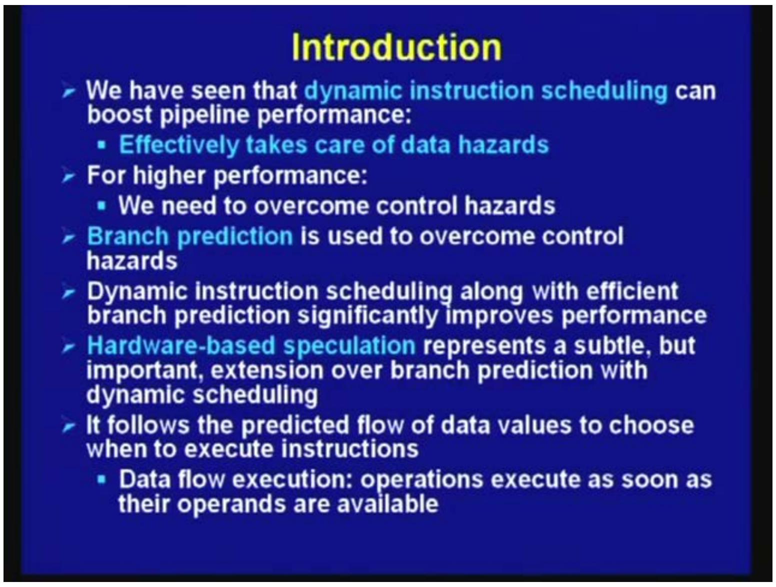
High Performance Computer Architecture

Prof. Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 19
Hardware-Based Speculation

(Refer Slide Time: 01:01)



Introduction

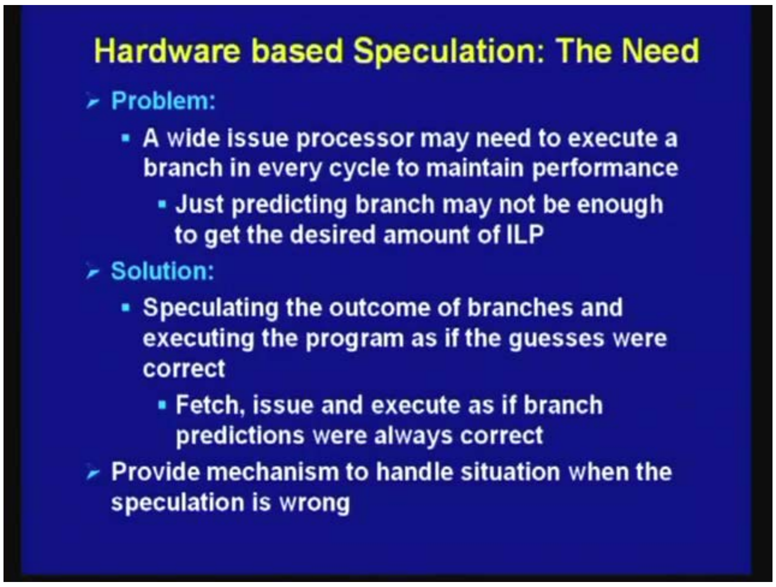
- We have seen that dynamic instruction scheduling can boost pipeline performance:
  - Effectively takes care of data hazards
- For higher performance:
  - We need to overcome control hazards
- Branch prediction is used to overcome control hazards
- Dynamic instruction scheduling along with efficient branch prediction significantly improves performance
- Hardware-based speculation represents a subtle, but important, extension over branch prediction with dynamic scheduling
- It follows the predicted flow of data values to choose when to execute instructions
  - Data flow execution: operations execute as soon as their operands are available

Hello and welcome to today’s lecture on Hardware-Based speculation. We have discussed various techniques, we improve the instruction level parallelism and we have seen that, dynamic instruction scheduling can boost pipeline performance, particularly it effectively takes care of data hazards. Then we have discussed various control hazards for higher performance and particularly for higher performance, it is very important to control hazards that we have seen. And we have introduce the concept of branch prediction to overcome control hazards and by using sophistic I mean, suitable technique of branch prediction we have seen, how the performance can be improved.

And in the last lecture, we have discussed dynamic instruction scheduling along with efficient branch prediction and we have seen, how it can significantly improves performance. Now, hardware based speculation represents a subtle, so it represents a subtle, but very important extension of the idea of hardware based instruction scheduling with branch prediction. So, it is say, extension of the previous techniques that we discussed in the last lecture, this an a extension of our branch prediction with dynamic scheduling.

What it does, it follows the predicted flow of data values to choose when to execute instruction. In other words, it is a kind of data flow execution and operations execute as soon as their operands are available. So, it is not that way much different from the techniques that I discussed in last lecture that is, dynamic instruction scheduling along with efficient branch prediction, but there will be some important differences. And we shall see, what are the differences and how this dynamic instruction scheduling along with I mean, hardware based speculation is introduced.

(Refer Slide Time: 03:18)



Hardware based Speculation: The Need

- Problem:
  - A wide issue processor may need to execute a branch in every cycle to maintain performance
    - Just predicting branch may not be enough to get the desired amount of ILP
- Solution:
  - Speculating the outcome of branches and executing the program as if the guesses were correct
    - Fetch, issue and execute as if branch predictions were always correct
- Provide mechanism to handle situation when the speculation is wrong

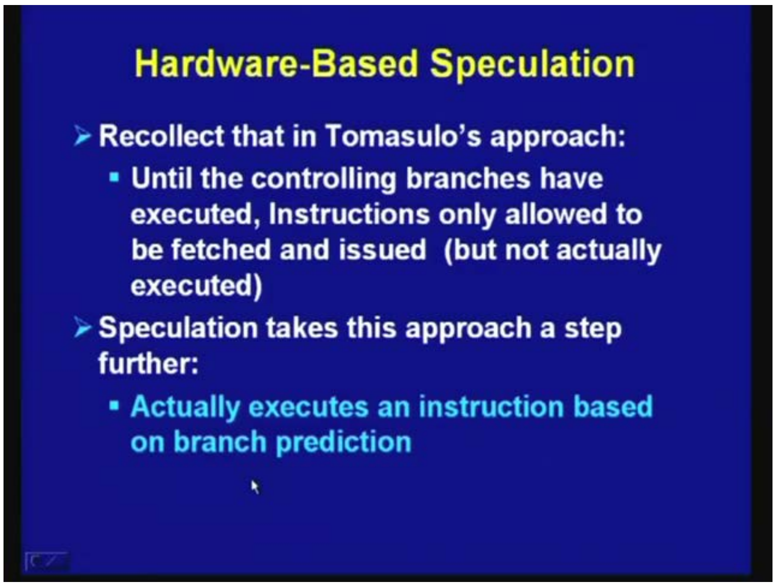
The need for hardware based speculation is coming from, particularly from wide issue processors. So, you have seen the superscalar processor, which are considered as wide issue processor that means, your issuing more than one instruction per cycle. So, in such cases we will find that, depending upon the application, the branch frequency that is present in the program, for in each cycle, one will encounter the branch instruction. That means, a wide issue processor may need to execute a branch in every cycle to maintain performance.

And just predicting branch may not be enough to get the desired amount of instruction level parallelism. That means, the instruction level parallelism that can be achieved simply by branch prediction is not enough, will not be able to keep various processing elements for functional units that is present in a superscalar processor. And we have seen in superscalar processor, you got multiple functional units, it is essential to keep them busy to get higher throughput.

So, the solution is speculating the outcome of branches and executing the program, as if guesses were correct. Earlier, the execution when not performed at the boundaries of control dependences, particularly until the control defenses were not resolved, subsequent instructions were not executed. But, here where going a step further, so execution will continue in the speculated direction and it will perform I mean, it will assume that, this speculation is correct and it will perform fetch issue and execute, as if branch predication were always correct.

So, you are going ahead with execution of instructions assuming that, your speculation is correct. However, whenever you do that, you have to provide mechanism to handle situation when the speculation is not wrong that means, speculation is a kind of guess, guess may turnover to correct, it may turnover to wrong. So, whenever it turns out to be wrong, you have to take appropriate step such that, you do not get incorrect result. That means, you have to undo the execution that has been performed and how that can be done, we shall discuss in detail.

(Refer Slide Time: 06:24)



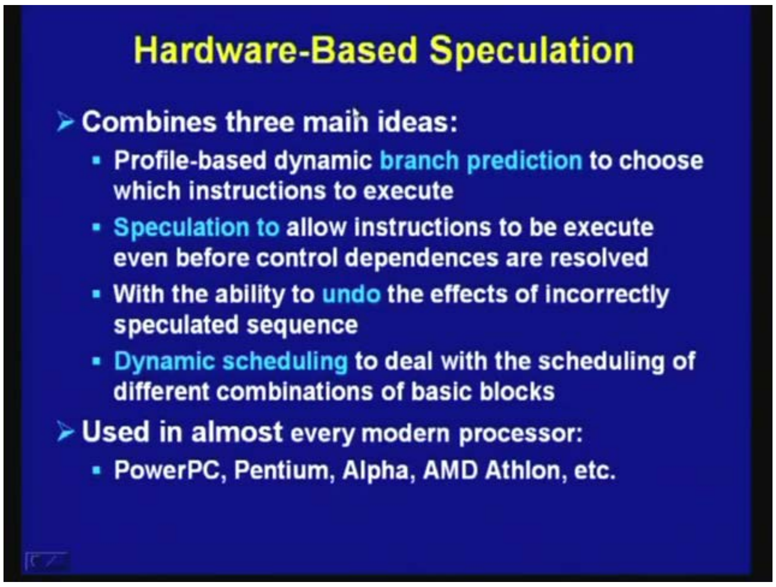
Hardware-Based Speculation

- Recollect that in Tomasulo's approach:
  - Until the controlling branches have executed, Instructions only allowed to be fetched and issued (but not actually executed)
- Speculation takes this approach a step further:
  - Actually executes an instruction based on branch prediction

In a recollect tomasulo's approach, which you have discussed earlier, so until the controlling branches have executed, instructions only allowed to be fetched and issued, but not actually executed. That means, the controlling branches must be executed only then the instruction can be executed. But before that, it can be fetched, it can be issued, multiple issue can take place, but execution will not take place. And as I mentioned, speculation takes this approach a step further, it will actually executes an instruction based on branch prediction.

