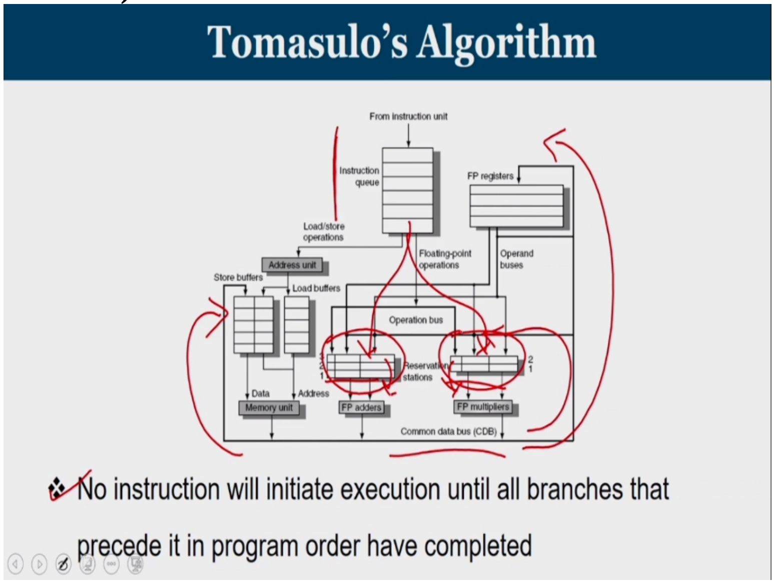
Advanced Computer Architecture
Dr. John Jose, Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati, Assam

Lecture 10
Dynamic Scheduling with Speculative Execution

Welcome to lecture number ten. In today's lecture, we are going to see the concepts related to speculative execution. Last 2 lectures, our topic of discussion was dynamic scheduling, wherein instructions are reordered in order to improve the performance of instruction pipeline. As a specific case we have seen how the Tomasulo's algorithm is used to take care of operand forwarding, register renaming and handling of other hazards.

One point which was put as a condition for the working of Tomasulo's algorithm was we cannot run instructions that are dependent on branches. So, instructions that come after a branch instruction cannot be executed on a Tomasulo's algorithm. But, we have seen that controls hazards also has to be handled in an instruction pipeline. Today's topic of dynamic instruction scheduling with the speculative support will make you aware how speculative execution can be done or how can you run instructions, which are dependent on branches.

(Refer Slide Time: 01:49)



Tomasulo's Algorithm

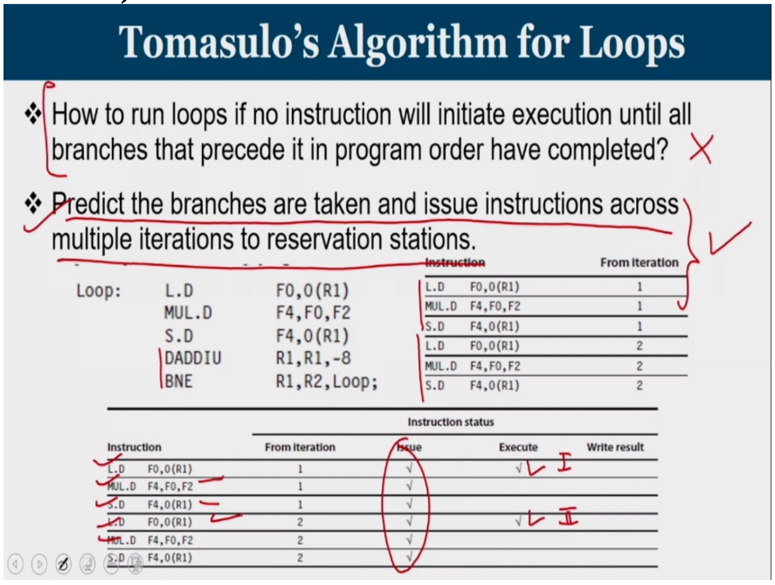
❖ No instruction will initiate execution until all branches that precede it in program order have completed

So, in this slide that is been shown, this is the general architecture, which is used by the Tomasulo's algorithm, where you know that instructions are there in the instruction queue. And then we have different functional units and these functional units are having reservation stations. So, during the issue stage you check for vacancy available in the reservation stations and then the instructions are being issued into them.

As on when the operands are available, then the instructions are taken from the reservation station and are carried out in this execution unit. And at the end of execution the results are pumped into common data bus, which will update the registers, which will update the memory if it is a store operation and which will update the pending instructions waiting in the reservation station.

We have seen that no instruction will initiate execution until all branches that proceed it in the program order have completed. The reservation station associated with their functional unit is not free it is considered as a structural hazard and issue will be blocked. Once the issue is blocked all other pending instructions are also blocked.

(Refer Slide Time: 03:00)



Tomasulo's Algorithm for Loops

- ❖ How to run loops if no instruction will initiate execution until all branches that precede it in program order have completed? X
- ❖ Predict the branches are taken and issue instructions across multiple iterations to reservation stations. ✓

