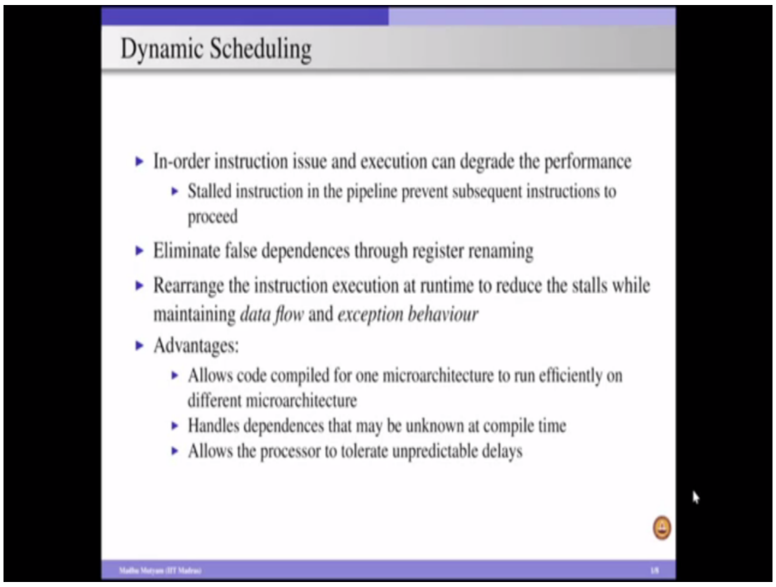
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 06
Lecture – 23
Superscalar Organization

So, this week we are going to discuss dynamic scheduling and superscalar organisation, which consist of various micro architectural components that help in achieving dynamic scheduling. And in this module, we are mainly concentrating on superscalar organization, and discuss various components in superscalar processor.

(Refer Slide Time: 00:36)



Dynamic Scheduling

- In-order instruction issue and execution can degrade the performance
  - Stalled instruction in the pipeline prevent subsequent instructions to proceed
- Eliminate false dependences through register renaming
- Rearrange the instruction execution at runtime to reduce the stalls while maintaining *data flow* and *exception behaviour*
- Advantages:
  - Allows code compiled for one microarchitecture to run efficiently on different microarchitecture
  - Handles dependences that may be unknown at compile time
  - Allows the processor to tolerate unpredictable delays

So, we know that the Inorder execution will degrade the performance, the reason is if the leading instruction installed, even though the trailing instructions are independent, we cannot issue these trailing instruction to the function units and as a result the overall performance will be degraded. So, in other words the Inorder instructions issue and executions will degrade the performance and we need to look at some other alternatives to improve the overall performance.

So, when we move from scalar pipeline design to superscalar pipeline design, we can issue multiple instructions, we can fetch multiple instructions, we can decode multiple instructions and we also have a support for multiple execution units, in a superscalar pipeline design. Once we have the superscalar processor, in order to expose the instruction level parallelization, we can take the help of either the compiler or the hardware. And we already discussed various compiler optimization techniques to expose instruction level parallelization.

Now, in this week we are going to discuss the hardware techniques that will expose instruction level parallelization to the underlying processor that is superscalar processor, so that the performance can be improved. So, in order to overcome the performance penalty associated with the in order execution, we are now going to the out of order execution and to execute instructions in an out of order fashion, so we need the support from the superscalar processor and as a part of this dynamic scheduling we are going to discuss various techniques associated with out of order execution of instructions.

So, we already discussed in the previous modules that the instruction dependences will limit the execution of instructions in an out of order fashion. So, instruction dependences consist of data dependences and name dependences. And the data dependences are the dependences where actually true dependence will be there, because the true data flow will happen between producer and the consumer. But were as in the case of name dependences, when instructions have name dependences there is no flow of information between producer and consumer and this also called as the false dependence. So, in order to eliminate the false dependence, we have to use techniques such as register renaming.

So, as part of this dynamic scheduling mechanisms we consider techniques to eliminate this false dependencies by using register renaming. Once we eliminate name dependences or false dependencies so as a result we can schedule instructions in an out of order fashion. So, we already discussed in compiler optimisations that the instructions at compile time can be rearranged in such a way that if the rearrangement of instructions are not going to create any data flow or any exception behaviours then we are fine with the rearrangement and then execute these instructions in an out of order fashion.

So, as long as we take care of data flow and exception behaviour, we can rearrange the instructions at run time after identifying the instructions are independent and then schedule these instructions on to the functional units, so that we can parallelize our execution or we can overlap execution of multiple instructions and improve the overall performance.

