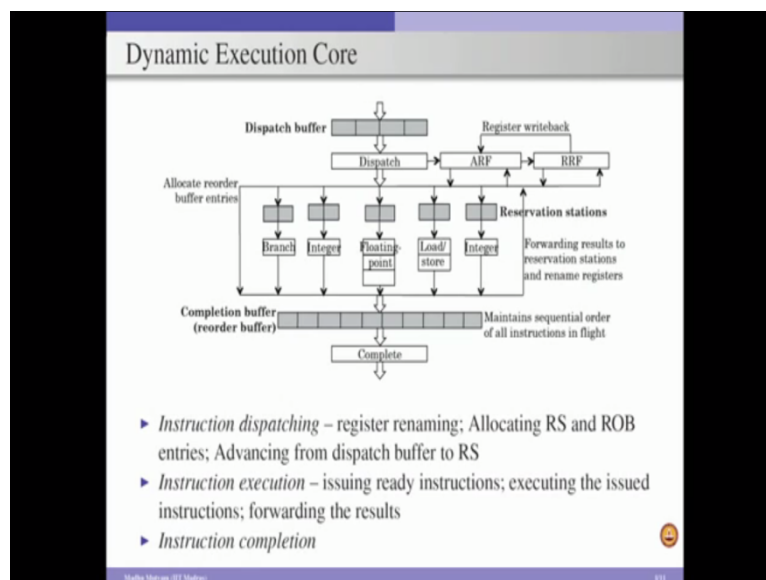


**Computer Architecture**  
**Prof. Madhu Mutyam**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Module – 07**  
**Lecture – 26**  
**Dynamic Execution Core**

In this module we are going to discuss one of the key components of Superscalar processors that is dynamic execution core. So, we know that in the Superscalar processor we have the Inorder fetch, Inorder decode, Inorder dispatch, but we have out of order execution. And finally, Inorder commit the Inorder retirement will be there. This dynamic execution core is the one which is actually implements the out of order execution of the instructions.

(Refer Slide Time: 00:50)



And this dynamic execution core actually resembles a refinement of Tomasulo implementation. So, we know that execute stage is sandwiched between the dispatch stage and the complete stage in our Superscalar pipeline design. So, in the dispatch stage we have the decoded instructions and these decoded instructions will be dispatched to various reservation stations. And at the time of dispatch the operands may be ready. So, we take the operands, if they are ready and dispatch these operands along with the instructions to the reservation station. And if the operands are not ready then we will dispatch the register tags along with the instruction to the reservation stations. So, when the operands are ready we can

read the operands either from the architectural register file or from the renamed register file. And that will be fed here to the reservation stations.

Again among the several reservation stations, we are going to select a suitable reservation station based on the type of the instruction. And each of these reservation stations is going to take care of issuing the instructions that are stored in the reservation station to the corresponding functional units. And here we have multiple functional units and different functional units may take different latencies. So, as a result the instructions can finish their execution in different number of cycles. And again an instruction can be issued to a functional unit only when both of its operands are ready and as well as the functional unit is free. If any of these things are not ready or available, then we cannot issue the instructions to the functional unit. So, as and when the instructions finish their execution then we can forward the computed value along with registered tag through this bus back to the reservation stations, as well as to the renamed register files.

This is mainly to ensure that all the dependent instructions will get the operands as soon as the parent instruction finishes its execution. And as and when the instruction is finished its execution then the value will be updated in the reorder buffer also. So, we have this completion buffer also called as reorder buffer and from there we will complete the instructions in order. So, in order to provide the in-order commit in this stage, we are always going to select the instructions which are at the head of the reorder buffer.

We take that instruction and then send it to the completion stage for the completion of the execution of that instruction. So, when I say an instruction is going through the complete stage. So, the values will be written to the architectural registers specified in the instruction. And also whenever the instruction is going through the complete stage, if the value is there in the renamed register file then we will write that value back to the architectural register file by using this mapping mechanism between ARF and RRF, that we discussed in the previous modules.

So, effectively the dynamic execution core consists of a set of tasks, one is the instruction dispatch, second one is instruction execution and third one is instruction completion. So, as part of this instruction dispatch, we take care of the register renaming allocation of entries in reservation station as well as reorder buffer and advancing the instruction from the dispatch buffer to the reservation station, as part of the instruction execution. We issue the ready

instructions to the functional units and execute these instructions on the functional units. And as and when the computation is done we will forward the results through using this forward bus back to the reservation stations as well as to the renamed register file.

