can basically get some you know favour from these type of a global branch prediction. So, this is what we this is one motivation this you will see in all the books this is one site variation this can be there. So, this is one motivation for why I should look at branch prediction, global branch prediction. And this global branch prediction is a part of some if else structures inside a loop, it is this global branch prediction may not be that useful or that hilarious if I am going to look at while loop, for loop or do while loops, fair enough.

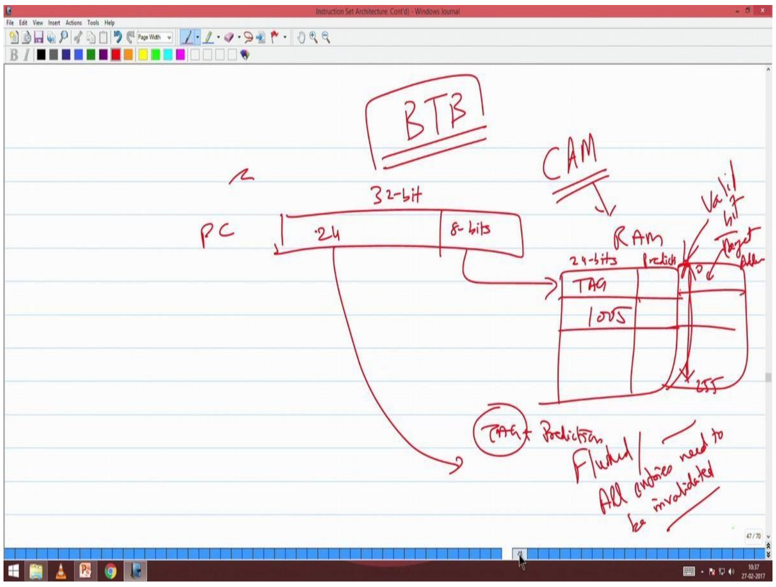
Now how do we how do we infer a global branch predictor, how will I infer that this branch is going to be basically decider upon by previous branches? Can I look at a one million line code in which there are several if else statements, and there can I go and make this type of inferences, this is one story. The second story is how do we actually implement this in practice, these are all concepts of advance computer architecture. So, you take the elective computer architecture M.Tech elective if you want in your subsequent semesters and there you can see how these things are constructed. So, I am also trying to tell you all of you that what you can expect in an computer architecture course. Now, you have done computer organisation architecture B. Tech [FL] should I do something more on architecture, yes lot of things are there. So, you have an advanced course, it will teach you much more. So, keep that in mind.

So, how do we construct the real architecture, how do I construct a real architecture

which will handle global branch prediction? How do I infer like somewhere I should do some pattern analysis and infer. So, these are all you know stories that I will leave for you, but here now we see that there are very modern computer architecture still using static prediction. So, this story that I have been telling and the story which I can take forward and talk more right on global branch prediction and etcetera. These are going to be something, which are not so practical, the very good to be on paper, but zero on wafer it can be a very good [FL] paper, but then we actually try and implement it on the wafer it goes for a toss, it will need another nuclear power substation to basically power this. So, much things that does for a branch does not matter.

Now, the next question comes how do we implement this prediction? So, next what should be the next immediate question that you need to answer any way, we have been answering these query. So, when we did this memory aliasing for load store. What was it that how did we conclude, how did we conclude what was my last statement regarding memory aliasing that it could be efficiently implemented and that we used a content addressable memory which had only the number of locations in that content addressable memory was equal to the number of load store units correct. And that is why we said that this can be implemented trivially because I will have only four or eight load store unit and having an eight address content addressable eight location based content addressable memory is very is very easy for us to implement.

(Refer Slide Time: 11:15)



Now, the story is now I am having branches how many branches will I have, I will have lot of branches not be 1, 2, 3 and all, will be many. Now, how do we implement this, how do I implement the branch target buffer? What is done is told now how do we implement it and why do we implement it what, how, why right? So, how do we implement this branch target buffer? What is this branch target buffer? The way I describe this was it is a content addressable memory. So, what will I do, I just give the address and then find out whether it is a branch or not and then I start making a prediction.