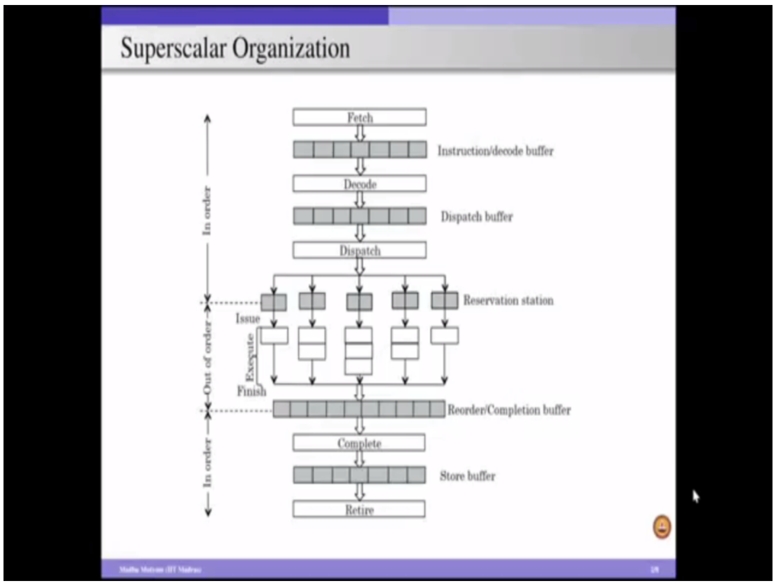
So, the main advantages with the dynamic scheduling compared to static scheduling are the dynamic scheduling allows the code compiled for one microarchitecture to run efficiently on another micro architecture. We know that for a same ISA we can have multiple microarchitectures. So, for example AMD processor and INTEL processors can use same ISA, x86 ISA, but the way the processor designed in AMD can be completely different from the way the processor designed by INTEL.

A single ISA can be realised through multiple implementations and each implementation is corresponding to one microarchitecture. So that, we can have multiple microarchitectures which have the same underlying ISA. So, when I say multiple microarchitectures for the same ISA, one processor may consider five stage pipeline, the other processor may consider ten stage pipe line. And one processor may consider single ported register file, the other processor may consider multi ported register file. One processor may consider a single issue, the other processor may consider multiple issue. So we can have several variations in our microarchitecture, but the underlying ISA can be same.

As long as ISA is same the code which is written based on that particular ISA can be executed on any of the microarchitectures or the any of the processors. And once we have the dynamic scheduling support in our system, so as a result we can efficiently execute code which is based on the same ISA on various micro architectures. The second advantage with the dynamic scheduling is, so it handles the dependencies that may not be known at compile time.

And the third advantage with the dynamic scheduling is the processor can tolerate unpredictable delays. So, in order to understand the dynamic scheduling part, let us focus on the superscalar processor organization in this module. So, once we have the thorough understanding of superscalar processor organisation then we can apply the dynamic scheduling mechanisms to expose significant instruction level parallelization that is present in our program, so that we can improve the overall performance of system. So we start with superscalar processor organization.

(Refer Slide Time: 07:15)



Superscalar Organization

So, this consists of several pipeline stages, starting with the fetch stage. So in the fetch stage we fetch multiple instructions from the instruction cache. And all these fetch instructions will be stored in instruction decode buffer, this is a buffer which is placed between instruction fetch, and instruction decode. And once we fetched the instruction stored in the intermediate buffer or the pipeline buffer, we take these instructions and give it to the decoder, and decoder decodes all these instructions simultaneously. So effectively, we have support for multiple instructions decoding in the decode stage. After the decoding is done we dispatch these instructions through a dispatch stage. And the main idea of the dispatch stage is instructions will be dispatched to the functional units.

In some processors they combine the dispatched buffers and the reservation station together and whereas in some designs they separate dispatch buffers and the reservation stations. So, again there are advantages and disadvantages with each of these designs, but in this particular organization what we are considering is after dispatching these instructions will be moved to different reservations stations and we have one reservation station for each of the functional unit. So, we know that after the decoding the instruction, the type of instruction what we are going to execute. And based on the type of the instruction, we dispatch this decoded instruction to the respective reservation station.

And in this particular design we are considering separate reservation stations for each of the functional units, but there are designs where they consider a single reservation station for all the functional units. We are going to discuss that in the next foil. So, once instructions are stored in the reservation station, if the operands are ready and if the functional unit is free then we can schedule one instruction at a time from the reservation station to the corresponding functional unit.

And here again, selecting one instruction from a set of instruction that are stored in the reservation station, we can apply a various heuristic mechanisms. We can consider an out of order the selection of instructions from this reservation station. So that is a reason why if you see in this a superscalar processor organisation, the fetch stage, decode stage and the dispatch stage are actually working in the inorder fashion. But whereas the selection of instruction from the reservation station to be issued to the functional units, happen in the out of order fashion and the execution of these instructions happen in out of order fashion on the functional units.

And finally, depending on the type of instruction, depending on the number of cycles instructions are going to take for executing, these executed instructions will be written to the buffer that is called a reorder buffer, in out of order fashion. So, effectively so starting from the reservation station output to the reorder buffer input, we have out of order flow in our instruction execution. And once we have instructions executed by different functional units and written to the reorder buffer, we select instructions from these reorder buffer and will complete these instructions in the program order again.

And some instructions, when we have memory instructions typically store instructions then we need one more stage after the completion stage, that is called as a retire stage. So, the complete stage is the stage where all ALU operations and the load operations will be completed, will complete their execution. But whereas the store instructions after completion stage, the store values will be written to store buffer and actual write operations to the memory happens only in the retire stage.

