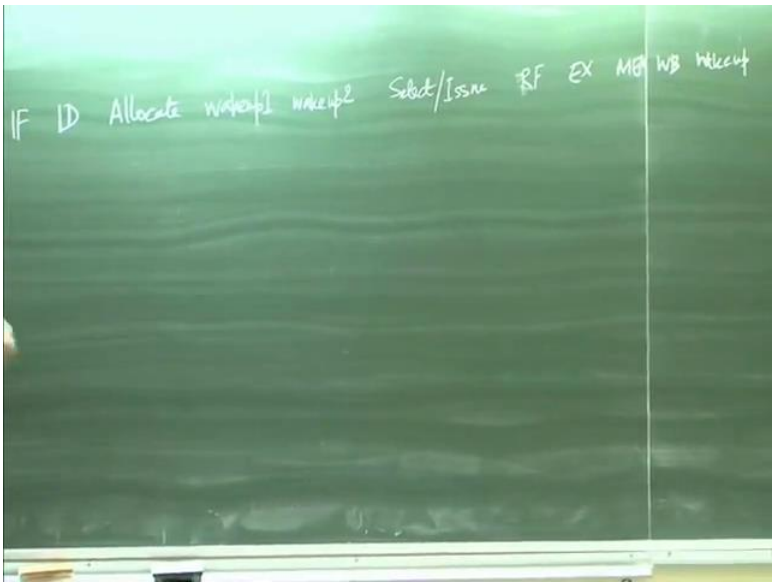
Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

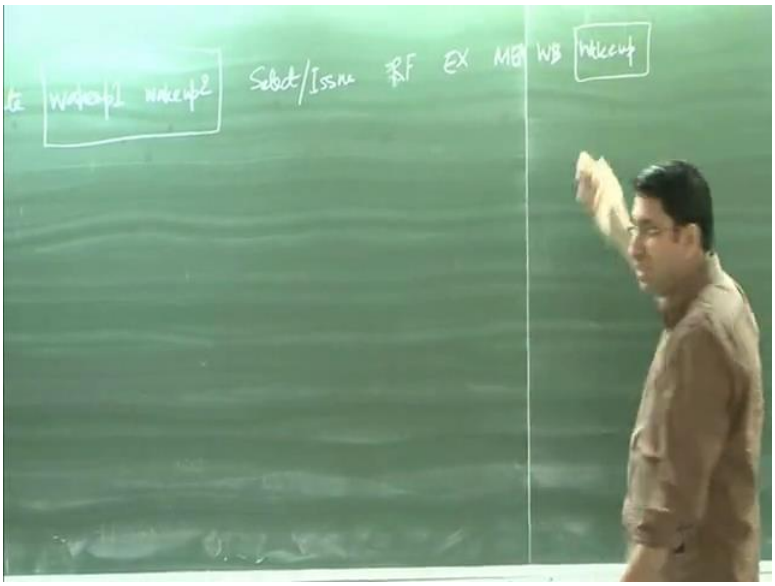
Module - 1
Lecture - 22
Dynamic scheduling, Speculative Execution

(Refer Slide Time: 00:41)



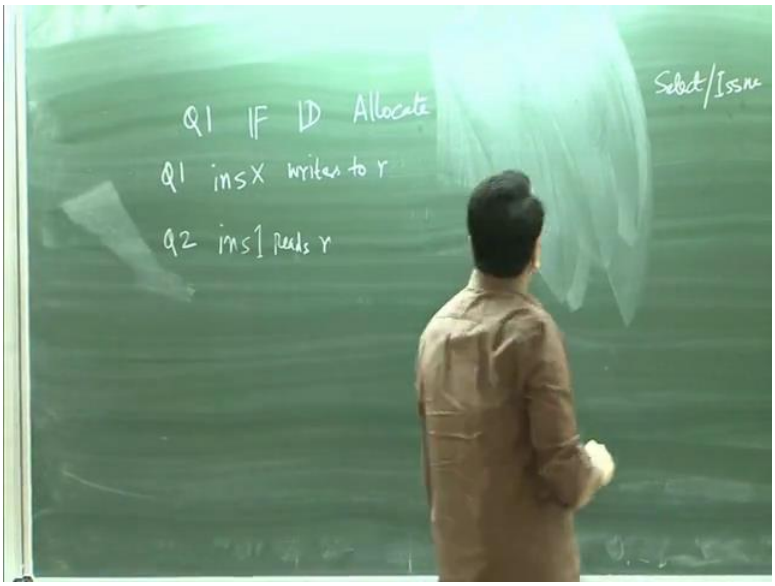
So last time we were discussing about a pipeline model, where we have a bunch of ready instruction, and we make a selection out of them and execute them. So, this was the pipeline that we came up with. So, you every cycle you fetch mother instructions, you decode them in the order that the compile race presented to you, and you allocate them in that order in an issue queue. And then and instruction may sit in the issue queue waiting to be issued, and when the instruction becomes ready, it takes part in selection algorithm, and the selection algorithm may or may not select a ready instruction for each. And eventually the instructional issue and read the register file, or the data from the queue from its parents. Execute them goes to the memory stage if it needs, and put the result back in the queue. And finally, when the instruction comes to the head of the queue, it will write the result back of the register file, and also wakeup (( )). And we discussed why we need two wakeup cycles here, why we need 1 wakeup cycle here. Of course, there is one solution, the solving the races there may be many other solutions.

(Refer Slide Time: 02:11)



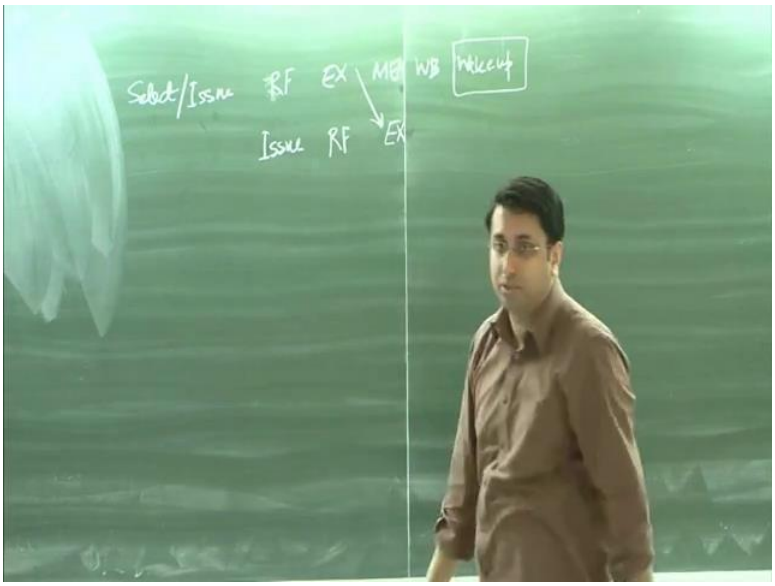
So, before moving on I just wanted to touch upon a few points, do you have any questions. So, often you will find that these two stages; usually not mentioned in the pipeline, over instruction, and the simple reason is that the number of cycles the instruction may have to wait in the issue queue is non deterministic, because it depends on when its immediate predecessor actually executes, until then the instruction is not ready. So, it is not like there will be these two cycles are waiting here. So, that is why, these two cycles are often not mentioned in the pipeline, the understanding is that, there will be non deterministic amount of delay between allocate and selection. So, that is what is implicitly understood. Similarly this cycle often is not mentioned, the reason is same; that this particular wakeup cycle get essentially club here, in this particular wait; however, many cycles instructions to wait in the issue.

(Refer Slide Time: 04:28)



So, that is one issue. The second thing is that. So, is the point clear to everybody that there will be a non deterministic amount of delay from the time the instruction enters in the queue, if the time it gets selected. It all depends on the selection procedure, and depends on when the predecessor executes. The second thing is that; last time we said that when the instruction completes execution, so the way wakeup work is that you will, you can essentially communicate your slot id to everybody in the queue, and they will compare against the dependence, that they depend on. If there is if the comparison passes then you know that the result is ready. So, we said that that happens when the instruction completes execution, so that is not really needed. What actually happens is, you can do it much earlier. So, I will remove this, because we really do not worry about these two cycles here. So, as soon as an instruction gets selected, the issue at that time, and at the same time it actually sends it slot id to everybody.