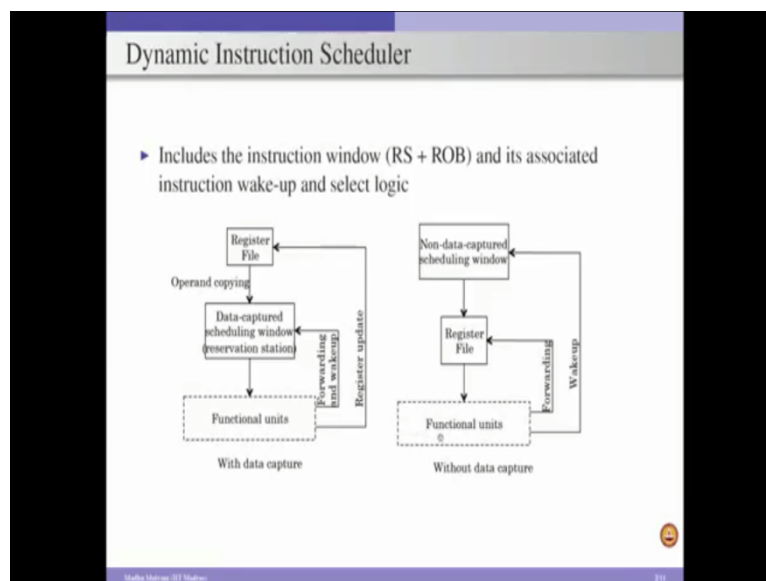
Once the instruction is finished its execution and if it comes to the head of the ROB, then we will complete the instruction by writing the values to the architectural registers. And we already discussed in the previous modules the role of renamed registers because in order to eliminate the name dependencies among the instructions we are going to use register renaming. So, as a result the both the output dependencies and the anti-dependencies can be eliminated from the instructions. And we allocate an entry into the reservation station because the instructions will be waiting in the reservation station after the dispatch and they will wait until both the operands are available and as well as the functionality is available. And we also allocate an entry in the reorder buffer for all the instructions which are having an infinite status. So, that is nothing but all the instructions that are there either in the reservation stations or that are executed by the functional units or those instructions which are finished their executions, but not yet completed will have an entry in the reorder buffer.

And we already discussed that reorder buffer is mainly helps in committing the instructions in the in-order. And so once we find a free entry in the reservation station, then we can dispatch an instruction from the dispatch buffer to the reservation station. If the reservation station is full then we cannot dispatch any instruction to that reservation station. And once the operands are ready as well as the functionality is available then we execute this instruction on the function unit. And depending on the type of functional unit, different instructions will take different amount of time. And as soon as they finish their execution we write these values to the corresponding entries in the reorder buffer as well as we forward these computed values to the reservation stations as well as the renamed register file. So, that any dependent instructions which are waiting in the reservation station for this computed value will get that value and then they can make their operands ready. And similarly, because the renamed register is waiting for this computed value to be updated. So, once we forward this computed values, then it will be updated in the register file as well as we update the other fields of the corresponding entry.

To inform that the value that is stored in the renamed register is the valid value. And we take one instruction at a time from the front of the ROB and we commit. And while we were committing the instructions, we are actually writing the values from renamed register file to

the corresponding architectural registers and that completes the execution of an instruction. And if the instruction is store instruction then in addition to the complete stage we also have our Retire stage. And in the Retire stage we are actually writing values to the memory. So, this is the core of the Superscalar processor the dynamic execution core is the critical component in the Superscalar processors, which actually executes instructions in out of order fashion and commits the instructions in the inorder. And because of these various functional units as well as because of these register renaming and the reservation station we can achieve out of order execution of instructions. So, that instruction execution can be overlapped for all the independent instructions and that will improve the overall performance of the system. So, we will see how the dynamic instruction scheduler can be designed. So, the dynamic instruction scheduler includes the instruction window and it is associated instruction wake up and the select logic.

(Refer Slide Time: 09:26)



We already discussed as part of reservation stations that, each reservation station consists of wake up logic as well as select logic associated with it. So, this dynamic instruction scheduler can be designed in one of the 2 ways. In the first design this is called as instruction scheduler design with data capture. So, here the functional units are here and the register file is here. And in between register file and the functional units we have this reservation station. So, here what happens is so, whenever functional unit finishes its execution, we forward the computed value to the reservation station and also to the register file.

And while we are forwarding this value from the functional unit to the reservation station, this also will be acting like a wake up logic for the entries that are stored in the reservation station. And in the case of forwarding value from the functional unit to the register file we are actually writing this value directly to the corresponding entry in the register file. And this register file consists of both the renamed register file as well as the architectural register file.

So, of course we are writing value only to the renamed register file. So, as a result once we consider this type of design, whenever we are dispatching an instruction to the reservation station, at the time of dispatch we read the values from the register file if the values are available. So, these values can be read either from the architectural register file or renamed register file provided values are available in the register file. If the values are not available in the register file at the dispatch time, then what we can do is we can forward the register tag and along with instruction. So, that this dispatch instruction will be stored in the reservation station with the corresponding register tags.

So, in other words the dispatch instruction will be staying in the reservation station or waiting in the reservation station until the corresponding values are available. Once the values are available for the instruction which is waiting in the reservation station, then we can issue this instruction to the functional unit. So, in order to know when to issue an instruction that is waiting in the reservation station to the functional unit, we actually use this make up logic.