We already know that for a store operation, if you are going to wait till the time when the store operation is writing to the memory, it is going to take significant amount of time. And if you are waiting for so long time then it is going to degrade the performance. So as a result in order to minimize the penalty associated with store operations we typically consider store buffer where after the completion stage of the store instructions, we write to the store buffer and so that the store operations is said to be completed from the processor point of view but the actual operation, actual write operation, happens only in the next stage that is the retirement stage. But for all other operations which are ALU operations or load operations because we are not going to the write to the memory so we can complete the execution of these operations at the end of complete stage.

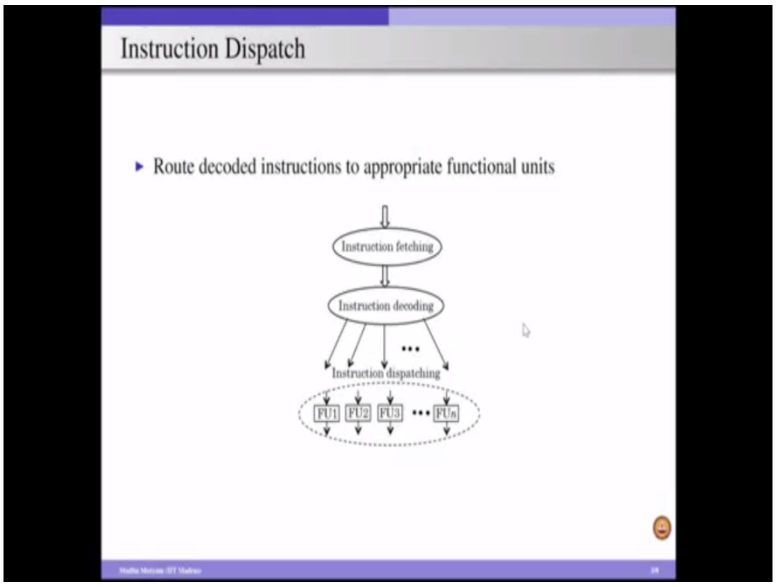
So, when we say ALU operation or load operation is completed, we are actually writing to the architectural registers of the processor and writing to the architectural register happens at the end of complete stage. So, each of these pipeline stages in the superscalar processor has a specific role to play, the fetch stage is going to fetch the instructions, the decode stage is going to decode the instructions, dispatch stage is going to dispatch the instructions to various functional units or associated reservations stations and reservation stations keep all the instructions which are waiting to be executed by the functional units or waiting for the operands to the available.

And the execute stage actually performs the actual execution of the instructions on different functional units, the complete stage is going to complete the processing of the instruction from the machine architecture point of view. And the retirement stage is the stage where we actually write the store values from the store buffer to the memory. So, out of all these stages the fetch decode and dispatch, we follow the program order and for the complete, completion stage or the retirement stage, we follow the Inorder or programme order again but for execute stage we will go by the out of order fashion.

So, because we can exploit the instruction level parallelization at the execute stage that is mainly because our processor may have multiple functional units and there may be independent instructions in the program. And as long as independent instructions are there and the functional unit is available, we can schedule these independent instructions in an out of order fashion and we can execute them on the functional units so that the execution of different instructions can be overlapped.

In other words we can exploit the parallelization of independent instructions, executions using this various execution units and that will improve the overall performance. And we need Inorder completion and Inorder retirement. This is required mainly to maintain the correctness of the programme. So, when we are going for the Inorder commits so as a result what happens is even if leading instruction is creating an exception, we are not committing any of the trailing instructions. So, we save the processor, we save the process state and we re-execute the instructions starting from that excepting instruction. So, having discussed this superscalar processor organization, now we are going to look at some of these stages thoroughly and we start with the dispatch stage.

(Refer Slide Time: 16:19)



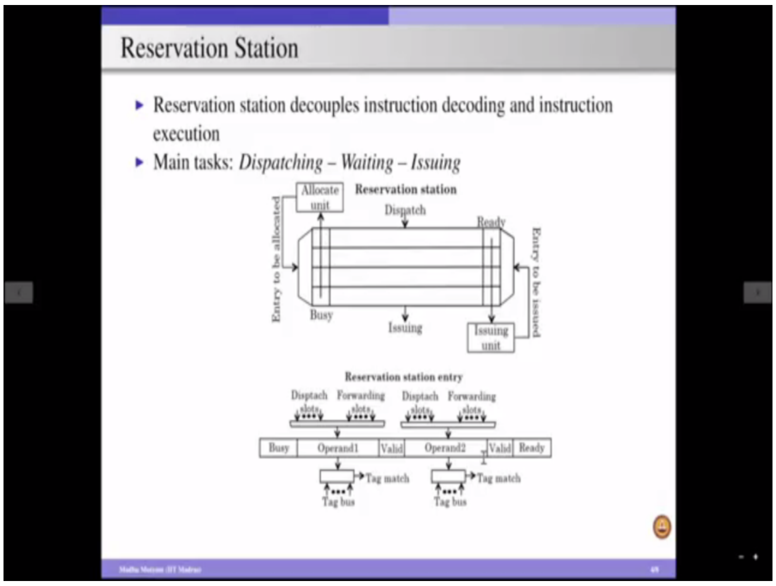
Instruction Dispatch

- Route decoded instructions to appropriate functional units

So, the main role of the dispatch stage is, it has to route decoded instructions to the appropriate functional units. Because after the decoding, we have the decoded instructions and we also know the type of instructions what we are going to execute. So, the dispatch stage is going to dispatch these instructions to appropriate functional units. Either you can directly issue to functional unit or we may consider a design where it will be issued to the reservation station and from the reservation station, we can issue to the functional unit.