So, here we are also doing branch prediction, but in the direction of branch prediction, earlier execution were not continued, but here we are continuing.

(Refer Slide Time: 07:19)



Hardware-Based Speculation

- Combines three main ideas:
  - Profile-based dynamic branch prediction to choose which instructions to execute
  - Speculation to allow instructions to be execute even before control dependences are resolved
  - With the ability to undo the effects of incorrectly speculated sequence
  - Dynamic scheduling to deal with the scheduling of different combinations of basic blocks
- Used in almost every modern processor:
  - PowerPC, Pentium, Alpha, AMD Athlon, etc.

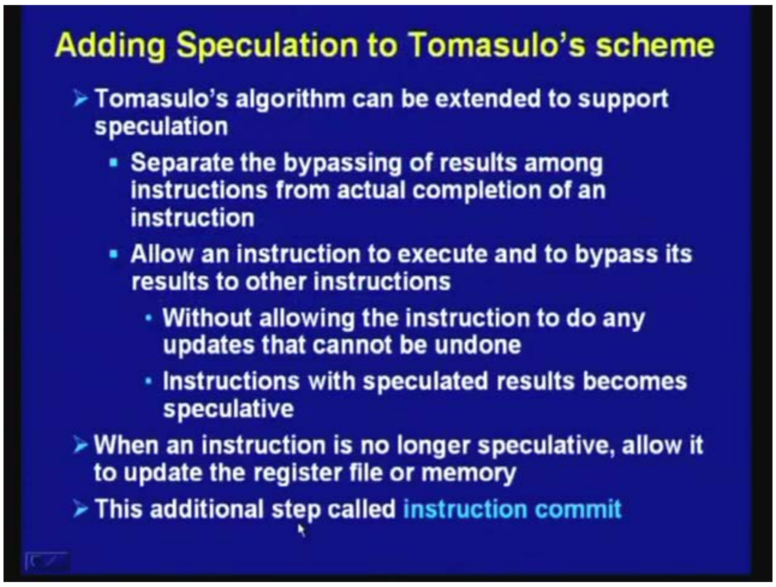
And essentially, this hardware based speculation combines three ideas, the three ideas are number 1 is profile based dynamic branch predication, to choose which instruction to execute. So, you will be using some profile based prediction technique, may be that correlating predictor or tournament predictor, some subsequent prediction technique, to choose which instruction to execute, then speculation to allow instruction to execute even before control dependences are resolved.

So, earlier I mean, dynamic scheduling with branch prediction, we were not allowing this, but now we are allowing to execute before control dependences are resolved. So, earlier, the instruction were not execute until the control dependences were resolved, but this is the step we are doing in case of hardware based speculation. Now obviously, in such a situation as I have told, this can be done with the ability to undo the effects of incorrectly speculated sequences.

Earlier, what was the meaning of execution, by execution what we are doing, we are performing with the execution, writing the result in the memory or register, but if you do that, you cannot undo it. So, permanent irrevocable changes technique place, so that has to be somehow stuffed and dynamic scheduling to deal with the scheduling of different combination of basic blocks. So, it is doing branch prediction, speculation along with dynamic scheduling to deal with the scheduling of different combination of basic blocks.

So, at the branch point, you will be going through different direction and you will encounter different basic blocks. So, in the predicted direction, it will continue to execute the basic blocks and this particular approach have been used in almost all modern processors like power pc, Pentium, alpha, AMD athlon, etcetera, almost all processors, modern processors.

(Refer Slide Time: 09:51)



Adding Speculation to Tomasulo's scheme

- Tomasulo's algorithm can be extended to support speculation
  - Separate the bypassing of results among instructions from actual completion of an instruction
  - Allow an instruction to execute and to bypass its results to other instructions
    - Without allowing the instruction to do any updates that cannot be undone
    - Instructions with speculated results becomes speculative
- When an instruction is no longer speculative, allow it to update the register file or memory
- This additional step called instruction commit

Now, what we shall try to do, we have already discussed in this in details, the tomasulo's scheme, but tomasulo's scheme cannot be exactly followed, we have to made some modifications. So, what kind of modification can be done, you can extend the tomasulo's algorithm to support the speculation by doing two things. Number 1 is separate the bypassing of results among instruction from actual completion of an instruction, so what we can do, we can execute an instruction, produce result.

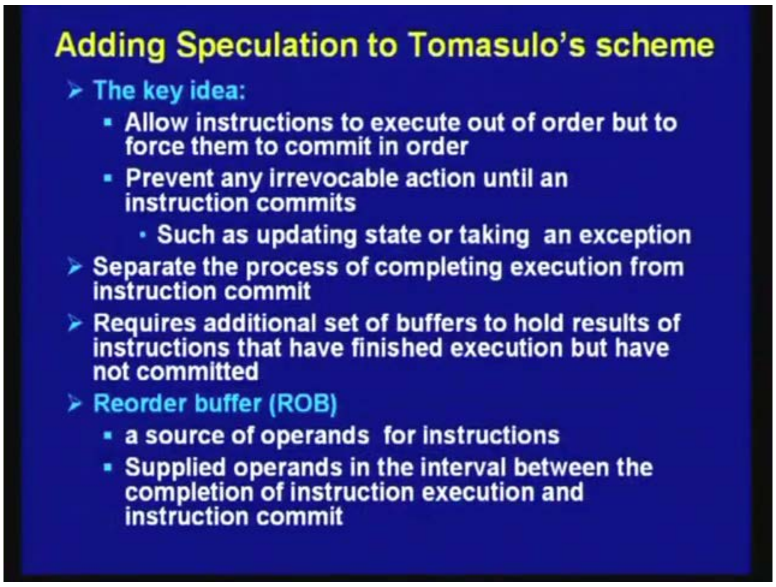
But, those results will not be written in the memory or in registers, but they will continue to pass values to other instruction as when they need. And another step you have to follow, allow an instruction to execute and to bypass it is result to other instructions as I have told, without allowing the instruction to do any updates that cannot be undone. That cannot be undone means, if you perform writing into a register or in a memory location, that cannot be undone, that is a permanent change.

So, that will be allowed instruction with speculated result becomes speculative that means, it remains speculative until writing is done. When it is been done, when an instruction is no longer speculative that means, the outcome of branches is already decided then it is no longer speculative. Now, we know that, these instruction were supposed to be executed, so only then when an instruction is no longer speculative, allow it to update the register file or memory.

So, we have to reach that point when an instruction is no longer speculative, it is actually known that, that instruction will be I mean, those control instructions that means, the condition has been evaluated, the target at this is known, only when these two are known, we know whether an instruction is speculative or not speculative. So, when these two are known then an instruction is no longer speculative, only then you can make the parliament changes.

And this additional step is called instruction commit that means, what we are doing, we are breaking up the execution into two parts. We are performing the execution, we are doing the competition in a shadow, but permanent changes taking place in the commit stage, we are adding in additional stage known as commit stage.

(Refer Slide Time: 13:04)



Adding Speculation to Tomasulo's scheme

- The key idea:
  - Allow instructions to execute out of order but to force them to commit in order
  - Prevent any irrevocable action until an instruction commits
    - Such as updating state or taking an exception
- Separate the process of completing execution from instruction commit
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- Reorder buffer (ROB)
  - a source of operands for instructions
  - Supplied operands in the interval between the completion of instruction execution and instruction commit

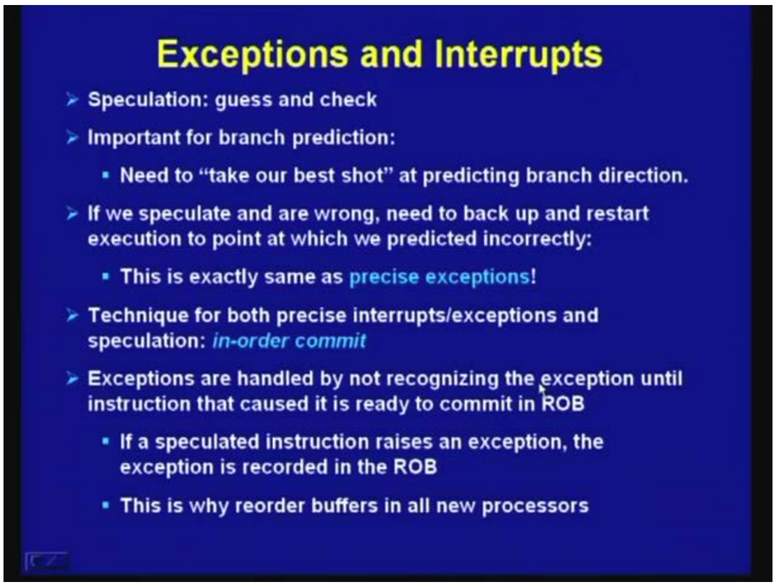
So, the key idea is allow instructions to execute out of order, but to force them the commit in order, so this is a very important statement. That means, we are allowing the instruction to execute out of order, but we are in forcing the instructions to commit in order, as it is present in your program order such as, updating state or taking an exception. These are done only if, in this will prevent any irrevocable action until an instruction commits.

So, this is the basic idea behind this hardware based speculation, when it is implemented on top of the basic tomasulo's approach. So, you have to separate the process of completing execution from instruction commit. So, completion after instruction means, an instruction have been executed, it has produce result and commit means, although results have been produced, the result have not been return into the registers or in memory location, that is performed in the commit stage.

So obviously, this will require additional set of buffers to hold results of instruction that have finished execution, but have not committed. That means, he will obviously, without any additional buffer, you cannot do it, you have to give the results value generated by instructions to be passed on to other instruction, unless you store it in some buffer, you cannot do it. So, you will require addition buffer and from those buffers, it will pass on to other instructions and that particular buffer is known as reorder buffer.

A source of operands for instructions, so reorder buffers will be source of operands for instructions, it will supply operands in the interval between be completing of instruction execution and instruction commit. That means, even before the I mean, the memory or register update take place, from the reorder buffer, it will provide the operands for different instruction. And for this duration, that between the completing of instruction execution and instruction commit, reorder buffer will be the sources of operands.