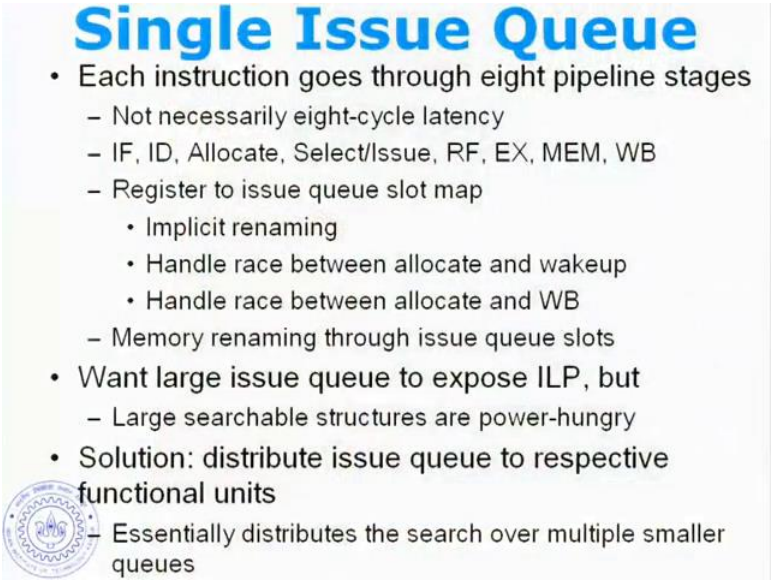
(Refer Slide Time: 05:42)



So, the point is that suppose there is an instruction, which requires register r, some instruction, and there is some instruction before it, which produces register r. So, writes to r and this one reads r. So, last time we said that this instruction will send its. So, this instruction will have. So, let us suppose this queue slot for this instruction is q 1 this is q 2. So, q 1 will be broadcast to everybody, and q 2 will have q 1 as producing r, and they will make a comparison.

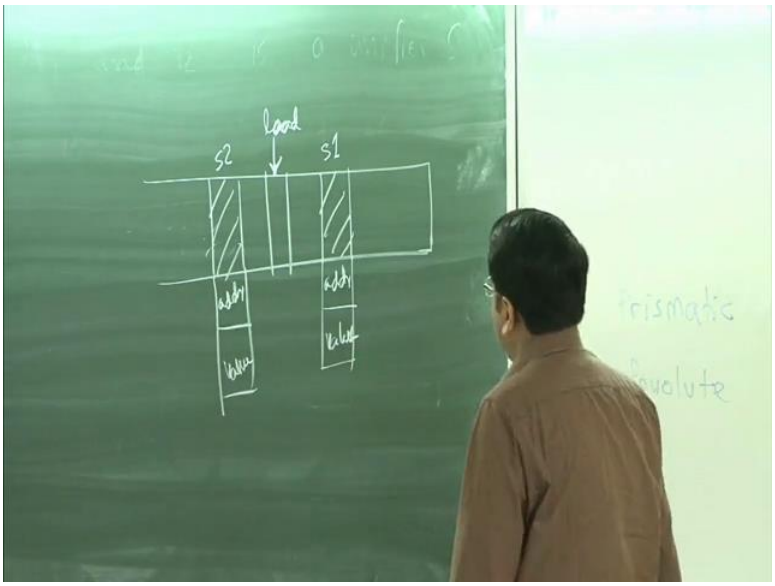
And at that time you get to know that q 2 is ready, and q 1 sends its slot id only after it completes executions. So, that is not needed what you actually do is, as soon as instruction q 1 issues, it will actually send this slot id, and in the very next cycle instruction q 2 may issue. So, essentially what will happen is that your. So, if this is q 1, then q 2 will essentially issue here in the next cycle; that is the earliest it can issue. It will read the register file here will get a wrong value and will pick up the correct value from q 1. So, last time we were not making use of the bypass straw. We are saying that, you wait until the execution is done and then only you broadcast, so that the instruction will read the value from the queue slot itself; that is what we needed, you do not have to wait. So, much you can issue it in the next cycle, and the value will only picked up from the by parts. Is it clear?

(Refer Slide Time: 07:56)



So, these are the few points that we left out last time. So, from here will move on. Yes because of purpose of that is to make sure that no instruction indefinitely ways, and need to tell the. Because see, if it is written back then you need to know where to read the data from. So, that is a purpose of that wakeup others that the data is now will be read from the register file. You can possibly avoid that cycle some extra machinery, but complex to do is pretty much be same. So, this is the summary of what we discussed last week, each instruction goes through 8 pipeline stages. Not necessarily 8 cycle latency, because there maybe bunch of cycles here which are not shown. So, that is what is implicitly understood, and also there maybe bunch of cycles here, from the time an instruction completes execution with the time it writes back. It may happen, waiting in the queue for a long time, until it comes to the head of the queue. So, fetch decode allocate, select issue, register fetch, execute memory write back.

(Refer Slide Time: 09:21)



And we also talked about the map table, which maintains register to issue queue slot mapping, because it tells you which register is owned by which queue slot; that is which instruction is producing what register, so, that does and implicit (( )). It renames the same register to different queue slots, depending on the production of the register. We also talked about how to handle races between allocate and wakeup, how to handle races between allocate and write back. We also talked about memory renaming, through issue queue slots. So, just to remind you this particular process, which said that you may have. So, this is my issue queue, you may have two store instructions; store 1 and store 2. So, you may have two store instructions, and the way the stores execute, we said that when the store is issued. So, when do you issue a store, when it has the value to be stored is ready, and also the register required to calculate the address is ready ,which requires two operands. when both are ready you issue the store, what the store instruction does is that, it reads the value to be stored from the register file as usual, or from other queue slot or from the bypass, and it will also calculates the address in this stage, but it does not really access the memory straight to the pipeline.

So, it stores the address here in the queue slot, stores the value also here, similarly here. So, even though these two address are identical. These two stores can execute out of order. These two stores can execute concurrently without any problem. They will execute both will compute the address, both will put the value here; however, the write to memory will happen in order, when they go to the q1 of the other, that would preserve ((

)). So, this is implicitly doing a memory renaming, it is essentially this instruction is renaming this particular address to this queue slot. This instruction is renaming this same address to another queue slot. So, that pi is concurrency which was not possible. Now if we have a load instruction; let suppose somewhere here. And the load instruction requires just one registry operand being issue, which is required to compute the address. So, what may happen is that this address may turn out to be exactly same as this address. So, there is a danger that this load if it is allowed to execute without any regard for this particular store, it may get a wrong value from memory, because of value this value is not yet in memory, still sitting here.

So, what you said was that the load execution is with the two parts. So, when its ready it issues, it reads the operand from register file, or picks it off from bypass, or picks it off from the queue slot. In this particular stage it computes the address, and before it accesses memory, it takes this address and compares an address with all previous stores, and if there is a match, then something special has to be done. So, in this case for example, the value can be taken from directly here. It is not going to memory, because this is the value actually you know it.

