Advanced Computer Architecture
Dr. John Jose, Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati, Assam

Lecture 8
Dynamic Scheduling to Explore Instruction Level Parallelism

Hello, hi everybody, today in lecture 8, our discussion is on dynamic scheduling to explore instruction level parallelism. In the last lecture, that is the seventh lecture, we have seen how compiler can help the architecture in scheduling the instructions such that the number of stalls that is happening at the time of execution due to structural hazard, data hazard and control hazard can be minimized.

What we saw there was the compiler is playing a proactive role in trying to understand what are the instructions that are dependent and if needed, you how to keep those dependent instructions sufficiently apart. And this was done by finding out some other instruction down the stream and picking up and then keeping in between 2 dependent instructions. In this way, any kind of a stall that can naturally happen if the instructions are normally scheduled can be reduced or eliminated by an effective involvement of a compiler.

So for a compiler scheduling to work the internal architecture of the hardware need to be informed to the compiler. So, compiler writers should basically know this is the hardware in which this is going to work. And these are the dependencies that can happen this many cycles is the number of stall that will happen if these 2 instructions come. So, a lot of hardware information is needed for compiler to effectively schedule a code.

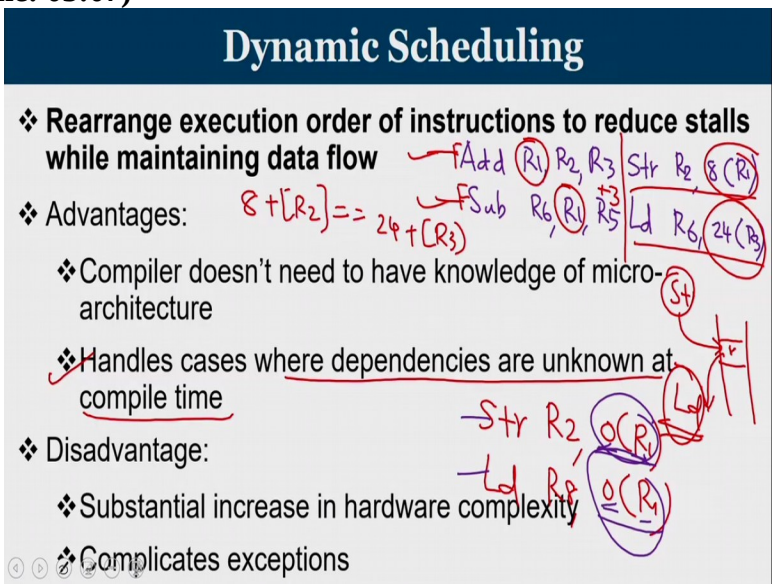
Since, generally the kind of companies who manufacture processors or hardware and companies who develop compilers are not the same. It means for an effective compiler scheduling we need a closer interaction between hardware manufacturing companies and compiler writing companies. In all the cases it may not be possible to have such a kind of a closer interaction.

So, what is the solution when a hardware company is not willing to share its internal architecture details to the compiler such that compiler can re arrange the code that is not possible then hardware has to improve and the techniques that we use in hardware should be

sophisticated such that I can reduce the number of stalls that can happen due to pipeline execution of instructions.

So, today's lecture with this keep these things in background, todays lecture is going to be on, how can dynamic scheduling be possible or how can you facilitate the hardware to exploit instruction level parallelism more.

(Refer Slide Time: 03:07)



So, the whole idea of dynamic scheduling is to rearrange execution order of instructions to reduce stalls while maintaining the data flow. So, let us try to understand in the case of a compiler scheduling, is also known as static scheduling versus hardware scheduling or also known as dynamic scheduling. So, in this case compiler is going to give you some sequence, it may be these are the set of the instructions given by the program and compiler is going to reorganize these instructions and this is what is coming to hardware.

So, hardware is going to run instructions only in the same order what compiler has given, that is what is called compiler scheduling. So, eventually the order of instructions is changed by the compiler, but once it comes to the hardware, hardware is strictly going to follow the order given by compiler. Now, whereas, in the case of hardware scheduling, these are the order of instructions, what is given by the programmer. Now, when it comes to hardware, hardware is going to reorder the instructions in different format. There is an idea.

So, rearrange the execution an order of instructions to reduce stalls while maintaining data flow. So, this rearrangement of instructions before giving for execution is what is there in compiler scheduling or static scheduling rearrangement of execution order of instruction to reduce stalls is basically what is called dynamic scheduling. Now the advantage of going for these kinds of approach is.

Here compiler does not need to have knowledge of micro architecture, the hardware manufacturing company need not share its internal details to a compiler development company. So, compiler does not need to know all about this knowledge. And there are certain dependencies which we cannot find out at compile time, we will come to know such kind of dependency only at runtime.

I would like to draw your attention to one such dependency where we cannot know whether there exists a dependency or not. So, consider the case that you are going to have an Add instruction on R 1, R 2 and R 3. Now I am going to perform a subtraction instruction on R 6 with R 1 and R 3, R 5. Now this is actually you are going to have a dependency because the name R 1 you can see that it comes us the resultant or destination register for ADD instruction and the same R 1 come as the source operand for the subtraction instruction.

So these kind of dependency while the compiler generates the instruction, these kind of similarity in names of the operands can be easily understood. And if at all, you require any such kind of operands. So imagine that use is going to be a floating point add and floating point subtraction. Even with the operand forwarding and FAdd and FSub cannot happen in adjacent cycles, I have to wait for a minimum of 3 cycles. These data dependencies are easy to resolve at compile time.

