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

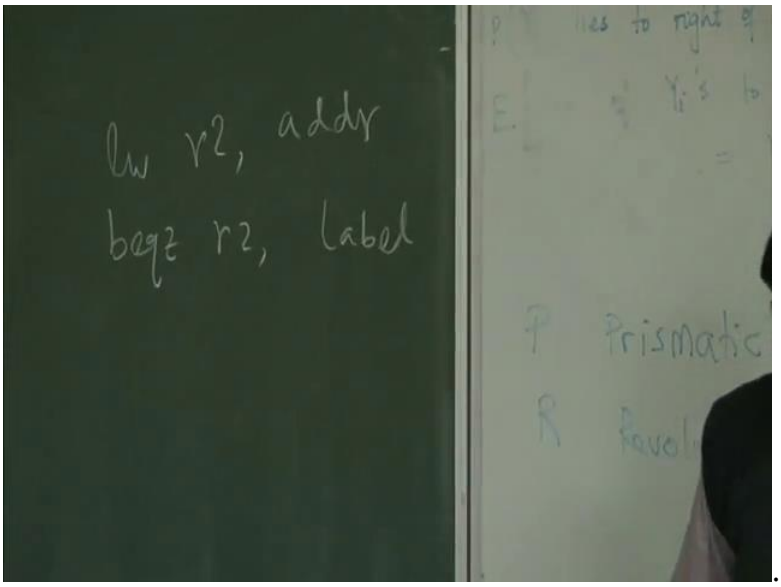
Module - 1
Lecture - 17
Basic Pipelining Branch Prediction

So, today hopefully we will grab this up, with little bit of direction. So, if you are interested in wonderful tribute, what should be your focus; that is the whole point on discussing this. So, as you have already mentioned deeper pipelines need much better predictor, because you know misprediction penalty increases, then the distance from the time you make a prediction to the time you execute the branch, essentially the cycles loss between them, because during this time between fetching along a long path. So, just to give you some example, although these are kind of old example, which r 10 k has a 5 stage integer pipeline, so that is just like what we are doing, alpha 2 1 2 6 4 has a 7 stage integer pipeline, Intel Pentium 4 extreme edition has a 31 plus stage integer pipeline. So, this was actually when Intel picked in terms of frequency. So, it was 4 Giga hertz, around 4 Giga hertz frequency. So, get frequency they were making the pipeline deeper into deeper. So, what this means is that essentially, that you can assume that in this 31 plus stage integer pipeline. We will make a bulk prediction quiet already in the pipeline, and this is where the branch executes, after 31 stages.

So, during this many pipeline stages, essentially which means these many cycles, fetching from the wrong path, essentially fetching wrong instructions, that ultimately leads to wastage of cycles. So, it is why as you make your pipeline deeper and deeper, you need better predictors to need more accuracy. Today's Intel processors do not have such a deep pipeline. It is actually short the pipeline is much shorter, but let me see bigger than 10, 10 series, so still need better predictors. So, what is branch misprediction penalty; that is a number of cycles between prediction and verification. So, from the time you know when the prediction was correct or wrong. So, until verification you will believe the prediction, and fetch along that path. On verification in the final you made a mistake, in which can be throughout all the world that you have done, and start a fetch from the corrected path. So, essentially you have a misprediction this is the amount of time that you use. This is a cycles that is used. And remember that this is the minimum

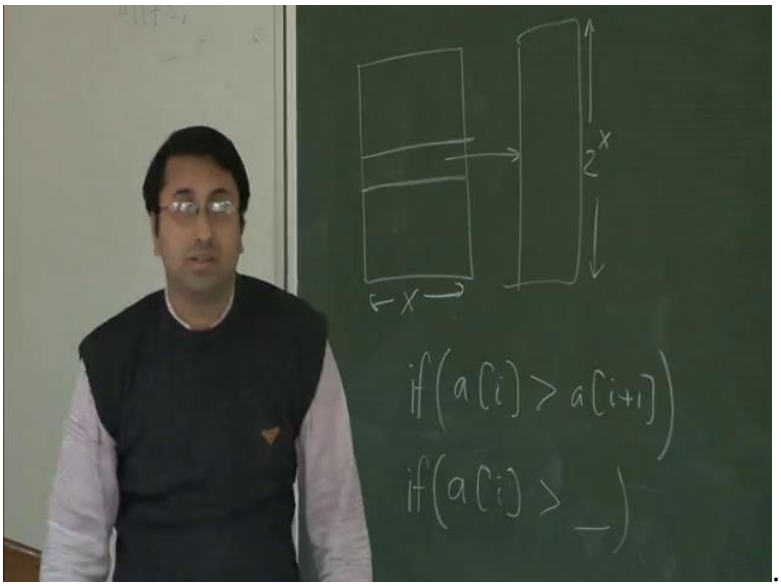
misprediction penalty, it can be larger. For example, there may be a branch, which require some data to execute, and that data may get delayed, because it may have a cache miss or something. So, for example, if you have a load instruction, that load addressing register, addressing register 2, and then there is a branch instruction which computes r 2.

