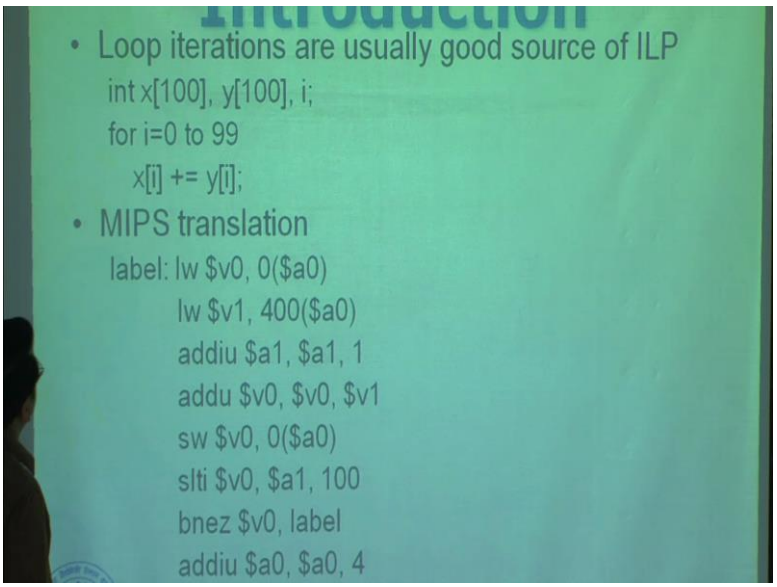
Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 20
Dynamic Scheduling, Speculative Execution

(Refer Slide Time: 00:13)



So, this is a loop, which just adds 2 vectors who is the result in all the vectors, right, and this is the MIPS translation. So, 0 dollar a0, is the base of x and base of y is 400 dollar a0. So, these two instructions loads a values of x i and y i into two respective registers. This is the loop iterator, incremented by 1. And this instruction adds these two, puts results in these two. So, this one could not be put here because, the load delays lot, all right, we have to be delayed by 1 cycle ok.

Then, this one stores a value back to x i, same as this address, and this one compares dollar a1 against 100, if it is already there then, its sets dollar v0 to 1. This one checks if v0 already 1, if not goes back and executes it. And these are branch delays lot, which is always executed, which increments the address by 4 because these are integer arrays, all right to take it to the next element of the array.

So, now the question is that, with our model of, any question on this translation? Is it clear? So, with our model of execution, we are fetching at a certain rate. All right, and the execution model is that, whenever an instruction is ready to execute, I will execute it.

And I also leave the constrain that, in a particular cycle if can only execute one instruction. You can execute unlimited number of instructions in a cycle.

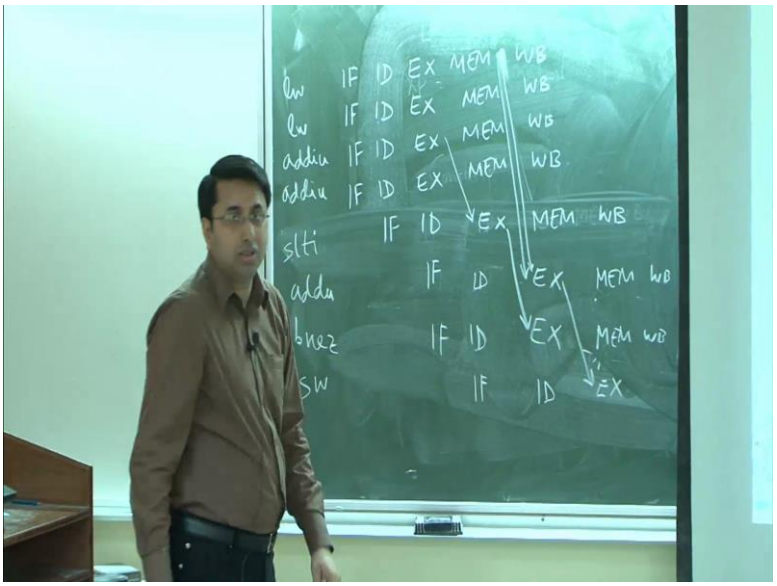
So, the question is, what is the minimum number of cycles required to complete, let say one iteration. So, what will you do in the first cycle? Which instructions you can execute? You can send these two for execution because there are independent. You can also send this one for execution. You can also send this one, but there is the small problem, if you send this one. This one violates, what kind of hazard is this, between this and this, these two instructions? write after read right.

So, which violate a write after read dependence, if we actually, so essentially by executing these, I essentially, what I am doing is, I am reordering this instruction ahead of this one. I am saying that this, this, this, and this. So, this instruction will be put here essentially, all right. And that would violate this particular dependence all right, how can I fix it? Any simple solution? I still want to execute this one in the first cycle.

There is a slight problem because this one is my sort of its connected to the loop iterator. So, how do you really translate then, hope. So, I you are saying that, I change the target to something else. What do I do in the next iteration? I cannot use a0 anymore right. So, my translation breaks. There is a very simple solution what can I do? Can I change this to minus 4 this offset? It will have the same effect right. I increment it first and then, I put it, put a minus 4 here instead of 0, ok right?