(Refer Slide Time: 15:59)



Exceptions and Interrupts

- Speculation: guess and check
- Important for branch prediction:
  - Need to "take our best shot" at predicting branch direction.
- If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly:
  - This is exactly same as precise exceptions!
- Technique for both precise interrupts/exceptions and speculation: *in-order commit*
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
  - If a speculated instruction raises an exception, the exception is recorded in the ROB
  - This is why reorder buffers in all new processors

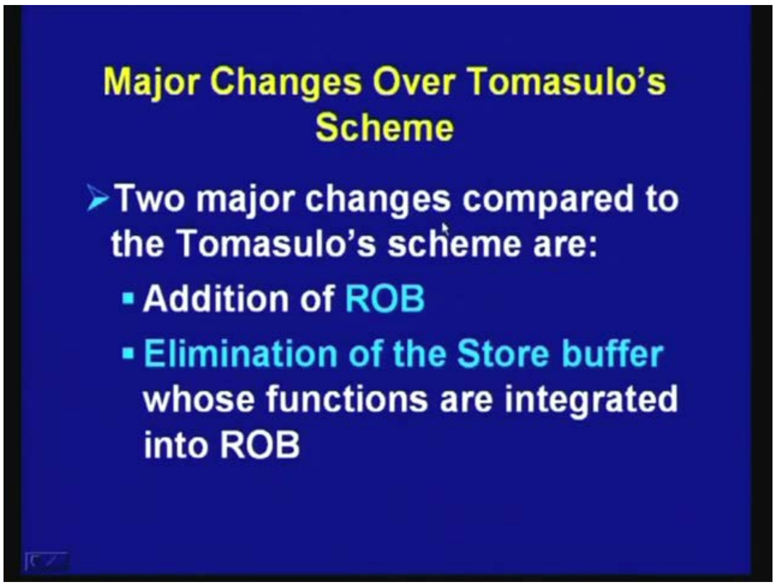
Of course, you have to take into account an exception and interrupts as well, speculation means, it will guess and check important for branch prediction, need to take our best shot at predicting branch direction. So, this is a very important thing, first of all your performance will heavily dependent on the correctness of your prediction. So, you have to use sophisticated branch prediction techniques and if we speculate and are wrong, need to backup and restart execution at the point, at fetch we predicted incorrectly and this is exactly the same as precise exception.

That means, if we can do that then this will overcome the precise exceptions, so precise exceptions essentially wants this and if something is wrong, we have to back up and restart execution to the point at which we predictor incorrectly. So, this is how the precise exception are overcome and techniques for both precise exception and interrupts exception and speculation is same. Essentially, by using this approach in order commit, we are able to overcome the precise interrupts and exceptions.

An exceptions are handled by not recognizing the exception, until instruction that caused it is ready to commit in ROB. So that means, whenever an exception is detected and it is kept pending, until it reaches the commit stage in the ROB. So, whenever before commiting is done and if a particular instruction need to be discarded then the exception operations to perform is also discarded. So, both are discarded, not only the instruction, but the exception conditions.

So, if a speculated instruction raises an exception, the exception is recorded in the ROB and that is why that means, the ROB keeps an information about the exception. And if it needs to undone then whenever it will be I mean, we can keep that information until commit and only when it is committed then that exception is allowed to happened. So, that is why, reorder buffer is present in all new processors.

(Refer Slide Time: 18:49)

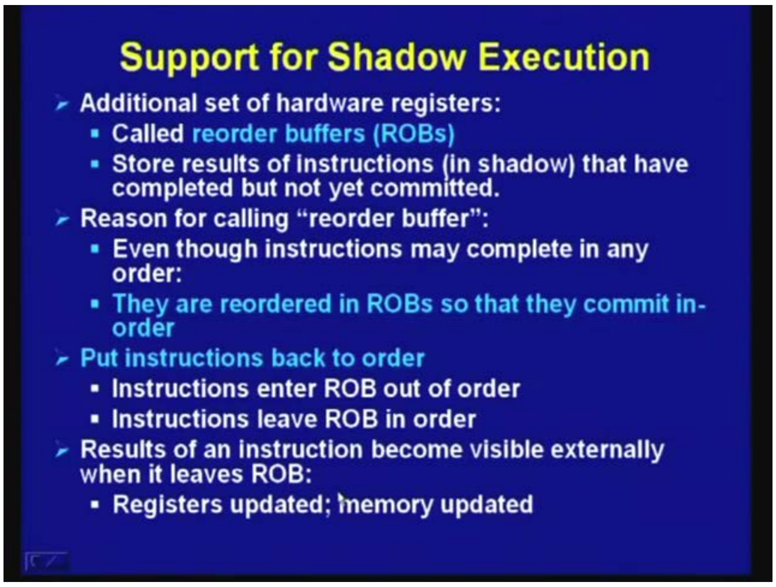


Major Changes Over Tomasulo's Scheme

- Two major changes compared to the Tomasulo's scheme are:
  - Addition of ROB
  - Elimination of the Store buffer whose functions are integrated into ROB

So, whenever we go for reorder buffers, there are two basic changes, two major changes compared to the tomasulo's scheme, number is he will require reorder buffers that is, an addition. And elimination of the store buffer, earlier we were having, you may recall that, in our tomasulo's basic approach, we had store buffer and load buffer separate. But now, the store buffers are no longer required, the reason for that is, the store buffer function is taken care of, why the buffers present in reorder buffer. So, this leads to elimination of the store buffer, whose functions are integrated in the reorder buffers.

(Refer Slide Time: 19:43)



Support for Shadow Execution

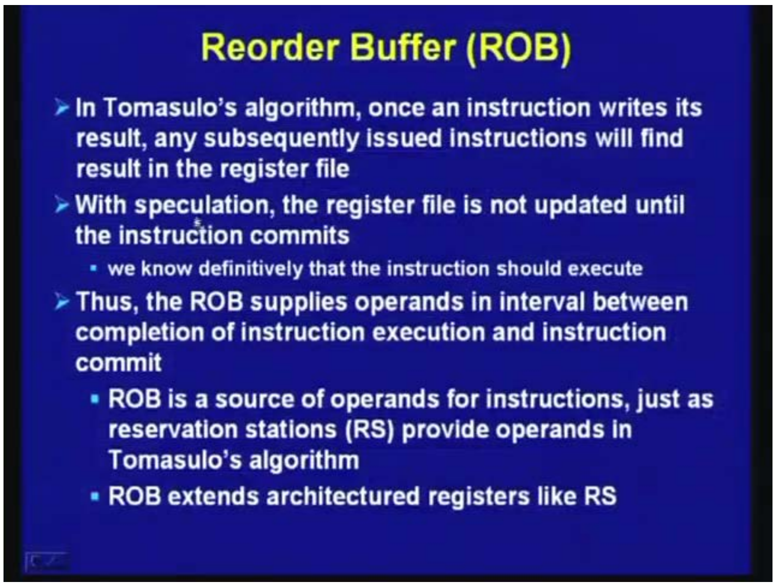
- Additional set of hardware registers:
  - Called reorder buffers (ROBs)
  - Store results of instructions (in shadow) that have completed but not yet committed.
- Reason for calling "reorder buffer":
  - Even though instructions may complete in any order:
  - They are reordered in ROBs so that they commit in-order
- Put instructions back to order
  - Instructions enter ROB out of order
  - Instructions leave ROB in order
- Results of an instruction become visible externally when it leaves ROB:
  - Registers updated; memory updated

So, you require additional set of registers called reorder buffers, store results of instructions in shadow that have completed, but not yet committed. Why we are calling them reorder buffer, there is a name assigned to it, the reason for that is, even though instructions may complete in any order, they are reordered in reorder buffer so that, they can committee in order. That means, instructions are executed, their results are stored, but in the order, in which commit is changed with the help of this reorder buffer and that is why, the name calling it reorder buffer.

So, it puts instruction back to order, instruction enter ROB out of order, instructions leave ROB in order. By in order, out of order mean, with respect to the program order, in the program they seen, this sequence in which they appear with respect to that and results of an instruction becomes visible externally when it leaves ROB. So, until an instruction leaves ROB, it does not become visible to outside world, outside world means, to the programmer, who is a executing it is program.

For example, if a program does single stepping, executed instruction one of the other, so in a such case, until that register operation or memory operation is performed, it will not be visible to the programmer. So, that is why, results of an instruction become visible externally when it leaves ROB and that means, registers updated and memory updated.

(Refer Slide Time: 21:38)



Reorder Buffer (ROB)

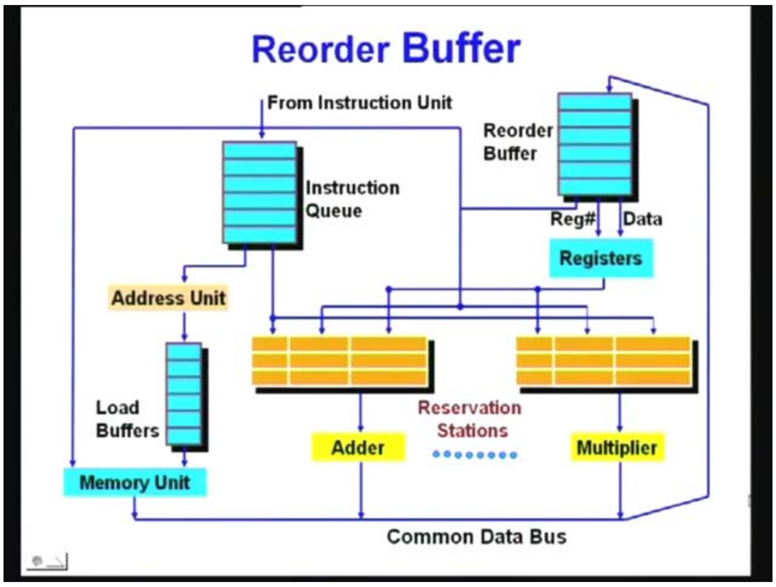
- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
  - we know definitively that the instruction should execute
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
  - ROB extends architectured registers like RS

So, in case of tomasulo's algorithm, once an instruction writes, it is result, any subsequently issued instructions will find result in the register file with speculation, the register file is not updated until the instruction commits, I have repeated in several times. And we know definitely that, instructions should execute thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit.