Now, here the point what they are telling what are the advantages of dynamic scheduling is sometimes when the dependencies are unknown at compile time. Let me to draw your attention to one such case where dependency is not known as the runtime. So consider the case that you are going to perform a store operation

Store R2 ,8 (R1)

and then I am going to have a load operation

Load R 6, 24(R3).

So, look at this case, there is a store operation and immediately after the store operation, we have a load operation, now the store or load dependent. What do you mean by dependency of a load and store. So think of a case this is memory location. Now, my store instruction is going to write into this memory and my load instruction is going to read from the memory location.

So, if this load and store are dependent, then the store instruction will write into a location and the same location is read by the load. So in this case, I cannot exchange the order of the load and store, because they have dependent the loaded value is same as what is stored, meaning the load instruction can happen only after the store instruction is over. So when can I say that they both are same, they both are same.

It is very obvious in this case, let us say there is a store instruction of R2 that says 0(R1). No, I am telling there is an load instruction on R8, 0(R1). So in this case, the store is going to write an instruction, whose memory address is given by 0 + R1 and the load is also going to read from a location whose memory addresses 0 + R1. In this case, right from the name of the base register and the displacement looking at this similarity, it is clear that they both are pointing into the same location.

So, this is a clear case of dependency, which even at compiler can understands because the name of the base register R 1 is same and the offset is also same. This is actually dependency. Though the challenge is, this a dependency. This will become these 2 instructions are going to be dependent if 8 + content of R 2 is equal to 24 + content of R 3. That is the dependency, these can be seen.

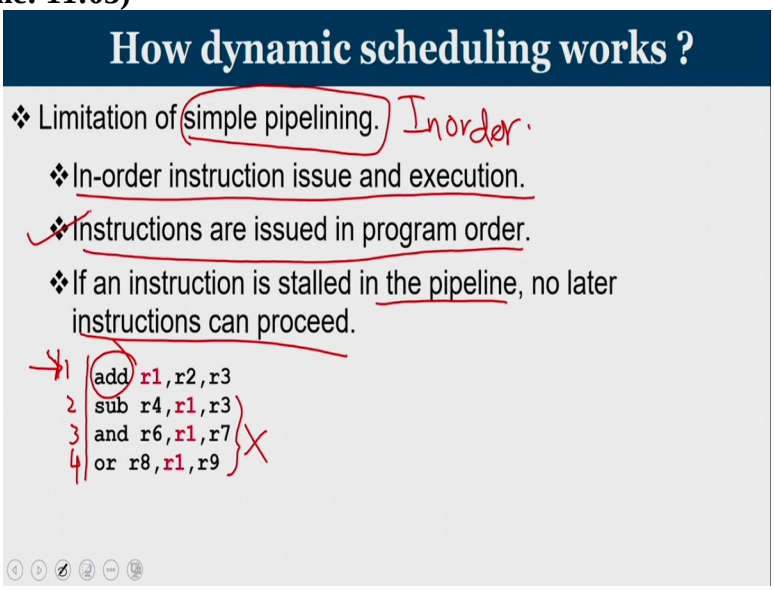
We will come to know only if you add up the value of R 1 with 8 and R 3 + 24. So, by simply looking at the instructions and the operands, we do not find there is a dependency. These dependencies we will come to know only after a computation of effective address. This is the case what is mentioned here handling cases where dependencies are unknown at compile time this can be resolved only at runtime.

So compiler cannot reorganize. So, if compiler try to reorganize them seeing that there are no dependency between them, then it is going to create troubles. Now, what are the disadvantage of hardware scheduling, there is substantial increase in the hardware complexity that is one

thing, the hardware is going to have more complexity because hardware need to understand hazards and hardware need to have mechanisms by which it will try to reorder the instruction. And then whenever there is an exception that is happening that is also going to complicate this scenario. So, why it is dynamic scheduling. Dynamic scheduling is basically the approach in which given a set of instructions hardware will understand, what are the dependencies, what are the instructions that can go, what are the instructions that can be reordered and based upon that execution of instructions is been reordered.

Based on the requirement which a target of trying to reduce the number of stalls, trying to explore instruction level parallelism. The advantage is with respect to combine them scheduling or the limitations of compile time scheduling is runtime dependencies are not known at the compile time scheduling which will be knowing at runtime. And the second aspect is the micro architectural feature of the hardware need not be exchanged with the compiler. Need not be informed with the compiler, but naturally the hardware is going to become more complex and it will take more power or more area in the chip once you have a more intelligent hardware that can understand hazards.

(Refer Slide Time: 11:03)



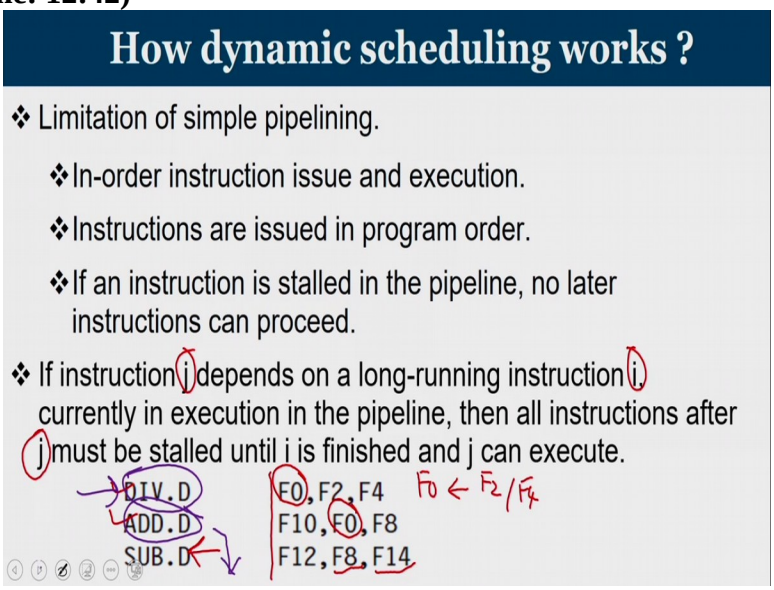
Moving further let us try to understand how dynamic scheduling works. Now whatever pipeline we have learned so far is called a simple pipeline. So, what do you mean by a simple pipeline, consider the case that you have instructions like this and all these instruct let us I am going to number the instruction 1, 2, 3 and 4. So, these instructions are going to be executed in-order, that is called in-order instruction issue and execution.