So, that will allow me to execute these four instructions in the first cycle. Is it clear? What about the second cycle, what can I execute. So, if you want me to show you the pipeline. So, this is the first load, second load, add i u, add i u.

(Refer Slide Time: 04:06)



So, if I have a five stage pipe, so what will, what I will do in the second cycle? So, my second cycle fetch will be here, what are the instructions that are eligible, that will be ready I know that that will be ready. What about this one, will it be ready? They will be in the delay slot of the loads, they cannot issue. This one? Of course, not because, this one, this one depends on this one, this one. No, this one also depends on this one, this one? No, this one depends on this one. So, I have to put a stall cycle, right there is no other option actually.

So, after one cycle stall, what can I do? So, now, I can probably send this one, why because, this one will pick up the value from the bypass, right. So, this is add u, what else can I send? This one no because, this one actually stores the value that I am computing in this cycle, and everything else, wait, no, sorry.

Let me back track a little bit. No, why could not I send this instruction in the, in the, in this cycle? You saw, you just comparing dollar a1 against 100, and dollar a1 is being computed here. So, as such if I fetch it here, execute it here, it will actually pick up from the bypass. What is the problem then?

Student: Write after read is here because add u is probably.

What is it called, write after? So, there are two hazards that are been violated here. One is write after read, and another one is write after write, right same problem. So, dollar v0

is written here, and write here. And I am actually, if I, if I boost it this instruction before it, I will be violating both the, both the dependencies.

Right exactly yes, this one also has a source dollar v0 yes exactly, but this is somewhat unfortunate, that is very important that is the important observation that you should make. Because as such these two instructions had nothing to do with this one, absolutely nothing. This is the completely independent operation. It is just comparing dollar a1 against 100. It does not use any results from these two instructions. So, how, what is, what could be a solution?

Student: We can use some other register.

Sorry.

Student: We can use some other register.

Use some other register, except v 0 right. So, use some other register except anything used here, so that you will not violate any dependence. So, these are often called name dependences. These are not true dependences, there are actually false dependences. They arise only because, compiler happen to choose this register, for allocating the result of these operation, all right.

So, we will come up with systematic solutions to get rid of name dependences, the point is that name dependences are false dependences, and they should never hamper your ILP all right. So, let us for now assume that this is actually some other register, all right. So, I should be allowed to issue it here right without any problem.

So, this is slti, I do not think I can issue anything else in this cycle. So, next cycle can, right, and can send the branch also right. So, addu and this one will pick up the values from the bypass, and bnez, this one will pick up the value from the bypass. So, what are we left with, but actually I could, all right I ((Refer Time: 09:32)) left with only this one right everything else is done, I believe. 1, 2, 3, 4, 5, 6, 7, 8, yes. So, only left with the store, which will now going to the branch delay slot. All right, is that clear?

So, that is pretty much the best we can do, even if I give you the, freedom of executing unlimited number of instructions every cycle. So, question is how can we go, beyond this? Can we, or is it the theoretical limit? What you think? Given this particular loop

which executes 100 times. We have looked at just one iteration. Can you improve this, any further? So, any question on this, anybody has any question on this particular schedule? Yes somebody was saying something?

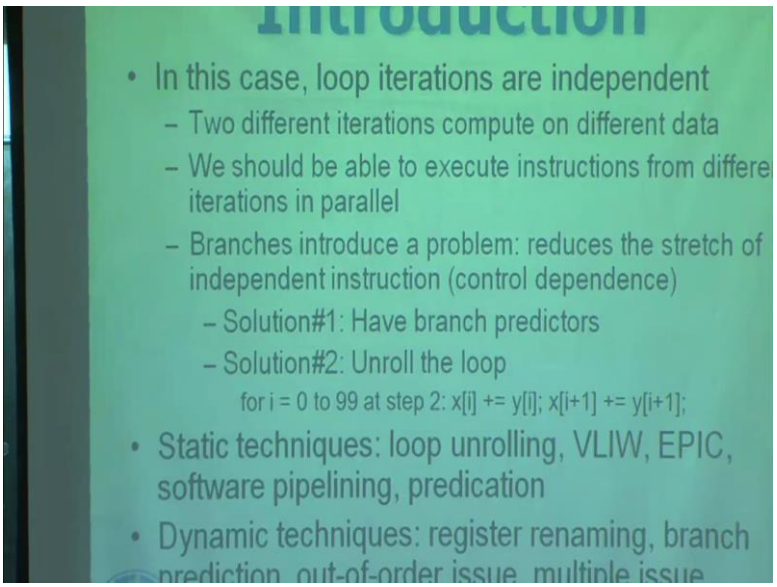
Student: So, we can run the different maintenance like x1 plus equal to y 1 and x2 plus equal to parallel.

And why can you do that?

Student: Sir, because they are not dependent on each other.

Exactly right very good.

(Refer Slide Time: 11:05)



So, loop iterations are independent in this case. Two different iterations compute on different data all right. So, we should be able to execute intrusions from different iterations in parallel. The only question is, how to really systematically do that because the biggest problem is the branches. So, although here you can clearly see that, this branch will be taken 99 times or 100 times, and the last one will be not taken.