(Refer Slide Time: 03:13)



Then what will happen is that, even if the branch has reached its designated pipeline stage for execution, it may not be able to execute, because r 2 is not yet available, because the loan instruction before it is still pending, does not took a cache miss when to memory to fetch the data. So, this is minimum, it may be larger from any branch instruction. So, you put the pipeline is more work is lost due to misprediction. So, what are the challenges. So, these are basic research problems in this particular area. First 1 is, how do you remove destructive aliasing in branch history tables. So, you remember that, in the example that we discussed yesterday in the g a g example. We had severe aliasing history entries. So, question how do you relieve get rid of this problem.

(Refer Slide Time: 04:54)



So, in essence this is really a hashing problem. We have to come up with smart hash functions. So, I will discuss some of these today when all of them just. And the second 1 is there is a need for larger history, because larger history of the (()). So, if you can do it over a large window of history, normally gives you much better idea about what is happening instruction; however, here the problem is that, there is an exponential relationship between the history lengths and the storage that you require. Just to remind you. So, if this is the x, then this is 2 the power of x . So, this is the exponential relationship. I want this to be bigger. So, that I can encapsulate bigger history, but that makes it tall, and these grow the exponential. So, that is a big problem actually. So, that is what makes this very challenging. The question is, what kind of other predictors that I can come up with, where this exponential relationship goes away, and can appear this thing, but I have a predictor which does not have this problem.

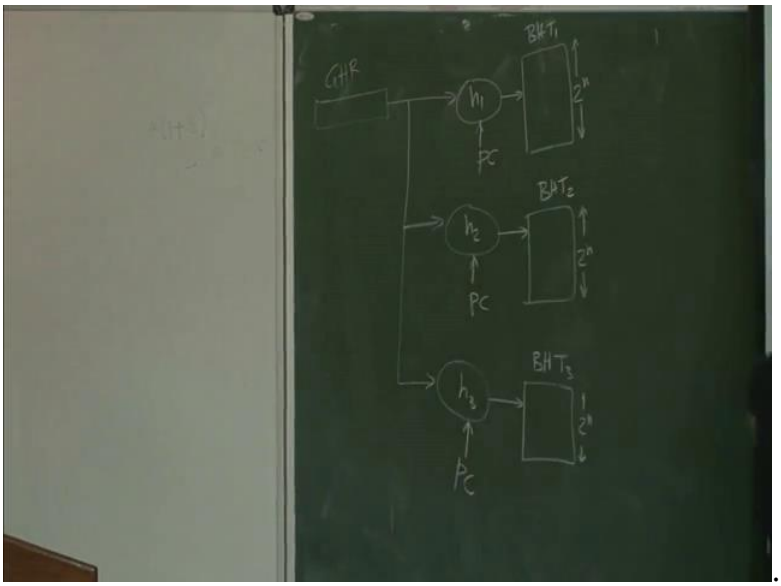
Sir bigger steady state, (( )) more time to read the steady state.

Yes that is true. Well bigger learning time, but that is hopefully you know ones you change your trade. It is just one time, every time you go over a face stage, this will happen. Once you learn then you can make better prediction for data dependent branches. So, this is a very big problem. So, for example, you might remember from your sorting algorithms. So, a typical comparison that you do there is. You often do this for example, bubble sort. It is extremely difficult to predict this branch. It depends on the

pattern of your data that you have in the data. So, this may even be better, but take about doing something like a i commutating some value here, some constant let us say. So, this kind of branches are very frequent, in programs. So, these are data dependent branches, and these are the branches that know your prediction accuracy. Because initially what you are asking is, how predictable is this value, that is what really you are asking; that is extremely difficult to predict.

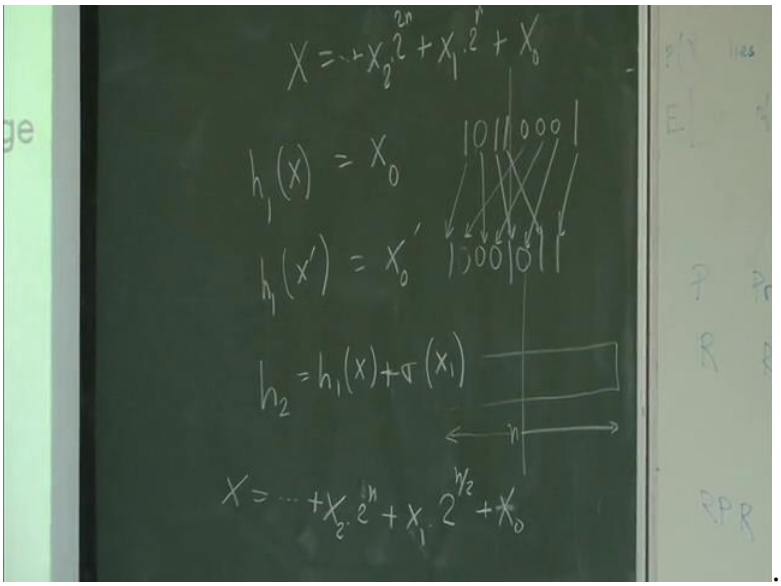
The third problem is not really about direction predictors, but it is about efficient handling of indirect calls. So, if you have a direct call; that is fairly easy to predict with the help of branch standard buffer, will be correct every time, target is constant. For indirect calls the target will vary, depending on the, of your program, because these are essentially function pointers, or virtual matters. The target will depend on what you were really doing. And your b t p will be extremely in accuracy, and remember that for calls procedure calls we really cannot take any help from the predictor. So, the only machinery that you have a predicting call, is the that is b t p. There is nothing tells actually. So, this is a big problem, and this is a big problem essentially, especially your object oriented languages, because they are virtual methods in the your c plus plus or java language. So, programs these languages lot of (( )). So, this has to be addressed. So, pretty much these are the 4 problems that are challenging today. So, let us go through work each of them quickly, some solution at least. It is impossible to give you know what has happened in all these 4 departments as such. So, there is a family of predictors known as gskew predictors. They are trying to address a first problem of reducing B H T aliasing.