So, ROB is a source of operands for instructions just as reservation stations, earlier we were using reservation stations to provide the operands to the functional units, you may call that, but now, we are using reorder buffers, from where the operands are available. Now, essentially what it is doing, ROB extends the architecture registers like reservation station. So, as I mention earlier, reservation stations are performing the role of additional register, which is not present in processer.

Similarly, reorder buffers are also essentially extending the register set, registers not available in the processer architecture is now may be I mean, it is partially available in the reorder buffers. So, you may consider, it extends architecture registers like reservation stations.

(Refer Slide Time: 23:22)



Reorder Buffer

So, this is the schematic diagram, this shows how the tomasulo’s algorithm I mean, have been extended, you may recall that, we had one store buffer here. But, that store buffer has been removed, in place of that we have put reorder buffers at the top and from the you can see, reorder buffer is connected to the common data bus. That means, whenever in the functional units, memory unit or adder unit or multiplier unit, when they produce any result, they straight away go to the reorder buffer.

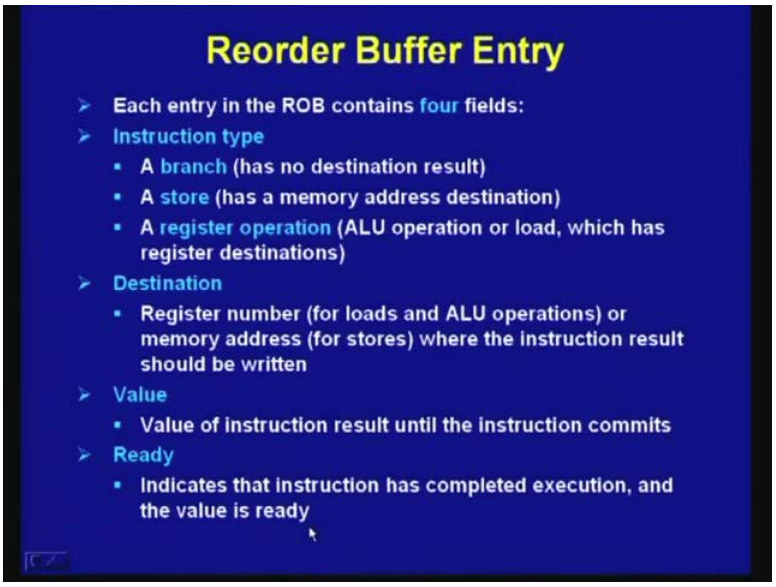
So, they are not going earlier, they were going to different buffers present in the reservation stations, they were going to the other buffers. But, now we can see, that interconnection has been removed, it goes only to the reorder buffer and from the reorder buffer, they can be stored, they will go to the register, they will go to different reservation stations.

It will go to the reservation stations, it will also go to the memory unit I mean, in case of I have to write it I mean, if it has to written in the memory in case of stored I mean, from reorder buffer then it will be stored, that is why this store buffer is not present here, it will come from the reorder buffer. However, load buffers are necessary, because they will provide the load addresses, they will compute the addresses it will present here, then load buffer will be present here, it will come from the address unit.

And the data may be available from here, which will be written I mean, in case of load, address will come and then it will go to the reorder buffer and from where, it will go to the register, so this is how load operation will perform. You cannot really get rid up reservation stations, although we have reorder buffers, the reservation stations cannot be removed, because they will provide the operands into the multiplier. However, earlier you were keeping track of, from fetch functional units, the output will come to the reservation stations.

But here, you have to keep track, you have to keep kind of tag, from which reorder buffer will come I mean, there is a reorder buffer number. Reorder buffer 1, 2, 3 and so on and it is a entered in the form of a first in first out, FIFO and the reorder buffer number will be tagged here. And so from the corresponding reorder buffer number, operands will come and then it will be stored in the reservation stations and from the reservation stations, it will go to the different functional units. So, that part is not changed and that is the reason, why we cannot get rid up reservation stations. Although partially the functions earlier performed by reservation stations or the performing reorder buffers but not fully.

(Refer Slide Time: 27:03)



Reorder Buffer Entry

- Each entry in the ROB contains four fields:
- Instruction type
  - A branch (has no destination result)
  - A store (has a memory address destination)
  - A register operation (ALU operation or load, which has register destinations)
- Destination
  - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
- Value
  - Value of instruction result until the instruction commits
- Ready
  - Indicates that instruction has completed execution, and the value is ready

So, with this basic structure in mind let us see, what are the different entries that is present in the reorder buffer. So, each entry in the reorder buffer contains four fields, there will be 4 fields in reorder buffer, number 1 field is instruction type. The instructions can be broadly divided into 3 categories as you can see, number 1 is branch,

branch has no destination result. Second is a store has a memory address destination that means, you have to store in a memory location.

And branch means, simply it will jump a particular address, so there is no destination result and register operation, which are related to ALU operation or load. So, ALU operation on load is performing in the same way, as we have seen in this diagram. Whenever you perform ALU operation, it will be data will go to the registers, in the same way data will go to the memory unit, it will come from the reorder buffer. So, ALU operation and load is somewhat similar, it is different from store, so because they have registered destination.

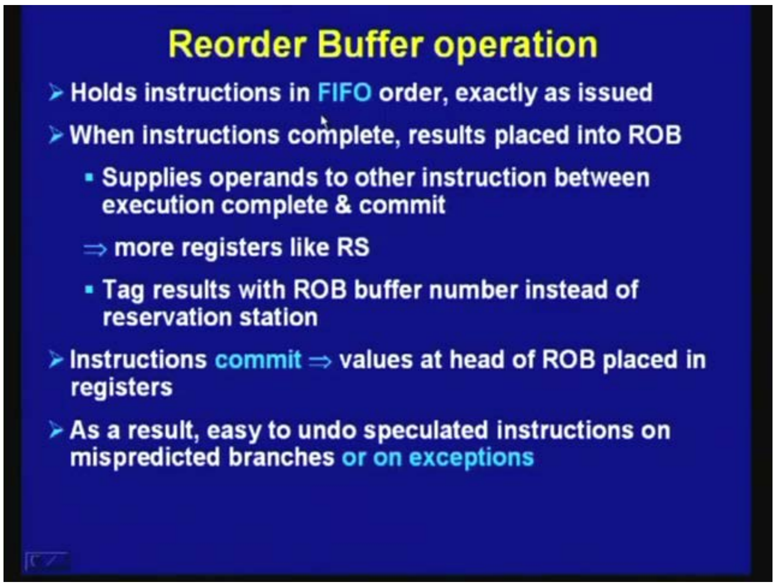
So, these are the three different instruction types and that you have to, that is kept in one of the fields of the reorder buffer. The second field is the destination, the registered number for loads and ALU operations or memory address for stores, where the instruction result should be written. So, there is a destination field, which gives information about the register number, in case of register operation and memory address in case of store, where the instructions result should be written.

Then, the third field is a value, value of instruction result until the instruction commits, so as I mentioned repeatedly, from the common data bus, the values will come to the reorder buffer, as the functional units does some computation and the computation result, those values are stored in the reorder buffer and this is the field, where the value will be stored. Then you have got another field, which is ready, so it indicates that, instructions has completed execution and the value is ready.

So, whenever an instructions is issued, it gets an entry in the reorder buffer and as it goes to two different stages like execution, right. Right means, right on the common data bus and until it commits, it remains in that particular reorder buffer and that is the reason, why you have to keep track of it. So, indicates that, instruction has complicated execution and value is ready, value is ready means, it has to be written, either in the register or in the memory location, to be written in the destination.

So, destination is known and when it is ready, it will go to the destination and the value is also known. So, the value is known when the ready flag becomes 1 then that value has to be written in the destination register or memory location. So, this is not cell, the information or the data structure that is being maintained inside the reorder buffer, these are the four fields, it is been done.

(Refer Slide Time: 30:51)



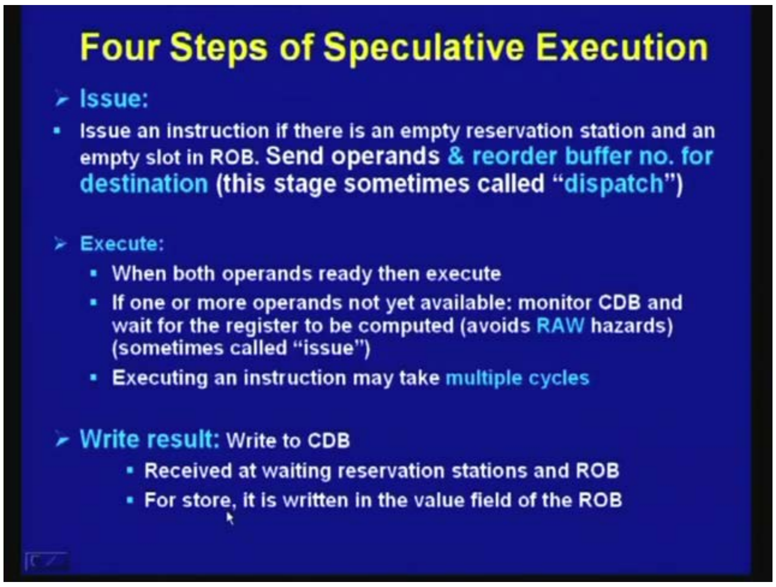
Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit
  - ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒ values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

Now, lets us see, so on the hold instructions in FIFO order as I told, first in first out order, exactly had issued that means, the way the instructions have been issued in the same order, it is kept in the reorder buffer in the form of first in first out I mean, that is your data structure, first in first out data structure. When instruction complete, results placed into ROB that means, supplies operands to other instructions between instruction execution complete and commit, as I already told.

So, he will requires more registers like reservation station that means, the number of registers that is present in the reorder buffers will be much more, that is present in the architecture registers present in the processor. And tag results with ROB buffer number instead of reservation station, so that means, ROB buffer number that means, you have to keep track, where the data will be available. So, that is done by keeping that result with ROB buffer, number instead of reservation station and instruction commit, when values at the head of ROB is placed in the registers. And as a result, it is easy to undo speculated instructions on mispredicted branches or exceptions, we shall discussed it in little more detailed later.