The question is how does, how does a hardware figure out at run time, that this is going to be the case. Now, the hardware hits this particular branch and has to figure out, should I go down or should I actually go back and execute this one, right. Sorry, right exactly,

so but predictor will actually help you, but it is not the, it is not really an oracle. It will not give you the correct answer all the time.

So, point is that, branches introduce a problem that one has to recognize. It reduces the stretch of independent instructions. What it introduces is, control dependence, there is a data dependence as such, but all it is doing is it is putting bunch of instructions, that become dependent on a certain branch. Solution one somebody has already proposed that, have branch predictors, but once in a while they will make mistakes, so we have to do corrective measures. The second solution is unroll the loop.

So, what if I change the loop to something like this. You go from 0 to 99 at a step of 2. And within an iteration, you actually do two things. So, instead of having 100 branch instructions, you now have 50 branch instructions right. So, that definitely helps. Is there any down side of bring this. So, does everybody see that it will now help because, now I will have four load instructions, which are completely independent, I have a lot of freedom in selecting my instructions.

Is any down side of this? So, I will let you see this one. So, imagine that this is just doubled, right everything will have now, every instructions will get replicated, every instructions will get replicated.

Size of, size of codes, yes that is one problem. Your size of code will double, if you have if you,, so by the way this is called a loop with an unroll factor of two, unroll twice. If you move in a step of four, and I have 4 such things, the unroll factor will be four and. So, on and. So, forth you could have an arbitrary unroll factor, if the unroll factor does not divide your loop tree count, you will have to have some code at the end of the loop, to fix up that, you know to do, to execute the residual iterations. Yes. So, she has suggested this will definitely inflate your code size.

So, that is a big problem. So, why you actually duplicate this instructions? To be able to use them, to be able to execute them in parallel, they must have different targets right. They cannot use the same target clearly. So, that puts more pressure on your register, all right. So, often you will see that, when you start unrolling more, your false dependences will start showing up more and more because the compiler is actually running out of registers. So, it is trying to reuse same register over and over.

So, that is not problem, and In fact, beyond the certain unroll factor, the compiler will have no register to use. So, essentially what we will do is, it will take register values and store them to memory, and use that register for doing some computation and when that value is needed, it will load it back from memory to some register.

So, that increases your code size even further, and what it worse it will actually increase your number of memory operations. You will have no extra memory operations which are not because of the code, it is because of shortage in registers, all right. But in general if you can choose your unroll factor wisely, it gives you, it exposes more ILP especially, in this type of loops where iterations are independent.

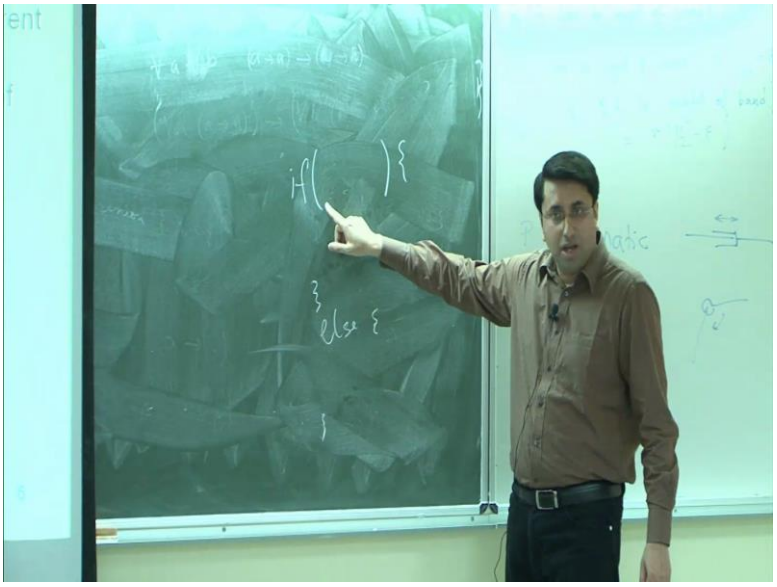
So, essentially so this just one technique. So, loop unrolling that is also called a static technique, which is compiler driven because the compiler will actually do this transformation. However, of course, not all loops are like this, they are not so well behaved when iterations are independent. So, for those you require more sophisticated solutions, and as I told you last time if I get time I will get into this otherwise not.

So, I will just name a few things. So, these, this stands for very long instructions word computers. So, this is again a compiler technique, where compiler analyses your code, and prepares long instruction packets, these are essentially combination of multiple instructions. So, in a VLIW machine, one instruction would be for example, combination of four instructions that will go in parallel, all right. So, that is how the compiler will prepare instruction packets, and that is where the name comes from, these are the very long instruction word computers.

Epic is very similar, it is stands for explicitly parallel, explicitly parallel instruction computers. So, here you do the pretty much same thing, the compiler prepare this explicit parallel instruction packets. The only thing, only difference between this and this is that, in VLIW the packet length is usually constant. That is at the design time you decide that, one instruction packet will have four instructions, one long instruction packet will have four instructions, which essentially means that, if in a, in a particular slot, compiler cannot find four parallel instructions, it will put no ops in the empty slots. To make sure that every slot is four instructions long. All right like for example, here four long VLIW would be happy with this packet, but it will have three no ops in this packet. Will have only slti and three no ops all right.