That is happening in the case of a simple in-order pipeline that is known as an in-order pipeline. Now, in the case of an in-order pipeline instructions are fetched in-order, decoded in-order, executed in-order. That is the way how it has been done. Now in multi cycle pipeline, there can be cases of instructions completing out of order, but still the issue happens in-order. Instructions are issued, issued means giving the operands of the instructions to the functional units they are issued in the program order.

Now, if by chance you just imagine that your ADD instruction cannot proceed let us say the adder is not available. So, if an instruction is stalled in the pipeline, whatever be the reason, let it be a structural hazard, then no later instruction can proceed. This will actually prevent all the subsequent instructions from getting issued. So, in a conventional in-order pipeline, all instructions are issued in-order, issue is in the program order, if at all any instruction is stalled.

Due to a hazard it can be structural hazard or it can be data hazard whatever it is, all other subsequent instructions are also be stalled.

(Refer Slide Time: 12:42)



Now, consider the case you have a division instruction on 14 point as I have mentioned in the last lecture, whenever we have registers which are starting with F, F0, F2, F4 like that, they are floating point registers, meaning the data is doubled. So, think of a case that you have a division operation on F2 and F4. So essentially this means F0 will be loaded with F2 divided by F4 is a division operation and we have seen that division will take close to 24 cycles.

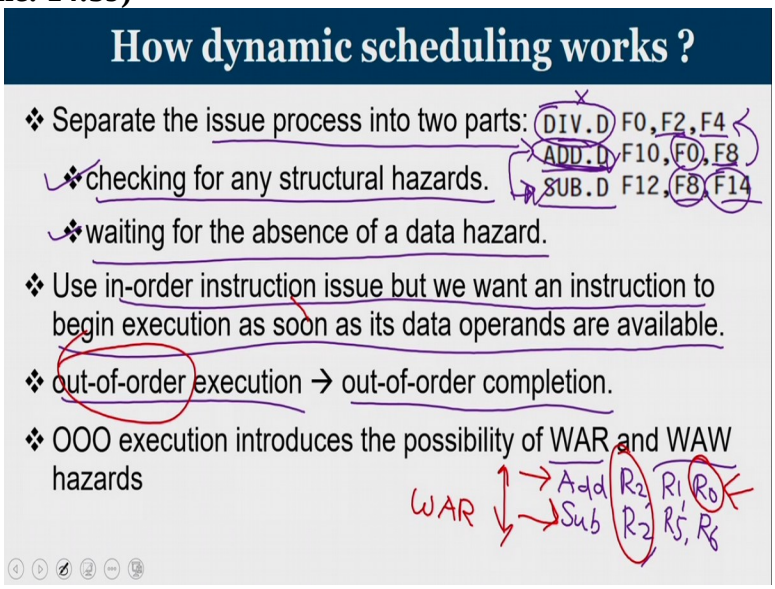
Now, the very next ADD instruction is having a raw dependency on F0. So, even though the adder is free, the second ADD instruction cannot proceed because a division instruction that is there is taking longer time. Now, if you look at the subtraction instruction, let us assume that adding and subtraction has been done in a different unit or let us say you can actually do the subtraction because the operands of subtraction F8 and F14 are available.

So, as far as the subtraction operation is concerned, the functional unit that carry out the operation that is there, that is free, the operands on which the operation needs to be done, the contents of F18 and F14 are ready. But in the case of an in-order pipeline subtraction instruction is stalled because the previous ADD instruction cannot start, because its operands are not ready.

This scenario, if an instruction j depends on a long running instruction i currently in execution pipeline, all instructions after j must be stalled until i is finished and j can execute. So, if an instruction ADD that is been mentioned over here, if an instruction ADD is dependent on a long running instruction division that is already in the pipeline, all instructions after ADD, that is from SUB onwards down are stalled cannot be executed until i is finished first until division is over and followed by j can execute.

So, why there is a restriction can we think of an option which can still improve and that is what dynamic scheduling is all about.

(Refer Slide Time: 14:59)



So, consider the same set of instruction. So, what we do in dynamic scheduling is separate the issue process into 2 portion. Step number 1 is checking for any structural hazards. Second one is waiting for the absence of a data hazard. So, what is the structural hazard, as far as this instruction is concerned, if the division unit is not available, if it is not free, that is called a structural hazard. If division unit is available but if the values F2 and F4 is not ready, that is called a data hazard.

Now, in this case, you can see that as far as the add instruction is concerned, there is no structural hazard, but there is a data hazard, adder is available, but the value of F 0 is not available, because it is dependent on the previous division instruction. So if we can divide the issue process into checking of structural hazard and waiting for the absence of data hazard, that is how dynamic scheduling works. So, use in-order instruction issue, but we want an instruction to begin execution as soon as data operands are available.