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,Loop;
```

Instruction		From iteration
L.D	F0,0(R1)	1
MUL.D	F4,F0,F2	1
S.D	F4,0(R1)	1
L.D	F0,0(R1)	2
MUL.D	F4,F0,F2	2
S.D	F4,0(R1)	2

			Instruction status		
Instruction		From iteration	Issue	Execute	Write result
L.D	F0,0(R1)	1	√	√ ✓ I	
MUL.D	F4,F0,F2	1	√		
S.D	F4,0(R1)	1	√		
L.D	F0,0(R1)	2	√	√ ✓ II	
MUL.D	F4,F0,F2	2	√		
S.D	F4,0(R1)	2	√		

Tomasulo's algorithms for loops. So, when you have loops, how do you run loop if no instruction will initiate execution until all branches that precede it in the program order have completed. So, this is a very tough condition to satisfy. Because generally when you have a loop, we have seen it in compiler scheduling that they adjacent iterations of the loop is also unrolled such that we are not waiting for the instructions in the previous iteration to complete.

So, unrolling the loop is a very common approach that is used to improve the performance of instruction pipeline. But when in Tomasulo's algorithm, when you see that you cannot execute speculative instructions. So what we can do is here is the role of a branch predictor you predict the branches are taken and issue instructions across multiple iterations. Since the first line blocks you in executing instructions that are after a branch instruction.

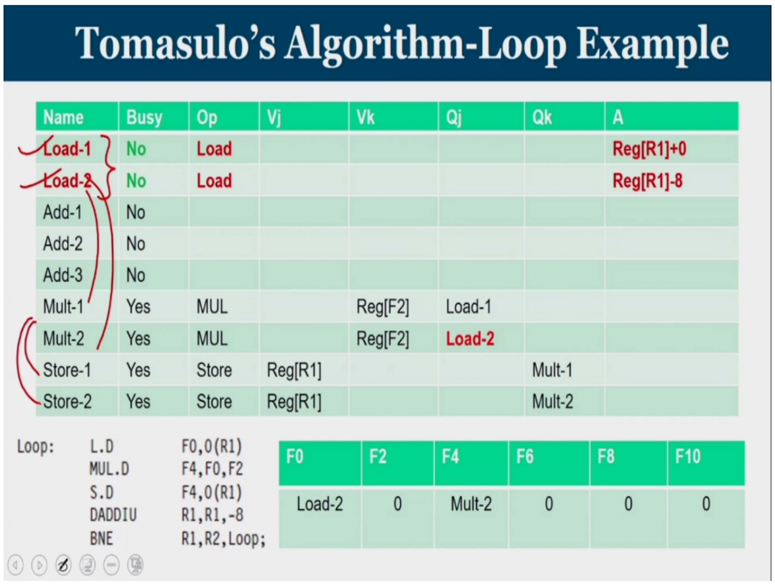
Which will stall the pipeline for some time until the branch instruction has been result the second condition what is mentioned over here, will permit us to predict that the branches will happen. If the branch is happening, we know what are the instructions that are to be executed execute them across multiple iterations. So consider the case we have a loop where we load a value and the value is coming to F0.

You are going to multiply the content of F0 with F2 storing the result in F4 storing the value back and then we have 2 instructions that the manipulate the loop, we are going to subtract the value of R1 until the value of R1 reaches R2. In this case, if I am going to unroll the loop 1 more time. So you have load MUL and store that is part of first iteration and again load MUL and store or this part of the second iteration.

Now, you can actually issue all these instructions that is what has been shown across iterations, but only the first 2 loads may be executing you are actually waiting for the load to complete. So, think of a case that you have 2 loads store unit. The first load is carrying out in the first load store unit, the second load will be carried out in the second load store unit but you are not sure whether the second load is actually required or not, because it depends on the condition of the branch.

So, we are we are predicting what is outcome of the branch and then go and execute instructions that are generally coming after the branches condition has been resolved. So, in this scenario, the instructions MUL and store that are part of first iteration are not yet complete, but an instruction that is load which is part of the second iteration can be executed.

(Refer Slide Time: 05:55)



Tomasulo's Algorithm-Loop Example

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load-1	No	Load					Reg[R1]+0
Load-2	No	Load					Reg[R1]-8
Add-1	No						
Add-2	No						
Add-3	No						
Mult-1	Yes	MUL		Reg[F2]	Load-1		
Mult-2	Yes	MUL		Reg[F2]	Load-2		
Store-1	Yes	Store	Reg[R1]			Mult-1	
Store-2	Yes	Store	Reg[R1]			Mult-2	

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,Loop;
```

F0	F2	F4	F6	F8	F10
Load-2	0	Mult-2	0	0	0

So, this kind of a mechanism requires some kind of an extra support. So, consider the same program loop that has been discussed now we are loading a value multiplying a constant to that loaded value that is called F0 and F2 has been multiplied and then you store the result and back into the same location and then go to the next element. So this is the register status indicator that is being shown in the conventional Tomasulo's algorithm.

So here, the register status indicator is giving the status of 6 registers F0 F2 F4 F6 F8 F10. Now I will take you through a quick snapshot of what happens to the reservation station entries when you execute multiple iterations of this loop. So what you see here in this table is your reservation station. Where we have operand value V j and V k will contain the operands is the operands are readily available and Q j and Q k will tell you the functional units if operands are not available, and A will carry the address.

I am going to talk about the very first load instruction is going to write the result into F0. So register status indicator of F0 should be pointing to load-1 and the address is

0 + content of R1

which will be carried out in the first load unit. Assume that you have 2 load units to store units, 2 multiplication and division unit and 3 Adders. Now you are going to the multiplication instruction, where in one of the operand is available in the register file that is why Vk value which will have the second operand F2.

But F0 is not available since you a value of F0 is not available I am specifying the functional unit that will produce of load-1 is going to produce a result that has to be returned to F0 and the multiplier unit is essentially waiting for an output to be produced by the load-1 unit. Moving further for a store operation, imagine you have a store unit and its first operand is R1 which is used for computation of an address and then it is waiting for a value that will be produced by the multiplier unit.

So, this multiplier unit has to produce something and then it has to be put in CDB for the store one unit to do. Now these 2 instructions that is a multiplication instruction and the store are actually waiting, but we are going for a loop unrolling. So, you can see that since the multiplication operation is going to write the result to F4 the RSI register status indicator of F4 points to multiplication of.

Now, the second load operation that is actually the first instruction of the next iteration, we know that this load for a second iteration will happen only if this branch condition is met. So the branch condition tells that R1 and R2 value should be equal to 0. Now, when I predict the value of the branch, and it is a prediction say that the branch will be taken that means the control will be transferred to loop where in the load MUL and store will be executed for the next iteration.

So, generally such an execution will happen only after the branch condition is resolved. Here we predict that the branch will happen we are not waiting for the outcome of branch, we are going to the next iteration and trying to issue the instruction since you have one more load unit I can issue the second load, but we can see that the value of R1 and offset has to be appropriately adjusted.