Similarly, in this packet, it will have two instructions and two no ops and so on and so forth. Epic allows you to terminate instruction packets early. So, epic will have a variable size instruction packets, all right. Software pipelining and predication these are again two techniques, to expose more ILP, predication is very much related to branches. This is not predictions this is slightly difference this predication essentially the idea is that, you convert control dependence to data dependence. So, the idea is that you evaluate this expression, put it into a predicate register, and you tag all these instructions here…

(Refer Slide Time: 18:05)



with predicate register value true, and tag all these with predicate register value false. So, now, essentially what you are saying is that, there is no control dependence. All these instruction depend on this predicate register. Similarly, all these instructions depend on this predicate register. And this instructions will finally, survive if the predicate register value turns out to be true, and this instruction will survive if the predicate register value turns out to be false.

So, we are converting a control dependence to a data dependence. So, it will essentially look like exactly same, predicate registers to be bypassed to your, you know required instructions and so on and so forth. The only thing is that, the extra thing that will require is will execute all these instructions, some of them will get cancel later, when the predicate register value becomes available.

So, again as I told you, if I have time, I will get into this, but this really belongs to a compiler course. So, this is not really priority here. So, what is the dynamic techniques? So, that is these are essentially hardware techniques. So, as I have already hinted, one important thing is register renaming, where you try to remove name dependences that we have already seen. Branch prediction, we have talked about it that length, so i will not get into this anymore. Out of order issue we will talk about that.

So, I have already shown you some flavor of these here, what is happening is that I am reordering instructions. Multiple issue, that is also shown there, I am issuing multiple instructions every cycle, and some more advanced techniques for exposing more ILP. So, these going to be your add in socially. This is what we are going to focus on, and you might ask well, will I ever get an effect of loop unrolling in this, in this particular loop?

The answer is yes, can somebody see how is that possible with dynamic techniques? That the amount of ILP that is exposed here, will also get exposed in dynamic techniques. How is that? But the code that the, that the machine will see is exactly this.

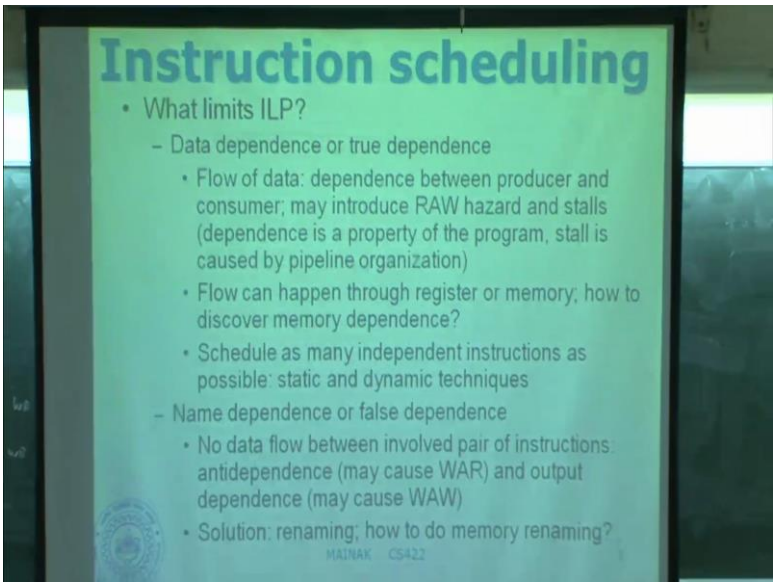
What will happen exactly? So, I keep on fetching right. I when I hit this branch, I ask the predictor. So, in this case the predictor will fairly accurate. We will say that go back. So, you continue fetching. So, now, essentially what will happen is that, at any point in time your processor, the instruction pool is going to have a large number of iterations already fetched. So, what matters now is that, the hardware has to pick up the independent instructions. So, automatically it will pick up these two loads from one iteration, and some other two loads from the other iteration, and execute them in parallel.

So, it is going to have the same effect, but there is one condition here that is, the predictor has to be accurate. The predictor tells you something wrong, your instruction pool will get propagate with useless instructions. And after that whatever you do is does not matter, you are doing essentially wrong things.

So, eventually your branch will resolve and you will get to know that your predictor sets something wrong, you have to cancel everything, flash everything from instruction pool and start over again, all right. So, you are going to get the same effect pretty much, depending on how good your predictor is. The compiler removes that particular condition by analyzing the code, and telling you that you know this code can be unrolled, and you can get good ILP from that.

So, to summarize what limits ILP just whatever we have discussed here, one is data dependence or true dependence, that was the reason why we could not execute more than one in this particular cycle because, whatever we had here by dependent on something that were executing all right.

(Refer Slide Time: 22:00)



So, this is often the primary concern because these are the dependences that you would have to obey because, these are needed for correctness. You cannot really valid any of these dependences. So, this is essentially this concerns flow of data dependence between producer instruction, and a consumer instruction, and it may introduce read after write hazard and stalls. So, I say that it may because the reason is that dependence is the property of the program, stall is calls by pipeline organization.