(Refer Slide Time: 08:58)



So, here the idea is that we have a number of branch history tables, and these are all g h g predictors. So, I will be a little more general here. So, suppose this is your global history register. And this is my branch p c. I have 3 different hash function h 1 h 2 and h 3, that index into my B H Ts, which means B H T 1. So, what I want is. Finally how do you combine these 3 predictions, any suggestion. That is why I wanted to have all number of translations. So, I take a majority vote and that is my final predictor. So, the question is that, how do I get rid of this particular problem over that. So, there is a requirement on these 3 cache boxes. If there are two addresses that conflict in B H T 1, because of h 1. We should try to design h 2 and h 3 such that they do not conflict, in these 2 B H T. So, even if this B H T for that particular those two addresses, this B H T gives you very poor prediction, this two will actually be correct, and you win in the majority vote, is this clear. So, that is essentially a problem of designing hash functions. So, one hash function that (( )) one family of hash function. So, I will simplify this a little bit. So, I will assume that the input to the hash function is some value x. Let us first fix these sizes, suppose this is some power of 2 B I Ts. So, I will write. So, x is essentially a combination of g h r and p c which is that input to the hash function.

(Refer Slide Time: 11:20)



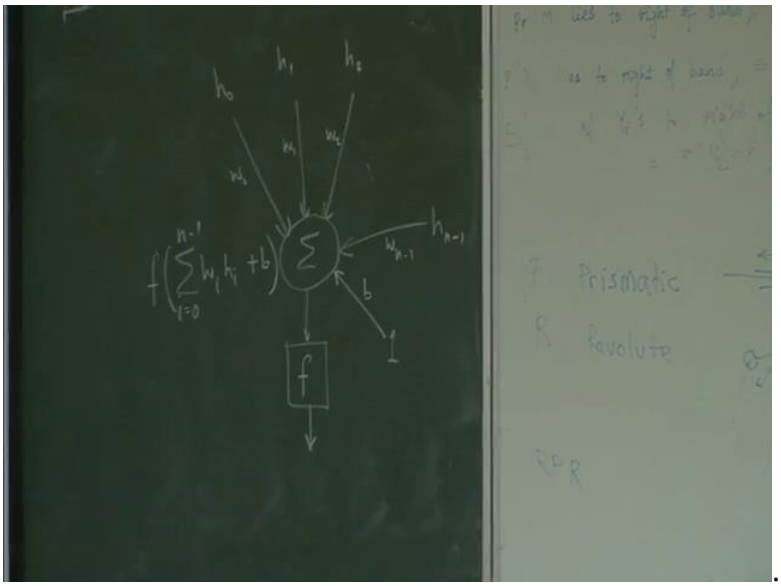
So, I will write x in phase 2 to the n. So, I can write it as this is x naught plus x 1 in 2 length in 2 n etcetera. So, if I do a module over 2 to the power n hashing here. So, if I give, if I just give h 1 x, where h 1 is the modular hash what is the result of that; x naught. Is that clear to everybody, that h 1 x. If h 1 x is just a modular2 to the power n hash functions. Now what I want is, suppose I have x and x point, then both of them have the same x 1, in that case they will alias only to the first table. I want these 2 not to alias from h 2 and h 3. So, what he will often use for h 2, how many of them perfect shuffle. Did you heard of perfect shuffle of a big stream. So, what it means is that if you are given a big stream. There are two types of perfect shuffle. You divide into two parts. So, let us suppose this is n bits. So, you divided in two parts n by 2 and n by 2, and what you do is you do not have bits from here alternatively. So, you just insert them here. So, for example, if you have 1 0 1 1 0 0 0 1. So, you divided in this way. Insert the 0 here, so it becomes 1 0. Then you will have this 0 here, insert the next 0, 1 here next 0 1 1. So, what am I done; this one here, this one here. Its alternatively I take bits from this side and this side, that is the perfect shuffle of this particular stream, got it. So, I could do it in different way also by starting with this particular bit.