So, remember the wake up signal is just by passing the register tag to the reservation station. Any instruction which is waiting for the operands for the corresponding register tag will get a wake up signal and the value will be written to the appropriate location in the reservation station. So that, that instruction can be now ready for executing on the functional units. So, that we will pick the selected instruction or we pick the ready instruction which is selected by the selection logic and we send it to the functional unit.

If we consider this type of design our reservation station is very wide because it has to keep the register contents in each of the reservation station, as well as the forwarding path between the functional unit to the reservation station also needs to be very wider because we are going to forward the values on this. This is one type of design, but some of the processors are following the other types of design.

That is dynamic instruction scheduler without data capture. So, here what we do is we reorder these 2 units. So, we have non data capture scheduling window here. And we have a register

file here and functional units here. So, this non data capture instruction scheduling window locates all the instructions that can be dispatched to the functional units or issued to the functional units. And whenever this logic gets a wake up signal from the functional unit then it selects the instruction. And then it is going to issue to the functional unit, but while it is issuing to the functional unit, it also goes to the register file and read the appropriate values and then send it.

So, in this particular design, our reservation station will have only the register tags, but not contents of the registers, whenever functional executes an instruction it forwards the data through this forwarding path to the register file, but it forwards the register tag of the computed value to the reservation station. So, this tag can be acting like a wake up logic or wake up signal for all the instructions which are waiting in the reservation station. For example, consider an instruction ADD R1 R2 R3 where R2 R3 are source operands and R2 is already available and R3 is not yet computed because R3 is going to be produced by a previous instruction and which is actually executed on this particular functional unit.

So, as soon as this functional unit finishes the previous instruction. So, now it forwards the R3 value through this forwarding path to the register file, but R3 tag is forwarded through this wake up path to the reservation station. So, that now this instruction which is waiting the reservation station will get a signal from the functional unit saying that R3 is now ready. So, as soon as it gets a ready signal here because already R2 is ready in the instruction.

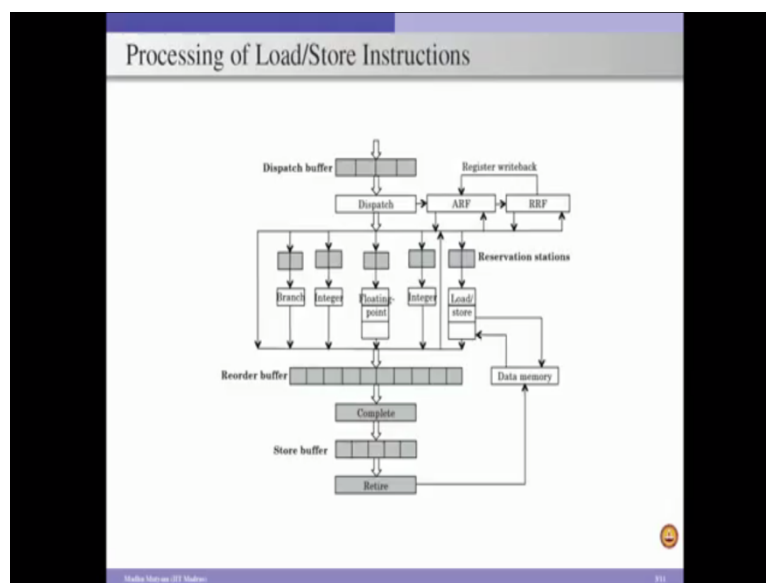
So, now this instruction is now scheduled to the functional unit. And while it is being scheduled on this functional unit we read the value of R2 and R3 from the register file and these values will be supplied to the functional unit. So, then this instruction will be executed on functional unit. So, because of this non data capture mechanism our reservation station will be very thin compared to very fat reservation station, whatever we consider in the previous design because here with each entry of the reservation station, we need to store only the registered tag, but not the contents that are stored in the register.

Also wake up logic path between functional unit to the reservation station will be very thin because here all we require is, we need to pass only the register tag as part of the wake up logic, but whereas in the previous design we have to pass both the register tag as well as computed values. So, this is going to take wider bus but whereas, here this is going to take very thin bus.

So, these are the two different mechanisms in which we can design our instruction scheduler, but the problem with without data capture design is, so register file read is in the critical path because before dispatching any instruction to the functional unit or before issuing instruction to the functional unit, we have to read content from register file. So, as a result this register file requires multiported design and that is going to increase the execution time also.

So, again these 2 designs have their trade-offs. One design is eliminating the register file read from the critical path. So, as a result it can improve the performance, but it is going to take more area because of the fat reservation station as well as the wider forwarding path, but whereas, the other design. So, it is not going to take significant area because we adjust storing the register tags, but the register file read is in the critical path for issuing an instruction to the functional unit.