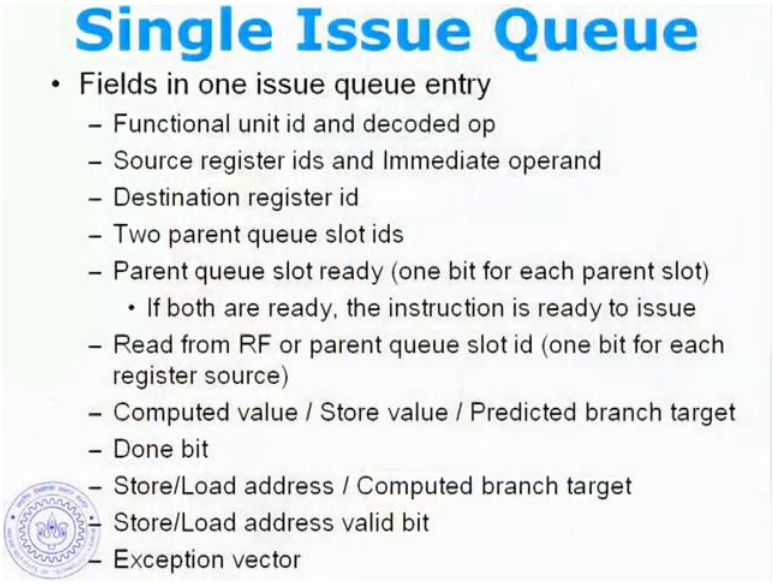
So, so memory naming is also done through issue queue slots. So, at the end what you really want is, to have a very large issue queue to expose instruction level parallelism, because if we have very large queue, the probability of picking independent instructions actually increase, but the problem is that, large searchable structures are power hungry. So, that is the known fact, I cannot explain it here at this point, because many other prerequisites, but this is a big problem actually, which is why and it is a searchable structure, which you have to search this for many purposes; one is for figuring out dependence, one is for figuring out load store dependence, and all other things.

Student: (( ))

So, usually the way it is implemented is that, you make a. so it is actually a comparison, that is what you wanted to do. So, you could as many comparators as needed before the number of two slots. So, all the comparisons will happen in parallel. So, time is not a concern. The problem is, the energy (( )); that is the biggest issue. This will be essentially accessing all queue slots, and switch in so many comparators. So, today what people do is, the solution is that you distribute the issue queue to respective functional units. So,

essentially what you do is, you say that well I have an issue queue attached to my load store unit only. So, that is supposed to be small issue queue, which is not only load store instructions. I will have an issue attached to my integer ALU, which will hold only integer ALU instructions. So, essentially you are distributing the queue in several parts, so that the the complexity of the design goes down. And also a load instruction need not be compared against all instructions. You can only compare against the instruction that can available. So, essentially distributes the search over multiple smaller queues. So, that is one implementation trick that is done to be in our processors as oppose to a single issue.

(Refer Slide Time: 14:57)

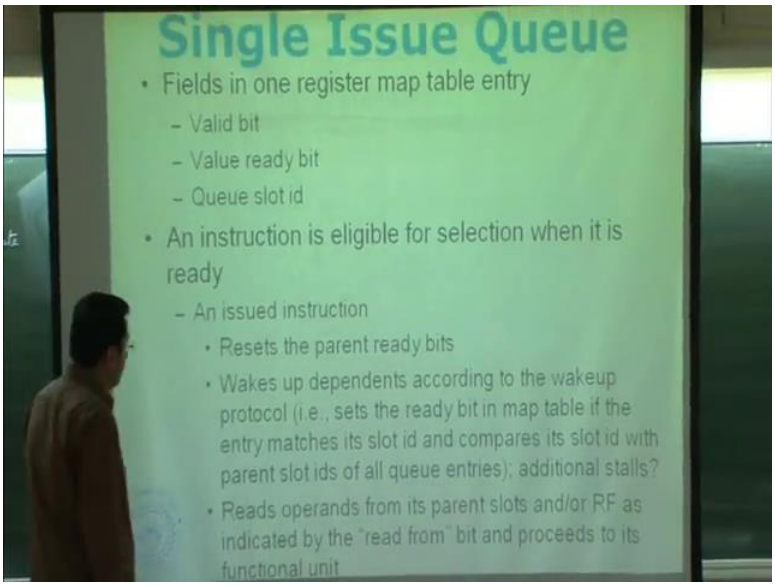


So, let us try to summarize what fields require in each issue queue entry, that might actually help you also remember what exactly needs to be done. So, each instruction needs to carry the functional unit id, and the decoded op code. So, that is has to go in the issue queue entry, because when the instruction issues, it has to figure out what to do and where to do that. Source register ids and immediate operand, is a needed for reading a register file, and this needed for computing components. Destination register id required for write back. Two; parent queue slot ids that will hold, which are the instruction that will immediate predecessors, as far as these two sources are concerned. So, more the instructions are producing, these two source registers before the parent queue slot ready. So, essentially that tells you when these two slots are actually ready, so they are 2 bits. If both are ready then the instruction is ready to issue. Whether it should read from register

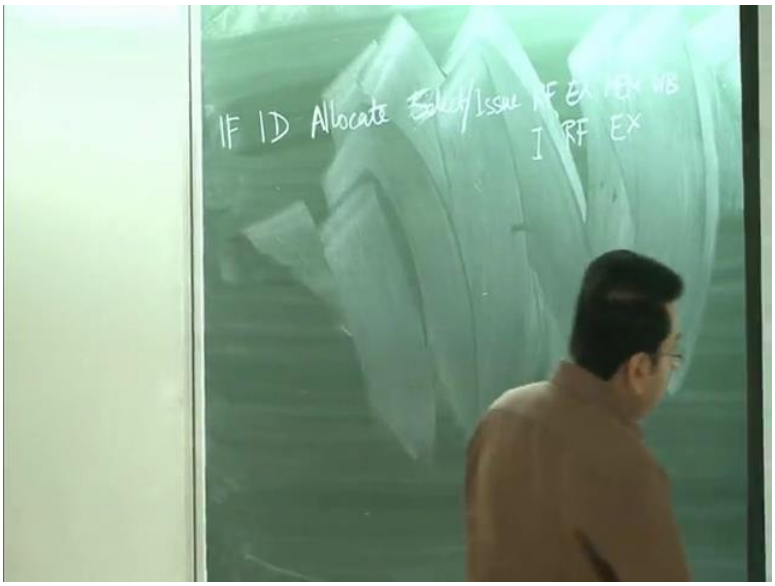
file or parent queue slot id, this also we discussed last time. So, when an instruction is written back, you can read the value from register file as the waiting for the parent id, so 1 bit for each register source. And then you need space for storing the computed value, because the value goes back to register file only when you write back, not before that and the same field is actually used for storing the store value, this particular value, because the store instructions do not compute anything so you can use the same field. Also is used for storing the predicted branch targets, because you have to figure out whether the prediction was correct or not.

There is a done bit which tells you the instruction is complete, execution. You also have to store the store load address for this purpose, when a load issue is jut to compare is with every store before it. Also you store the computed branch address in the same field. And also you need a values bit for this particular thing, and you have exception bit, to figure out if the instruction to or not. So, the summary of one issue queue entry. You may require many other small bits here and there, depending on implementation, but these are the main things that you remember; execute one , wakeup its dependence and everything. Any question.

(Refer Slide Time: 17:48)

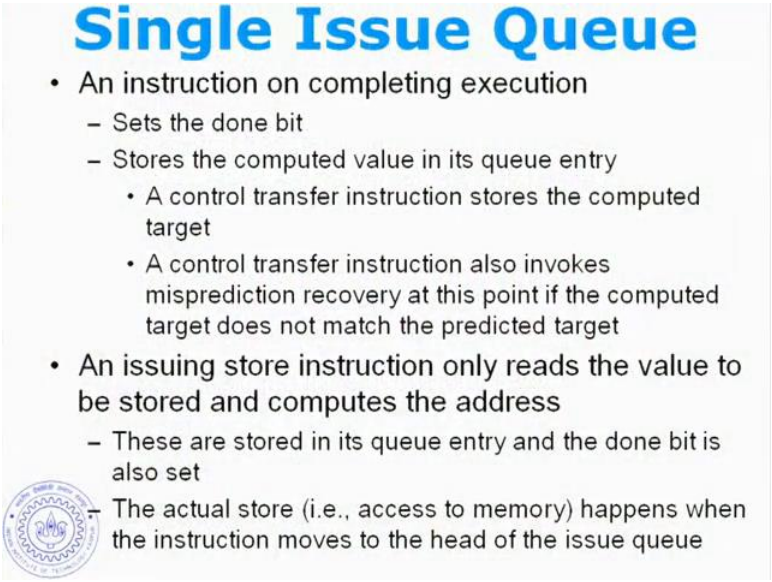


(Refer Slide Time: 18:45)