So, issue means if you are free from structural hazard perform the issue and as on when the data is available, let us run. So, if an issue is in-order, ADD will be issued that means it is going to the adder and it is waiting in the input of adder for the value of F0 to be ready. So, with F8 value, I can wait in the adder and as on when F0 is available, ADD can proceed. In the meantime for subtractor, that says the functional unit is available and if F8 and F14 is there. So there is no structural hazard, the adder or the subtractor unit is free, operands are also ready so this will permit the subtraction operation to start execution, before the adder starting execution. So, this will lead to out-of-order execution. See this is the order division is the order, so division is in progress and adder has not started but subtractor will start or subtraction instruction will start. This will lead out of order execution and naturally it will lead to out-of-order completion as well.

So, out-of- order execution will naturally result in the possibility of WAR and WAW hazards which we have seen. So, what is the WAR hazard, a WAR hazard is a case wherein you have a write after read hazard, if an instruction is going to write on a data, after some other instruction is going to read on a data.

So, one such example is if we have one instruction, let us say an ADD instruction
ADD R 2, R 1 , R 0

Now you have a subtraction instruction

SUB R 0, R 5, R 6

Now, in this case, if you look at the instruction sequence, you can see that there is an R 0 from which the ADD instruction is going to read and there is an R 0 to which the subtraction instruction is going to write.

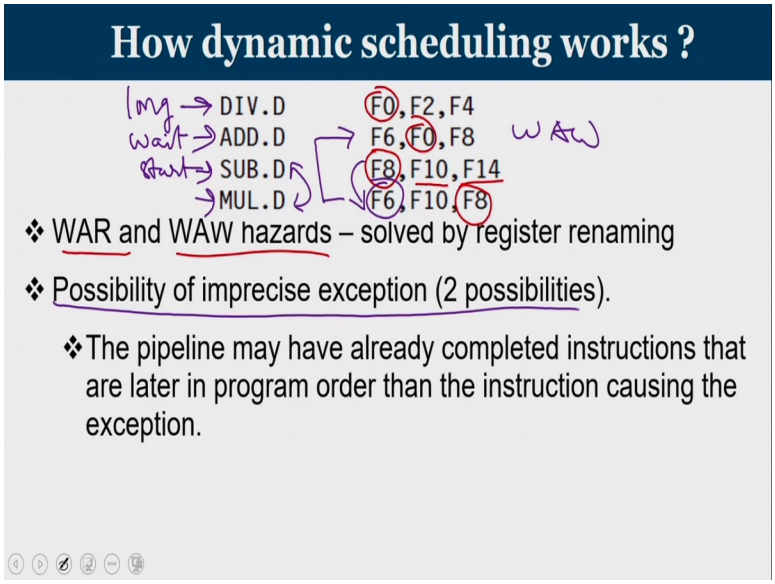
Normally, if you see if it is an in-order pipeline, then the ADD will read the contents of R 0 and R 1 and then only the subtraction is going to write. Because reading from R 0 and R 1 happens in the ID stage of the ADD instruction and writing into R 0 will happen in the WB stage of SUB. Now, if SUB is permitted to run before the ADD, then SUB will write into R 0 and when ADD tries to read, ADD is going to read the wrong R 0.

And that is exactly what a WAR hazard is interchanging of these ADD and SUB instruction will result in WAR. But so far in the in-order pipelines we have seen such a kind of an interchanging will never happen. So, when ADD is waiting, SUB is also waiting. So, when you permit out-of-order execution, then these kind of scenarios can happen. Similarly, we can how WAW hazard as well.

So, this WAW hazard typically happens when you have a scenario where your subtraction is also going to be on R 2. So, if subtraction is R 2, then both the ADD and subtract are going to use the same destination register. So ideally, this means the subtraction should write into R 2 only after ADD writes into R 2. So if the order is violated, then that can lead to write after write hazards.

So dynamic scheduling is the process by which out of order execution is permitted and this out of order execution can result in WAR and WAW hazards.

(Refer Slide Time: 19:36)

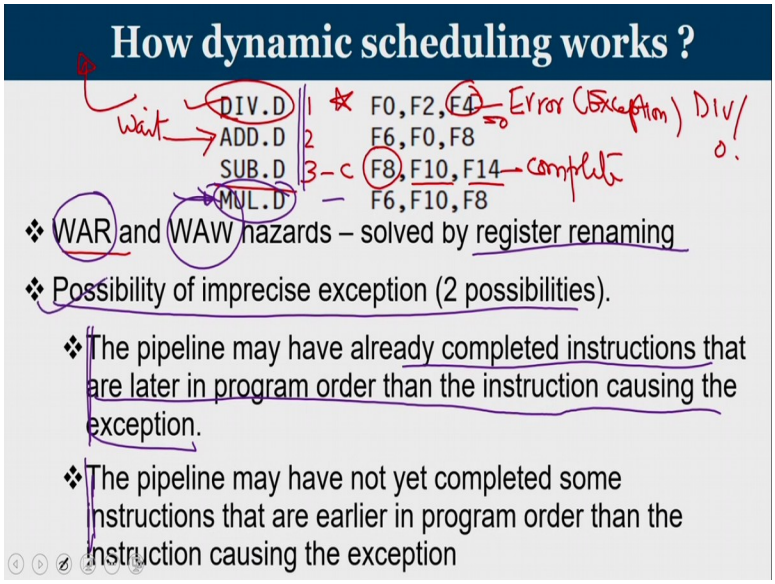


The how dynamic scheduling works. WAR and WAW hazards has to be addressed if you wanted to go for dynamic scheduling. So think of a case what happens here when you have a division instruction, which is dependent on F 0 and then you have a sub instruction, which is not dependent on anything. So, this is F10 and F14. So, this is going to write into F8 and this F8 is what has been used.