How many cycle you will stall will depend on for example, how deep my pipe is all right? Certain dependence is may not cause any stall, simply, because the distance between them is large enough. So, that I can actually issue the consumer without any stall at all right. So, I mean you can see that here. So, I could, I could separate these particular instruction from this one. So, that there was no stall, so adjust like that.

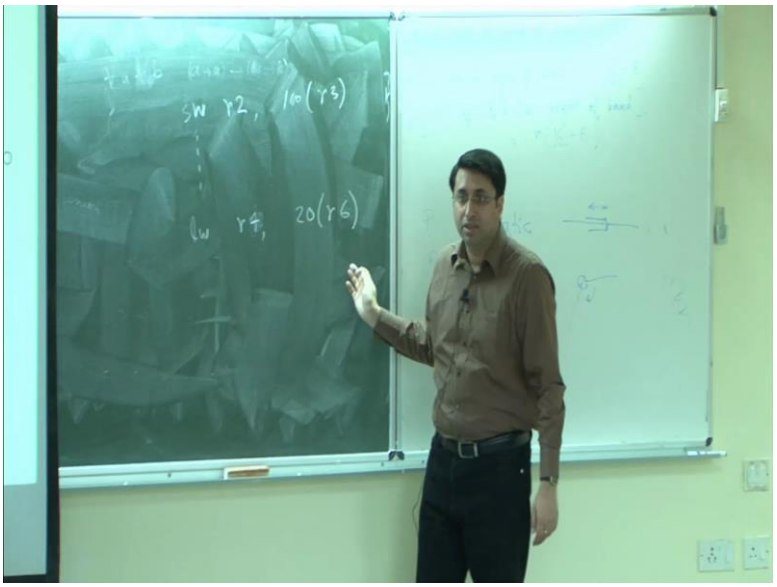
So, is this point clear to everybody, that dependence is may not always cause a stall that is very important to understand actually. And flow can happen through register or memory. So, in this particular example, we have, we are only seeing flow through registers right for example, the value flows from this instruction to this instruction

through dollar v0. Value flows from this instruction to this instruction through dollar v1 and so on and so forth. But you can have data flow through memory also. How is that, can somebody give an example?

student- To store a register in the memory and then…

Exactly. So, you store something to memory, and later read that value from memory through a load instructions. So, that establishes a memory dependence, a dependence through memory. So, the question is, how to discover memory dependence? So, let me show you why this is not as easy as register dependence. So, how do we discover register dependence? Exactly, so you just compare right, the registers, the source of one instruction and target of another instruction. If they match we know that there is a dependence.

(Refer Slide Time: 24:56)



Here this is not so easy, so let us take a look at two instructions that are in the dependence. So, let us say, as we has mentioned we have a store, which lets say stores register r2 to some address 100 r3, all right. And then there are bunch of instructions and I will load which loads from this address into this, turns out that these two are actually same address, it can happen right.

The problem is by looking at these two instructions, there is no way to know that actually, they are in a dependence. Is that clear to everybody? It is just that, the values of

r3 and r6 are such that, these two addresses turn out to be same. And there is nothing that stops a compiler from generating this code. In fact, compiler does generate this code many time. In fact, given the availability of registers when it is generating this particular load instruction. It may not have, may not use r3 again because, r3 may be may not be allocating to some other variable by then.

So, question is how do you really discover them? That two instructions are in dependence because I may make a mistake, I may think that well I am probably, this probably safe to send these two instructions together or even worse, I can execute this instruction before this store even, which may end up this load getting a wrong value. So, how do I discover this? What is the simplest solution? Reorder buffer. Reorder what is that? Reorder buffer. What is that?

No I am not, no let us not get there. You may be knowing what the reorder buffer is, but I am not really asking for a solution. What I am asking is give me new way of figuring out that these are dependent. What you are saying is that well, I can actually go ahead and execute the load before the store, but eventually, I will be able to figure out, that is what you say well. So, we will get to that solution, but I am saying how does normally, how do you really. So, even your case right suppose, let us take that case. You said that the load I have, I have executed this load already right. So, when the store executes, I figure out that there is a problem. How do I figure this out?

student- If both are and then just see if both of them executed the final value gets.

Final value?

student- The final address.

Ok.

student- The one that.

So, the point is that, before this instruction can go and access memory, the address will have to be computed right. So, when that is done, I am ready to figure out whether it is a, it is in a memory dependence or not. The point is that I have to wait till then whereas, in case of register dependence, even in the decode time I know, which are the instructions

that are depend on each other, but here I have to wait until the address gets computed, then only I will get to know whether this instruction depend on this one all right.

So, that is makes things a lot complicated because, which means, my instruction pool which is, which is having instructions which are already decoded right. So, my job was just to go through this pool and pickup independent instruction for execution. And it seems that is not sufficient anymore, because of this memory dependence. Because, from the decoding instruction I cannot figure it out anymore. I have to actually partially execute this instructions, to compute the addresses and then only I will get know.

So, that poses a lot of problem that clearly hinders your ILP, because essentially, it says that if you do not have any other machinery, you have to be conservative. Whenever you find a load, you say that well, I cannot really execute it, until all stores before it have been executed right. So, that is correct, that will guaranty correctness, but that will make sure that, you will not get as much ILP as you would have got, if you could actually execute this load earlier.