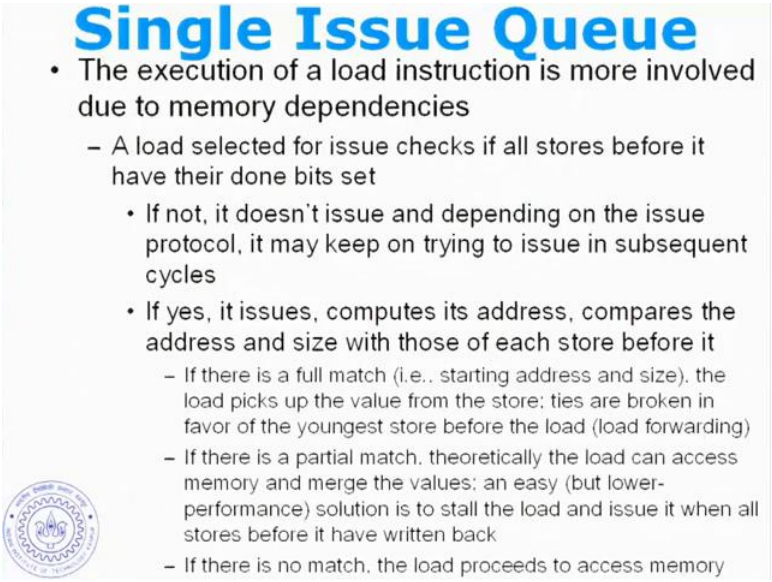
What are the fields in one register map table entry, valid bit. You need a value ready bit which we discussed last time for resolving a race, and you need one queue slot id, which tells me which slot id correctly owns this particular instruction. So, an instruction is eligible for selection, when it is ready, and what does an issue being instruction do. It resets the parent ready bits first of all, because this instruction has already selected and we should, so you do not need these bits anymore. Wakes up the dependents according to the wakeup protocol; that is sets the read bit in map table if the entry matches its slot id, and compares its slot id with a parent slot ids of all the queue entries. So, this one we discussed last time how to wake up instructions. There may be additional stalls even if your parents become ready because of interlocks. You may have to install implement that, so that is what we talked about earlier; the load interlocks and all. So the point is that, suppose your issuing a particular load instruction in cycle, and the next instruction depend on the load. So, if that instruction issues here, these the file you cannot really get the data at this point, because a data is produced only here. So, there may be additional interlock cycles that the selection hardware may have to implement, so that is what work here.

(Refer Slide Time: 20:02)



As soon as the parents become ready it may not be enough to issue the instruction. There may be other scheduling constraints. So, once the instruction issues it reads operands from its parent slots, handover register file as indicated by the read from bit, and proceeds to its functional unit. And instruction completing execution sets the done bit, stores the computed value in its queue entry. A control transfer instruction stores the computed target, because that we needed for figuring on these predictions. A control transfer instruction also invokes misprediction recovery at this point, if the computed target does not match the predicted target. An issuing store instruction only reads the value to be stored and computes the address. These are stored in its queue entry, and the done bit is also set, so that is what we discussed here. The actual store that is access to memory happens when the instruction moves to the head of the issue queue not before that.

(Refer Slide Time: 20:42)



(Refer Slide Time: 21:40)

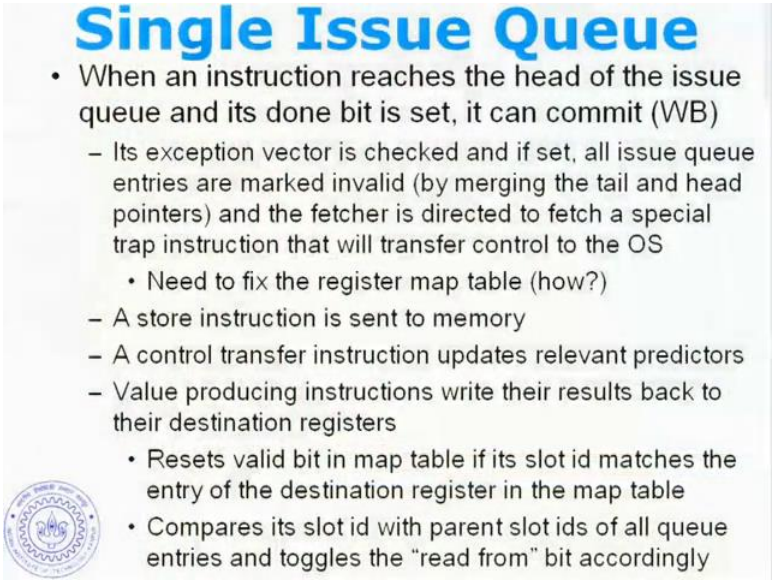
The execution of a load is more involved due to memory dependencies. So, I just explaining that here, so this is summary of that. a load selected from issue checks, for issue checks if all stores before it have their done bit set, if not it does not issue, and depending on the issue protocol it may keep on trying to issue in subsequent cycles. So, what I saying is that, suppose when the load becomes ready, this store is not ready to execute. So, what does the load do, it has to get; that is what this cycle. If we has; that means, all the stores have already executed before it, computed the address, then the load issues, computes its address compares the address in size with those of each store before

it. If there is a full match that is starting address and size, so a load essentially does what. It loads bunch of bytes. So, there is a size. So, there is a starting address a and there is a size. So, a full match happens when the load finds a store before it, which has the exact same size and the same starting address; that is the full match. So, this is an easy case, because in this case the load picks up the value from the store. It just contributes this value, that is it.

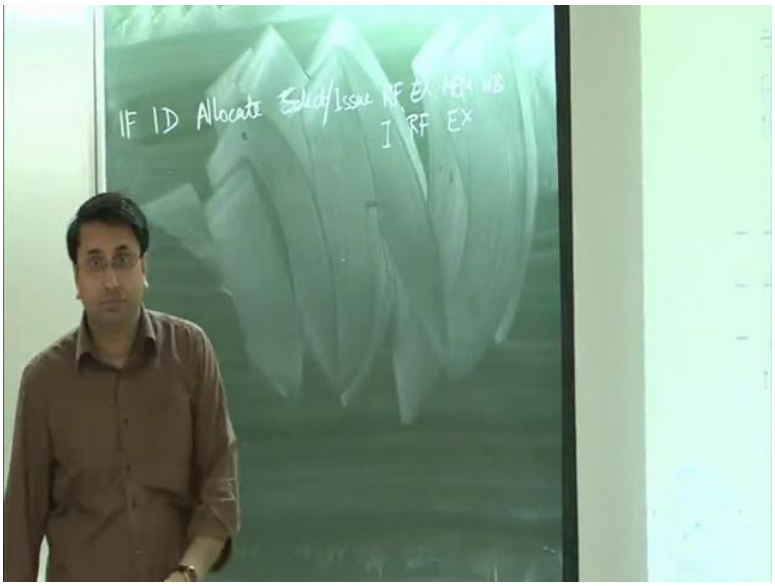
Ties are broken in favor of the youngest store before the load, that makes sense, because the load must be contribute the value from the immediate predecessor. So, this called load forward; is it clear any question. If there is a partial match. So, when talking about the case where. So, this is my load, and the store happens to be like this. This is the address and this is the size. So, load over lash partially with the store. So, these are very problematic situation. So, theoretically, the load can access memory, and march the values, that is possible actually.

So, the load could actually read out these bytes from memory, and pick up these bytes from the store, to prepare the final load, but these, so this is possible; however, an easy solution would be to stall the load and issue it when all the stores before it have written back, this much load performance, but much simpler actually, because this merging hardware it may sounds very easy to implement. It is not that easy actually, because I am just talking about one store here. There may be multiple stores; there may be another store like this. So, how many things would you merge now? So, this merging hardware gets more or low compliments that we think about cases. So, often this is what the processors implement. So, they would just let the load wait until all the stores complete, and then read from memory.