(Refer Slide Time: 18:02)



So now, we are going to discuss the processing of load store instructions using our dynamic execution code. We know that the load store unit is also one of the functional units in our dynamic execution core. And this unit is going to process all load and the store instructions that are there in our program. So, we consider load store architecture where ALU operation will be performed only on registers, but the number of registers in a typical processor is limited. So, as a result we cannot keep all the required data in our limited number of registers.

So, as a result some of the required data will be stored in the memory and we have to go to the memory to get those data. And to go to the memory we are going to use load and store

instructions. Load instruction is going to load the data from the memory in a specified location and the store instruction is going to write the specified data in specified location in the memory. So, in order to process this load and store instruction we have separate unit called Load-Store unit. And this load-store unit is connected to the data memory. So, that it can perform a read operation or a write operation from or to the data memory.

And if it is store instruction we already discussed earlier, that the data will be written to the memory only when the instruction is processing through the retirement stage. Similar to the ALU instructions the memory instructions also have these dependencies among them. So, a memory data dependence exists between the 2 load and store instructions, if both these instructions refer the same memory locations.

(Refer Slide Time: 20:04)

**Ordering of Memory Accesses**

- ▶ A memory data dependence exists between two load/store instructions if they both reference the same memory location
- ▶ **Option #1: Execute all loads/store instructions in program order to enforce memory data dependences**
- ▶ Stores must be executed in program order
  - ▶ Preserve the sequential state of the memory to recover from exceptions

Store	:	X
Store	:	Y
Load	:	Z

Example #1

*Load can bypass earlier Stores*

Store	:	X
Store	:	Y
Load	:	X

Example #2

*Store can forward data to a later Load*

- ▶ **Option #2: Out-of-order execution of loads is the primary source of performance gain**
  - ▶ Make sure that RAW dependences are not violated

But how do we know whether two instructions are actually referring to same memory location? Any load and store instruction typically consists of 2 components, one is providing the base address and other is providing the offset. We have to add this base address contents with the offset to get effective address. And this effective address is an address location in the memory generated by program. And when we have a system supporting virtual memory this address is effectively the virtual address. So, we need to convert this address into physical address that is the address stored in main memory. And from there using that address we go to the memory and get the data from that particular location.



Or we have to write the data to that particular location. So, as a result, so in order to identify the data dependencies among memory instructions, first thing what we have to do is, we have to compute the effective address. Once we compute the effective address then we know whether 2 memory instructions are dependent on each other or not. So, in order to enforce this data dependencies, one option what we can do is execute all the load and store instructions in the program order to enforce this memory data dependencies.

We should not opt for this option one because it is going to degrade the performance significantly, as we execute all the load and store instructions in the program order to enforce the data dependencies. So, we can relax the option one slightly by considering this ordering only for the store operations. So, once we enforce that all the stores must be completed or executed in the program order we can preserve the sequential state of the memory, to recover from any exceptions happening in our program execution.

In the case of loads, we do not have to enforce this constraint. So, that effectively in order to improve the overall performance in our Superscalar processor, we can execute load instructions in out of order fashion to improve the overall performance, but this out of order load instruction can be done as long as we maintain the true dependence among the instructions and so on.

So, from this we know that rather than forcing all the load and store instruction to executing the program order to enforce memory data dependencies, we can just enforce only the store instructions to execute in the program order. So, because of this relaxed constraint now we have the option to execute the load instruction in out of order fashion. Now, we will consider 2 examples. First example is a piece of code consists of a store instruction writing to a memory location address x.

There is a store instruction after some time writing some data to an address location y in the memory and after some time there is load instruction in the program which is going to read data from address location z. Now, in this case we clearly know that this load instruction is independent of these store instructions, but if we are following the option one which says that we have to execute all the load and store instructions in the program order. If it is so then we have to execute this load instruction only after we execute store x and store y.

That is actually going to degrade the performance because we know that this load instruction is independent of the previous stores, there is no point in delaying this load instruction until

the store instructions are finished their execution. So in this type of scenario what we can do is, we can go ahead with load bypassing. So, this load instruction can bypass the previous store instructions. So, that this load will be executed in a speculative way in an out of order fashion and it finishes its execution.