So, when we consider a system where, if you are dispatching the instructions directly to the functional unit, then dispatch and issue both will be same, but in some designs they consider separate stage in between the dispatch stage and functional unit execution. So that is called as a reservation station, there dispatch means dispatching to the reservation station and issue means, selecting an instruction from the reservation station and giving it to the functional unit.

(Refer Slide Time: 17:32)



Reservation Station

- Reservation station decouples instruction decoding and instruction execution
- Main tasks: *Dispatching – Waiting – Issuing*

The main role of the reservation station is to decouple the instruction decoding and the instruction execution. So, once we decouple the instruction decoding and the instruction execution, so we can eliminate the stalls at the instruction decoding stage, as well as the penalties at the, or the starvation at the, instruction execution stage. The main role of reservation station is dispatching the instructions from the decode stage to the one of the entries in the reservation station.

And second role of reservation station is to keep all the non ready operations, non ready instructions, to be waiting in the reservation station itself. And the third one is select one ready instruction from the list of instructions that are there in the reservation station to be issued to the functional unit. In order to do these three tasks our reservation station will be designed like this. It consists of multiple entries and it has Allocate unit, Issue unit and set of fields in each of the entries.

Whenever an instruction is decoded and this instruction needs to be despatched to the reservation station. So, in order to dispatch instruction to reservation station, all we have to do is we have to check whether there is a free entry in the reservation station. So, as a result this allocate unit is going to scan through all the entries in the reservation station, to see if there is a free entry. So, this can be identified by using single bit information per entry, and that is called as a busy bit.

Whenever we issue an instruction or whenever we dispatch an instruction to a reservation station, we set the busy bit of the corresponding entry. So that next time if any other decoded instruction is to be dispatched to the reservation station, we are not going to write that decoded instruction in to this entry. So, as a result we are not going to overwrite the existing data with the new data as long as if the busy bit is set. So, allocate unit always scans through all the entries of the reservation station and identify the free entry which is specified by the busy bit, that is reset.

And once we identify free entry we dispatch the decoded instruction in to that particular entry. At the time of dispatching an instruction to the reservation station, the operands may not be available. So sometimes all operands are available at the dispatched time sometimes partial operands are available and in other times none of the operands are available. So, as a result, if the instruction whose operands are not ready at the time of dispatch, we have to send the register tag to the reservation station.

So that, whenever the producer instruction produces the value to the corresponding register, we can send the data to the waiting instruction that is stored in the reservation station. Instructions will be waiting in the reservation station to be selected by the issue unit and we know that each instruction will have two source operands. And if both the operands are ready, then we set the ready bit associated with that particular entry. And the issue unit is always going to scan through all the ready bits of all these entries in the reservation station, and it can select one instruction from all the ready instructions and this selected instruction will be issued to the functional unit. So that is a role of issue unit.

So, the allocate unit always keep track of busy bits to identify a free entry in the reservation station and the issue unit always keep track of ready bits to identify the ready instruction to be issued to the execute stage. And if the instructions are not having their operands ready then there will be just waiting in the reservation station itself. So each entry of this reservation station will look like this, it consist of a busy bit, operand one, a valid bit, operand two, a valid bit and ready.

So, here as soon as an instruction is dispatched to an entry, we set the busy bit and this operand one is the source operand one for the instruction and if the operand is already sent or the operand is available or the operand is ready then we set the valid bit. And if the operand is not available at the time of dispatch to the reservation station, so we reset this bit and if both of these valid bits are set that indicates that two operands for this instruction are ready, and then we set the ready bit. So that the issue stage will keep track of this and it can select this instruction to be issued to the functional unit.

And we already discussed in the previous modules that in order to improve the overall performance or in order to minimise the stalls because of the dependencies, as soon as the operands are produced by the function units, we can forward these operands to the dependent instructions. So, in order to support that operand forwarding, so what we do is for this reservation station, we have a connecting path from the output of the functional unit to the inputs of each of these entries. So, here we have forwarding slots and whenever a functional unit executes or completes its execution it forwards the computed value back to this reservation station along with this computed value it also forwards the tag associated with the operand.