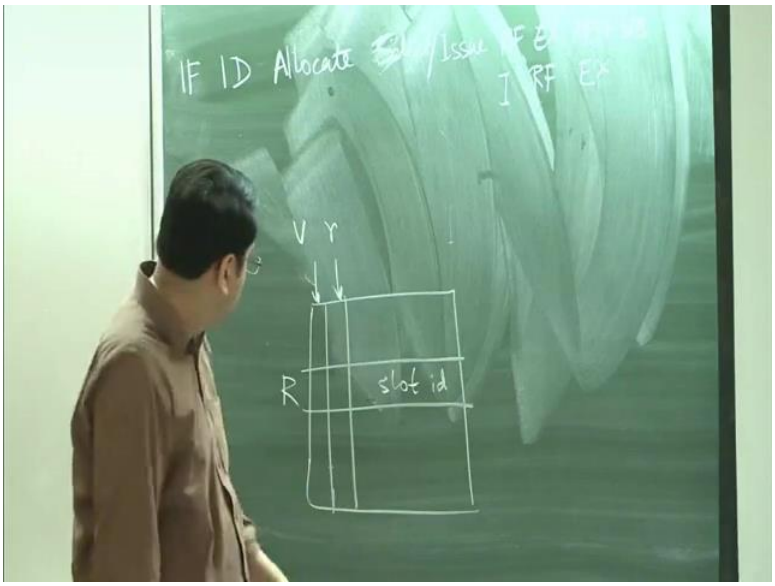
(Refer Slide Time: 24:14)



(Refer Slide Time: 25:15)



(Refer Slide Time: 26:39)



If there is no match, the load proceeds to access memory. So, this is the usual case. There is no store that matches, so there is no intersection at all, the load can read from memory. Is it clear, how a load executes. And finally, when instruction reaches the head of the issue queue, and it is done bit is set, it can commit. So, that is the technical used it is called an instruction committing; that is essentially the write back of instructions. So, what happens here. The first thing that you do is you check its exception vector, and if set; that means, the instruction must have taken the exception somewhere inside (( )). What you do is, all issue queue entries are marked invalid, by merging the tail and the head pointers. And the fetcher is directed to fetch a special trap instruction, that will transfer control to the operand system.

So, essentially you are removing all instructions after this particular instruction from the processor, and preparing processor 2 and in the etcetera. The big question is how do you now fix the register map table, because the register map table has been modified in this particular stage, and it points to, I mean the status corresponds to last instruction allocated here. How do you fix the table now. It has to roll back to point 2 this particular instruction, whatever the state was at that point. So, this is my, is the state of issue (( )). So, this much data I have, can I recover the map table from this. So, is this problem clear to everybody, what we are talking about. An instruction is taking an exception; the map table has run much further high actually. Now you have to role it back to this point. Can I recover it from these states. For each instruction in the queue I know all these things.

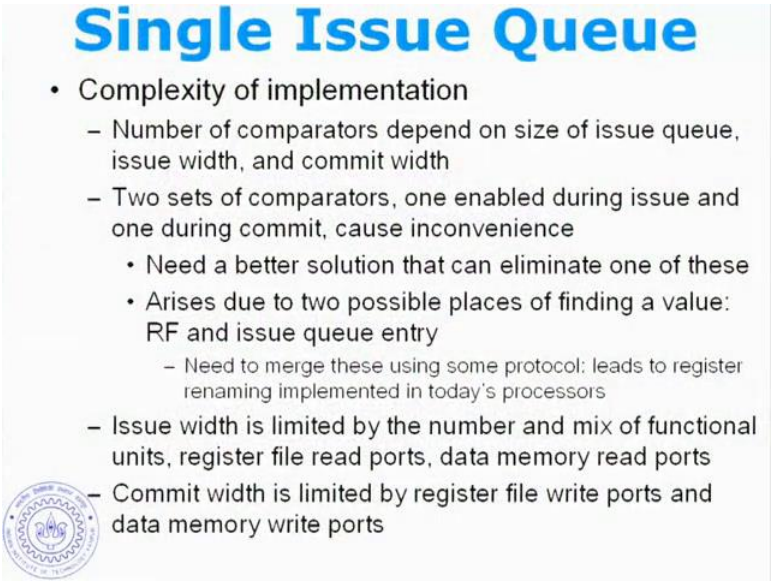
So, this is my map table, each, every has three pins. So, for this register; register r let say, it tells me which slot id was the one to produce this register last. The last instruction to produce this register, this particular slot. This field tells me if this instruction is already executed, and the register is ready in the slot id, the value is ready in the slot id. And this one tells me this bit if this particular map is at all valid or not. And to remind you this bit is turned off when the slot writes back, provided this slot is still there in this registers map. So, how do I now recover this entire map table, to a state of the accepting instruction.

So, when accepting instruction was allocated, I want back to that state. So, can I do one thing. So, this ready bit here. It is 0 when the value is not ready in the slot, and otherwise this one right when it executes. So, can I sit all this bits to 1. The only instruction that is currently held on queue is excepting instruction, everything after it will be killed, they will be fetched again and execute again. What will be the still the valid bit. So, forget about this 1, I am just asking about this two states, what should they be after you recover. Sorry both 0 sorry what. Which one is 0. r should be 0 yes. What about the value bit. 0 why (( )) exactly. So, point is that, when the exception handle starts all the values produced by the program till this point, are all in the register file. There is nothing in the queue at this point. So, this map table essentially has no meaning. So, I can mark everything to 0, all the valid bits and I do not have to or anything else in this table. There is no map that is valid. Is it clear to everybody, then I can just clear this valid bit column, and I am done. So, when the whenever exception handle starts, it will reestablish all the mappings one after the other. Clear or not clear.

What was the purpose of the map table. The purpose was that when instruction comes which needs to read from register r, it will know which what its parenting actually. So, now, there is no parent actually, everything is written back to the register file. It is like a reorder pipeline, the one that we discussed. So, I need just clear this bit and bit done. You do not you have do anything else. So, this is easy, fixing the register map table. So, that is exception check, assuming that there is my exception. Store instruction is sent to memory at this time, when the instruction commits. A control transfer instruction updates the relevant predictors, it is the time to update the predictors, because you know the correct outcome, and you know that this instruction is about commit. Value producing instructions write the results back to the destination registers. So, it resets the

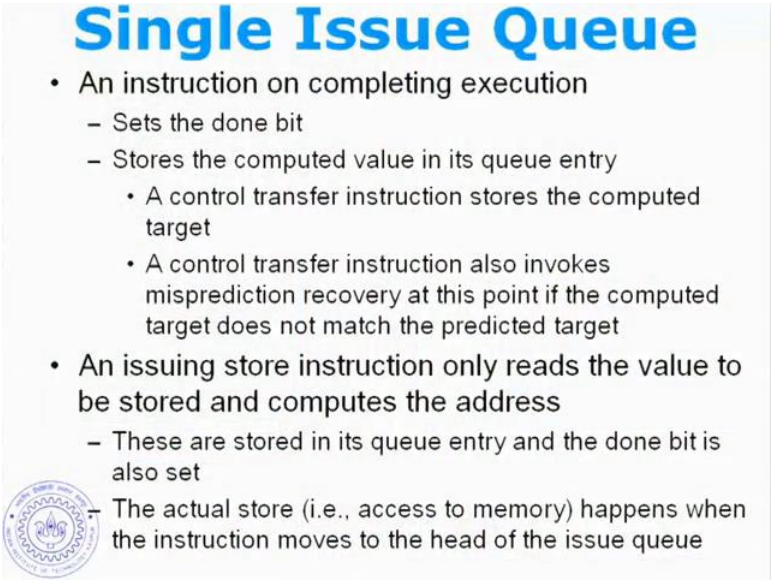
valid bit in the map table, if its slot id matches the entry of the destination register in the map table. Compares its slot id with parents slot ids of all queue entries, and toggles the wait from bit accordingly. So, these are this is the wakeup stage that we talked about, clubbed with the write back. Any question. So, that is the commit protocol that is what happens at commit stage.