So now if you look at here, this division is taking long time. So ADD is going to wait SUB will start execution and by once SUB is over that will permit multiplication to complete its operation and once multiplication is completed, then it will write into F6 because multiplication is only dependent on the subtraction, so once subtraction is over, it will permit multiplication to carry out its operation, it will write to F6 and if you write to F6 and later only ADD is going write to F6 and that will happen as a WAW hazard.

The possibility is, sometimes you can get an imprecise exception. So we will try to understand what is an imprecise exception all about. A pipeline may have already completed instruction that are later in the program order than the instruction causing an exception.

(Refer Slide Time: 21:08)



Let us try to understand this imprecise exception with the same example itself. Now think of a case we are currently doing division instruction and because of that, the ADD instruction is waiting. Now imagine that you are subtraction instruction is permitted to run. So that means F10 and F14 will go into the subtractor and the result is produced in F8. And imagine that your subtraction instruction is now complete.

Once a subtraction instruction is complete, assume that the division cause an error. It can be an exception. Let us say F4 value was equal to 0 division by 0 might have happened. So if the operand if the divisor is going to be 0 then division is not defined, it will lead to an exception division by 0 exception. So let us imagine this is the first instruction the ADD is third, the third instruction is completed, it will update the value of F8, but the first instruction will come to know that the division operation is not possible.

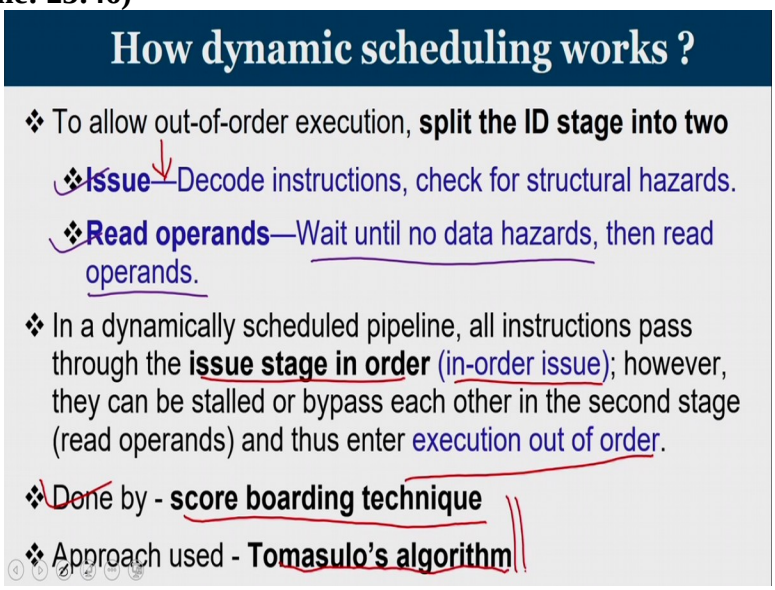
So first instruction encountered with an exception in the meantime, the later instruction has actually completed. Such an exception is called an imprecise exception, we cannot handle an exception generally what happened when you get an exception is, you have to roll back all the instruction that are partially over and make sure that all instructions before division is complete.

Since the subtraction has already changed, the state of the registers meaning F8 is updated any kind of a recovery mechanism will become difficult. The pipeline may not how yet complete some instruction that are earlier in the program order that instruction causing the exception.

Yet another way is a form of exception is imagine that your multiplication as encountered an exception. Can I handle the exception. Can I go for an exception service routine?. No. You cannot go to an exception service routine on the multiplier until all instructions before this multiplication is over. So, these two cases a pipeline may have already completed instruction, that are later in the program order, than instruction causing exception, the one that I told. Division is creating an exception when subtractor is already complete or the pipeline may have not yet completed some instruction that are earlier in the program order.

The instruction causing the exception. So, value deal with dynamic scheduling we have to deal with WAR and WAW hazard that will happen and that is done by register renaming concept and then you have to deal with exceptions that is causing.

(Refer Slide Time: 23:46)



Now to allow out of order execution what we do is we split ID stage into 2, stage 1 is known as issue. Decode the instructions, check for structural hazards if any and if there is no structural hazards, then you go to the read operands faced, wait until no data hazards and then correspondingly read operands. In a dynamically scheduled pipeline all instructions pass through the issue stage, stage number 1, this is stage number 1, issue stage in-order that is called in-order issue.

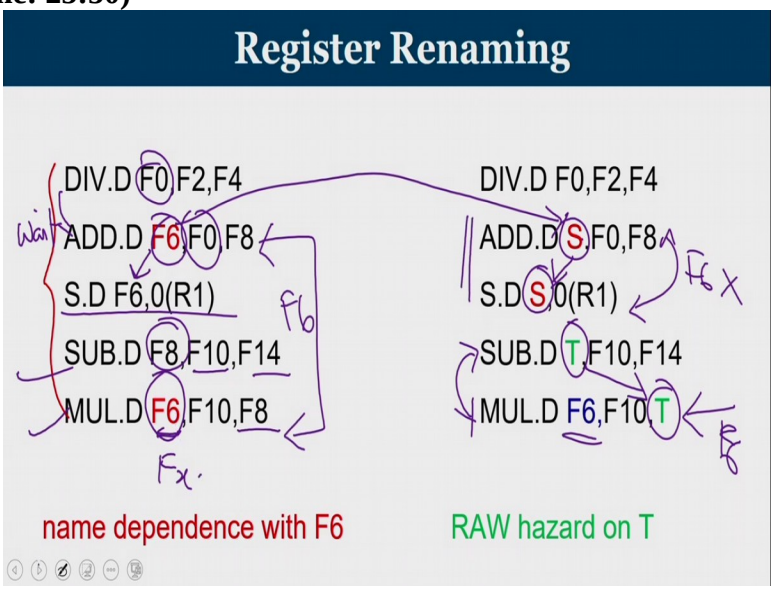
However, they can be stalled or bypass each other in the second stage or read operands thus entering the execution out-of-order. Now we will see, as far as 1 instruction once the fetching is over the next thing is try to understand what the operation is let us say to come to know it

is a multiplication, next stage is go to the multiplier get the operands perform the operation. So, issue means trying to understand what operand that is called decoding.