So, instead of inserting 0 here, I started 0 first and then inserted 1 and so on correct. So, I can all write in two different ways. The function that I apply your x that in h 2 is h 1 x plus. Sorry you have to addresses that back to the same slot for h 1. This will have actually a very low probability of that happening; this particular hash function, for these

two addresses. So, I will not going to approval from that. So, perfect shuffle is one of the hash functions that have these particular property, that if two addresses are aliasing 1 of the B H Ts, they will not alias in the other 2. So, for h 3 you can choose h 1 x plus sigma square h 1 . So, you can just apply shuffle twice, and they have a nice properties of shuffle, if we keep on shuffling ultimately it will get back the same stream after some time. So, if you have too many hash functions, you cannot do this actually in the (()) . First one about hash function, it is3 hash functions. So, it is the basic things here what I require here. Whenever hash function h 1 which have a conflict, under that condition I will make sure that h 2 and h 3 will not have a conflict, when the majority function will be this 1, the correct prediction the aliasing prediction.

And why is it called gskew predictor, is because actually if you make the tables of different sizes, you get even better prediction accuracy. So, essentially what happens is that, these B H Ts will now have different size of history inputs. Now well the papers are try to look at the relationship between this histories. For example if this history is n bit, how many history should I put in n B H T to n B H T 3 to get better accuracy. So, there are papers which look at geometric history lengths that increase a geometric length, and that transverse to be a best actually. So, anyway. So that is your gskew family of predictors, they are extremely good and have very high accuracy. if you want to read up on that, I can give you some references and email. Now the second problem is the history relationship. So, here essentially we have the same problem I have n bit history here. I have an exponentially large table 2 to the power of n entries. So, how do you get rid of that. So, neural networks have some readable properties, and that actually come very clearly. So, how many of you have heard of perceptron’s work.

(Refer Slide Time: 17:28)

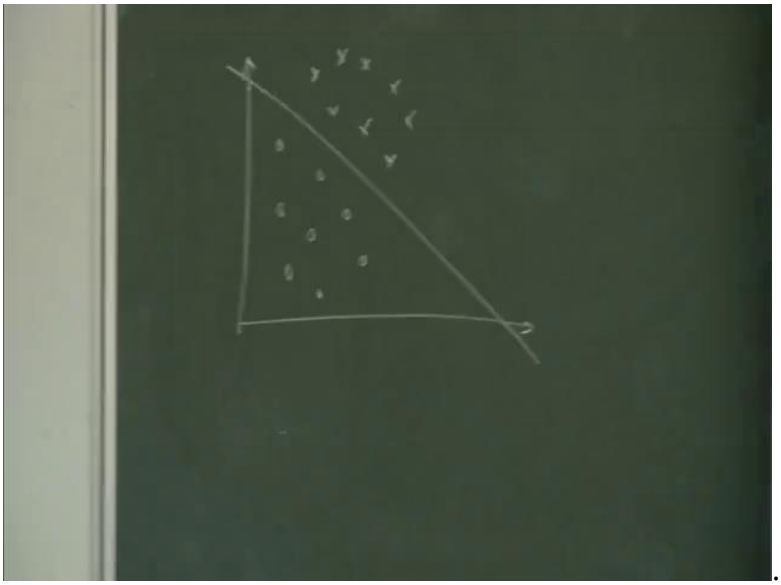


So, it is a simple thing actually, let me think and keep an overview of what perceptron does. So, perceptron is the simple of a neural network. So, which are bunch of inputs. So, for our purpose here, the inputs are history bits, bits of history. And what I do is, I weigh them in some weights, and there is an extra input called a bias. So, what comes out here, this is just a sum of this. And on this we apply a function. So, this one is going to be a real number, from what you get, because of weights are all real numbers, and what you apply a function which gives you a binary output . So, often the function applied here, is the sign function, sign by sign I mean not trigonometric sign function. You take the sign of this value positive or negative. If it is negative, output is 0, if it is positive the output 1. So, what have we got. So, the perceptron is same as f of summation w i h i plus the bias.

So, why is this relevant at all. Can somebody guess? Some say this is my branch predictor now. I give some history, I get a binary output, and is relevant, because a branch predictor essentially a Boolean function, lot of Boolean function. So, if history can be seen as a vector, in the n dimensional space if you want to input the bias, it is actually in plus one dimensional space , this one also .it is a vector . And what we are trying to see is, given this vector you tell me whether this vector belongs to, the taken side of the space or the non-taken side of the space. So, this one here is a plane, it is a hyper plane in the space, this 1. So, essentially what you are trying to do is, you are

trying to learn this particular plane. So, in a two dimensional space, if I have two history bits h naught and h 1.

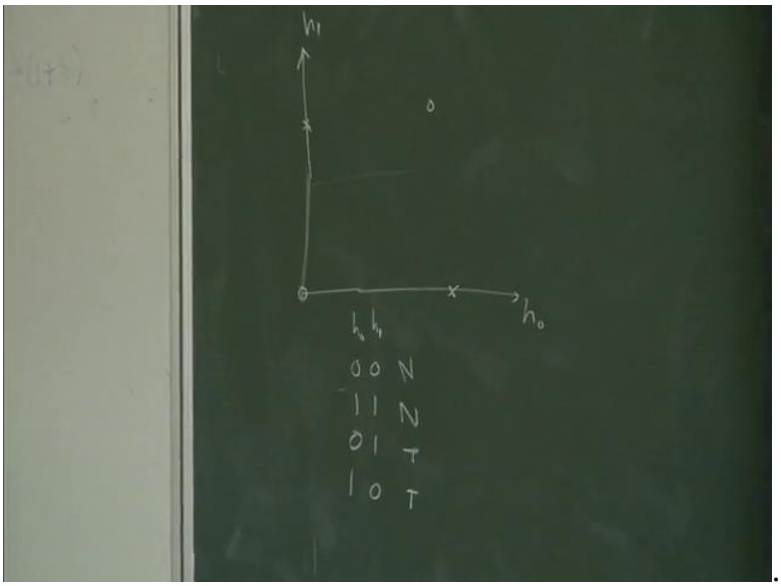
(Refer Slide Time: 20:22)