(Refer Slide Time: 32:21)

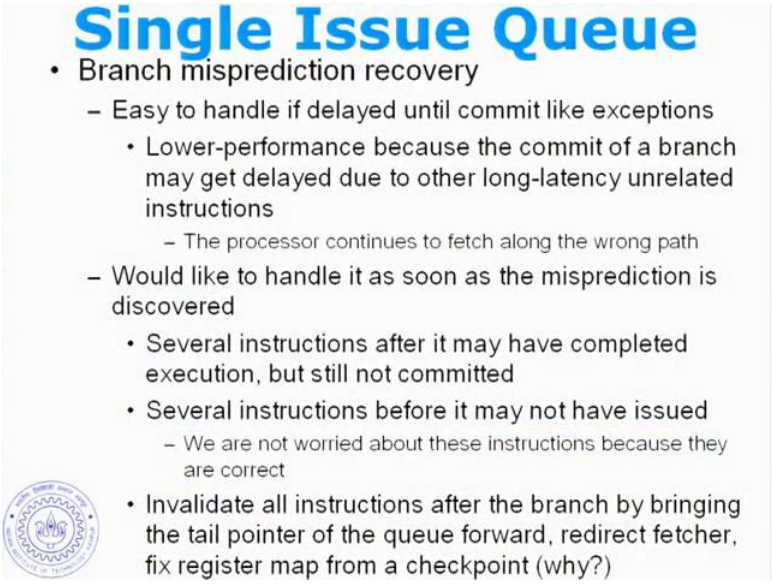


So, how complex is this implementation. Number of comparators, depend on size of issue queue issue width, and commit width, why is that. So, issue width is the number of instructions that you issue in a cycle. And each instruction will send its queue id to everybody for comparison. And everybody will make two comparison, because they may have two dependence. So, number of comparisons in this particular wakeup stage is, two times issue width times the length of the queue. And when instruction commits, it will have to do one more wakeup, so that depends on the commit width, how many instructions I can commit in cycle. So, that is again twice commit width times the number of queue slots. So, two sets of comparators; one enabled during issue, and one during commit cause inconvenience. So, you need a better solution that can eliminate one of these. And why we need these 2, it arises due to two possible places of finding a value. Currently the a value maybe in the register file, or maybe in the issue queue entry, depending on a state of instruction that produces the value.

(Refer Slide Time: 34:21)



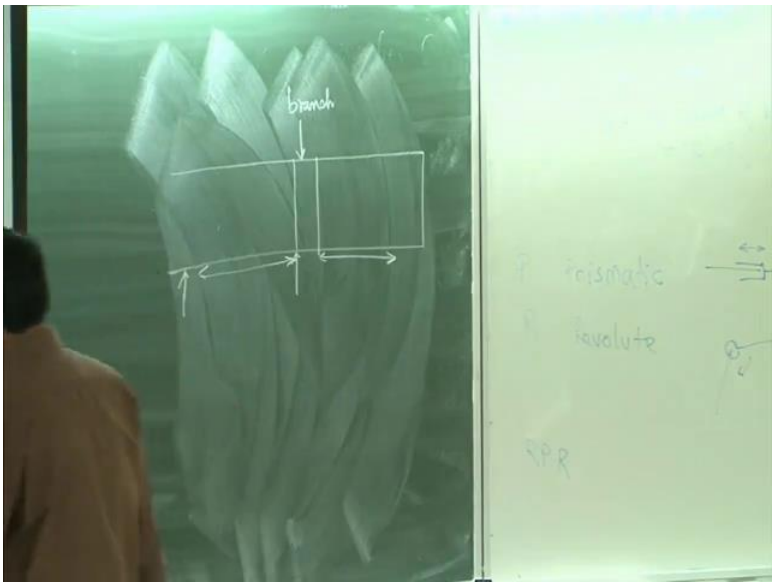
(Refer Slide Time: 34:50)



So, you need to merge these, using some protocol, and that is what today's process do, based to register renaming implementing in today's processors; that is what we discussed very soon how exactly you merge this two things into instruction. Then you can get rid of one of one set of comparators. Particularly this one you do not really need, at the time of commit there is no need to wakeup actually. What dictates issue width. Issue width is limited by the number and mix of functional units, register file read ports data memory read ports, we discussed this last time. Commit width is limited by register file write ports, and data memory write ports, this also discussed last time. So, one small thing that

is left in this processor is, how do you recover from branch mispredictions; that is small thing that we need to discuss. So, just to remind you, we said that you invoke this when a branch instruction completes execution. It invokes the misprediction recovery at this point, if the computed target does not match the predicted target. So, when the branch instruction completes execution, you have both, you compare them. If they do not match you know that something else got wrong. So, how do you recall.

(Refer Slide Time: 36:17)



So, that is easy to handle if delayed until commit, just like exceptions, because say that well why do have to recall for immediate just wait until the branch commits. So, then it looks exactly like an exception. You can say that when the branch instruction is taking an exception, so I will just do the same thing as I was doing for handling exceptions. Its lower performance, because the commit of a branch may get delay due to other long latency instructions, because remember that branch can commit only it comes to the head of the queue. There may be other instructions sitting here before the branch, which may take a long time to complete, which are completely unrelated. And essentially if you delay so much the processor to continue to fetch along the wrong path, and all those instructions will have two memory. So, this is not really done. This is not acceptable at all. The main point here is that, for exception this is acceptable, because exceptions are rare, but this is, this may not be rare, because it is not really the property of an application. Its property of the branch predictor, how smart it is. If I do not have a good branch predictor this going to be very; this particular event. So, we would like to handle

it as soon as misprediction is discovered, and the thing to observe is. So, essentially we are talking about the situation where. Let us say we have the branch instruction here, which has been issued and has executed. And you find that this branch has mispredicted , so the predicted outcome does not match the computed target.

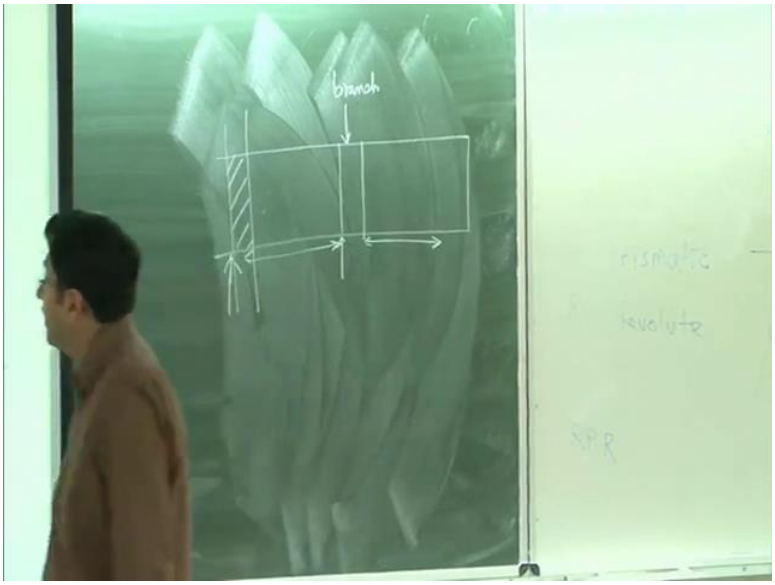
So, point is that first thing to observe is several instructions after it, may have completed execution, but still not committed. So, there may be several instructions here that I have already completed, but have not you know committed of course, because they cannot commit before the branch. Several instructions before it may not have any issue actually yet. So, here several instructions which were you waiting for some instruction here. They will be dependent on some instruction; they have not any issue yet.

So, given this particular issue queue, we would like to recover from this misprediction. So, the first thing to notice is that we are not worried about the instructions before the branch, because they are correct. We are worried about the instructions after the branch, because they are not correct actually; they are on the wrongly predicted path. So, what you do you all instructions up to the branch, by bringing the tale pointer of the queue forward. So, you bring the tale pointer probably here, or bring it out here. So, the next instruction will be allocated here, so that these will get over it, they will never committed.