Generally, the value from which the reading happen is

0 + content of R1

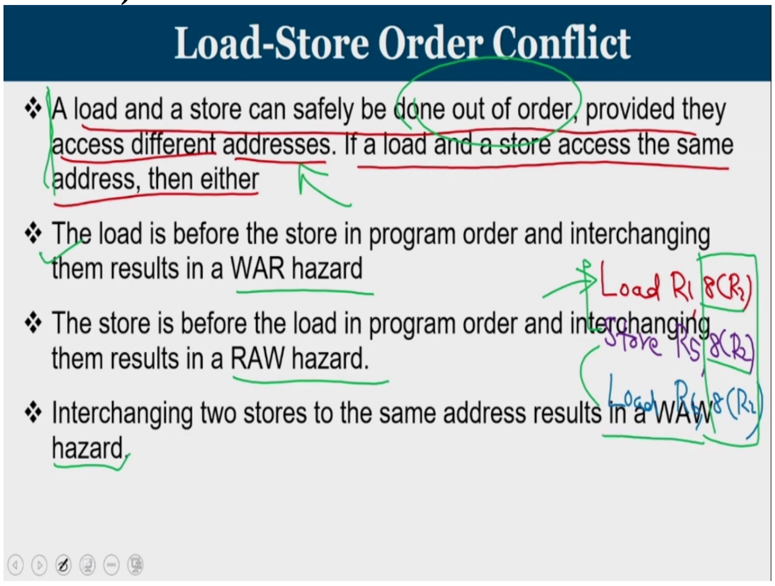
And now when you go to the next iteration, it will be content of R1 + (-8) because we are decrementing the value of R1 by -8. Now moving further the multiplication instructions, since they have 2 multiply units I can carry out the multiplication. So, the second multiplication unit is waiting for the output of load-2, where as you are first multiplication unit is waiting for the output of load-1.

So that is the dependency there and this is the dependency 3. So, if you have more number of functional units, I could carry out instructions across iterations and they all will wait for the appropriate values. Now, coming to the store instruction, the case of a store instruction you can even since there is 1 most store unit, the store unit is in turn waiting for an output produced by a multiplier unit 2 and we have to see that now at this point F0 values and therefore register status values are updated to load-2 and MUL 2 respectively.

So, the second iteration, when you are going to execute the second iteration, the RSI value changes and when you are going to execute the first iteration, the RSI value has a different set of index. Now think of a case that your load instruction is going to get over so both load-1 and load-2 units will be free. So, the moment load-1 is free then you are multiplication instructions can actually start and once multiplication over, the corresponding stores will do.

So store 1 and store 2 essentially will complete the operation together and then your instruction execution will continue in the normal fashion. So, it is kind of a speculation we are predicting that the branch is happening.

(Refer Slide Time: 12:07)



Load-Store Order Conflict

- A load and a store can safely be done out of order, provided they access different addresses. If a load and a store access the same address, then either
- The load is before the store in program order and interchanging them results in a WAR hazard
- The store is before the load in program order and interchanging them results in a RAW hazard.
- Interchanging two stores to the same address results in a WAW hazard.

Now, can we execute instructions of adjacent iterations without knowing whether such an iteration will take place or not? Can you bypass control hazards, these are basically instructions with the loading and storing that we have seen in this example, let us try to understand, Can I perform a load of the second iteration, when the store of the first iteration is not yet overwe are trying to have a dynamic instruction execution, out of order execution, how much out of order is permitted and what are the consequences of out of order execution.

This is known as load store order conflict. A load and store can safely be done out of order, provided they access different addresses, if a load and store access the same address, then either the load is before the store and the program order, like this. So you are going to have

load R1, 8 (R2)

and let me take the second instruction. It is a

store R5, 8 (R2)

let me put up the third one another

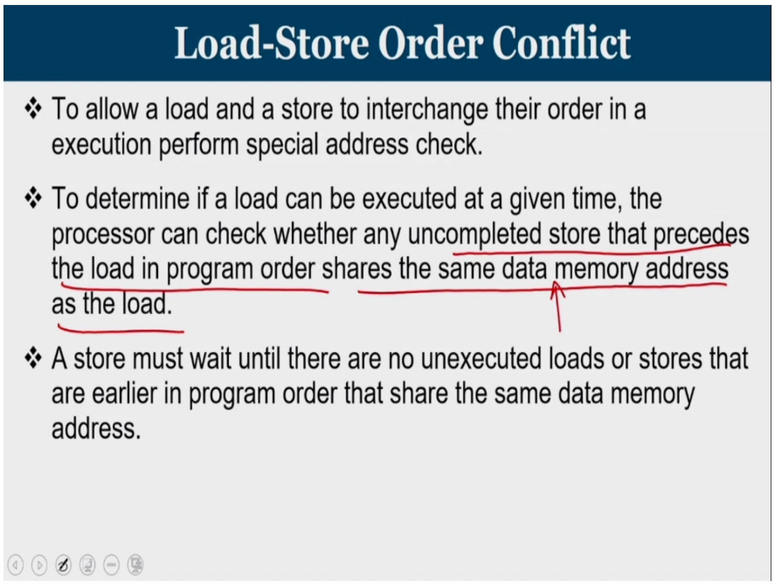
load R6, 8(R2)

So if the load is before the store in the program order, let us say this is my load and store, where the load is before the store and program order and if I try to interchange them, try to interchange means, generally the program order means the load has to be over, the loading a value and then you are going to store that means after the load operation is over some a new value is returned to the location 8 of R2. So, when you interchange that means when a store happens on the same location, but the order is changed, then the loading happening after storing.

So, store will write a new value and that is what the load is doing. So, the load is before the store in program order and interchanging of the result will result inside a WAW hazard. Similarly, if the store is before the load in the program order, so that is what we can see there is a store and the load. So, the interchanging will result inside the RAW hazard. So, we have to be very careful when you interchange the order of load and store provider they are going to access into different providers they are going to access to the same address.

As mentioned in the first point load and store can be safely done out of order, you can exchange the order provide that they are going to access different locations. So, in this example, what has been mentioned if the location where this load and store going to access if there are different interchanging of the order is no longer going to be problem. So, interchanging 2 stores to the same address can result inside WAW hazard as well.