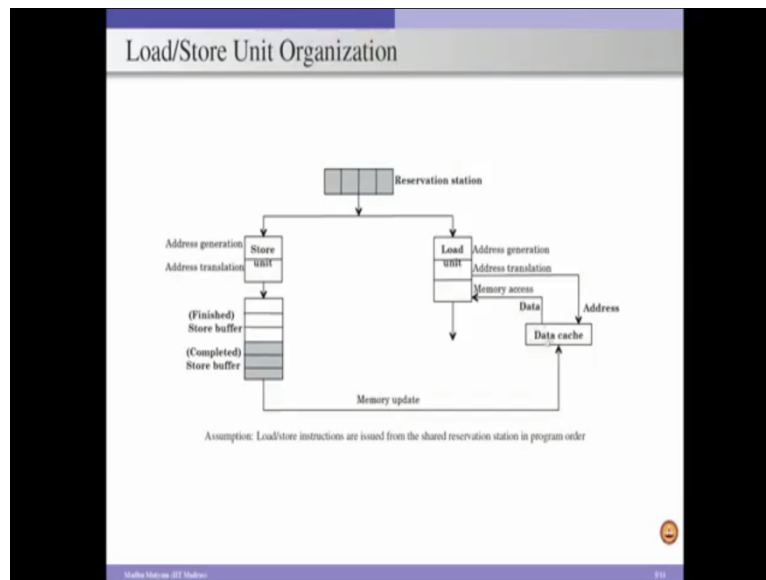
Now, consider another example. So, here we have a store instruction writing to an address location  $x$  in the memory after sometime there is a store instruction writing something to address location  $y$  in the memory and after sometime there is a load instruction loading some value from an address location  $x$ . Now, this load instruction is independent of this store instruction, but this load instruction is same as or dependent on the store instruction because both load and store are actually going to perform operation on same memory location.

So, in such scenarios this load cannot be executed before these 2 stores. And this load actually can get value supplied by store instruction. So, as a result we can exploit the other optimization to deal with the load instruction that is the load forwarding. This store instruction is going to supply the value to the trailing load instruction which is also pointing to same address location in the memory. As a result this load instruction need not access the memory to get the value and this load can be executed quickly.

So, effectively when we have such scenario where sometimes loads can be independent of previous stores or sometimes load can benefit from previous stores. So, we can exploit the load bypassing or load forwarding mechanisms to execute these instructions in these loads instruction in out of order fashion and improve the overall performance. Effectively the option 2 to deal with memory operations is will go for out of order execution of loads. So, that we can improve the overall performance, but when we are going for this out of order execution of loads, we have to ensure that read after write dependencies are not violated that is like true dependencies are not violated among the instructions.

As long as we ensure true dependencies are not violated, we can execute our load instruction in an out of order and speculative way so that, the overall performance can be improved. So, now in order to support this load bypassing and load forwarding we have to see how our load store unit needs to be designed in our dynamic execution core of the Superscalar processor.

(Refer Slide Time: 26:56)



So, here as the previously in our dynamic execution core we consider load store unit as a single unit, but we now separate load and store units because the operations involved with the store instruction execution will be different from the operations involved with the execution of load instruction. So, as a result we consider 2 separate units, one is a load unit which takes care of all the load instructions and the other one is store unit which takes care of all the store instructions. So, in order to support the store instruction execution and completion, we will consider a store buffer.

And here the store instruction which completes the address translation will be sent to this end of the store buffer and these entries are corresponding to the store instruction which finish their execution and whereas, these entries in the store buffer are corresponding to which completes there execution. In the case of store instruction we consider 2 staged pipeline for executing the store instruction.

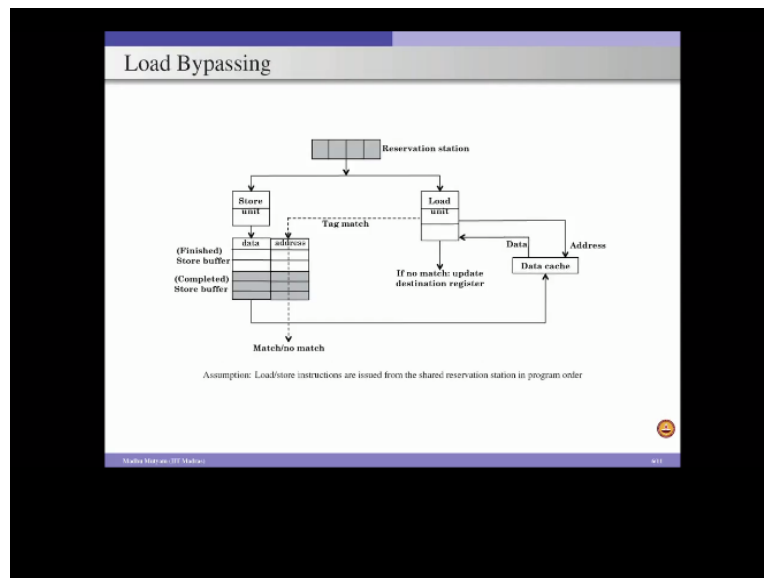
The first stage is corresponding to the address generation the second stage is corresponding to the address translation. And here the first stage we are going to complete the effective address calculation by adding the contents of base register and the offset. And the second stage of this pipelining is address translation where we go to the TLB and get the physical address for the corresponding the address whatever is generated in the first phase. And after that we say the store instruction is finish its execution. So, that we write the corresponding store instruction

to the store buffer and so that the instructions will be waiting in the store buffer as long as the cache or the memory is not available for servicing this store instruction.