Degrade the fetcher to the correct pc, and fix the register map from a check point. So, this is very different from the way we recover register map in case of the exception. So, except this point, is it clear the rest of the things what I do. So, here what I really need to do is not really going to be very easy, because now I really cannot just flash clear the valid bit and be done, I have to bring that register map up to the point what it was when the branch was allocated , and there is no easy way of actually. So, one solution here, is that for each branch instruction, whenever we allocate a branch instruction you make a copy of the map of table, so essentially that requires extra storage. And that puts the limit on how many branch instruction I can keep in the queue, because every branch instruction all comes to the check point, because there is a chances this branch instruction may mispredict. So, often the processors talked about; a sudden number of branches that you can, maximum number of branches that can remain uninvolved at any point any time. For example, which you say that I can have only 20 branches outstanding the queue; that means, whenever the decoder decode the 21st branch, the fetcher is going

to stall here, which cannot proceed any further, because there is no space for storing the check point for this particular branch instruction.

(Refer Slide Time: 40:02)

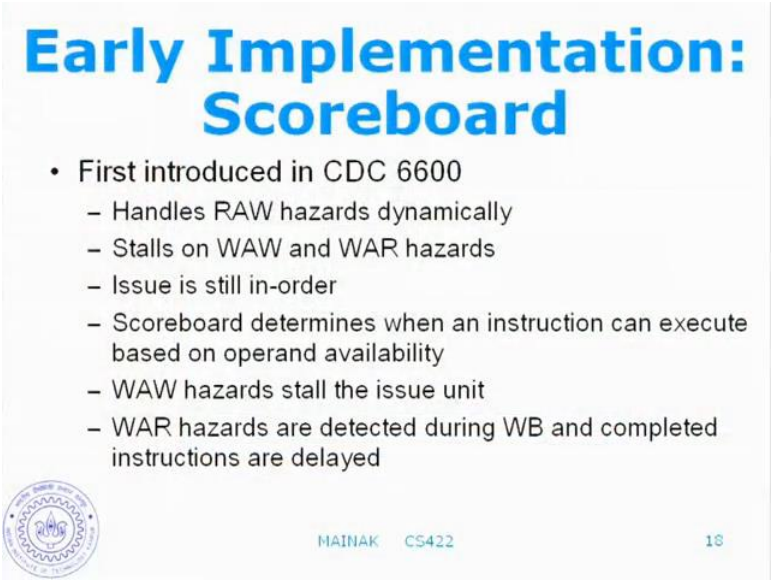


So, essentially a check point contains the copy of this table that is it. And whenever a branch mispredicts, I take that copy, copy the whole thing here. So, that I get back the map that I needed, is it clear to everybody how I recover register map. Is there any other way of recovering, which does not require a check point. So, this is what I have with these queue entries. So, currently my tell pointer is here, and allocated up to this point and; that means, I need to. So, my map table currently points to this particular state, and to bring it back here, can I recover it from the straight away instructions. Let us take the last instruction here. So, it tells me that it writes to certain register id, that sore in a queue entry. So, that is a register 20, which means if I go and look at the 20th row of this table, I must find this particular queue slot id there; that is what it being essentially

So, while recovering what I need, to make a progress I need to know, which was the previous entry that wrote to register 20. So, then what I can do is, I can change that 20th row to this 1, and then implementally move it up, I can do that. So, this request is search unfortunately. For example, register 20 I have to move forward and find out which was the previous instruction that wrote to 20. Can I improve that. Sorry say again. Exactly. So, when this instruction was allocated, what was the (()) of the 20th row, it was the previous slot id. I just remember it here in this cycle. So, I can quickly then recover, but

it still takes time, because you have to sequentially work this particular queue up to this point. It takes time and unmapped. It cannot be as fast as recovery check point, but it reads you of the check point completely. You can just recover it from the queue states; you do not need any other extra storage. So, there are two mechanisms, but today most processors actually do the check pointing, because of speed. Branch misprediction is often very frequent depending on the nature of the branch. And if the branch is towards the head of the queue, you may have to walk the queue for a long time actually, they may larger of instructions already allocated here, so that it requires.

(Refer Slide Time: 43:12)

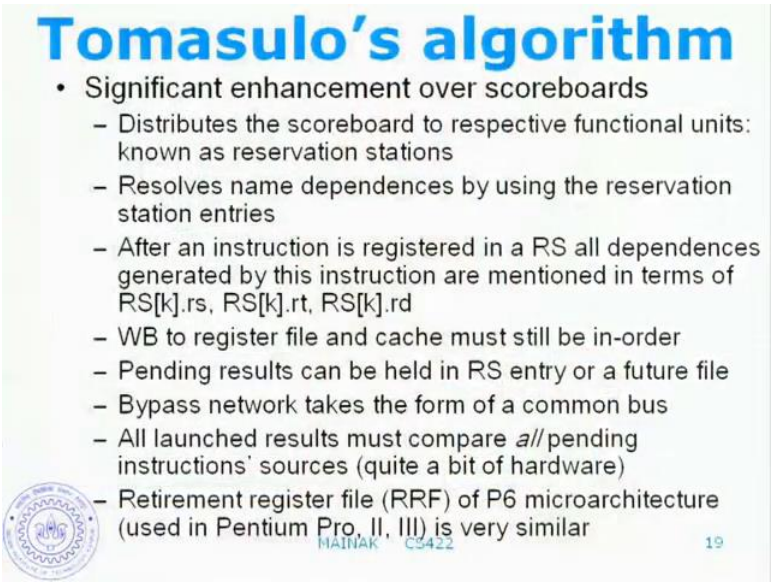


So, that is about our single issue queue processor, which essentially does out of order execution; that is program order. It gives you concurrency in terms of allowing you to execute multiple instructions in a cycle. So, any question on this. So, what we will do is now, we will track back bit try to see what people use to do in early years, when the sophisticated hardware's were not there. So, the earliest implementation of this idea was in terms of scoreboard. The first introduced in c d c 6600. So, that is a machine from control Data Corporation; that is what this stands for. So, they have a similar structure like our issue queue, which they called a scoreboard, and the name cast to the fact that, essentially each instruction gets a score which signifies whether the instruction is ready to go or not. So, it handles raw hazards dynamically just the way we have discussed, exactly same actually, it gives track of the dependence. it stalls on waw and war hazards, so the decoder actually figure out, if there could be a possibility of any of these

happening, and if there is a chance then the decoder would actually you know introduce interlock stalls, until that is, and it discussed how to do this using register and all .

The issue was still in order, you could not actually jumble instruction order. Here we have what have discussed in order in our single issue queue model is that, you can issue any ready instruction order, and the order at a time of commit , but here even execution was in order. The scoreboard determines whether instruction can execute based on operand availability waw hazards stall the issue unit, war hazards are detected during write back, and completed instructions are delayed. So, the processor previously we saw that war hazards can be written in decoder, but here the delay it until write back. So, the instructions are allowed to go, and when they come to the write back stage, they detect the war hazards, and they completed instructions are delayed. So, they have a buffer there, where they can put the results and until (()) result they can.

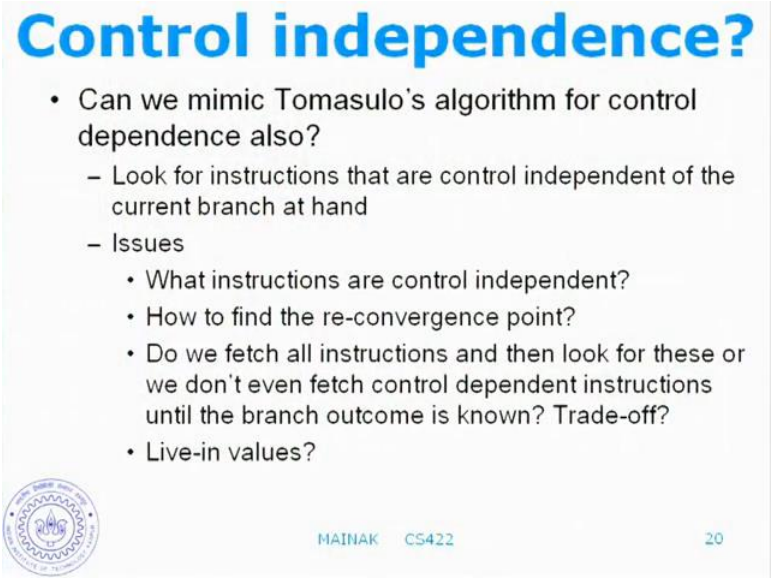
(Refer Slide Time: 46:05)