So, I may have bunch of histories. So, after this history, I always see a 1; that is the tail whenever a branch comes up with this history I see that this branch is tail. So, I mark this history as crosses here, and I may have histories like; these history points whenever they show up the branches actually on tail, and what you are trying to come up with a plane that separates these 2. So, for example, these two this is what this perceptron is trying to learn in the ways. So, if you come up with the ways, so that you take this particular plane, and then when you apply the sign function, it will clearly separate these 2. And if its negative so you belong to this particular part alight. So, your ways will be adjusted. So, there is algorithm to adjust the ways, which I will not discuss here.

So, here what is to be observed is that I do not have a B H T anymore. I can make my history as large as I can, only thing have to remember is the ways. So, now, in storage becomes linear in your history size. If I have an n bit history I need n ways to remember that is it. So, now, of course, a single perceptron cannot predict all branches. For example if I have a pattern like this. So, whenever I see a history of 0 0, the branch is not taken after that, whenever I see a. So, this is h naught and this is h 1, where h naught and h 1 whenever I see a history of 1 1, the branch is not taken. Whenever I see a history of 0 1 the branch is taken. Whenever I see a history of 1 0 the branch is taken.

(Refer Slide Time: 22:02)



There is no plane that will separate this one that anything possible you cannot, whatever you try, you will always mix up something. If you draw this plane two different points are equal to sides, if we try this plane, same problem, this plane, same problem. So, person from whom you will predict this particular branch. You need multiple processor. So, I will not get you that. The point here is that, this problem is actually sound. I can now have large history, and my storage word is linear in a history length. So, longer exponential. There is a predictor called prophet critic predictor. So, these are very general kind of prediction technique, which are often called overwriting predictors. So, let me give you an analogy. Suppose you are in a city, you are navigated in a city in, and when you are driving the car (( )) you, and your trying to get for destination, why did not have been before. So, the, but you just why do not remember clearly how to get that.

So, what happens is that, you are trying of different ways, and whenever you get into a lane if you proceed a little far, your friend tells you that it does not seem like we have been here before, seems very. So, you quickly back track starts some other. You go around that path friend again say h looks new bit, I do not think we have seen before. So, point here is that after a branch prediction is done, we will start going along some part, suppose we get some of those history bits from the long path, or from the correct part through can that improve prediction accuracy. So, you predict a branch you start fetching from the predicted path, very soon you will volume of branch on the predicted path, look

at out from this branch, and then well you take this branch, and you go along the predicted path again, and count another branch.

Suppose I give you this two extra bits to you, after your bit prediction must match, can you now tell me what is the prediction was correct or not. So, it is actually incorporating a little bit of future information. Notice that this is not same as giving in two extra history bits on that size. So, what I am saying is that suppose you have a prediction of n bits of history, and there you get to see too more branch. This is not same as giving you n plus 2 bits of history. I can attach too more history bits there, but it is not. Because here its telling you, that after this prediction have you ever seen these two predictions or not , when you went along the correct path did you see these two prediction to happen in the bus. It has its own; that means, you must have data misprediction here. So, this is essentially overriding predictor where. You make a prediction first, then observe for a while, and then you try to correct that prediction. So, that first predictor is call that prophet, the second one is often the critic of that prediction that was correct or not.

So, second predictor actually predicting the correct of the first. So, this one is very helpful in often predicting data dependent branches correctly , but anyway nonetheless predicting data dependent branches remains a bit problem in. So, it is a huge problem and there is no good solution. So, the gap is huge. The second the last one is how do you handle indirect calls. So, here1 solution that is often used is you use the path history to index into the b t b instead of using the p c 2 index into the b t b. So, essentially what I am saying is, that before we reach the call you must have seen a few branch p cs. So, that defines your path to switch we actually make the call. So, that has a very good correlation with where you are going to going to when you make the calls.

(Refer Slide Time: 27:35)

So, what I am saying is that, suppose you have a counsel f and there is a counsel pointer here, which takes you somewhere. What I am saying that the path that leads to this particular call, determines has a very good correlation with where you are going whether you gone some function x through this pointer also a parts of function y through this pointer determines. So, which path you came. So, instead of using the p c of this branch to index to the b t b, you use a hash of these p c s that you have seen, along the path to this particular call. So, that is that is also seen to prove accuracy.

So, of course, this is very hand, if you want details let me know I will give the papers you can read up on that. So, one problem that I s that cross cards all these prediction result is criticality of branches, and here the main point is that not all branches are equally important . So, you might end up invest to the lot of time try to predict the branch, which is not important. So, that is not really a good. So, predicting some correctly, is critical to performance while others have very performance. So, it is known that all branches are not equally critical; the question is why is that? So, here is the list of factors that often influence your branch criticality. So, one obvious thing is miss prediction.