Whenever the data cache or the memory is available, then we will select one instruction from this store buffer in the program order and then we will write it to the data cache or the memory. So, here the store buffer is actually working in the FIFO order because we already discussed on the previous point, that all the store instructions are executed in the program order. So, as a result we will just always select instructions in the stored buffer in the FIFO order and here in the case of load instructions. So, our load unit consists of 3 pipeline stages.

First 2 pipeline stages are same as the store unit pipeline stages there is address generation and address translation. And once we translate the address now we have the effective address in the main memory. Then we can go to the memory or if our system is supporting the data caches then we will go to the data cache using this translated address and then we will read data from that particular location to this. So, effectively the third stage of this load unit pipeline is the memory access, where the actual access from the data cache happens.

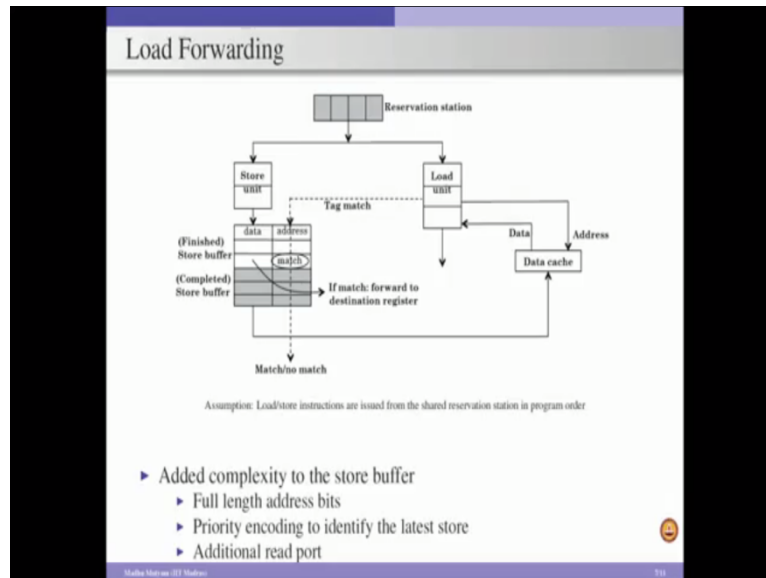
(Refer Slide Time: 30:05)



In order to exploit the load bypassing, we have to modify our store unit slightly by considering an address field for each of the entries of store buffer, by adding this extra components which has a partial address of the this store instructions, we can compare the partial address of the subsequent load instructions with this and to know whether there is

match or not. If there is no match then we are going to execute this load instructions in out of order fashion with respect to the previous one, previous store instructions.

(Refer Slide Time: 30:46)



And now in order to support the load forwarding unlike the load bypassing mechanism, in the load forwarding we are going to actually forward the data that is stored in the store buffer to the trailing load instructions. So, as a result here we need to consider the full address associated with each entry of this stored buffer. So, note that in the case of load bypassing we consider only the partial address associated with each of these stored buffers. So, when we consider the partial address sometimes the partial address may match, but the full address may not match.

That is not going to create a significant penalty in our design, but that considering the partial address is going to save the area overhead associated with the store buffer design, that is the reason why in the load bypassing case we consider the partial addresses, but in order to consider the load forwarding we have to consider the full address associated with each entry in the store buffer. So, that only when all the bits of store address is matching with the load instruction address, then only we can forward the value stored in the data field of the corresponding entry in the store buffer to the load instruction.

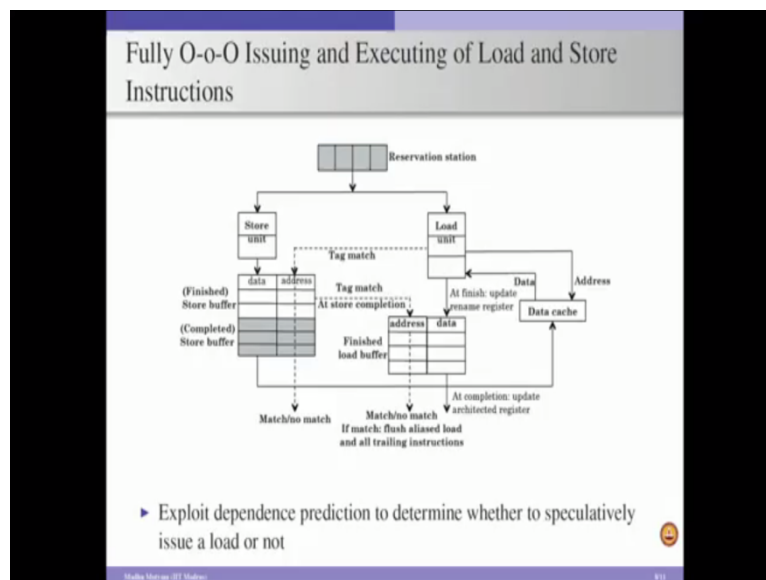
So, here whenever a load instruction comes here we just compare the tags. And if there is a match then for this load instruction we do not have to go to the cache or the memory to get the data actually the store instruction supply. So, whenever there is a match we forward this

value directly to the load instruction. Of course so, this extra component is going to add some complexity to the overall design of the store buffer because here in the case of load forwarding we have to consider full length address bits for each of the entries of the store buffer.