(Refer Slide Time: 14:40)


Load-Store Order Conflict

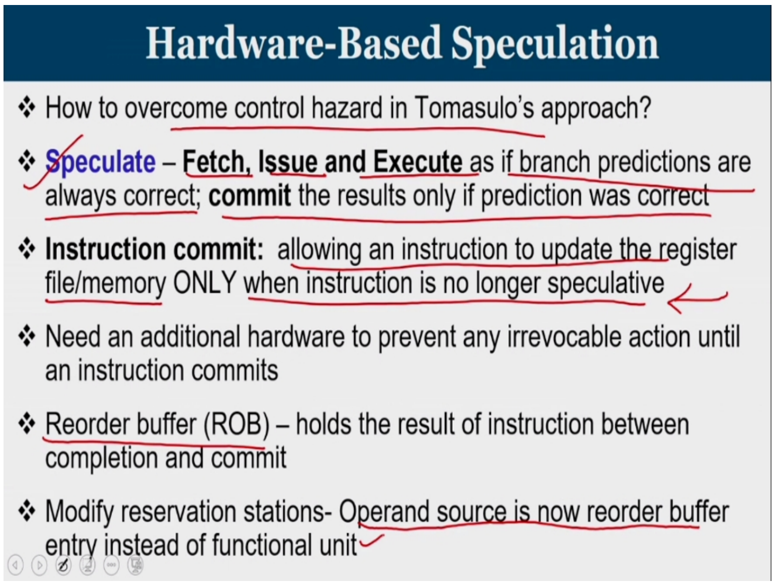
- To allow a load and a store to interchange their order in a execution perform special address check.
- To determine if a load can be executed at a given time, the processor can check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load.
- A store must wait until there are no unexecuted loads or stores that are earlier in program order that share the same data memory address.

Now, to allow a load under store to interchange their order and execution you have to perform a speciality check and what is a checking all about. To determine if a load can be executed at a given time, the processor can check whether any uncompleted store that precedes the load in program order share the same data memory addresses that of the load. Meaning suppose if I wanted to run a load instruction now, processor has to check any store operation that is before this load is still under execution.

So, all uncompleted store that proceeds the load, which having the same address if the address is not same it is not at all a botheration. Similarly, a store must wait and until there are no unexecuted loads or stores that are earlier in the program order that saying this share memory address. So, when you are going to deal with loads and stores any kind of interchange in the order in which they are executed you have to take care of 2 things.

Point number 1, if you want to put a load instruction somewhere processor has to make sure that there are no uncompleted stores before this load instruction, which are operating on the same address. Similarly, a store cannot happen at that point until all previous uncompleted loads or stores, which are pending. So, as long as you have pending loads on stores and store instruction cannot come there with the same address. So, if this checking is being properly done, exchange of load and store can be fairly easily managed.

(Refer Slide Time: 16:41)



Hardware-Based Speculation

- How to overcome control hazard in Tomasulo's approach?
- Speculate – **Fetch, Issue and Execute** as if branch predictions are always correct; **commit** the results only if prediction was correct
- **Instruction commit:** allowing an instruction to update the register file/memory ONLY when instruction is no longer speculative
- Need an additional hardware to prevent any irrevocable action until an instruction commits
- Reorder buffer (ROB) – holds the result of instruction between completion and commit
- Modify reservation stations- Operand source is now reorder buffer entry instead of functional unit

Now, coming to hardware based speculation, let us try to see what speculation is, whatever control hazard that we have discussed now, how I can you do or solve in Tomasulo's approach, the technique is called speculation. Speculation means fetch, issue and execute instruction as if branch predictions are always correct, but branch predictions can go wrong. So, you should commit the results only if the prediction was correct.

So, there are 2 approaches or basically there are 2 things that we have to keep in mind. When you go for speculation, you should have a mechanism wherein you can fetch, decode and execute instructions as per the directions given by a branch predictor. But the prediction happened to be wrong, you should have mechanism to revert back whatever actions you have done or you should commit operations only if branch prediction is correct.

So you require additional circuitry in the conventional Tomasulo's algorithm which has reservation stations, which has registered status indicator and which is common data bus that has been connecting to the registers and the reservation station. Extra circuitry is required to have a check on these 2 points that we discussed right now. So what do you mean by Instruction commit.

Instruction commit means allowing an instruction to update the register or memory only when instruction is no longer speculative. So think of a date, when are you going to update the register, when you have a load operation or when you have ALU where the result is to be written to a register. Similarly, when you have a memory operation like store operation, you are going to update the memory.

These updation in registers and memory should happen only when the instructions are no longer speculative. So when you run a speculative instruction, what do you mean by a speculative instruction. When I am going to run a load instruction, which is been predicted by a branch predictor, the branch predictor tells that you can run the loop. Let us say the first instruction in the loop is the load instruction.

So you are going to run this load instruction, as per directions given by the branch predictor. So at that point of time, this load instruction is a speculative instruction. Meaning I am not sure whether it is must execute or not, branch predictor tells that you can execute. But let us assume that after few cycles, the condition of the branch is no longer speculative means we know what is the outcome of the branch.

And if the predictor was correct, at that point of time, the load instruction is no longer speculative. So certain instructions, which come as a follow through of a branch or as a target of a branch, for some time, they are speculative instructions when you run and execute. So essentially, what we do is we run these instructions, store the result in some temporary place, when you come to know that these instructions are no longer speculative or the outcome of the previous branch is known.

And to sure that these instructions are to be executed, whatever is the result of these instructions, which was stored in a temporary place can now update the corresponding register and memory. And that is the core idea behind speculative execution. We need additional hardware to prevent any irrevocable action until an instruction commit. And that the extra hardware is known as the reorder buffer also known as ROB.