So, we will come up with more sophisticated solution, which can actually get rid of this problem, but is this problem clear to everybody? This particular one, memory dependence problem? Any question on this, is it clear? So, how do you really go around data in true, data true dependence well, you try to schedule as many independent instructions as possible, and that is what we have been trying to do there and we will see dynamic techniques to do that.

The second type of limiter is name dependence, these are also called false dependence as we said because there is no data flow between involved pair of instructions. So, there are two types of false dependence, one is called anti-dependence which may cause write after read hazard. So, this is just opposite of flow that is why it is called anti-dependence. So, you can look at these two actually. So, these two instructions are in an anti-dependence.

So, this one is reading from dollar v0 it is writing to dollar v0. There is no as such data dependence between them. It is a totally false dependence happened only because compiler ended up choosing dollar v0 as the target of these instructions all right. And these are anti-dependence because, there is a read before a write, just an opposite of the

flow dependence. Does anybody see any other anti-dependence in this code? There are more, where are the instructions in anti-dependence? Louder.

Student- Fourth and sixth.

Fourth and sixth, fourth and sixth, anything else? First and last. So, dollar a0 written here read here. So, the other type of false dependence is called output dependence, which may cause write after write hazard. So, these are essentially dependences when you are trying to write the same register for example, these two instructions are in output dependence all right. And this again happens unfortunately because the compiler chose v0 as the target of these instructions all right.

So, dynamic techniques are there to get rid of this as he as suggested, just rename them. Use some other register right. So, there has to be systematic way of renaming. So, that because you can see that if I change this one, this register to something else, my iteration flow will break immediately, because next iteration will have a problem then.

So, we have to be very systematic in renaming these registers. So, we will talk about algorithms to do that. So, solution for both is renaming, and again you have to answer this question that is, you may have false dependence in involving memory operations also, how is that possible? Can I have an anti-dependence involving to memory instructions, I can have right. I can switch these two pairs, and they will be in an anti-dependence. Or I can introduce another store, and if we turns out that these two addresses are same, these two instructions are in output dependence right. So, we have to also do memory renaming, to get rid of these dependence instructions.

So, we will answer all these questions. Any question? And the third thing that limits ILP is control dependence, which we have already discussed in great detail simply because data flow alone is not sufficient for program correctness, control flow must also be preserved. Because that decides along which path you execute, and the solution to that is branch prediction, which we have discussed, and what if you go wrong we will discuss that, we have not discuss that in much detail, we will have to be, we will have to have systematic ways of recovery from mispredictions all right.

Because now, what will happen is that, in this case although, in this particular case it may not show up, but when you have two iterations club together, what may happen is

that, a large number of instructions might have been issued on the wrong path. When the branch finally resolves, and this will especially happen if you have a very long pipe.

So, there has to be a systematic way of removing this wrong instructions from the pipeline, and fixing certain other things which are related for example, you will now start renaming registers and all, that will have to be also fixed on the wrong path which whatever, you have renamed. So, we will see all these problem very soon. So, this one is not easy, that is the whole point. It is not as easy as we have seen in the in a, in a single instruction fetch execute linear pipe. Is much more involved and it gets complicated, because of various types of renaming that we will do.

So, any question? So, essentially, our goal would be to,, so ultimately you know I mean,, so the ultimate limiter, is essentially this one. Assuming that you have an oracle to decide your control dependence, and this is false. You have unlimited number of registers to get rid of all name dependence. So, then the ultimate limiter is going to be this one only, which will actually limit your ILP. Everything else will go away, and that is what the goal of a processor is, that you know I designer would like to live only with this, and get rid of everything else.

So, the goal is to design a good branch predictor, and the goal is to design good mechanisms to do renaming all right. So, we will focus on these two things, branch prediction is already talked about. So, I would not go to get into that. Questions?

Student- Yes, store dependencies.

Sure.

Student: So, the limiter of this actually. So, stalls who are also going to limit the performance.

Yes. So, that is the limiter and then the source is flow dependence right.

Student: Just improving on the branch predictor and the, is not sufficient if we have.

No that is what I am saying. So, if you remove all these suppose, you have an oracle predictor here, which is 100 percent accurate, and you have enough registers to have no false dependence. Then what are you left with, flow dependence right. And all stalls in

your pipeline will come from only this nothing else. So, that is what you are saying probably. So, that is that is what I am also saying that, this is the final limiter which you have to leave with because, this actually guaranties correctness of your program.

Student- Sir, but we can you can improve error by reducing the number of stalls.

How?

Student- We have a, I mean the pipeline.

By making the pipeline shallower you are saying.

Student: Yes.

Yes of course, sure you can do that yes, but there are other trade options over there, as you make the pipe shallower, you may lose frequency. So, that will have to be looked into more carefully. So, what I am saying is that ultimately, this is what will limit your ILP. So, any question on this? So, of course, we will, we will also talk about techniques. So, this is often called the data flow limit of a processor.