Try to understand what is a functional unit I wanted to perform operation on multiplication is multiplier freely available, then I am free from structural hazard. This much levels and I am issuing I am just going and waiting in the queue for the multiplier, issue has to do in order. Now the operands may not be available. So, those instructions which are already issued, but if the operands are available, they will run but those instructions which are issued if operands are not available, they will wait.

So, issue has to be order and execution can be out-of-order based upon the availability of operands. So, in short dynamic scheduling means divide the instruction issue into 2 sub stages, read, make sure there is no structural hazard issue in-order and then wait for the data assignment data is available, carried with the execution. How it is been done, it is been done by a technique in order score boarding techniques and approach use this Tomasulo's algorithm.

(Refer Slide Time: 25:50)



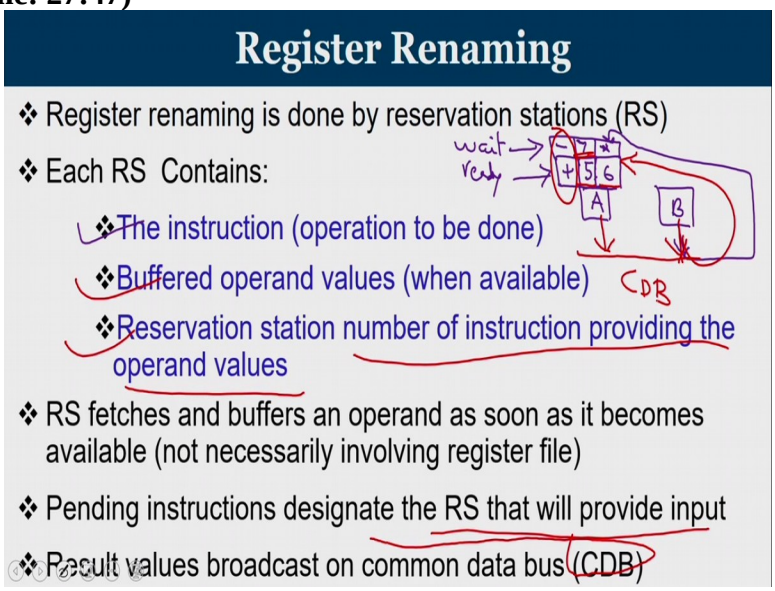
We will see how register renaming is done. So consider there are, there is a sequence of program of instructions where in your ADD is going to write into F6 and multiplication is also going to write into F6. Now look at the sequence of program there is a dependency between the division because of this F0 dependency. Because of that, this ADD is going to wait and then we are supposed to store the value, since the value of F6 is not available I cannot even perform this store.

Now the subtraction operation is going to work F10 and F14 and F8 is the resultant assignment F8 is available multiplication is going to write into F6. So, here F6 is the problem the ADD and multiplication are going to operate on F6. It is only a name dependency, had this F6 been replaced with some F x then there is no issue like a WAW hazard here. So, this is something that has been created.

So what I can do is these 2 are the same thing that is going to work and these are the same thing, that is going to talk about so as on when, this S is available then only, the store can go so as on when, the T is available then only I can go and this is the actual F6. So, now we have a RAW hazard that is one T so they will be dependent on each other. I am renaming this F6 with some temporary register.

So the ADD value will be returned to a temporary register S which is used by the store in order to write. So they need not update on real F6 that is not happening, only the multiplication instruction is going to update on F6. So temporarily the name F6 that was used here is been now using a different one. That is the way how it has been processed. This whole step is known as register renaming rather than using the name F6, some temporary register is being used.

(Refer Slide Time: 27:47)



This register renaming is done with the help of the reservation stations. So, every functional unit will have reservation stations, which tells the instruction or operation to be done, it will buffer the operand values when they are available. The reservation station number of

instructions providing the operand values. Reservation station fetches and buffers and operand as soon as it becomes available, not necessarily involving the register file.

And pending instruction designates the reservation station that will provide an input. So consider, this is 1 functional unit A and this is another functional unit B. The reservation station of A means, I am supposed to perform a '+' operation on 5 and 6. So 6 and 5 are the values as on when A is free then this has been taken. Similarly, another entry of the reservation station can be told us how value to be 7, lets say 7 is my operand. And I am waiting for a value to be produced by B.