That maybe like 32 bits or 64 bits depending on what are the processor we are considering. And also whenever we have the translated address we have to search in all the entries. So, for that we have to actually use a priority encoding logic to identify the latest store. We may have multiple stores writing to the same location memory, but we always have to consider latest stores. So, for that we have to use a priority encoding logic.

Also in the original design of the store buffer we have only one read port, but now because of this added component we need to have 2 read ports for the store buffer. And increasing the number of ports is also going to increase the access time of the store buffer as well as the power consumption, in addition to the area over head. So, these are the extra added complexities to the store buffer, to support our load forwarding mechanism.

(Refer Slide Time: 33:27)



So, once we have the support for the load forwarding or load bypassing, by using these extra components to the store buffer. Now, we can execute instructions, we can execute load instructions in out of order fashion. Here in this particular design we consider a separate load buffer to take care of all the finish load instructions. Again here we are splitting the load instruction execution into execution finish and execution complete. And at the time of

execution finish, we write the data to this buffer. Now, we can consider a simple example we have store instructions store x and load y. So, this load instruction is actually coming after store instruction.

Now because this load instruction is independent of the store instruction, so we can execute this load instruction in out of order fashion. So, in such scenario what we can do is, while the store instruction is issued to the store unit, we issue the load instruction to the load unit. After the second stage of the load instruction pipeline processing we have the translated address. And so, once we have this translated address, we give this translated address to this address unit of the store buffer to see whether this instruction is matching with any of the previous store instructions.

While we are doing this, we will proceed with the third stage of the load instruction pipelining by going to the data cache or the memory to read the data from the location specified by this translated address. So, that at the end of the third stage we have the data from the data cache. Now, what we are going to do is, once we have the data at the end of the third stage of the load instruction processing. We write this value to the updated renamed register. We write this value to the renamed register to update the renamed register content.

So, with that we actually finish the execution of this load instruction, but remember this load instruction is executed in an out of order fashion with respect to the previous store instruction. And we do not know whether this load instruction is actually dependent on the previous store instructions or not. That we will know only by comparing this address with the addresses of the corresponding store buffer entries. If there is a match with any of the data that is stored in the store buffer that indicates that this load instruction is dependent on the previous stores. And we have to flush this particular entry.

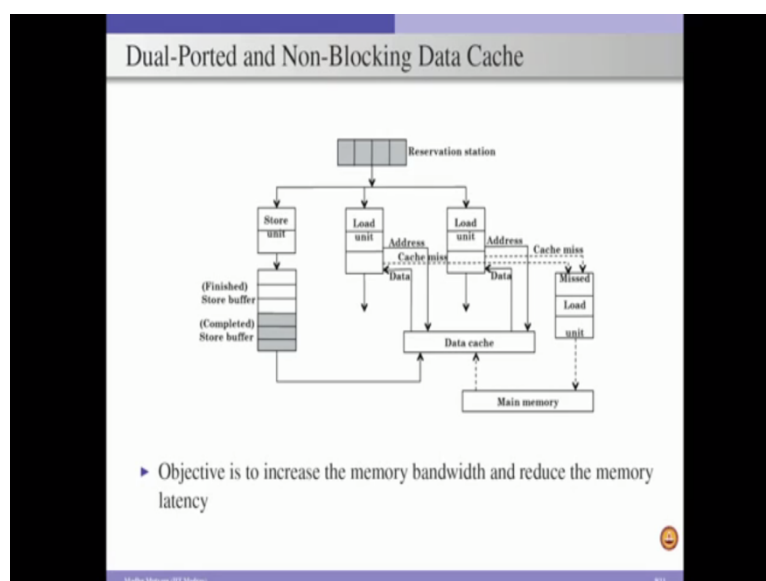
So, if there is a match happens then flush the alias load and all the trailing instruction. So, that we can revert it back to a state where we start executing again, from where we can start executing the load instruction, by taking the value from store instruction. And if there is no match happens between this translated address of the load instruction with any of the translated address of the previously finished store instructions then there is no problem and this load instruction can peacefully finishes its execution. And all the dependent instructions can take the value from this load instruction and execute their instructions.

So, that means if there is no match we do not have to flush anything here. And we can proceed and after sometime this load instruction can come to the head of the ROB and then we can complete the execution of this load instruction. And at that particular point of time we update the architectural register. So, that means we update the corresponding register in the ARF by taking the value from the corresponding register in the RRF.