But in practice how do I do this because I will have lots of branches and if I am going to make a content addressable memory say with 32, even 32 branches I am gone that will be a very huge hardware. So, the way this is implemented is that you take the program counter actually has a 32 bit address it has a 32 address I will just take the first say 6 or 8 bits and I will take this 8 bits into index into this memory. So, this is basically a RAM your cam is implemented as a RAM in the first eight bits are used for. So, this RAM will basically have 8 bits means 256 right. So, 0 to 255 and I will use the first eight bits index into this right. And here I will hold a tag of the remaining 24 bits. And then I will have the prediction bit. So, I will have that tag here, I will have the prediction bit. Are you able to get what I am trying to say?

So, now what will happen is every time the eight bits the last eight bits of the pc is indexed into it and you get the data you get the tag bit. The tag is you get the tag plus prediction. And if this tag matches this tag, the remaining 24 bits then we know that this is a otherwise what it will what we will do we need to we will think it is not a you know it is not a branch and then we will proceed. So, every time my 8 bits will be taken for example, in this case and the index using that 8 bits into this memory and then get the remaining 24 bits and compare it with the actual 24 bits of the pc. And then if it matches then it is a conditional jump otherwise it is not a conditional jump, this is how the cam is implemented.

So, this is now strictly a cam, but this is the easiest way for me to implement because if I go beyond 32 and reach the cam becomes very costly to implement. Now, so what are the problems in this implementation?

Student: If two things have the same last eight bits

Yes, if two things have the same 8 bits then I will collide; then I have to completely replace this that entry the previous entry and put here.

So, if I put, if I want to kill this technique, I put two branch statements exactly to 256 apart. So, this will replace that that will replace that so every time this jump will come it will not find itself it will find the other jump and vice verse correct. Are you able to get this? So, I can go and if I have a direct map like this I can have a scheme which will basically kill the performance; on the other hand, a compiler which is trying to fine tune this will see that the jumps are not 256 bits aligned, it also tried to checkout that these fellows are not 256. So, this is another intelligent thing which the compiler can do. so that I could get better branch performance. Are you able to follow? So, moment I make some restricted assumptions like this I start hating certain very interesting cases like this. So, the compiler can see what is compiler optimisation for an architecture, it is a very, very interesting important question we have talked of a couple of optimisations already correct. What was the optimisation that we talked in the previous class? If I have always not taken as a static branch prediction, then a do while statement can be.

Student: (Refer Time: 16:17) converted to a for while.

Converted to a for while for or while statement I can. So, this is was one interesting optimisation that we saw in the previous class.

Now, this is another optimisation I will not put two branches exactly 256 apart. If I am having such type of a predictor the next thing what should be the next refine matter. What could happen if this process goes and another process comes and sits in the same location. One process is completed the things are freed, another process comes and sits in the same location, and then I start executing it. And then if this branch target buffer is not cleaned then you go and say that this is a jump instruction. Actually it may not even be a jump instruction if entirely new program comes and sits in the same place of the old program then what we need to do your entire branch target buffer should be flushed that means, all entries have to be invalidated. So, what do you mean by invalidating the entry should I go and make all these as zeros can I go and make all the entries here as zeros.

Student: What is the need for flush it, what is the need to flush it.

Because otherwise what will happen some other instruction may be add instruction that will be loaded and the same as the previous jump.

Student: Suppose when we do loading automatically takes care of (Refer Time: 18:08)

What takes care of what?

Student: load an instruction.

When I load an instruction who is going to take care of adjusting the cam right. See previously in 1005 there was a jump instruction that got loaded as 1005 here with some prediction right. Now, I remove that code and I am putting some other code there operating system that program completed it is gone. Now, I am putting some other code there or I did some garbage collection and I moved that code above the different reasons for why a code should not stay in the same place, it can move it can migrate it, it can be pushed. When I do that type of a what will happen if I am not going to flush this BTB then when some 1005 can be some add instruction that will be recognised as a jump. And for some stupid reason I have to add some other thing would be fetched rather than this. You are getting this if I do not flush it then after add some arbitrary instruction will be fetched now only it will be fetched if that particular instruction is outside your segment, it will give a segmentation violation, please understand it correct. The target need not be within the segment right it is a new program. So, suddenly 1005 go and fetch something in 5000, in the new program it is an add. So, when I after add you suddenly see this going to some 5000, and it creating a segment jump.