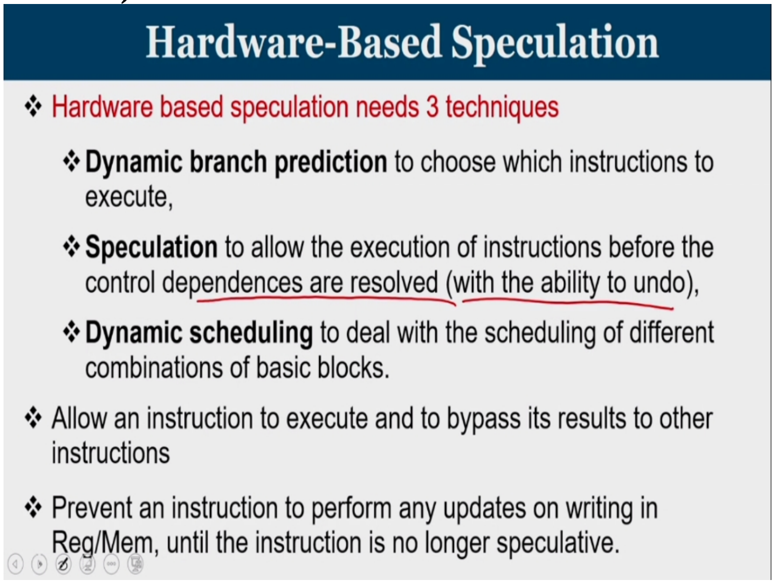
ROB holds the result of instruction from completion to commit. So when you perform a speculative operation, you will do the operation get the result store in a place called reorder buffer. And when you come to know that this instruction is no longer speculative from the reorder buffer copy into the memory or to the registers. So now there is a small modification in the reservation station, operand source is now a reorder buffer entry instead of the functional unit.

So previously, you are actually waiting for the functional unit to produce the result. So you are pinging the CDB the common data bus waiting for a result produced by the execution unit

1 or execution unit 2. Now we are not pinging the CDB. By looking at the number of execution unit, we are pinging the CDB we are looking at the reorder buffer entry that needs to be put.

The difference is, with speculative execution when you produce a result rather than telling execution unit one produce the result, the change is, the result that is to be generated by this functional unit has to go to reorder buffer entry 2, so everybody whose waiting for reorder buffer entry 2, can appropriately get them.

(Refer Slide Time: 21:27)



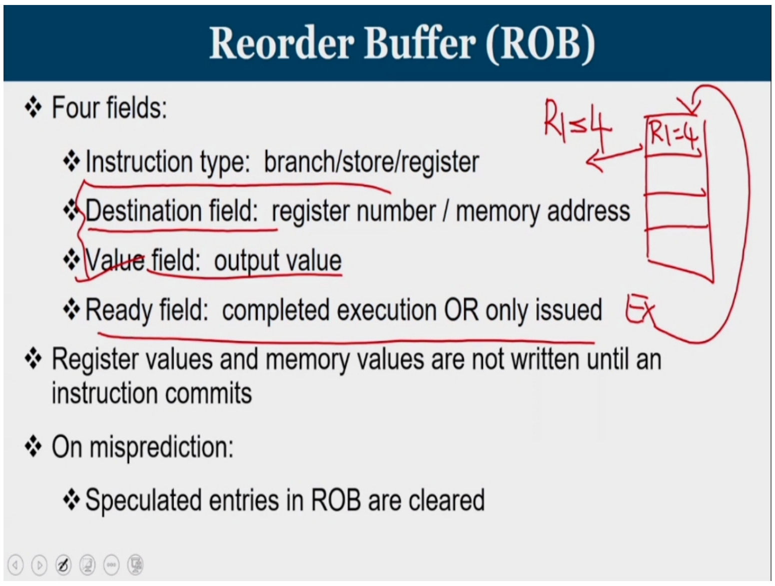
Hardware-Based Speculation

- Hardware based speculation needs 3 techniques
  - **Dynamic branch prediction** to choose which instructions to execute,
  - **Speculation** to allow the execution of instructions before the control dependences are resolved (with the ability to undo),
  - **Dynamic scheduling** to deal with the scheduling of different combinations of basic blocks.
- Allow an instruction to execute and to bypass its results to other instructions
- Prevent an instruction to perform any updates on writing in Reg/Mem, until the instruction is no longer speculative.

Hardware based speculation needs 3 techniques, you need to have a dynamic branch predictor, which will tell you which instruction to fetch and execute. You need to permit speculation to allow execution of instructions before the control dependence are resolved, control dependence are resolved means before the branch outcome is not. At the same time, you need to have an ability to undo whatever speculation is going wrong.

And then capability for dynamic scheduling to deal with scheduling of different combination of basic blocks. Overall, we are allowing an instruction to execute and bypass its result to other instructions and prevent an instruction to perform any updates on writing in register or memory until instruction is no longer speculative.

(Refer Slide Time: 22:11)



Reorder Buffer (ROB)

- ❖ Four fields:
  - ❖ Instruction type: branch/store/register
  - ❖ Destination field: register number / memory address
  - ❖ Value field: output value
  - ❖ Ready field: completed execution OR only issued
- ❖ Register values and memory values are not written until an instruction commits
- ❖ On misprediction:
  - ❖ Speculated entries in ROB are cleared

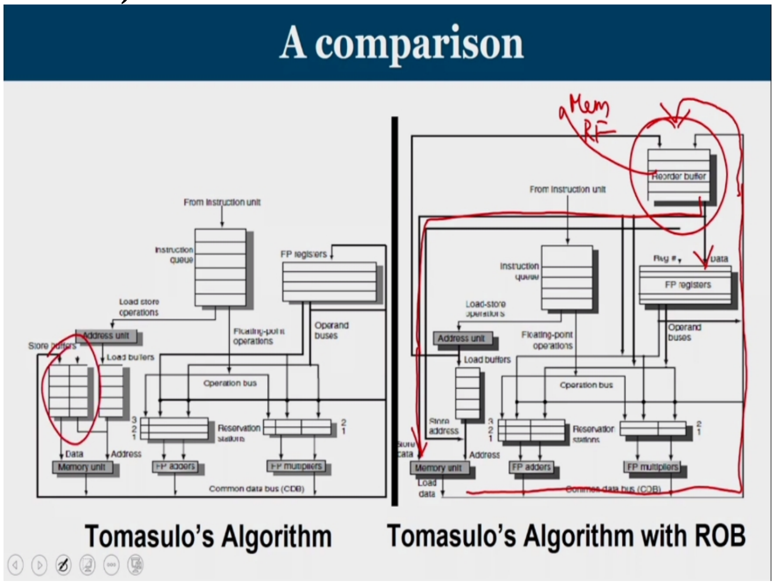
Now let us try to look into what is the structure of reorder buffer. So reorder buffer is basically the extra hardware. It is a queue which has 4 fields, it will store the instruction type, the destination field. If you have the reorder buffer to which the outcome of execution unit is going to come, we have to tell the value of R1 is going to be 4. So R1 value is entered and 4 is entered.