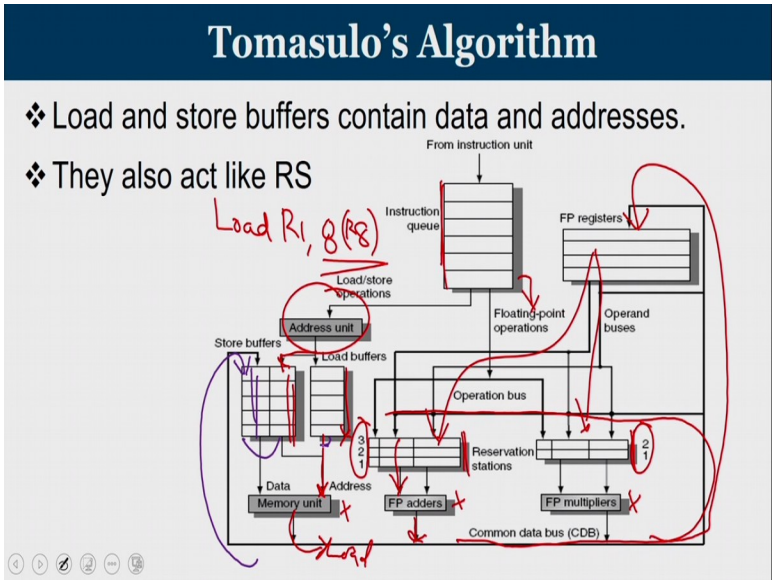
So as on when B produces the result that value will come over here. So, this minus instruction or subtraction instruction is actually waiting for an operand, which is to be produced by functional unit B whereas this is ready as on when the functional unit is ready this will go. So reservation station is nothing but it is a set of buffers that are associated with functional units and this buffer contain the operation that needs to be performed. So, in this case, this + and - are the operations, second one it will buffer the value.

So, if the value is available, you take the value and keep it ready. If the value is not available, So, in this case, 1 operand is available, where the second operand is not available, it is yet waiting for the operand due to it is, it is basically because of raw hazard. So, if it is waiting, it has to tell for which reservation station it is waiting. So it is actually waiting for a result that is to be produced by B. So reservation station number of instruction providing the operand values.

So reservation station fetches and buffers and operand as soon as it becomes available, all pending instructions, you will tell which reservation station will produce. We will learn about how this the reservation station works, while we work with the Tomasulo's algorithm, that will come in the next lecture. Now the peculiarities all these things the result of each of these functional unit has to be broadcasted on a common data bus that is what has been mentioned overheads called CDB.

So, every functional unit is connecting to a common data bus and everybody is waiting in this you will see the architectural diagram also.

(Refer Slide Time: 30:42)



So, this is how it looks like, you have the instructions that is been fed. This is the instruction queue and the during the decoding operation, let us say these are the adders, these are the functional units. These are the multipliers, another category of functional unit. This is the memory unit which will take care of load and store. So, you pick an instruction from here look at the operand and then you feed them into the reservation station.

So, you can see these are all queues, your adder floating point adder, there is a 3 entry reservation station, this is a 2 entry reservation station. So, the values if at all if the options are available, you go to the registers and you put the values in the reservation station. Now as an when the functional unit is free, and the values are available, the entries from the reservation stations are have been taken and are put into the adders or subtractors or multiplies whatever functional unit and they are going to produce the result.

Now we have to see that this common data bus is connected to the input side of the reservation station. So, if anybody is waiting for any value, you will get the value at the same time if the result is to be returned to the registers, it is going to be updated there. Now look at the load and on store you have an address computation that happens. So if you have a load instruction, that is a

load R1 , 8(R8)

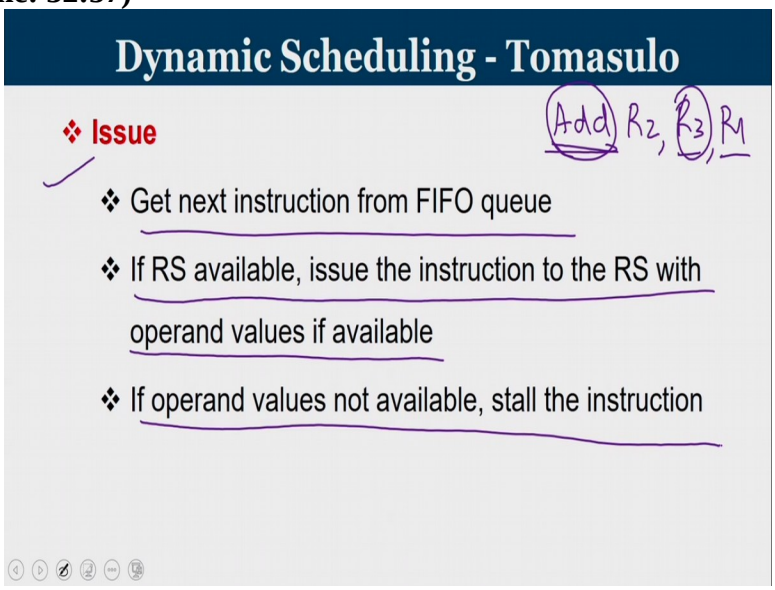
So, the address unit commutes R8 + 8 and that will give you the effective address.

Now be the effective address you go to memory unit and this is the process of load. Now, if it is a process of a store, the address unit will produce the address, this is the place where you

store the address and the data to be written that is available here. So, that is generally produced by these functional units and you get the values to be written. So, store has a value as well as an address whereas the case of a load, it is only the value that is available.

So, the output of memory unit is also connected to the same common data bus if a value with an loaded value is to needed, then that also is being obtained in the input of the reservation stations.

(Refer Slide Time: 32:57)



So in Tomasulo algorithm, which is a most prominent algorithm that will take care of dynamic scheduling. We have a detailed session one lecture exclusively for Tomasulo algorithm working. During that time, it will be more clear. Today before winding up and just summarizing what are the important aspects of Tomasulo's algorithm. The Tomasulo algorithm issue means you get the next instruction from the FIFO queue.