So, when you miss predict a branch how many cycles you will use before you well on the time path, and every branch does not have equal misprediction penalty, the reason we have already discussed one of the reasons is that if the value is data dependent and the

data is not ready the branch will have to wait longer . So, minimum is the number of byte stages between last direction prediction, and branch execution. So, remember that you can have multiple direction predictors in multiple pipe stages; the last one to make a prediction, is hope to be the best one. From that point all over it through the time you execute the branch, is the misprediction penalty. Certain branches may get delay due to data dependence, and predicting this correctly is important for performance, because here if you mispredict, you are going to lose a large amount on this. So, predicting these branches is very important, if you have a data dependence branches. The second point criticality factor is, cache pollution due to wrong path execution. So, when you are going along the wrong path, you are bringing in data instruction into the data instruction caches. Conflict instruction in data workings is, along the two branch paths will cross this problem.

So, for example, suppose along one path where, certain number of certain instruction in the cache. When you look at the wrong path these instructions will conflict with this instructions in the cache, and may actually evict these instruction all over the cache. So, next time when you come along the correct path, in starting this instruction caches, same way happened to the data actually. And criticality of the correct path that is for example, if the correct path always starts with an instruction on data cache miss. Indirectly you want these two correct, because otherwise what will happen is that, you go along the wrong path, you find out that you made a mistake, you cancel all instructions, you start fetching from the correct path, and the very beginning you take a instruction caches. So, this entire (( )) you are executing and you cannot do anything; actually you do not have the instruction to proceed. The pipeline is empty; we are waiting for the instructions. So, that also adds to a criticality of branches. So, the branch is that often have these things become a predictor, and those of the branches like you definitely want to predict

So, how do you go about doing this. So, critical branches need high prediction accuracy. Identify branches that mispredict frequently is actually easy, it is not very difficult, and the reason is that you can just have a small cache of recently miss predicted branches. You can maintain a cache, at any point I will tell you whichever branches that miss predicted to; however, all of these may not be critical. So, that is what is most important. A branch we have a very high misprediction accuracy, but even if you increase its accuracy by a large amount, it may not effect for all those match, because it is not

critical. So, discovering good features that correlate well with the behavioral critical branches is difficult, and this is a very hot research topic. So, if you want to invest time on branch predictors, you can look it to that. So, here by good features I mean, what program property to tell you that this branch is actually critical. Most of these branches are actually data dependent; that is what it turns out ultimately. And prediction accuracy depends on the entropy of the data they depend on. So, as you know if some pieces of data have high entropy, if you do not they are usually not really predictive. You must know entropy in a data predictor.

Today's best direction predictor has very high prediction accuracy, which means small number of branches, actually cause most of the mispredictions, and these are highly critical branches. The performance gap between such a predictor and the oracle is large. So, that is normal which means even if you have 97 percent prediction accuracy the major of chance (( )) is left, then this is the (()) actually. So, there is a big room for improvement, and this is one of the big problems, but identifying critical branches. So, quick summary of control hazards, redirect fetch from various stages of the pipeline with increasingly better prediction. So, some of the machineries that you have discussed are branch target buffer, return address stack, direction predictors. The fetcher selects the most appropriate next p c every cycle, from among different indication, coming from different stages. Research problems, focus effort on critical data dependent branches, what features correlate well in the behavior of these branches. Can the compiler often help in anyway.

1 of the things that that is very interesting is, can I re-execute branches in a separate fetch. So, today we have a lot of trends and codes in our processors. So, what I can do is that I can run my program on 1 code, and only execute the branches in some other codes. So, you can pre-execute certain number of branches, and actually can know the outcome before it, before this 4 actually gets to that. Of course it is not as easy as said. You have to figure out the instructions that lead to this branch here which we have to execute actually, but there are proposals that run to (( )). Any question before I move on to some other types of hazard. So, you may not have the questions. So, the other type of hazard that is important in the pipelines, is a is data hazard, and this arises because pipelining disturbs a sequential thought process, and data dependencies among the instructions starts. So, here is an example. So, the first instruction adds r 2 and r 3, puts the result in r

1. The rest instruction subtracts r 5 from r 1 and then puts in that r 4 third one adds r 1 are 7 puts in r 6, and this one or r 1 and r 9 puts in r 8 and xors r 1 r eleven put in r 10.

So, here you can see that result of add, is needed by all the instructions. So, in the pipeline what you can. So, this is shown in pipeline actually. So, this is the add pipeline which is going through here, that instruction will execute here, but back to the register file here. Other instructions will read r 1 to this stage . this is the stage where you read a register file. So, as you can see here; this 3 instructions marked in red, will actually re (( )). This exor instruction will read the correct value, because it happens after the value is written back, is this topic clear. So, how do you solve this problem; that I stall these 3 instructions. So, this instruction can be stalled by how many cycles. I can move the decode here. So, it can stall by 3 cycles then everything will be fine. Well after that everything will flow to have here, while solve cycles better solution. Anything else that can be done, why is the value produced r 1 which 5 stage.