(Refer Slide Time: 32:31)



Four Steps of Speculative Execution

- **Issue:**
  - Issue an instruction if there is an empty reservation station and an empty slot in ROB. Send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
- **Execute:**
  - When both operands ready then execute
  - If one or more operands not yet available: monitor CDB and wait for the register to be computed (avoids RAW hazards) (sometimes called "issue")
  - Executing an instruction may take multiple cycles
- **Write result:** Write to CDB
  - Received at waiting reservation stations and ROB
  - For store, it is written in the value field of the ROB

Now, let us look at the four different steps that is being perform whenever you go for speculative execution or hardware-based speculation, so it is also called a speculative execution. So, first step is issue, so issue an instruction if there is an empty reservation station and an empty slot in ROB. So, here you see, structural hazard will result, if either there is no empty reservation station or there is no slot in the ROB. So, both has to be available for issuing an instruction and instruction cannot be issued, until both the things are available.

So, send operands and reorder buffer number for destination, this stage is sometimes called dispatch, so you may have heard of this term dispatch. So, this issue is essentially dispatch, which sends operands and reorder buffer number for destination. Then second step is execute, so whenever an instruction is issued, it can perform execution when both operands ready then it can be executed.

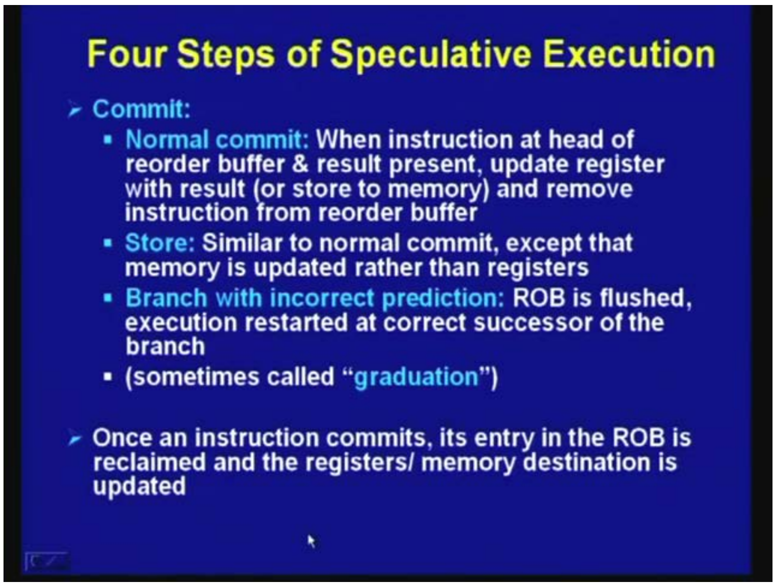
If one or more operands not yet available then what has to be done, we have seen, whenever an operand is not available, it will be obviously be produced by some functionally unit and that functional unit will write it on the common data bus. So, you have to keep on monitoring the common data bus, so monitor common data bus and wait for the register to be computed that means, it avoids read after write type of hazards and this particular step is sometime called as issue.

Then, executing an instruction may take multiple cycles, as I have already mentioned depending on the instruction type, if it is multiply, it will take more number of cycle than addition, if it is division then it will take still more number of cycles than multiply. So, that whenever execution takes place, it may take multiply cycles depending on the instruction type. Then comes the write result, here whenever we say write result, earlier write result was essentially writing into the memory location or registered, but here it is not so writing means, writing on the common data bus.

So, whenever execution is complete, the result can be written on the common data bus and it is received at waiting reservation stations and ROB. So, the data goes to the waiting reservation stations and ROB, obviously it goes to through ROB and for store, it is written in the value field of the ROB. So, if it is store instruction, we have seen that, store buffer is no longer present.

So, since store buffer is no longer present, it is written into the value field of the ROB and subsequently, in the commit stage, that writing into the memory will take place. It will store the data in the corresponding memory after the address is note.

(Refer Slide Time: 36:16)



Four Steps of Speculative Execution

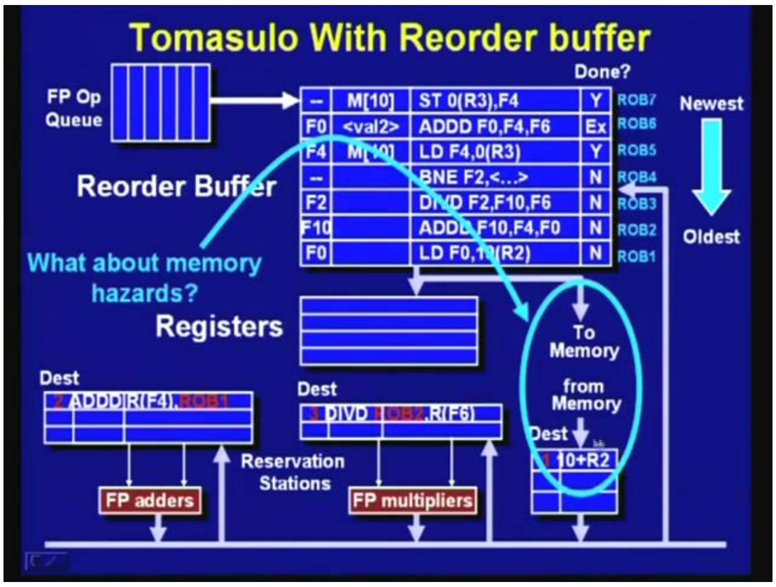
- Commit:
  - Normal commit: When instruction at head of reorder buffer & result present, update register with result (or store to memory) and remove instruction from reorder buffer
  - Store: Similar to normal commit, except that memory is updated rather than registers
  - Branch with incorrect prediction: ROB is flushed, execution restarted at correct successor of the branch
  - (sometimes called "graduation")
- Once an instruction commits, its entry in the ROB is reclaimed and the registers/ memory destination is updated

Then, comes the last step that is, the commit step, so commit can be broadly divided in three different types, normal commit. Normal commit, when instructions at head of reorder buffer and result is present that means, it will reach, it is FIFO First In First Out, it will reach the head of the reorder buffer at some point of time, as the clocks are progressing. And at that point, if the result are present, it will update register with result or store to memory and remove the instruction from the reorder buffer.

So, you see, as soon as an instruction is, in the first step, if that instruction is entered into the reorder buffer. And at the commit stage, at the end of the commit step, it is removed from the reorder buffer after performing permanent change in the processor. That means, writing into the register or in the memory location and store is done, is somewhat similar to normal commit, except that memory is updated, rather than registers. So, in the previous cases, we have seen that, in normal commit, essentially register are updated, but here you are updating memory.

Then, branch with incorrect prediction, so now you have to take necessary steps so that, a branch which was executed computation was performed, but it has now turnover to be incorrect. So, ROB is flushed, execution restarted at correct successor of the branch and this is sometimes called graduation, so commit step is sometime called graduation. So, once an instruction commits it is entry into the ROB is reclaimed and the registers memory destination is updated, as I have told, this is performed in the commit step.

(Refer Slide Time: 38:24)



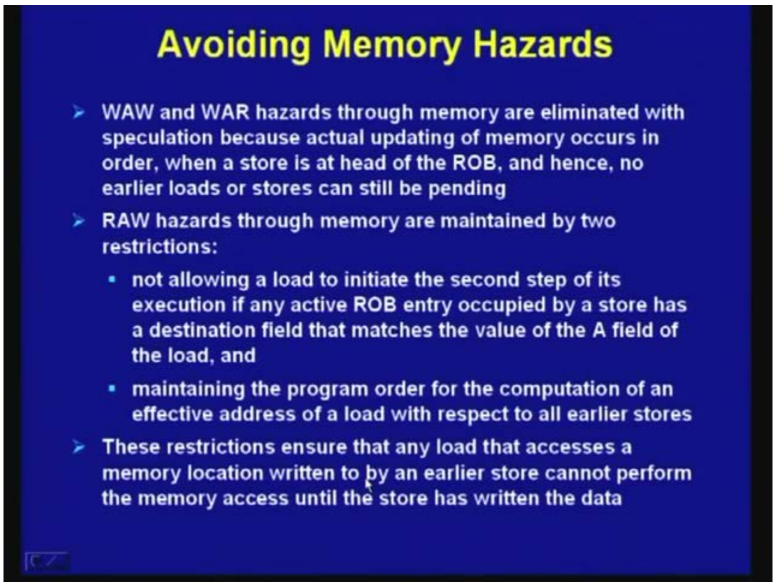
So, this is the simple I mean, this source how it takes place in different cycles, so first instruction, one instruction, load instruction has been entered in the ROB ReOrder Buffer number 1. And accordingly, destination registers in the reservation stations, update is perform, you see here it is mentioned ROB number 1, earlier it as mentioned that, execution unit that will produce result, but here it is not so it is tagged with the ROB. So, the ROB will write into the register F 0 and load operation will perform that is being shown.

So, the destination is F 0 and here you can see, it will be required in the second instruction and here that address, it will be loading into the, that destination address is 10 plus R 2 that is, the address from where loading will take place. So, that is you have to read it from memory, so it will go to the memory. Then the third instruction division, here accordingly, the updating is done in the reorder buffer and also in that store buffer and corresponding modification had done from memory.

And from memory means, this will read this from this, so that is been mentioned and the floating point adders, this addition is performed by these floating point adders and the corresponding entry into the reservation stations is performed and in the reservation stations corresponding to this deficient is performed here. So, 3, 2, these are all mentioned here that means, swap the result will go and from which reorder buffer it will get operand, these are all mentioned here in the reservation stations.

So, you can see, we are using reservation stations buffers and also we are using reorder buffers and in this way other instructions are entering the reorder buffers and the computation is performed. And you can see, how the values or being stored here, this is the value field where the values are stored and after performing addition, that value is also stored. So, you can see, this is how the computation is done, now what about memory hazards, because we have to overcome memory hazards and let us see, how the memory hazards are avoided.