So, you have killed, you have understand what I am trying to say. So, when a particular program finishes then all of these has to be wiped off. What do you mean by wiped off, it has to be invalidated how do you invalidate it could should can I go and write all zeros in all the locations then it is going to be extremely costly. So, in all these type of entities where there is a need for me to invalidate it at some point of time, I will have something called a valid bit. If that valid bit is zero that means, the entry is irrelevant; if the valid bit is one then only this entry means. See so the moment I switch on any hardware all these entries will be some random zeros and ones correct. So, each one of them will represent some tag or other, but then what will happen.

So, what we will try and do is that we will go and make this valid bit zero then we say that all the entries are invalid. So, for me to flush the BTB, flush the branch target buffer or invalidate the branch target every entry, I need to go and set only the valid bits of all these things to 0 then automatically everything will be in place. Rather than going and making all the entries as 0 I go and make the valid bit alone as zero so that even the process tries to access it next time or the program tries process tries to access it next time it will end up with here. So, we will also have this valid bit and that is very, very important.

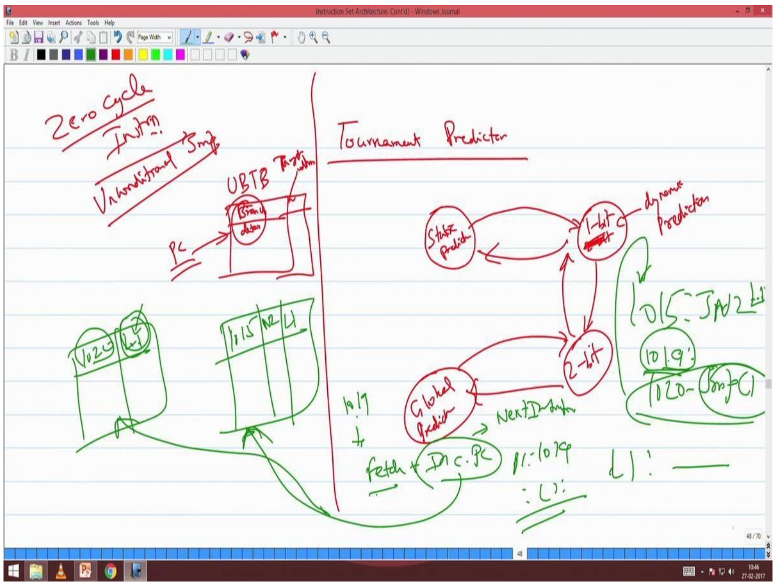
So, one thing is to flush the branch target buffer to invalidate all the entries in the BTB you can use this. And very interestingly when the system boots up right what is the reset value for all, what is the reset value for all, already when the system boots up, there is lot

of activity that is happening in there that is why when we switch on the system the initial power is a serge it actually consumes lot of power. So, that makes the entire problem very, very complex.

So, if I am going to say on these at all these tag bits prediction everything has to become to zero then it will take enormous amount of power for getting these term. So, what normally we need to do is when I switch on the power, I do not want to invalidate entire table I will just go and make the valid bit for the entire table as zero that is enough for me as you see here. So, I need not go and make all the tag bits, r bits, predict bits everything zero rather than doing it I can go and say right go and make this valid bit zero that becomes very easy. And the reset pulse also is should be a thug wire because it is going to reset say some half a million transistors at one shot. So, normally right the reset is a reset will be a tree, reset tree.

So, quickly we will reach each of these nodes and reset it. So, the reset in this case what do you mean by reset in this case, reset in this case has to go and make this valid bit alone zero. So, the amount of thugness, the reset needs that is not that is reduced here considerably because the reset is not responsible for going and erasing everything inside that RAM, but the reset will just go and make the valid bit. Once I know the valid bit is zero the entire other entries in the RAM in the branch target buffer becomes useless.