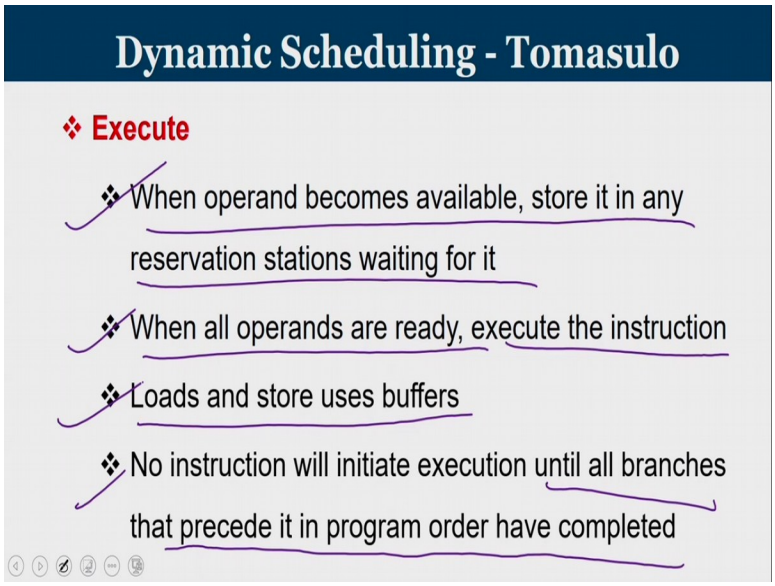
If reservation station is available issue the instruction to the reservation station with whatever operands available at that point of time. So, consider the case you have an ADD instruction

ADD R2, R3, R1

So, if the reservation station is available means, if the adder has a reservation station which has space available, then I will issue into the reservation station is the value of R 3 and the R 1 is there go with them.

If you have only R 3 then go with R 3 and R 1 you will get it from some other instruction. If operand values are not available, then even if it is going to the reservation station it cannot enter into the execution stage.

(Refer Slide Time: 34:06)



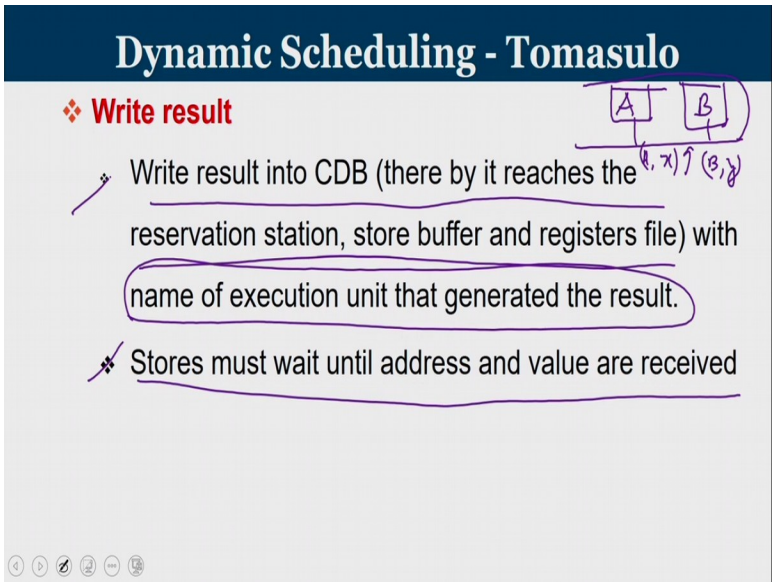
So what happens in the execute, in the case of execute these are the operations that are waiting into reservation station when the operand become available, store it in the reservation station waiting for it. That is what you have to do, when all operands are ready as far as an instruction waiting in the reservation is concerned you execute the instruction. So, when the execution is over, update all the reservation stations some of the following instructions may be waiting for the result of this.

When all operands are available for instructions waiting into reservation station carry out the operation. Loads and store has to be using buffers. That is what we have seen in the diagram, no instruction will initiate execution until all branches that preceded in the program order has completed. So this is how you deal with control hazard. Think of it as that there is a branch instruction that is halfway.

So if we have a branch instruction and there are some other instructions after the branch, I cannot execute those instructions because at the point of execution, if the branch condition is not resolved, once I start execution, it will produce the result, it will change the value of my registers, it is very difficult to unroll it back. So, execution of instructions after a branch will happen only if you are sure that this instruction is must execute.

That means as far as the outcome of a branch is concerned, this instruction will be executed. So, whenever there is a branch all other subsequent instructions are going to wait and execution is been delayed.

(Refer Slide Time: 35:38)



And how will write the result? We have seen that all the functional units are connected to a common data bus. So you write the result into common data bus thereby it reaches the reservation station, store buffer and register file. Now, like the way I told, this is a functional unit A and this is another functional unit B and if there is a common data bus. And how will somebody know these values. Let us say I am producing a value X in the bus, how will somebody know the value X that is been return into the bus is produced by A or B. Because somebody may be waiting for a result produced by A and somebody may be waiting for a result produced by B. So it is very important that the name of the execution unit that generated the result. It is basically like A produce the value X similarly B produced the value Y.

So along with the value, you have to produce which functional unit has generated the result. And stores must wait until address and values are been received. So that is the way how you perform the write result. So, with that, we are coming to the end of this lecture, a quick summary of what we learned today, on the previous class, our focus was on how can compiler helps in reorganizing instructions to improve performance of pipeline and in today's lecture, we were looking at yet another alternative model that is called dynamic scheduling.

In dynamic scheduling, you are trying to execute instructions out-of-order. So when 1 instruction cannot proceed due to a dependency or a hazard all other instructions after that need not be waiting. So instructions that how functional unit ready that how operands ready are permitted to execute. For that the stage has been divided into 2 issue happened in in-order

and reading of operands can happen out-of-order and this is implemented using Tomasulo's algorithm.

And the concept of register renaming will take care of your WAR and WAW hazards and Tomasulo's algorithm will take care of dynamic scheduling. So with that we conclude. Thank you.