(Refer Slide Time: 41:32)



Avoiding Memory Hazards

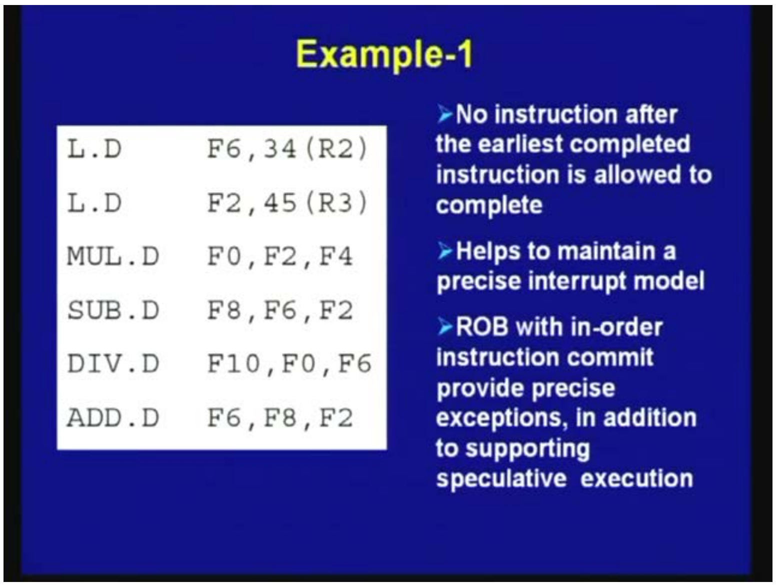
- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
- RAW hazards through memory are maintained by two restrictions:
    - not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a destination field that matches the value of the A field of the load, and
    - maintaining the program order for the computation of an effective address of a load with respect to all earlier stores
- These restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

WAW and WAR hazards through memory are eliminated with speculation, because actual updating of memory occurs in order, when store is at head of ROB and hence, no earlier loads of store can still be pending. So, you can see that, in reorder buffer, since it is here the entity take place in order in the same program order and since the commit operation is performed with the help of the ROB, so these two hazards, WAW and WAR type of hazards through memory are completely eliminated.

And read after write hazards through memory are eliminated by two restrictions, what are the two restriction, by not allowing a load to initiate the second step of it is execution, if any active ROB entry occupied by a store as a destination field that matches the value of the A field of the load, so this is how read after write hazard is eliminated. Another is maintaining the program order for the computation of an effective address of a load with respect to an earlier stores, so the read after write hazards are also overcome in this way.

So, these restrictions ensure that, any load that accesses a memory location written to or by an earlier store, cannot perform the memory access, until the store has written the data. So, this is how, by committing in order, you are able to overcome all the different types of hazards that may occur, while execute programs.

(Refer Slide Time: 43:22)



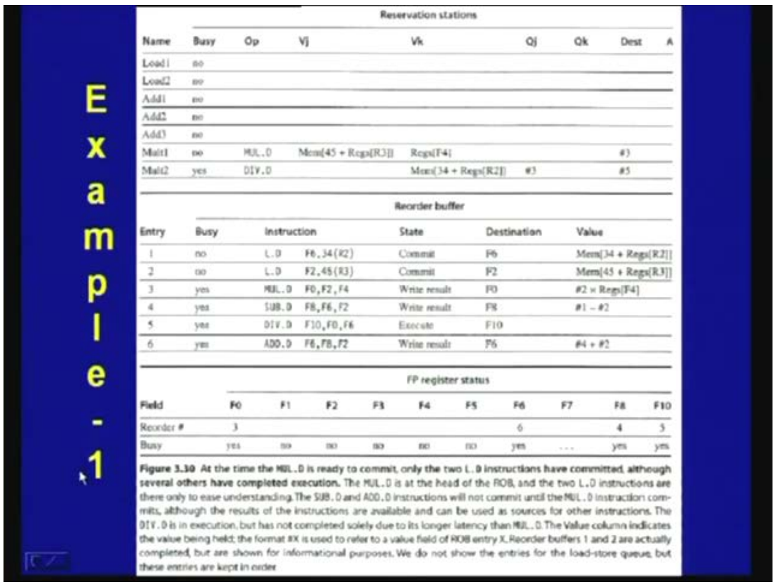
Example-1

```
L.D     F6,34(R2)
L.D     F2,45(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F6,F2
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

- No instruction after the earliest completed instruction is allowed to complete
- Helps to maintain a precise interrupt model
- ROB with in-order instruction commit provide precise exceptions, in addition to supporting speculative execution

So, let us take up few examples, so this is a simple example, this is a straight line code, two load followed by, multiple followed by, it is followed by subtraction, division and then addition.

(Refer Slide Time: 43:38)



Example-1

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34 + Regs[R2]]	#3		#5	

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6,34(R2)	Commit	F6	Mem[34 + Regs[R2]]
2	no	L.D	F2,45(R3)	Commit	F2	Mem[45 + Regs[R3]]
3	yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8,F6,F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10,F0,F6	Execute	F10	
6	yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	. . .	yes	yes

Figure 3.30 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although several others have completed execution. The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed, but are shown for informational purposes. We do not show the entries for the load-store queue, but these entries are kept in order

Let us see, this is the, in this particular case as you can see, only the first two load instruction has completed and it has perform commit, so it has perform commit and the result has been written into the registers F 6 and F 2. So, the value that was obtained is given here, 34 plus the content of the register R 2, that was added to get the address and from there, the value was obtained and that is being written into the register F 6. Similarly, the value was obtain from the memory location by adding the content of R 3 with 45 and then after getting the value, it was written into the register.

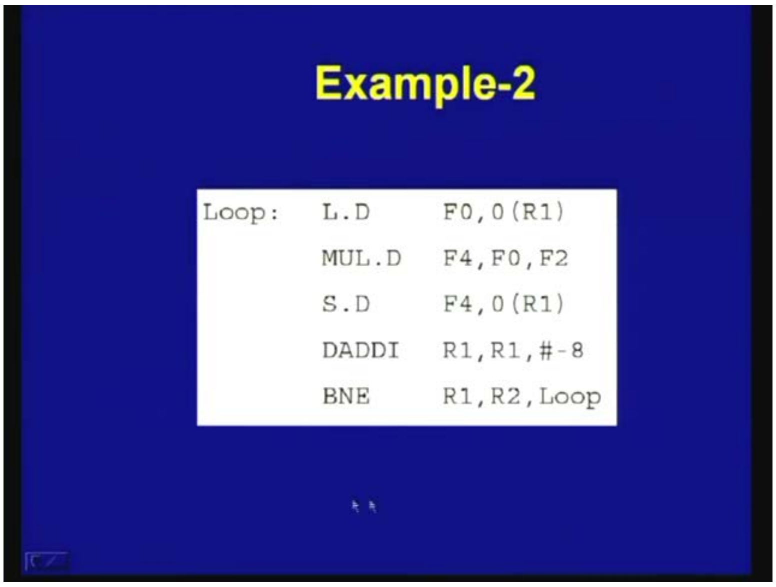
And although the other instructions are I mean, the execution has started, although several other some computed execution, the multiplication is at the head of the ROB. So, these two have completed, now this one is at the head of the ROB and this will take some time to get to complete. So, the subtraction D, this particular instruction, 4 th instruction and the instruction 6, these two, they will take smaller time.

So, they have already completed their execution, so but they are not allowed to commit due to the letters you have multiplied D. Because, if their committing is done in order, since this is at the head of the ROB, unless this performs commit, the other two although they have produced result, they have completed execution, but they will not perform the commits step.

The value column indicates the value being held, so here you can see, this particular value being held is here and the value for this instruction is also held here, so the values available, which can be passed on to other instructions. If other instructions need them, but they will not be committed into the registers, until this multiplication D is performed, which is at the head of the ROB at this point. So, the value column indicates the values, but as shown in the informed, we do not show the entries for the load store cube, but the entries are kept in order.

So, this is a snap shot of a particular step, when we have seen these two instructions multiplication and division, these two instructions have not yet completed executions, others have already completed execution. And the state of reorder buffer is shown here and the status of the floating point registers are also shown in this particular diagram.

(Refer Slide Time: 47:08)

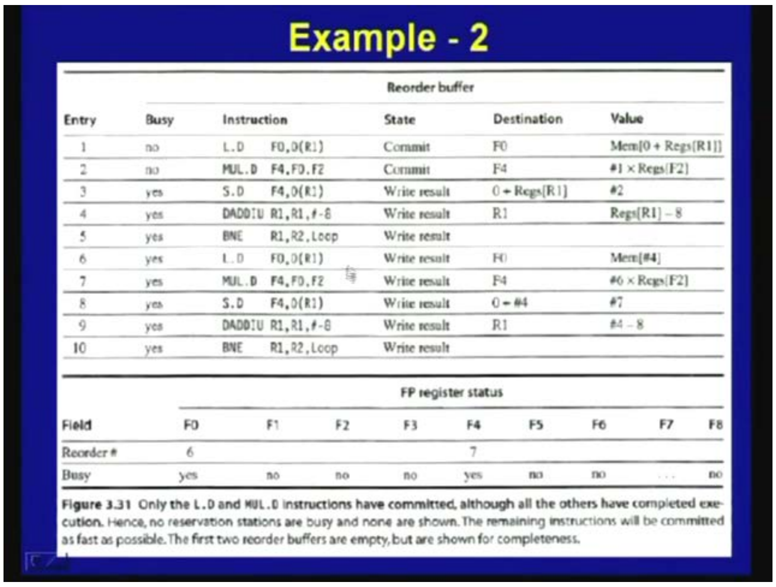


Example-2

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

Now, let us consider another example involving a loop, so this particular instruction is involving a loop. And we have seen, we can go I mean, beyond the loop boundary in the predicted direction.

(Refer Slide Time: 47:22)



Example - 2

	Reorder buffer				
Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	no	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D F4,0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDIU R1,R1,#-8	Write result	R1	Regs[R1] – 8
5	yes	BNE R1,R2,Loop	Write result		
6	yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D F4,0(R1)	Write result	0 + #4	#7
9	yes	DADDIU R1,R1,#-8	Write result	R1	#4 – 8
10	yes	BNE R1,R2,Loop	Write result		

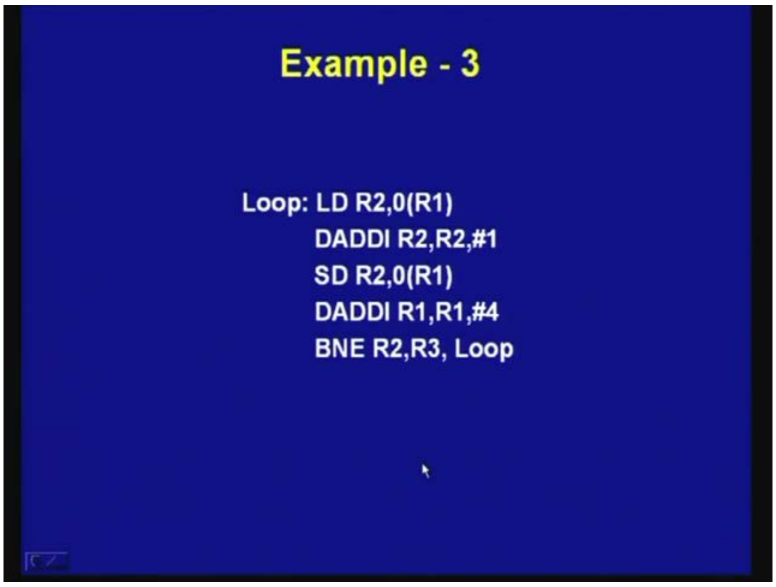
	FP register status								
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	. . .	no

Figure 3.31 Only the L.D and MUL.D instructions have committed, although all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

And this is the reorder buffer and these are the F register status, we can see that, these two instructions have committed load and multiple D has committed. Committed means, they perform writing into the registers F 0 and F 4 and however, the other instructions have, although all the other instructions have already completed the executions and hence, no reservation stations are busy and none are shown. Reservations stations are not shown here, the remaining instructions will be committed as fast as possible.

So, these instructions, execution have been completed, but they have not yet been committed, so they will be committing as soon as possible. And the first two reorder buffers are empty, but are shown for completeness that means, these two reorder buffers are empty, but it is shown here, they are no longer busy, you can see the reorder buffers. So, you can fill two more instruction here and they can be pushed out, so these two are, now instruction 3 is at the head of the buffer. So, this commit will be performed and then this commit will be performed, so casually one after the other the commit operations will perform.

(Refer Slide Time: 48:56)

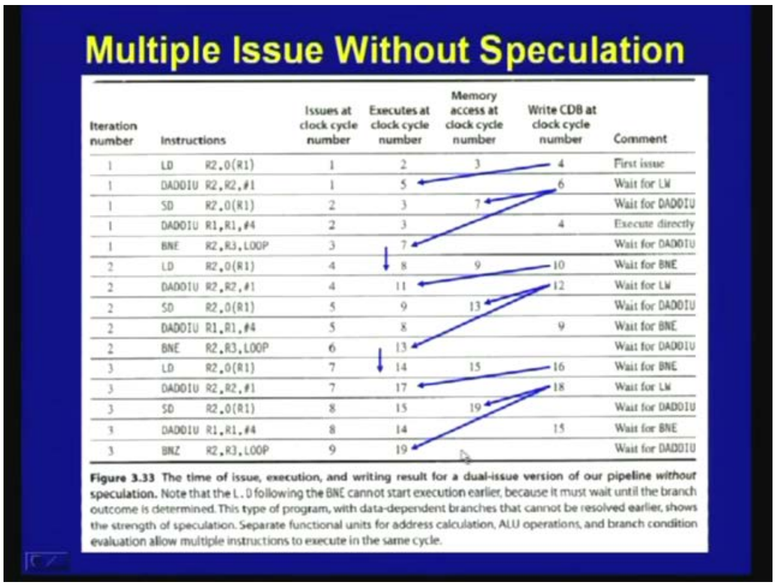


Example - 3