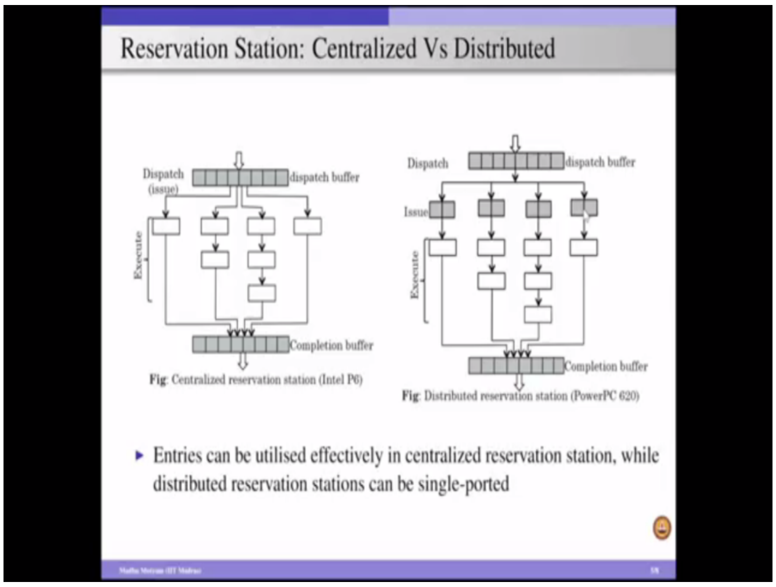
In other words, we forward the register tag as well as the computed value that is going to be stored in the register and using these forwarding slots, we compare the tags of this computed value with tags of the operands that are stored in the reservation station. If there is a match, we are going to write the computed value on to that location and we set the corresponding valid bit, so that this value is available in the reservation station. And next time when both the operands are ready, we can issue this instruction to the functional unit.

And also, so here we have support for the dispatch slots because whenever, as soon as we decode the instructions, we have to dispatch this instruction to the reservation station. So as a result the dispatched slot is actually going to take care of that where to dispatch which entry or the decoded entry to be dispatched, so that will be taken care of this dispatched slots. And whenever the value is computed by the functional unit this forwarding slot is going to take care of forwarding that value to a particular entry in the reservation station.

We know that if the functional unit is not available, reservation station is not going to issue an instruction to the functional unit, even though the operands are available. So, in other words, because there is no availability of free functional unit, we are not issuing the instructions to that functional unit. So that indicates that the reservation station can take care of structural hazards properly.

And similarly, if the operands are not available even though the functional unit is available, we are not going to issue this instruction to the functional unit. That indicates that this reservation station is taking care of the proper data flow or otherwise this reservation station is taking care of the read after write hazards.

(Refer Slide Time: 26:22)



Reservation Station: Centralized Vs Distributed

Fig: Centralized reservation station (Intel P6)

Fig: Distributed reservation station (PowerPC 620)

- Entries can be utilised effectively in centralized reservation station, while distributed reservation stations can be single-ported

So, reservation station can be implemented in a centralized way or distributed way. In the case of a centralized reservation station, which is typically implemented in INTEL based processors. Especially this example is for the Intel Pentium pro, so which has a centralised reservation station. So, once we have a centralised reservation station, so we combine our dispatched stage as well as reservation station together. So, the decoded instructions will be dispatched to the dispatch buffer and from this dispatch buffer instructions will be selected and they are send to the execute stage.

So, here the input to the dispatched buffer is coming from the decode stage in the Inorder, but the output from this dispatch buffer is going to the functional units in the out of order fashion. The main advantage with the centralised reservation station is that we can utilise the entries in the reservation station efficiently. And also because here we have a combined stage for both the dispatch and reservation station together, so as a result the input side of this dispatch buffer is called as a dispatch operation and the output side of the dispatch buffer is called the issue stage. Or we can interchangeably use the dispatch and the issue, especially when we are dealing with the centralised reservation stations.

And but the main disadvantage with centralised reservation station is whenever we want to select ready instruction from this entire list of instructions that are stored in the reservation station, we have to scan through all the entries and that is actually going to consume significant amount of time as well as the power. Because we know that, here in this particular design, we have four functional units and each of these functional units may require one instruction in a cycle. So in other words in a single cycle we may have to dispatch four instructions from this dispatch buffer to all these functional units.

So, in order to support that we need to have multiported dispatch buffer from at the output side. So we need to have four ports, so that four instructions can be issued or dispatched to the execute stage. And we already discussed in the cache memory design once we have increased number of ports, it is going to increase the overall execution time and also it is going to incur the hardware overhead as well as the power consumption.

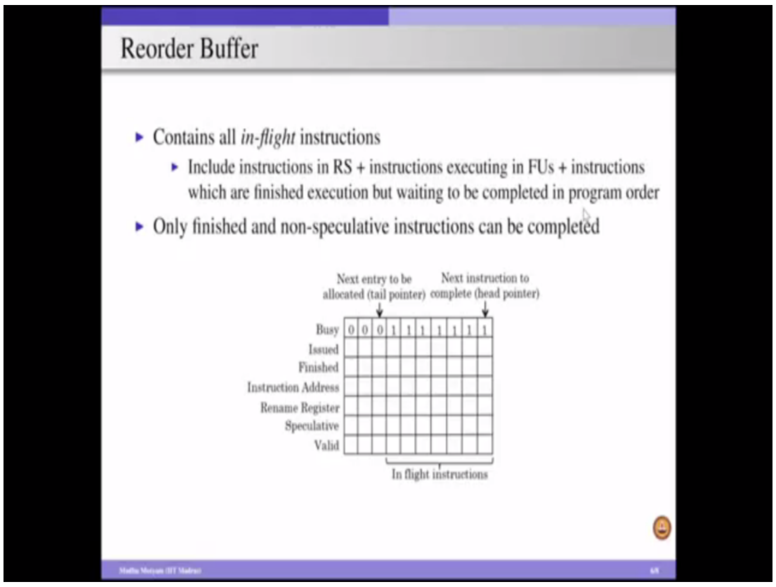
So, as a result in order to eliminate the problems with this centralised reservation station, we can go for other alternative, where we have a separate dispatch buffer and then we have reservation station separately. The output from the decode stage will be written to the dispatch buffer and from this dispatch buffer, we are going to dispatch instructions to different functional units depending on the type of instruction. And here in this particular design, we are considering one reservation station for each functional unit. So, as a result, whenever we want to issue an instruction to a functional unit, all we have to do is we have to search only in a limited entry reservation station. And each of the reservation station requires only a single port.