Meaning and instruction produce a result which has to be updated in register R1, and the content value is 4. Actual registering of R1 will happen only when this instruction is no longer speculative. So when an instruction is in a speculative stage, I stored the result in reorder buffer and reorder buffer is a temporary register, which will store the value or the outcome of an instruction before commits stage, but after complete stage, that is called a value field.

And then you can also tell a status whether the instruction is issued only or it is completed. Register values and memory values are not written into the registers and memory until an instruction commit. So as on when the instruction commit, there is a copying that happens from reorder buffer into registers and reorder buffer into memory. What happens if you having a miss prediction, speculator entries are only so the result of the executed that the speculated instructions that are executed are available only in the reorder buffer, I can just flush off the corresponding reorder buffer entries?

So even if you run, even if you execute a speculative instruction, the result is available only in the reorder buffer, the result is not updating your memory or registers. So any kind of misprediction that happens, it is only clearing of reorder buffer, no memory or register contents are being updated with wrong results.

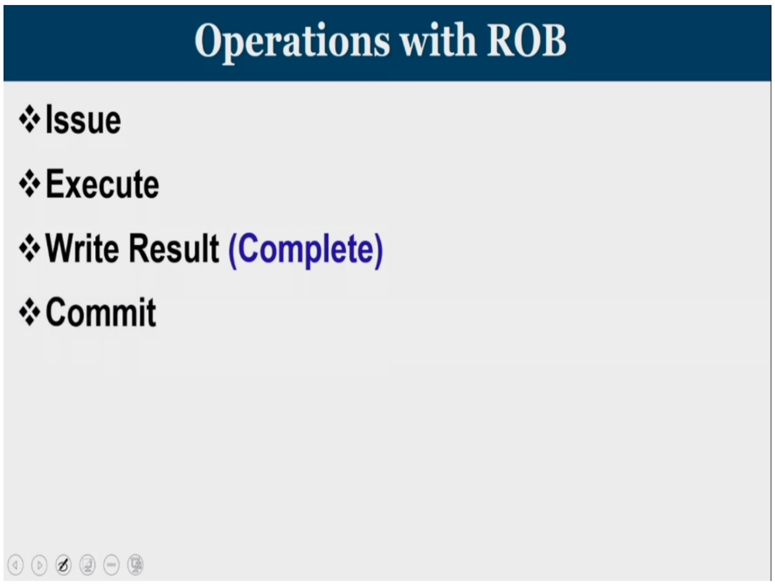
(Refer Slide Time: 24:13)



So this diagram tells on the left side you can see the conventional Tomasulo's algorithm where speculative execution is not permitted. And on the right side, we have the Tomasulo's algorithm with reorder buffer entry. So what is the basic change, here you can see there is a reorder buffer. So the outcome of this of the execution, which is returning to common data bus, now it is not going into floating point registers it is coming to this reorder buffer.

From the reorder buffer the moment you come to know that an instruction is complete. If it is a store operation, you are going to update in the memory. If it is an ALU operation where the registers are updated, then it happens. So from the reorder buffer, you update memory, if it is a store operation, or you are going to update the register file, if it is an instruction wherein the result is written to registers. So, the store buffer concept which was there is not here we have only load buffers.

(Refer Slide Time: 25:20)

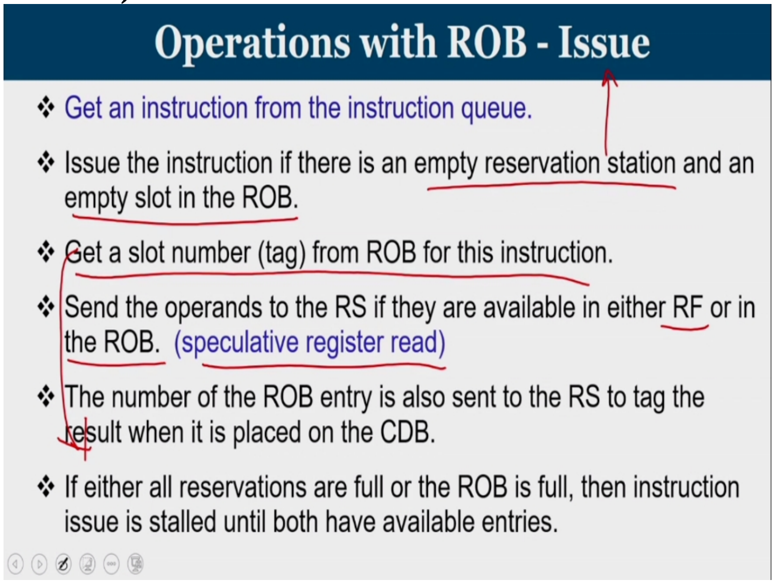


Operations with ROB

- Issue
- Execute
- Write Result (Complete)
- Commit

What are the operations with the reorder buffer first is the issue operation, execute operation, write result operation or it is also known as complete operation and the last stage is called commit operation. Let us see each one in detail.

(Refer Slide Time: 25:31)



Operations with ROB - Issue

- Get an instruction from the instruction queue.
- Issue the instruction if there is an empty reservation station and an empty slot in the ROB.
- Get a slot number (tag) from ROB for this instruction.
- Send the operands to the RS if they are available in either RF or in the ROB. (speculative register read)
- The number of the ROB entry is also sent to the RS to tag the result when it is placed on the CDB.
- If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries.

So what is issue to when you deal with an issue operation, you are to get an instruction from the instruction queue. Once you get the instruction from the queue, issue the instruction if there is an empty reservation station and an empty slot and reorder buffer. This is the new extra point.

So the issue operation in a speculative execution is slightly different. I need to check for 2 entries. Do I have a space in the reservation station associated with the operation, it said my operation is multiplication and how to check whether there is an entry free in the reservation station of multiplier. If so I have a space there. Second, if my multiplication is over, do I have a space in the reorder buffer where I can temporarily store the result.