(Refer Slide Time: 23:31)



The next thing that we want to see is called tournament predictor. So, this tournament predictor is an example of some sort of machine learning that we do. So, what happens for a particular branch, I will start with static prediction. Some time when I see this is growing, we can make it as a dynamic prediction. So, I can use 1 bit, 2 bit predictors. So, here itself I can make a one bit then I can move if that it is not suiting me I can make it 2-bit. I can go ahead and make it global, I can return back the same way. So, essentially I am trying to play a tournament here and depending upon my status, depending upon the you know the type of game a branch is playing, the predictor also suggest to meet that game.

So, if it is a for do type of things then it will meet it to always not taken type of thing, but if it is something beyond for do while where I gave some examples then some sort of global prediction can also happen. So, the hardware decides for every address which stores a branch, it will decide for every such branch based on that addresses, it will find out which mechanism of the available things will suit that particular branch. So, when I go in for that particular branch, when I am trying to predict I will use that mechanism like if a particular branch has to be evaluated globally then I go to the global predictor; otherwise I can do it on the normal way. So, this is the basis of what we call as tournament predictor.

So, the last bit is of course, what we yesterday called the zero cycle instruction which is nothing but the unconditional jump. Suppose, I have like a branch target buffer, I also have an unconditional branch target buffer. In that unconditional branch target buffer, I actually store the branch details and the target address. I store the branch details and the target address. Branch details in the sense my program counter well and the target address immediately I will have the target address. So, let us say at say some 1015, I have jump on non zero statement, jump on non zero some L 1. Now, what I will store here say I will store here 1015, I will say if it is non zero and then the target address is L 1.

Now what happens whenever this execution this instruction is executed the second time 1015 is given to the branch target buffer and that fellow will tell you whether it should be taken or not taken etcetera. So, this particular table is for unconditional sorry conditional jumps. Now, I could have another table for unconditional jump it says say in 1020, I have just jump just jump this is an unconditional jump. So, I can now say that 1020 and this is the target address say L 1. So, how do I execute the instruction I execute, I fetch an instruction plus increment pc correct, this pc will now point to the next instruction.

Now, what will happen is when I fetch an instruction say let me say I have fetched the instruction at 1019 then I increment pc it becomes 1020 the moment I increment pc I will go and give it to this table this table will sorry I will go and give it I will just go and check this table. The moment I increment pc currently I am going to execute 1019. So, I have fetched instruction from 1019 and I increment pc the moment I increment pc it will go here and here it will find 1020. So, 1020 it will also L 1 that is the target address.

So, at the end of the moment I increment pc and I find this instead of loading L 1 sorry instead of loading anything, I will go and load pc with L 1. So, the next instruction that will come right will be L 1 will be the one in L 1. So, when I am trying to execute 1019 after I finish 1019 I increment my pc once I increment the pc in that cycle itself I will go and check whether the pc is stored there it is not stored here. So, 1019 I am executing; after 1019, I increment pc 1020. I go and check here yeah 1020 is loaded I am not even fetched 1020. What I will fetch is the instruction at L 1, I will not even fetch 1020. I am executing 1019 I have fetched 1019. Now, I increment the pc that became 1020. I now go

and ask that fellow is 1020 is stored, he says yes then I know for sure it is an unconditional jump I will say then give the target address he returns back L 1. So, the next time I am going to load will be the instruction at L 1.

So, this 1020 [FL] never even executed or never even fetched into the system and that is why we call this unconditional jump as a zero cycle instruction, because it has executed the unconditional jump has actually taken place. But what has happened, it never executed correct, because L 1 was not even fetched without even fetching L 1 that jump sorry without even fetching this jump L 1 which is there in 1020. I have finished execution of it, correct.

Let me just very quickly explain; 1019 I am executing at that time I increment the pc it became 1020. The moment I increment I give to this fellow and if he has an entry corresponding to that address he will tell me what is the target address. The target address L 1 I will load, so that the next. So, the pc was holding 1019 the next thing it will hold is going to be you know the L 1. So, pc will never point to 1020 at all, so that means, this instruction that this jump instruction never came in and got executed that is not here, it was just at the periphery level. So, this type of jump execution of not every jump can be done in zero cycle, the execution of this particular instance is where the unconditional can be done in zero cycles. Because jump I did not spend I did not even fetch the jump into my system, but it the affect of its execution is already got here. Are you able to follow? So, this is a notion of what we call as a zero cycle instruction.

Thanks.