So, compared to this centralised reservation station the entries in the distribution reservation station may not be utilised efficiently, but because we are going to have only single ported reservations stations for each of these functional units, the access time will not be significant. So, effectively there is a trade off, so whether we want to efficiently utilise all the entries in the reservation station or whether we want to have lesser access latency to select an instruction from the reservation station. This type of distributed reservation station is implemented in IBM PowerPC. But note that these two are two extremes of the designs, one is using the complete centralised reservation station, the other is using a complete distributed reservation stations.

But as an intermediate to these two designs, we can come up with the clustered reservation stations, where rather than having one reservation station per functional unit, or rather than having a single reservation station for all the functional units, we can cluster or we can divide the reservation station into multiple clusters and each cluster can feed in multiple functional units associated with that.

So, in other words we can consider an intermediate design where this reservation station is divided into two sub reservation stations and one reservation station will take care of supplying the instructions to these two functional units and the other reservation station can supply the instruction to the other two functional units. So, again because the whole computer architecture is a design space explanation, so we can come up with an efficient reservation station, which is not going to underutilise the space in the reservation station, at the same time it is not going to consume the significant access latency and the area overhead. So as I mentioned earlier, so the centralised reservation station can utilise the space efficiently, but the distributed reservation station will have low access latency because of single ported design we consider. So, having discussed this reservation station now we will move on to the reorder buffer.

(Refer Slide Time: 32:46)



Reorder Buffer

- Contains all *in-flight* instructions
  - Include instructions in RS + instructions executing in FUs + instructions which are finished execution but waiting to be completed in program order
- Only finished and non-speculative instructions can be completed

The reorder buffer is required, whenever we are supporting the out of order execution in our system. So before that we will define what is an in-flight instruction? An instruction that is despatched to the reservation station and instruction that is there in the functional units or the instructions which are finished their execution, but not yet written to the architectural registers are called as in-flight instructions. And for each of these instructions we are going to maintain an entry in a table and that table is called as “Reorder Buffer”.

And we know that only finished and non speculative instructions can be completed. When I say finished, an instruction finishes its execution, for example when I give an add instruction "ADD R1 R2 R3". So performing of the add operations R1, R2, R3 contents if it is done then we say this add instruction is finished its execution. But when we write the final value to register R1 then we say that this add instruction is completed. Because of out of order execution a trailing instruction may finish its instruction, but it is waiting for all the leading instructions to be completed then only this instruction will complete, will write to the architectural registers.

We can complete only those instructions which are finished their execution and those instructions which are issued in non-speculative way. In order to enforce this constraint and also at the same time in order to provide out of order execution, we are going to use a buffer called as reorder buffer. And reorder buffer is implemented as a circular queue, which consist of multiple entries and when an instruction is dispatched to the reservation station, we allocate an entry in the reorder buffer.

And each entry in the reorder buffer consists of multiple fields one is the "Busy bit", this is going to say whether the entry is allocated to the instruction or not. So, there is second field that is called as "Issued" and this is going to specify whether the instruction is issued to the functional unit or not. And finished field is there, which is going to specify that whether the instruction is finished its execution or not. And there is a field in the entry that is going to store instruction address and also we have rename register for each of the operands in our instruction, because we know that the register renaming is used to eliminate the name dependencies. And register renaming is one of the main components in our dynamic scheduling. So as result reorder buffer also has an entry associated with rename register.

And sometimes we issue instructions with some speculation or by predicting something is going to happen. So, all the instructions which are issued in a speculative way, we have to set the corresponding bit in an entry, so for that will have "Speculative" field in our entry and also we have a valid field. So, this valid field is going to specify whether the instruction is valid to be completed or not. Sometimes what happens is, because of some exceptions on we may have to invalidate all the entries. So, in those cases we require a separate field in our, for each entry in reorder buffer, and that is a reason why we consider this valid bit.

Effectively each entry consists of multiple fields and each field is going to do a specific task. And whenever we are going to dispatch an instruction to the reservation station, we have to see whether there is a free entry in the reorder buffer. So that will be indicated by this busy bit, if the busy bit for an entry is reset, then we can select that entry and we can store that instruction in that particular entry. And once we stored the instruction in that then we move this tail pointer towards left so that it will point to the next free entry in the reorder buffer, because we are implementing this as a circular queue.