X. So, the value is known here. So, can I do something.

Critically pass the value exactly. So, since the parallel is available in the pipeline somewhere, I can pass the value to this guy, I can pass the value to this guy, and then I can pass the value to this guy, and this guy will read the value from the register file. So, how you can avoid in c p i. So, stalling is clearly not acceptable. So, there is something called a phased register file, it solved the 3 cycle apart raw. So, what is that. what it does is that, just like the way we had a got rid of 1 branch bubble, by giving a phase branch execution, here what we can do is you can see that register writeback completes the first half of the cycle , and register read happens only in the second half of the cycle. In that case I can actually say this one. This one will actually get the correct value from the register file in a case. So, this actually solves 3 cycle apart raw dependence. By the way these are called raw dependencies, or read after write hazards you are reading after a write. So, these are 3 cycle apart raw hazard, happening in these instruction, that can be resolved by having a phase register file. You can completely raw register write it first half to cycle, then they lead with the second one , but still does not solve the problem with these two instructions. So, as somebody has suggested, why would you forward the correct value this 3 times.

So, these values produced here. I can directly forward it to the input of the A L U, and that is exactly when it needs. So, what will happen is that, it will need a wrong value in this particular stage, which will be overwritten by a new by the correct value here, before it enters the (()). Similarly, if you remember the pipeline, this value will be carried forward in the pipeline matches. So, here to be available here, and that can forward to this area; the same value r 1. So, read wrong value in I d r f stage, but bypass value overrides it the question is, how you will implement it. So, you always feel bypassed value to the A L U input, the question is how many sources in bypass network, do we need it bypass to memory stage also here I am showing bypass only to the execution stage. So, first let us take each of the questions at a time. So, how do you implement this bypass, let us focus on this bypass. I read a value from the register file which is wrong , if there are two questions now that is before I enter the value to the A L U, how I know that I take a wrong value. And if I can figure out that I get a wrong value, how do I overlap it. What kind of logics that actually I require. What else is that I am in a wrong value. So, if you have forgotten the instruction stream that I will show you. So, we are talking about these two instructions that is what. Here to here bypass yes.

Student: It is important that the (( )) to r 1, and now I am getting from r 1.

So, what kind of what kind of operation am I doing, can you resolve that. I am looking around value.

Comparing the destination which (( )).

Exactly. So, I need to compare both the sources with the destination of this 1. If there is any match I know that a wrong hazard has happened. So, so what does that mean? So, let me take it back to our. So, how does this change. I have two inputs coming into the A L U, and register file produces this value and this value. I have a multiplexer here, which depending on the opcode selects either the immediate of the register value, and here in that the next p c, p c plus 4 on the register file is having. So, how do I modify this, to need in the bypass. So, if I ask you to write a piece of program that describes this logic what would that be. That might be called. That might be easier to think actually in that way. I am writing a c program what should I write it, what will I compare it. How should it change. So, we will have to somehow change the inputs to the A L U, based on some indication that I made a mistake.

What was the (()) in destination register.

That is carried for remember that there is a latch. So, we have an add instruction, which is currently executed in this particular side. The sub instruction is currently here; one cycle behind it, reading the register file, so; that means, what. The add instructions source registers and this destination registers are in this latch. I will store in this latch you can assume that, and the sub instruction is currently being decoded.

Student: We compare the load registers (( )).

Where should I put the comparison which states it…

Student: In the decode stage I think.

In the decode stage, why you wanted to be in the decode stage

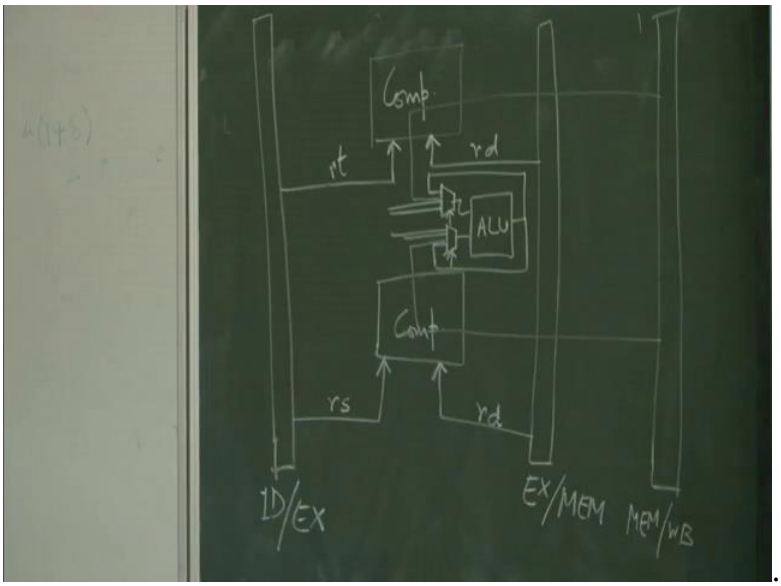
So, the decode stage will tell me whether the address is right or wrong.

And then while doing an execution I will get to know the other two input from, whatever the input of the decode stage is.

The problem with putting into the decode stage is that, you get to know my sub instructions source registers only after this decode. So, we probably get to know it from here. Can I do this execution stage, is that possible.