So, we do that and with that we finish the out of order execution of an instruction. In order to execute this trailing load instruction ahead of the store instructions and also minimize the penalty associated with flushing this load instruction or subsequent dependent instructions we have to use some data dependence prediction mechanisms, to see whether our load instruction is independent of previous store instructions and whether we can execute the trailing load instructions speculatively ahead of the previous store instructions.

If our prediction mechanism is correct then we can minimize the number of flushes as well as we can improve the overall performance associated with the out of order execution of the instructions. So, far we consider single ported data cache where at any point of time only one load instruction can go to the data cache and read the data from the data cache. In order to improve the memory bandwidth or in order to improve overlap the processing of multiple load instructions simultaneously by the data cache, what we can do is, we can consider a dual ported the non-blocking cache.

(Refer Slide Time: 38:43)

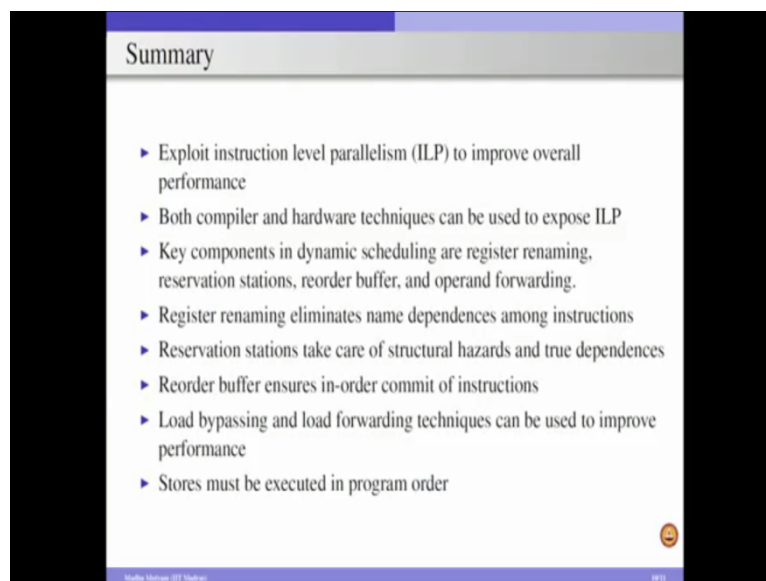




So, when we consider the dual ported data cache. So, simultaneously we can service multiple load instructions by this data cache. Of course, when we consider a dual ported we have to consider dual load units. In this particular design we can serve 2 read ports associated with the data cache. So, as a result at any point of time 2 load instructions can read data from this data cache in their third pipeline stage of load instruction processing. And also in order to improve the overall bandwidth, we can consider this data cache as non-blocking cache.

So, that even if the previous instruction is not completed or finished its processing, this data cache can take new request and start processing. So, effectively at any point of time the data cache can service multiple requests coming from the processor. So, because of this dual ported non-blocking data cache, we can improve the overall bandwidth and improve the overall performance of the system. So, with that we conclude the Superscalar processor design.

(Refer Slide Time: 40:08)



Summary

- ▶ Exploit instruction level parallelism (ILP) to improve overall performance
- ▶ Both compiler and hardware techniques can be used to expose ILP
- ▶ Key components in dynamic scheduling are register renaming, reservation stations, reorder buffer, and operand forwarding.
- ▶ Register renaming eliminates name dependences among instructions
- ▶ Reservation stations take care of structural hazards and true dependences
- ▶ Reorder buffer ensures in-order commit of instructions
- ▶ Load bypassing and load forwarding techniques can be used to improve performance
- ▶ Stores must be executed in program order

Hadoop Hadoop (11) Hadoop 10/10

In summary, exploit the instruction level parallelism to improve the overall performance of the system. So, by using this compiler as well as the hardware techniques, we can expose the ILP that is there in our programs, so that we can improve the overall performance by executing these independent instructions on our underlying Superscalar hardware units. And the key components in the dynamic scheduling associated with the Superscalar processor are register renaming, reservation station, reorder buffer and operands forwarding.

And we already discussed that register renaming eliminates name dependencies among instructions and reservation stations will take care of structural hazards as well as the true dependencies. So, as a result when we want to execute instructions in an out of order fashion, we have to respect the true dependencies. And as long as the true dependencies are respected we can execute instruction in out of order fashion, as long as functional units are available. So, that we can overlap execution of multiple independent instructions, by executing these instructions on different functional units available with our Superscalar processor.

Finally, in order to maintain the program correctness we have to commit the instructions in the inorder. So, to do that we take the help of reorder buffer which ensures inorder commit of instructions. And in order to improve the performance associated with the load and store instructions we can exploit the load bypassing as well as the load forwarding mechanisms. So, that we can execute instructions in an out of order fashion and improve the overall performance. But in order to maintain the program correctness we have to process our store instructions in the program order only. So, with that I am concluding the Superscalar processor unit design.

Thank you.