So, we will talk about techniques which actually try to go beyond that limit. Can anybody guess what it might be? So, I tell you that I have an oracle branch predictor, 100 percent accurate, I have enough registers. So, that I have no false dependence. So, what else can I do to go beyond the data flow limit? So, data flow limit is essentially this, the reason why I could not issue this instruction in this cycle, this cycle. That was because of a data flow limit, because the value would not be available on time, which is why I have to delay this instruction by one cycle.

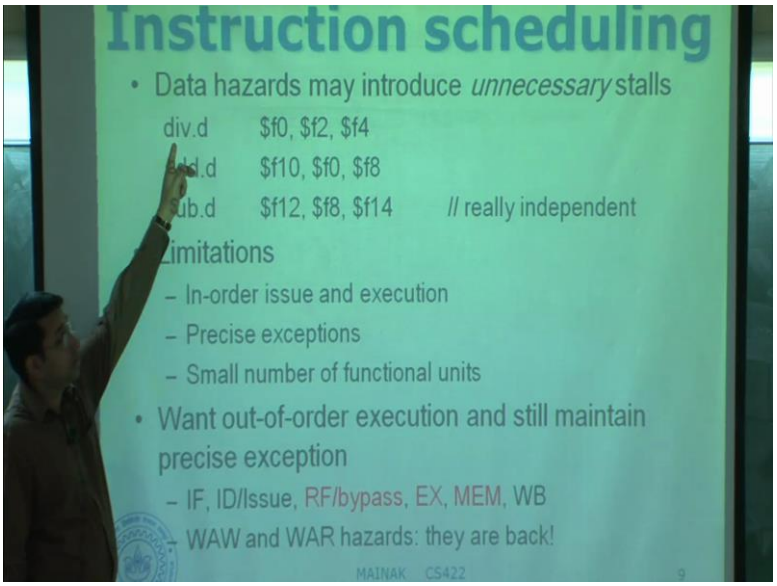
Suppose, I issue this instruction here, I fetch it here actually. So, by the time it reaches here the value is not yet ready, what can I do? Come on, you should be able to answer this question now. What make sense, other than stalling of course, predict the value exactly, why cannot I predict the value, and it is going to produce for memory. So, we will talk about that, a little bit though because it is a much harder problem than branch prediction. Yes.

Yes.

Yes. Exactly yes. So, accuracy is going to be low, but turns out that, there are many programs which load constant values. So, they are predictable, some values are predictable, some values are not, and turns out that one of the most loaded constant is 0.

So, anyway, so that comes from program analysis. We will talk about things that you know how does this, such a predictor look like, and what you can do when you make a misprediction? You have to have a fixing mechanism there also just like a branch predictor. So, that will allow you to actually go beyond this limit, this data flow limit. You can start predicting values correctly, and you can even execute instruction dependence before the producer is done. So, that is possible.

(Refer Slide Time: 38:48)



So, just to quick start the process, with the simple example. So, here there is a code snippet, which shows that data hazards may introduce unnecessary stalls. So, they listed to understand that first. So, I am, here I am talking about our traditional pipeline, which can fetch one instruction every cycle, and it will take the instruction to the pipeline.

So, whatever we have seen, so far, not like this that I can issue multiple things together. So, what is doing here? So, the first instruction is the division, a double precision division operation, which operates on f2 and f4 and produces value in f0, second one uses f0 produces value in f10, and this operation is using f8 and f14, and produces value in f12, which is independent of both of these right.

So, now whatever we have seen, what will happen, this instruction will have to wait until the value of, until the division operation completes and produces f0. And since we have a constrain that, we have to go in order one after another, this instruction will have to wait. It cannot be fetch actually, or even if it is fetched, it has to wait somewhere in the pipeline, simply because this instruction cannot execute at this point.

So, this in order issue an execution, and precise exception as you have talked about, actually this allows this instruction to overtake this one because if it overtakes this one then have many dangerous. One thing is that, this instruction may later raise an exception, some type of arithmetic exception by when this instruction is already completed then, you have to find out, we have already talked about solutions, how to fix those things.

So, I want out of order execution, and still maintain precise exception. So, what these means is that. So, this is the, this is a very generic pipeline that will find today. The front end of the pipeline, that is the fetch, decode and possibly issue are in order. So, they will fetch instructions in order, decode them in order, and issue them in order, by issuing I mean they will, they will put the instructions in some queue all right. And then, this particular middle part of the pipeline is going to be out of order completely.

So, they will pick up instruction from this issue queues which are ready, as an when they will become ready, they will be sends through register file to read the values, in the register values pick up the value from bypass, execute, lookup memory, and they will put the results somewhere. They will remember the result somewhere, let us for now assume that they will remember the result in the issue queue itself, the issue queue slot is maintained for this instruction, and the value will come back here, it will not yet right to the register file.

So, finally, what will happen is that, this issue queue will be drained in a FIFO manner one after another. The values will be transfer to the register file, and that is why they execute this particular phase. So, what I am doing is that, I am now decoupling the pipeline of an instruction into three phases. This is one phase, which is completely in order. This phase is totally out of order, we do not maintain any order at all here. And this phase happens sometime much later, when the instruction turn comes to complete.