We can execute (( )), the add instruction would be in this latch. And the destination of this compared to the sources of this. So, I need two comparators. I need to compare the destination of this latch, two sources. We need two comparator's. So, comparison output goes back. So, we have two comparators. And what does it compare. This is my (( )). So, you take the register id for r d, compare to bit r s r d r t.

(Refer Slide Time: 47:12)



So, it tells me a answer; the comparator r. I need to override the inputs to the A L U, this one is wrong, this one could be wrong, both could be wrong, one could be wrong, both could be correct.

Student: can you (( )) alu (( )).

These boxes. So, what you are suggesting is. So, I want one more input there. So, both these boxes are extra input which is output of the A L U very good. So, here is my A L U, now the output goes into two boxes. So, these boxes already have two more inputs. So, the selection logics will be opcode, and combination of this comparator outputs. So, here for example, if the comparator says yes, they actually match, I should pick up the A L U output, instead of getting up the one coming for the register file. The other input will remain unchanged, which will be selected by the opcode. If I am selecting the immediate then opcode it does not matter, whether there is a match or not. In that case how to match. Is it clear. So, this path is known as the bypass path, coming from the output of the A L U to the input of the A L U. This A L U is handling actually. Is it clear?

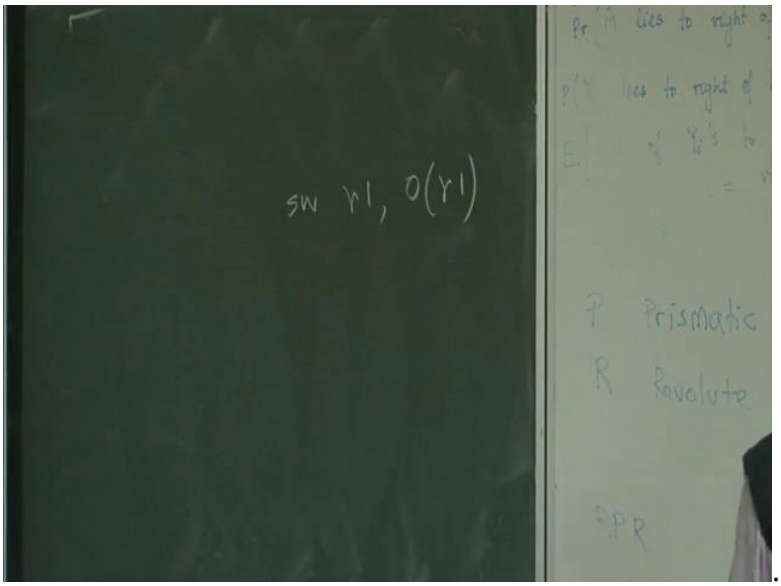
Now, how do I do this bypass. The next instruction, that also has to happen at the input of the A L U, but only thing is that, the input is coming from a different pipe stage. So, when this instruction is executing, this instruction is now in a write back; that means, the value that it is writing to the register is currently in this latch, which I need to bypass. So, what; that means, is you have to add one more input to this particular multiplexer , and

your selection will now be based on r d of this, and r t of this r d of this, and r s of this. We need two more comparators. So, now, there is a small problem, on this example when the problem arises you talk it. So, is this it. So, now how do we answer this particular question. So, we know how to implement that. We have some idea of that. We need bigger boxes in front of the A L Us, and it compare. How many sources in the bypass network? So, in this case I have shown two sources; one coming from coming from ex Mem latch. This is actually ex Mem latch, this is may be taken ex Mem latch, and one coming from a back latch

Other any other sources, in this particular pipeline from I need to bypass. And is it that the destination is always that. Is the destination always ex (( )) other destination, your question make sense or you are lost. So, do I need to bypass from the front stage, could there be a situation bypass to fetch stage something cannot be. I know what we are producing in fetch stage. I produce an instruction, and that cannot be input to some instruction. An instruction input. In the decode stage can produce some internal decoded sequence. They also cannot be.. Here I produce values, may require bypass. Here also I produce values, that may require bypass. Why you need a bypass from here. No are you sure. So, if I want to bypass from here, where do I bypass, what are the possible inputs. If I bypass from here to here, I can bypass from here to here. The next instruction that will come here, will have the decode stage here. You read the pipe here and you can correct value here.

So, I should never need a bypass from here to here, that is not needed. What about this to this and this to this, are they needed. If I want to bypass from here to here; that means, I am by passing the result of this instruction, to this instruction need to write back, that does not make you any sense , why should I do that actually. What about these two things. What does the memory stage do. It needs an address, and needs a value or store instructions. What if this instruction was a store instruction. It stores r 1. Suppose this instruction. So, I will write down the instruction. This r 1 computes the address and store the value r 1 in the values. So, this particular bypass I am talking about this instruction.

(Refer Slide Time: 55:02)



So, this particular bypass takes care of forwarding this particular value to number. The question is, why should I bypass, what will to delay it in this point, there is no reason. So, we are correctly. So, we need just these two sources ex and Mem; Mem will produce bypass two kinds of values; one is values produced by the A L U instructions here, which are bypassed one cycle in it, and values could put load instructions. So, that takes care of this particular question. So, we will come back to this question next time.