```
Loop: LD R2,0(R1)
      DADDI R2,R2,#1
      SD R2,0(R1)
      DADDI R1,R1,#4
      BNE R2,R3, Loop
```

So, this is a third instruction, third example, here we shall explain with the help of super scalar. That means, issuing two instruction and we shall show, how issuing can be done without speculation and with speculation.

(Refer Slide Time: 49:14)



Multiple Issue Without Speculation

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Figure 3.33 The time of issue, execution, and writing result for a dual-issue version of our pipeline without speculation. Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.

So, this is the multiple issue without speculation, so you can see, your issuing two instruction at a time I mean, across the loop. So, first two instructions are issued then the second two instructions are issued then third instruction is issued. Unfortunately I mean, 3 rd cycle this BNE instruction is issued, but unfortunately in the 3 rd cycle, you cannot issue this instruction, because there is a loop dependent I mean, this instruction is dependent on that.

So, the LD following BNE cannot start execution, because it must wait until the branch outcome is determined, so this type of program with data dependent branches. So, you can see, data produced by one loop is being used by another loop, that because of this dependency you can see, in the 3 rd cycle, you cannot issue this instruction. And later on, we shall see in the speculation step, when we shall so multiple issue in speculation, they are, how this can be overcome.

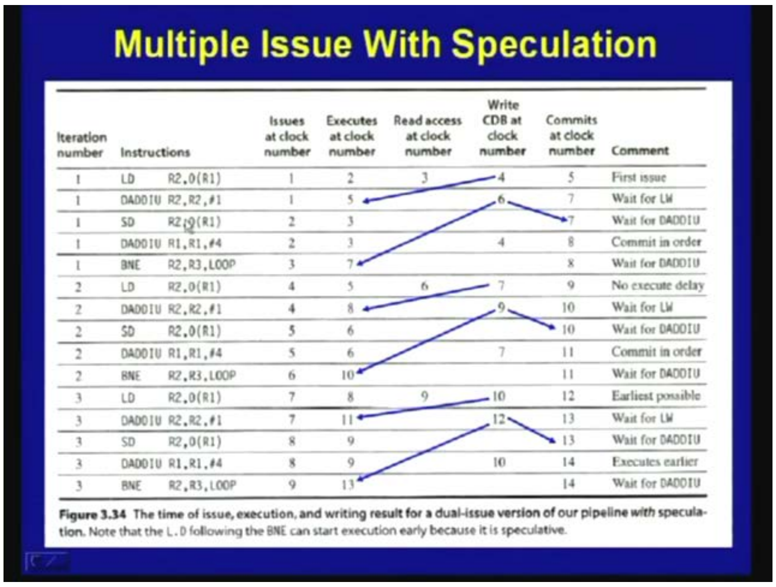
So, it will proceed in this way, in the different clock cycles, various operations will be perform and it will write the common DB at the clock cycle here, this is without speculation, so reorder buffer is not present. So, here performing everything with the help of reservations stations and it is writing into the common database. So then it is performing in the 5 th clock cycle, it gets the operand and in the 6 th clock cycle, it will perform the execution and these two are dependent on that, so the results will be passed on these two instruction.

And in the 7 th cycle, they will perform the execution and in this way you can see, it will proceed and however, as you can see, here it has to wait for one cycle, because of this data dependent branches. So, one cycle will be wasted here and later on, we shall see whenever we do it with speculation then this cycle will not be wasted, because it did not wait for this particular instruction, this control dependences are over come. So, this will proceed only when control dependences are overcome and that is reason, why you require this particular, this addition on cycle.

It cannot be issued, load instruction cannot be issued unless the control dependences are resolved. So, in this way, it will proceed 10 cycle, 12 cycle, it will complete this. Then it will provide data upto this store data and is the dependent on that, so it will get the corresponding data then here again, it has to wait till this control dependences is resolved. So, in this way, it will continue in the 16 th cycle, 14 th cycle, this load will initiate, it will effect two cycles to complete and then it will provide data to this particular instruction.

And then in the 18 th cycle, this will complete execution and data will go to that and in this way you can see, the multiple issue without speculation is performing. And it is requiring 19 cycles to perform the execution of this instructions that means, three locks are getting executed in 19 cycles.

(Refer Slide Time: 53:11)



Multiple Issue With Speculation

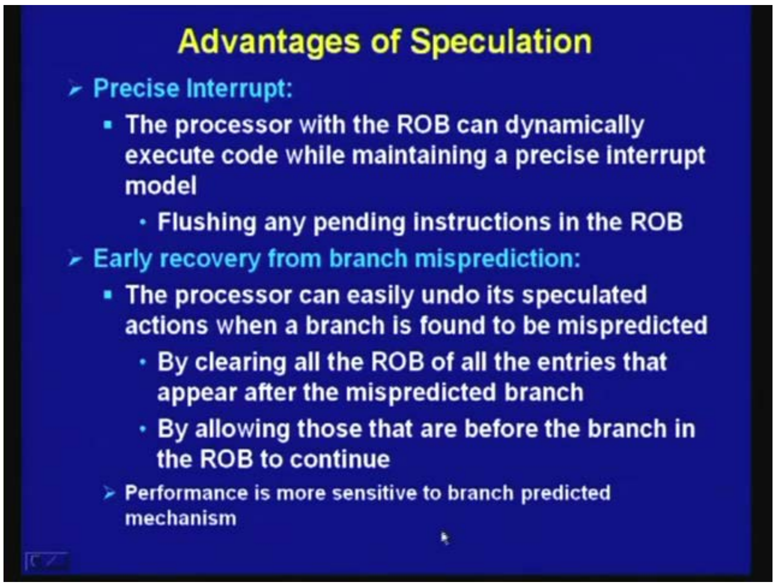
Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#4	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#4	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 3.34 **The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation.** Note that the L.D following the BNE can start execution early because it is speculative.

Now, let us see, how long it takes whenever you perform multiple issue with speculation, so here, different cycles as shown when an instruction is executed and when the read access is performed and when committee is being performed. So, first part is identical, but here you can see, this load instruction has already when issued, it did not wait for this control dependence to be resolved, because we are doing it with speculation. So, although this execution is completed in 7 cycle, but it has been issued in the 5 th cycle.

Execution here is done in the 4 th cycle, issued is 4 th cycle, but execution has taken place in the 5 th cycle. So, you can see, the advantage of speculation is demonstrated here, so in this way, it will proceed and you can see in 14 cycles, it will perform computation, compared to earlier we have seen, it required 19 cycles. So, you are gaining 5 cycles to complete execution of 3 loops by using speculation.

(Refer Slide Time: 54:37)



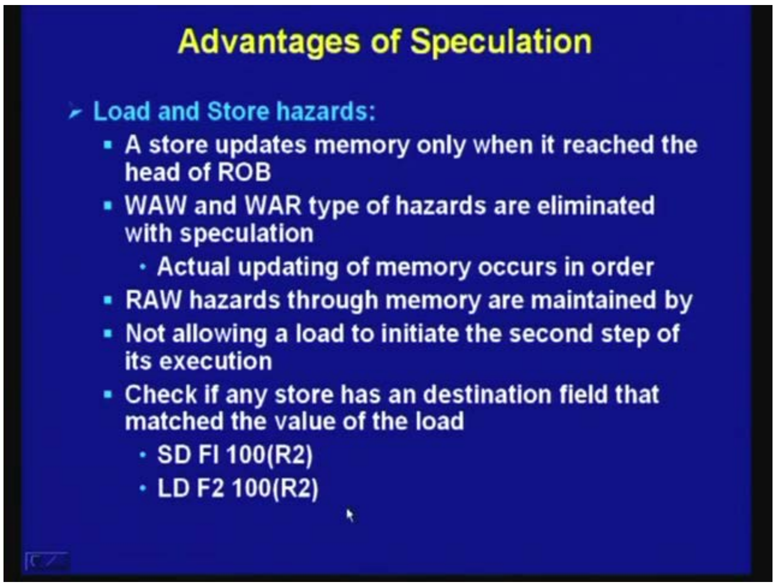
Advantages of Speculation

- Precise Interrupt:
  - The processor with the ROB can dynamically execute code while maintaining a precise interrupt model
    - Flushing any pending instructions in the ROB
- Early recovery from branch misprediction:
  - The processor can easily undo its speculated actions when a branch is found to be mispredicted
    - By clearing all the ROB of all the entries that appear after the mispredicted branch
    - By allowing those that are before the branch in the ROB to continue
  - Performance is more sensitive to branch predicted mechanism