So, this make sure that, you have no problem with precise exception at least, you can catch exceptions in the order they go, and you can cancel instructions which are after the excepting instruction. But you still have to deal with these two types of hazards, but does everybody see that or have I already solved them in some way?

So, I said that every instruction. So, model is very simple, I fetch an instruction, I decode the instruction and I allocate an issue queue entry for that instruction all right. And these entries maintained until the instruction is completely done, by done I mean has written the register value back to the register file. And from this issue queue, I pick ready instructions as an when it becomes ready that may not be in the FIFO order, they loop up the register file, they execute, they look up memory, and the value is stored back to the in instructions issue queue slot.

And finally, when an instruction is done and is at the head of the issue queue, it will move the value from the issue queue slot to the register file entry, and the issue queue entry will freed at that point. So, I tell you that I have not only solved the precise exception problem, I have actually solve these two also. Why is that? I am somewhere doing a renaming actually. What is the name space of these instructions?

Now, the name space of the target of this instruction. In the decode phase the name space was the register name space of course, whatever the register I d's are. Where do I store the value after the instruction has done executing? Issue queue entry, so that becomes a new name space actually.

So, two instructions with the same target register will now get to different issue queue slots, and they will store the values actually in those two slots. So, these two slots now serve us to different names for the same register. How do I guaranty true dependence in this particular model, but the dependence instruction should get the correct values also, how do I make sure that happens? Who goes in order, execution is not in order.

Yes, but let us go back to this example right. So, here this instruction right, so let us give them slots, you should give slots. So, let us suppose these are the slot I d's. Actually the slot I d's will be in this order I am sorry yes because the issue queue is assigned in order. So, let us see what are the slot I d's, so 0, 1, 2, 3, 4, 5, 6, 7. So, these are my issue queue slots taken by these instructions.

So, these load instruction will complete and put the value in slot 0 all right. This instruction will complete and put the value in slot two. The question is this instruction, so the slot five instruction must get the value from slot two how do I guaranty that this happens?

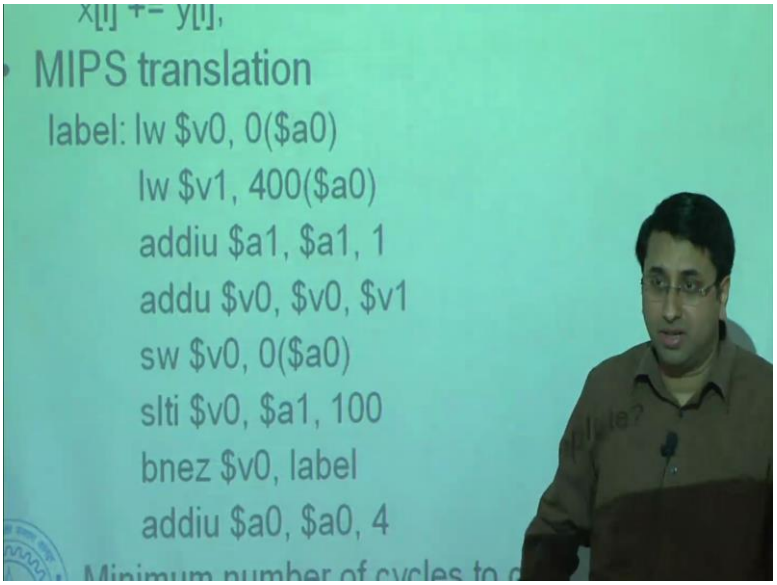
Student- Sir, the value in the queue, so you just fetch them in the last value is the greatest from the slot.

So, that means, when instruction 5 executes or issues you just look up the entire queue.

Student- Sir, look up the end of the queue.

Why should it be in at the end of the queue?

(Refer Slide Time: 46:40)



I am really looking for dollar a1, the latest dollar a1, that is what I am looking for actually right, which may not be at the end. So, I have to search through until I get the latest dollar a1. So, that is how I maintained true dependence in this particular model and the other two dependences are already gone because I have renamed them already.

So, for example, here you can look at we had problem with these two instructions right, we had anti dependence here. So, 4 so slot 4 here will pick up the value from slot 3 right, and this instruction here has already renamed dollar v0 to slot 5. So, that has decoupled

dollar v0 from this dollar v0, this dollar v0 really corresponds to slot 5 now, this is not dollar v0 anymore all right.

Similarly, if you look at this one here this is now really slot 7 has nothing to do with this dollar a0 totally different all right. So, finally, of course, your queue will be drained in order. So, this instruction will first go and write to dollar v0, this one will write to dollar v1, this one will write to dollar a1 and so on so forth. So, when writing back, in fact, even you can even nullify some of the rights for example, you can say that well by the time if dollar v0 is already consumed and overwritten, I may not even write it back it actually, it is enough to write back this one because this is the final dollar v 0 that is have ((Refer Time: 48:24)) all right.

But anyway, so those are some optimizations which are not really important, but is the model clear to everybody, how I resolve name dependence? and this one will also resolve memory dependence same way because the memory operations, two memory operations here we will get two different issue slots. Even if they have two different addresses, it does not really matter finally, they will write to the memory in order, but they can execute out of order without any problem. So, we will continue form here next time, we will try to concretize this model more.