So, this tail pointer will be advanced towards left, whenever we are inserting new instruction into an entry. And the completion of instructions will happen only in the In-order, that is in the program order, so in order to ensure the In-order commit, we are going to commit only from the head pointer side that is from this side. So, the head pointer will be pointing to the next instruction that is going to be completed.

And once this instruction selected and moved to the commit stage or a complete stage we can advance the head pointer towards left, so that it is pointing to the next ready instruction, next instruction in the sequence to be committed. And all these instructions, which are there between this head pointer to the tail pointer, are called as in-flight instructions. Because we know that these instructions will be there in different hardware components in our superscalar processor. Either they will be there waiting in the reservation station or there will be executed by the functional units or there will be just finished their execution but not yet completed their commit stage.

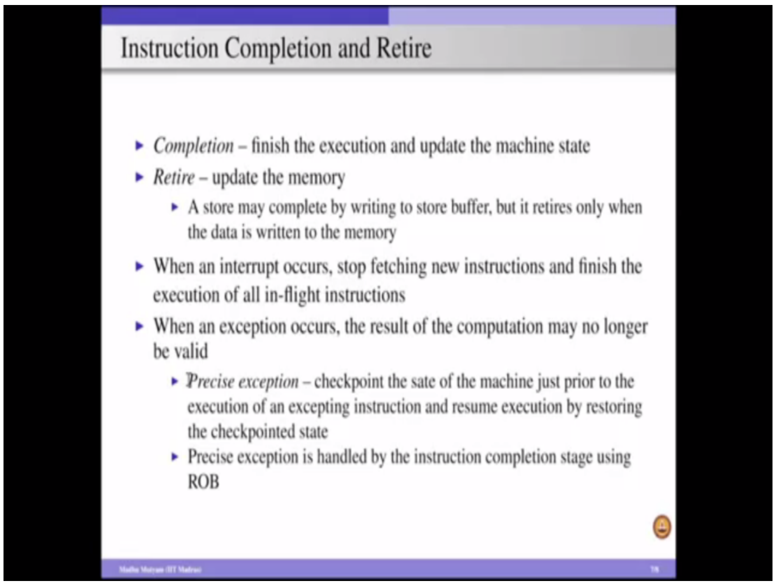
So, these are called as in-flight instructions and for all that in-flight instructions, we are going to have one entry in our reorder buffer. And using this reorder buffer we will ensure In-order commit, so that the program correctness is maintained, even when there is an interrupt happens or even when there is an exception happens. With the help of this reorder buffer we can execute instructions in out of order fashion, without creating any program incorrectness. When we are issuing any instruction in a speculative way, we have to set the speculative bit, but after some time the speculation is resolved. That means like for an example for a control instruction, we predict that the branch is going to take place and we dispatch instructions in the predicted path.

So, all these instruction which are dispatched predicted in path we have set a speculative bit one, and after some time the branched condition is resolved. And if our prediction is correct that means whatever we speculated earlier is actually correct, then in that case we have to reset all this speculative bits associated with these instructions, which are dispatched in a speculative way. So, those things will be reset so that these instructions can be committed once they finish their execution.

So that means like whenever we are going to select an instruction pointed by head pointer from the reorder buffer to complete, we have to ensure that the speculative bit for that instruction is reset, and also the valid bit is set and also whose finish bit is also set. Because there may be a scenario where instruction can come to head of this ROB but the instruction is not at its finish its computation, if it is so its finish bit is set to be a 0.

So, as a result we cannot proceed with this and even when there is a trailing instruction which finishes its execution, we cannot commit this second instruction unless the first instruction is completed. So because of this restriction, we are able to complete the execution, we are able to ensure the in-order commit. So, as a result, like a in the entire superscalar processor design, there are three important components one is reservation station, the second one is reorder buffer and the third one is dynamic execution core that we are going to discuss in the coming modules. These are the three critical components in our superscalar processor to enable the dynamic scheduling.

(Refer Slide Time: 41:57)



Instruction Completion and Retire

- *Completion* – finish the execution and update the machine state
- *Retire* – update the memory
  - A store may complete by writing to store buffer, but it retires only when the data is written to the memory
- When an interrupt occurs, stop fetching new instructions and finish the execution of all in-flight instructions
- When an exception occurs, the result of the computation may no longer be valid
  - *Precise exception* – checkpoint the sate of the machine just prior to the execution of an excepting instruction and resume execution by restoring the checkpointed state
  - Precise exception is handled by the instruction completion stage using ROB

Finally, so we have this completion stage. So this completion stage will finish the execution of an instruction and update the machine state. So, if I consider an example “ADD R1, R2, R3” where R2, R3 are source operands and R1 is the destination operand. So once add operation is finished then the instruction is set to be finished its execution, and when we write the value to the architectural register, that is R1, then we say that this instruction is completed. When we update the architectural register with the computed value then this instruction is said to be completed. In other words, when we are writing to the destination register, then we are actually updating the machine state.

And once the commit is done we cannot recover the previous state. So after the commit is done we overwrite the earlier content that is stored in R1 with the new content and once the over writing is done, we cannot recover the earlier data that was stored in R1. So that is a reason why whenever we are going to complete an instruction, we have to ensure that this completion is not going to create any problem with the machine state.