Now, very quickly let us look at the advantages of speculation, the process are with ROB can dynamically execute code while maintaining a precise interrupt order, and flushing any pending instructions in the ROB, so it maintains precise interrupt order. Early recovery from branch misprediction, so if there is a misprediction, the processor can easily undo speculated actions, when a branch is found to be mispredicted.

How it is does, by clearing all the ROB of all the entries that appear after the mispredicted branch. That means, after the mispredicted branch, all the other instructions are first out, but all the instructions before that mispredicted branch are retained. So, by allowing those, that are before the branch in the ROB to continue, so these flushing is done. And as a consequence, it can recover early from branch misprediction and particularly, the performance is heavily dependent on the branch prediction technique that I have already discussed.

(Refer Slide Time: 55:41)



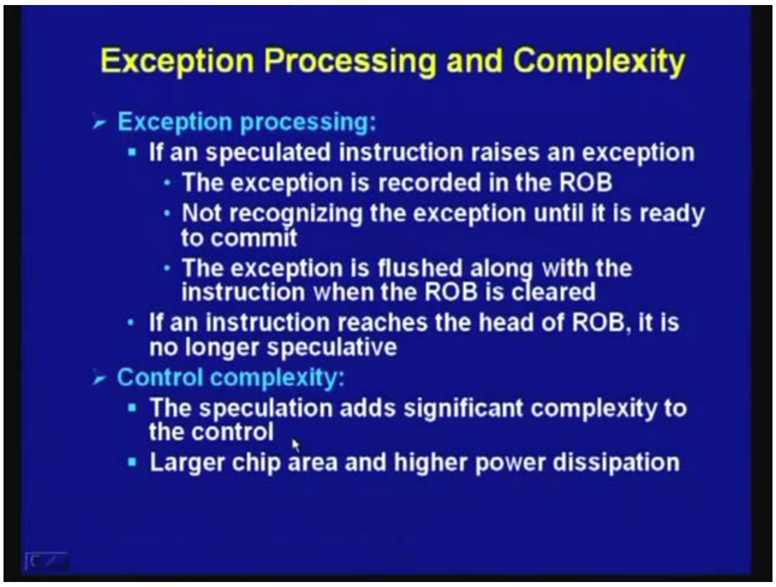
Advantages of Speculation

- Load and Store hazards:
  - A store updates memory only when it reached the head of ROB
  - WAW and WAR type of hazards are eliminated with speculation
    - Actual updating of memory occurs in order
  - RAW hazards through memory are maintained by
  - Not allowing a load to initiate the second step of its execution
  - Check if any store has an destination field that matched the value of the load
    - SD FI 100(R2)
    - LD F2 100(R2)

Then, load and store hazards are eliminated I have already discussed in detail, a store updates memory, only when it reaches the head of ROB. WAW and WAR type of hazards are eliminated with speculation and actual updating of memory occurs in order as I have told. And RAW hazards through memory are maintained and how it is maintained, not allowing a load to initiate the second step of it is execution.

And check, if any store has an destination field that matches the value of the load, as it is shown with the help of these instructions. So, you can see, the destination address is same for this load and store, so this checking is performed that effective at this calculation is done.

(Refer Slide Time: 56:31)



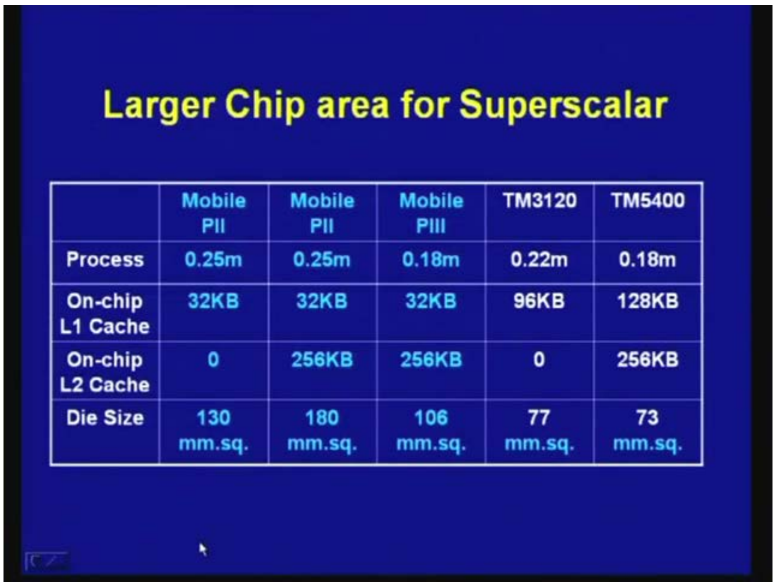
Exception Processing and Complexity

- Exception processing:
  - If an speculated instruction raises an exception
    - The exception is recorded in the ROB
    - Not recognizing the exception until it is ready to commit
    - The exception is flushed along with the instruction when the ROB is cleared
  - If an instruction reaches the head of ROB, it is no longer speculative
- Control complexity:
  - The speculation adds significant complexity to the control
  - Larger chip area and higher power dissipation

An exception processing is performed, if a specified speculated instruction raises an exception, the exception is recorded in the ROB as I told earlier and not recognizing the exception until it is ready to commit. So, until it reaches the head of the ROB, it does not perform the commit, so the exception is flushed along with the instruction, when the ROB is cleared.

And if an instruction reaches the head of ROB, it is no longer speculative, however all these things whenever you include, it had significant complexity to the control. That means, the control unit of the processor becomes very complex and as a result, it will lead to larger chip area and higher power dissipation.

(Refer Slide Time: 57:28)



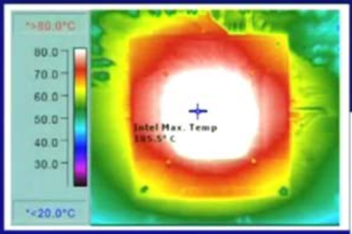
Larger Chip area for Superscalar

	Mobile PII	Mobile PII	Mobile PIII	TM3120	TM5400
Process	0.25m	0.25m	0.18m	0.22m	0.18m
On-chip L1 Cache	32KB	32KB	32KB	96KB	128KB
On-chip L2 Cache	0	256KB	256KB	0	256KB
Die Size	130 mm.sq.	180 mm.sq.	106 mm.sq.	77 mm.sq.	73 mm.sq.

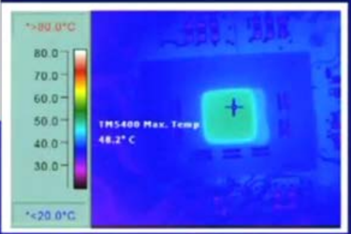
As it is a evident from this particular table, we can see this Pentium processors, which are best on super scalar architecture, they are requiring significantly larger chip area 130 millimeter square, 180 millimeter square, 106 millimeter square, in spite of the reduce dimensions used I mean, as the device size is reducing, so this is the area. On the other hand, for the processors which are not super scalar, which are VLIW, I have already mentioned about them, they require significantly larger chip area.

(Refer Slide Time: 58:16)

Comparison of Heat Dissipation



➢A Pentium III processor plays a DVD at 105°C



➢ A Crusoe processor model TM5400 plays a DVD at 48°C

So, chip area is more and particularly we will see, because of larger chip area, the power dissipation is also more. So, it is comparison of heat dissipation is shown here, this is corresponding to Pentium III processor playing a DVD at and the temperature profile is shown, which reaches 105 degree centigrade. On the other hand, in case of Crusoe processor, which is best on VLIW architecture, not a superscalar architecture, same program is running, but core temperature is reaching only 48 degree centigrade.

So, we have discussed in detail various techniques, which are used in implementing superscalar processors, in spite of the fact, they consume a larger chip area and high power dissipations, they are very popular. But, we are gradually reaching a point of diminishing return or we are reaching a point, where beyond which you cannot proceed, because power dissipation is reaching very high level.

So, what is the other alternative available, you will see the other alternative is not increasing the number of five line stages, not increasing the frequency, but to go for multi core, which I shall discuss in my subsequent lectures. But, before that, in my next lecture, we shall start with memory hierarchy, because performance of processor is not only dependent on the processor, it also depends on the memory, so that we shall start our discussion in the next lecture.

Thank you.