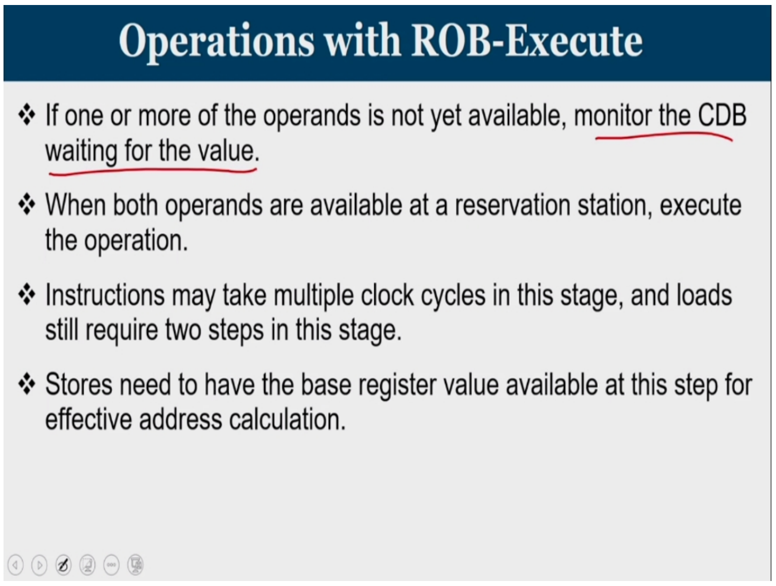
So I am pre booking a slot in the reorder buffer and then get a number, okay you are number is reorder buffer number 10. So, I will tell you whatever is the result that is been done by the multiplier unit kindly write into slot number 10 in reorder buffer. So 2 things are needed an entry in the reservation station and an entry in the reorder buffer. And the entry in the reorder buffer is used to tag the result, any output that is produced by your functional unit has to be returned to reorder buffer directly not to register file.

And when you write you should know where to write and that is what is called the entry in the reorder buffer. So you have to get a slot number or tag from the reorder buffer for every instruction and send the operands to the reservation station if they are available, either in the register file or in the reorder buffer. If you are taking a value from the reorder buffer that it is called a speculator register read.

The value is no longer available in the reorder buffer, then, you can go and take it from the register file. The number of reorder buffer entries also send to the reservation station to tag the results. That is why whatever is the value that you get the reorder buffer entry that is also added into the reservation station such that at the end of the operation, the result is forwarded to the appropriate reorder buffer entry.

If either all the reservation stations are full, or if the reorder buffer is full, the instruction issue is stalled. So, once the issue is stalled, all other pending instructions are also stalled because issue should happen in order.

(Refer Slide Time: 27:57)

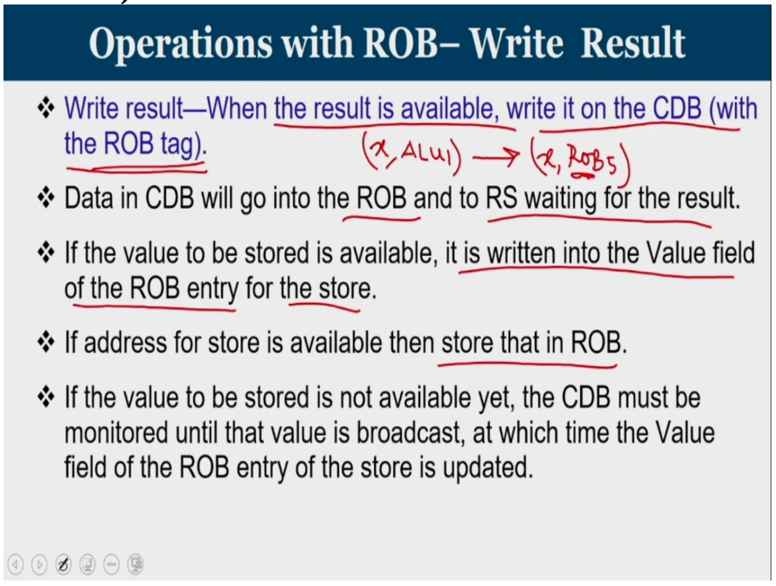


Operations with ROB-Execute

- If one or more of the operands is not yet available, monitor the CDB waiting for the value.
- When both operands are available at a reservation station, execute the operation.
- Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage.
- Stores need to have the base register value available at this step for effective address calculation.

So, what about execute, if one or more of operands is not yet available, then you have to monitor CDB waiting for the value. When both operands are available at the reservation station, you perform the execution of the operation. Instruction may take multiple clock cycles in this stage and loads still require 2 steps, one is address computation and 1 is accessing memory. Stores need to have base register value available to perform execute, because address calculation has to be done.

(Refer Slide Time: 28:28)



Operations with ROB– Write Result

- Write result—When the result is available, write it on the CDB (with the ROB tag).

$(x, ALU1) \longrightarrow (x, ROB5)$

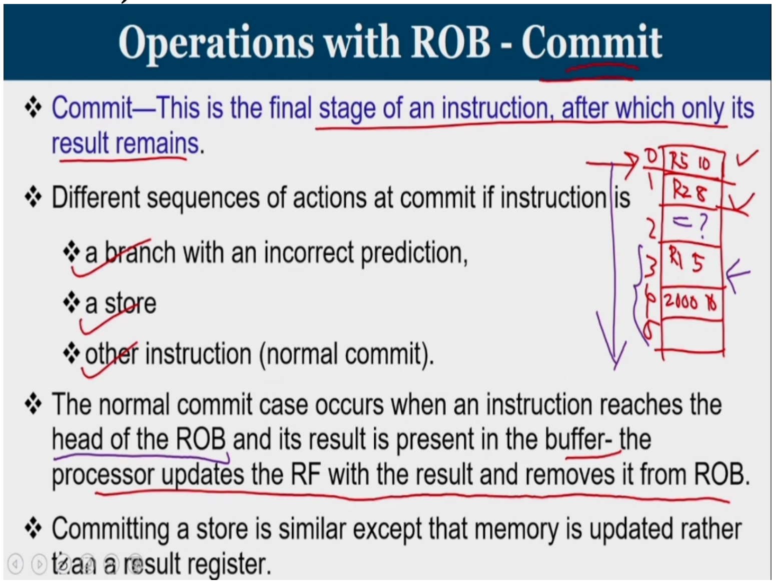
- Data in CDB will go into the ROB and to RS waiting for the result.
- If the value to be stored is available, it is written into the Value field of the ROB entry for the store.
- If address for store is available then store that in ROB.
- If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated.