That means like, unless we are sure that the instruction is needed to be completed, we have to keep that instruction waiting. Once everything is taken care, once we know that, once we are sure that we can go ahead with a completion of an instruction, then only we can proceed with the completion stage. If that is not the case we should not do that. So in other words for example, if we issue an instruction in a speculative way and the speculation is not yet resolved, then we should not proceed with this instruction for the completion stage. If you do that after completing speculatively executed instruction, we update the machine state.

That means, we update register content and after that if we identify that our speculation is wrong then we cannot revert the state back to the original state. So, that is going to create problems so that is a reason why we have to ensure that only the proper instructions will be going through the completion stage. So for that our reorder buffer will help in identifying suitable instructions which can go through the completion stage. And all the ALU and the load instructions will go through this completion stage and will update the architectural registers. In other words we also say that the machine state is updated.

So by the way machine state is nothing but the state of all the contents of all architectural registers and the contents of our program counter and another things. And we require this machine state whenever there is any exceptions happens or whenever there is any interrupt happens, we have save this machine state. So that when we resume the execution then we have to execute from the proper state that is the reason why we need to remember, this machine state. And for all the store instructions, we have to update the memory with the store instruction content, with the content specified in the store instruction.

So, we consider a separate the pipeline stage in the superscalar process that is called as a retirement stage. And the retirement stage is for only the store instructions so this retirement stage will take the instruction from the store buffer, and will be written to the memory whenever the memory is free. We also know that among the load and store instructions, loads are critical from the performance point of view and the stores are not so critical. As a result whenever there is contention for the memory to be used by both loads and stores, we have to give priority for the loads so that the processor will not be stalled for the load value and it can improve the overall performance. Since, we are giving importance for the load instructions, when there is a load and store instructions contending for the memory.

So this store instruction will be writing to the store buffer and from the architecture point of view, we can say that the store is completed, but the actual data will be written to the memory only in the retirement stage. So, another words store may complete by writing to the store buffer, but it retires only when the data is written to the memory. So, there may be several cycles between writing to the store buffer and writing to the memory because memory will be busy with dealing with the load instructions and so on.

So, as a result we will write all the contents of this store instructions to the store buffer as long as store buffer has enough entries. If store buffer is full, then we have no other option we have to take the control of the memory and write the data from the store buffer to the memory. So that we will get some free entries in the store buffer after that again we can to give the control of the memory to the load instructions, if there are any load instructions waiting to be serviced. And while we are executing the instructions, we may get an interrupt. Interrupt can be issued by the operating system or some other external device. So, whenever there is an interrupt occurs then we have to service, we have to take care of that interrupt. That is by executing the interrupt service routine associated with that particular interrupt.

So, effectively whenever there is an interrupt occurs, we stop fetching the new instructions into the system, but we are not going to flush the pipeline stages because different instructions may be there in different hardware components, and all the in-flight instructions, we are going to execute them completely and then we start executing the interrupt service routine. Only when the in-flight instructions are completed their processing then only we start servicing the processor interrupt service routine.

And so that is again with the help of reorder buffer, we are going to take care of this scenario. Because we know that which instructions are in-flight by using the reorder buffer all the in-flight instructions, are indicated by the pointer starting from the head pointer to the tail pointer in our ROB, all the instructions which are in-flight and all the instructions we have to complete their execution. But we stop fetching the new instructions until we service our interrupt service routine. This is about the interruption, and this can happen because of an external event or the operating system will issue an interrupt.

But there is another scenario where there is no external interrupt, but with the execution of an instruction we may get an exception. For example, when we are dealing with a floating point instruction, we may get an overflow or when we are dealing with an ALU operation, we may have a “Divide by 0” exception. When we have such type of Overflow, Underflow, Divide by 0 exceptions and so on, then we have to deal with those things in our execution of the program. So that is where an exception happens the result of the computation will no longer be valid and because we know that using superscalar processors, we execute instructions in out of order fashion.

So, when we are executing the instructions in out of order fashion a trailing instruction may be executed before the leading instructions and this trailing instruction may create an exception, such as “Divide by 0”. When such things happen, then we need to recover the state of the machine or we need to restore the proper state of the machine or we need to the go back to the proper state of the machine before the execution of this trailing instruction which created an exception. So, that means like we need to have the support in our processor to deal with this exception scenario.

If any processor which has support for this, then we can say that the processor supports the precise exceptions. If the system supports the precise exception then the system can check point state of the machine instruction by instruction so that at any point of time if any instruction is going to create an exception, we already save the state just before that exception instruction, so that we can compute without any problem.

So, again ROB is going to take care of these precise exceptions, and whenever we execute an instruction, which creates an exception. So, what we are going to do is all the instructions starting from that exception instruction, and all the trailing instructions which are issued starting from that exception instruction will be invalidated. Because we have one field in our ROB per entry that is a valid bit and we can invalidate the corresponding instruction so that these instructions are not going to have any impact on the machine state or the processor state.

Effectively, so using our ROB or reorder buffer we can take care of this precise exceptions. So, with that I am concluding the organization of superscalar processors and in the next module we are going to discuss another important component in superscalar processor that is Register Renaming. And we know that register renaming is used for eliminating the name dependencies.

Thank you.