And what will you do with the write result. In the write result, when the result is available, you have to write into CDB. Here is the difference. You write into CDB with reorder buffer entry, let us ALU 1 that is going to produce a result in previous algorithm in the ALU Tomasulo's algorithm, we are telling ALU 1 produce the result x. And now the difference is x is the value produced and that has to be returned to reorder buffer entry five.

So, rather than telling which functional unit produce the result, it is going to tell you what is the reorder buffer entry. So data in CDB will go to this reorder buffer entry that has been mentioned and to reservation station waiting for the result. If the value to be stored is available, it is returned to the value field of the reorder buffer entry for the store. If the address for the store is available, then you will store the address portion in the reorder buffer.

If the value to be stored is not yet available at the CDB must be monitored until that value is broadcasted at which time the value field of the reorder buffer entry of the store is updated.

(Refer Slide Time: 29:44)



Operations with ROB - Commit

- Commit—This is the final stage of an instruction, after which only its result remains.
- Different sequences of actions at commit if instruction is
  - a branch with an incorrect prediction,
  - a store
  - other instruction (normal commit).
- The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer- the processor updates the RF with the result and removes it from ROB.
- Committing a store is similar except that memory is updated rather than a result register.

So let us look into this additional last stage called commit. So in commit stage, it is a final stage of an instruction after which only is the result remains. So, once the instruction is committed, meaning the memory or registers are updated, you cannot revoke this action anymore. So commit happens only if you are sure that an instruction is no longer speculative. So all instructions which are executed, hoping that they will happen based upon the outcome of the branch predictor.

The result of all these instructions will be temporarily stored in the reorder buffer as on when you come to know that these instructions are no longer speculative means outcome of branch is known and whatever you have done is really required, then transfer the content from reorder buffer into memory or registers.

And there are different sequence of action that you have to do for a committed operation, if it is a branch instruction you have to do certain things. If it is a store operation, you have to do certain things. Or if it is any other operation that is also what to do. The normal commit case occurs when an instruction reaches the head of reorder buffer.

And its result is present in the buffer. The processor updates the register file with the result and remove it from reorder buffer. So consider, let me draw the reorder buffer here. So let us say the reorder buffer numbers 0, 1, 2, 3, 4, 5. Imagine a case where your reorder buffer entry 3 that value you got let us say 5 and then you got the reorder buffer entry where you are telling you are to write something into R2 and the value is 8.

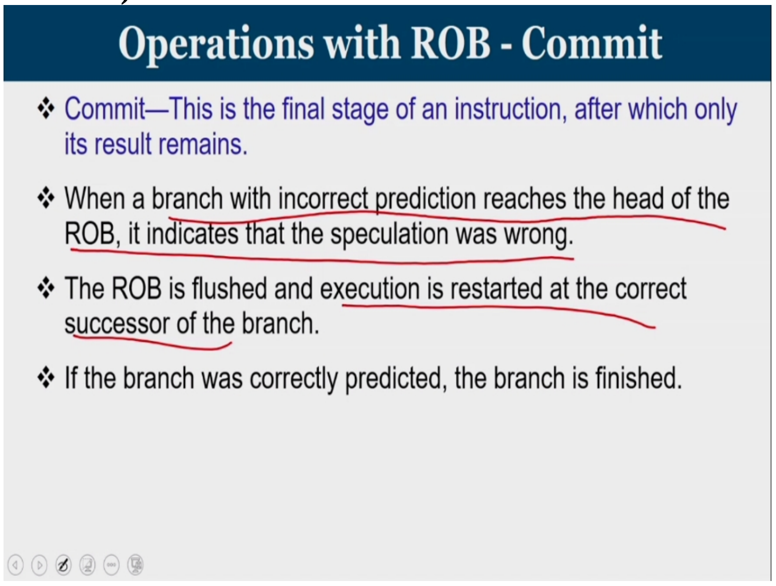
Then sixth instruction then the fourth instruction is getting complete it is telling you have to write some value to location 2000 as 10 and then you got a value that is there

$$R5 = 10$$

Now, the very first entry in the reorder buffer that is now complete that means this instruction is committed once that is committed this is committed, but I cannot commit you are R1 5 because that has not reached the front of the reorder buffer.

So, an instruction will get committed only if that instruction reaches the head of reorder buffer that has been mentioned over here. So all instructions 3, 4, 5 etc., will be there and reorder buffer until you get the value that is available for the reorder buffer entry 2. Because commit always happens in-, this is the program order commit happens always inorder. Committing a store similar to accept what has been mentioned rather than writing into registers. Now you are going to update the memory.

(Refer Slide Time: 32:26)



Operations with ROB - Commit

- Commit—This is the final stage of an instruction, after which only its result remains.
- When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong.
- The ROB is flushed and execution is restarted at the correct successor of the branch.
- If the branch was correctly predicted, the branch is finished.

And what will happen when there is a branch instruction, when I have a branch with the incorrect prediction reaches the head of reorder buffer, it indicates that speculation was wrong. Then the reorder buffer entry is flushed and the execution is restarted and the correct successor of the branch. So, the moment you come to know that your branch instruction was having an incorrect prediction flush out all entries, flush out all speculative instructions.

If the branch is correctly predicted the branch is considered to be finished. So, we had a quick glimpse of how the concept of speculative execution happens. So additional controls circuitries and storage is are required on conventional Tomasulo’s algorithm. So as to improve it to speculative execution. We learned about concept of reorder buffers and reorder buffer is a place where values of speculative instructions are been temporarily stored and during commit is always in order.

So when you look into the concept of dynamic scheduling, you fetch in-order, decode in-order, issue in-order, execute out of order, complete result out of order, but commit in-order, that is the order in which things are been happening. So with this, we come to the end of lecture number ten which talks about hardware based speculation. Thank you.