So, talks about an example in detail, so you can look at that actually, how the scoreboard actually works, but anyway I do not want to spend much time on this, because this is not really what is done today in. So, you can read the book, the book gives an example how the scoreboard works. And then what we have discussed in this single issue queue model is essentially the Tomasulo's algorithm, that also I mentioned last time. So, Robert Tomasulo was an IBM engineer, and he came up with this algorithm, when designing IBM 360 machine. Second the first inclination of Tomasulo's algorithm was very

different from what we have discussed here. So, again the book goes in great detail, discussing what that algorithm was, along with the examples. So, get the summary of that, again I would not give the detail, because this is not really what is done in any (()). So, it distributes the scoreboard to respective function limits, these are known as reservation stations. So, it already does the distribution, as opposed to having a single issue. And these distributed sections are called reservation stations. Essentially an instruction can go and occupy a reservation station, waiting for the parent to complete. It resolves the name dependences by using the reservation station entries, just like our queue slot entries. After an instruction is registered in a reservation station, all dependencies generated by this instruction are mentioned in terms of the reservation station ids, just like our queue slot id. Write back to register file and cache must still be in order; that is like what we have discussed. Pending results can be held in reservation station entry or a future file, so this just like what we did on our issue queue. Bypass network takes the form of a common bus. So, this is just specific implementation detail of the bypass network, how we implement it actually. All launched results must compare all pending instruction sources. So, that is exactly what we have discussed also. The retirement register file of p 6 micro architecture is very similar. So, will discuss p 6 very soon. We are almost ready to discus that.

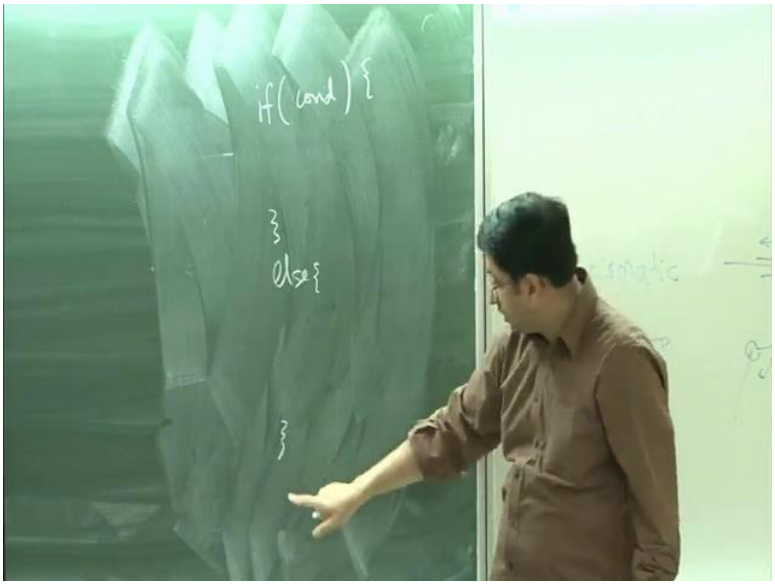
(Refer Slide Time: 48:23)



So, essentially what p 6 had was, they had a very similar thing like our single issue queue, but they did not issue from this queue actually, they had separate structure for

that. So, they had they used this queue for holding the results only, of computed instructions which have not it written to the register file. So, that that was called a retirement register file. I have two register files; one was a main file, other one was a rf, and rf values will be transferred to the main file when the instruction commits. So, will discuss p 6 in more detail very soon. We just wanted to mention it here, because it is very similar. So, one more thing about control prediction. So, if you look at the Tomasulo's algorithm carefully whatever we have done, what we have doing essentially is that. We are looking for instructions that are independent essentially. Independent of their predecessors and they themselves independent of each other, so that they can execute concurrently. So, that is essentially what we are trying to do.

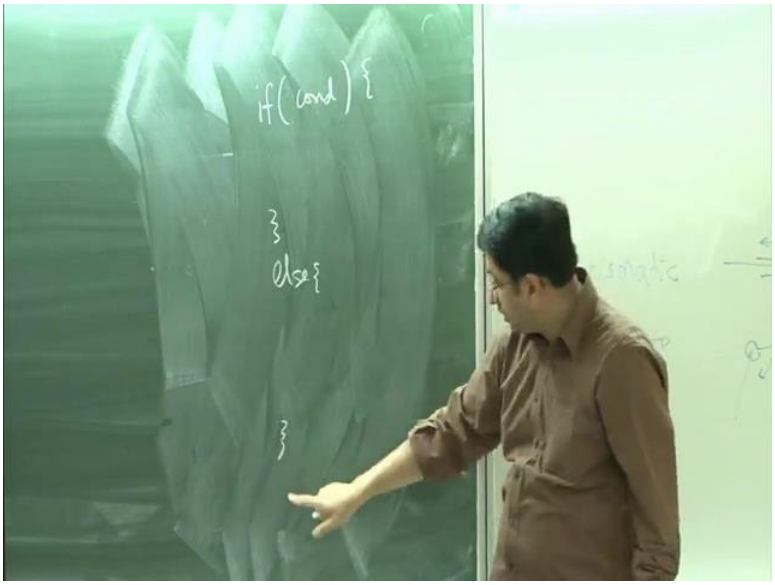
(Refer Slide Time: 49:19)



So, question is can we apply this same id of resolving branches. So, let me try to explain what I mean here. You essentially look for instructions that are controlled independent of the current branch. So, let us take an example. So, this one will translate to a branch instruction. Execution of this and this, are controlled dependent on this branch, but this is not. So, that is exactly what I trying to say. Is it possible to skip over these instructions, and start fetching from here. Later will fill up the gap, when the branch is closed, either from here or from here. So, that is essentially similar to Tomasulo's algorithm for control independence. So, this just for your, if you want to think you can think about them. Here are (( )) issues that will come up; first question is how do you figure out what instructions are controlling dependent on the branch. So, here your branch, this is control

independent, the question is it a static thing or it changes over time; that is a control independence set of instructions for a branch. Will that set change over time, or is it a fix set of instructions. If it is a fix set then that is that is good, because one side of detecting the set, then I am done. I can remember the set somewhere whenever I hit the branch I know where to fetch from again, what you think.

(Refer Slide Time: 49:19)



Student: what would be (( )) the change of the variable in (( )). No, we will come to that do not worry about the values yet. We just asking about the instructions, what are the instructions that are control independent, that is all I am asking. So, is it a fix set, or is the dynamic set. It should be static. So, for particular branch, the control independent set is actually a static set of instructions. So, which means I can remember it once I have discovered, and discovering it is also easy, depending on how to the compiler behaves. For example, in mips what will happen is that, this branch will be a taken branch if the else part is to be executed, and at the end of the if part there will be an unconditional jump, which will take you here actually. So, you can just look for this instructions, and you can easily find out what the control independent part is, that also answers the second question, how to find the re-convergence point. This is essentially this point, and you can apply this to other branches also for all values all this things.

The third question is do we fetch all instructions and then look for these; that is do we fetch all these, and then actually look for these instructions which are control

independent, or we do not even fetch control dependent instructions, until the branch outcome is known, what we do actually. So, there are two options, you can fetch everything sequentially, but do not do anything with this, start executing from here, and wait until the branch outcome is known. In which case I know which part to execute, and I can cancel the other part. The other option is, I skip over, I do not fetch anything, I fetch from here, and later I will fill in something, either this or this, when the branch outcome is known, which one is easier you think.

Student: we should follow the (( )).

Sorry follow, I should follow the right.

Student: You have the data dependencies also, (( )). So, there will be some data.

Well I am still not executing any of these remember that, because I do not know which one will execute ultimately. I am just talking about fetching part now, what do the fetcher do, which one is easier to do. First one is easier, first part is easier, why is the second part harder.

Student: There we need to start fetching.

Well that would be my branch target.

Student: You do not know the data convergence point (( )).

No I can find out, I look at the branch. Let suppose if it is an if else thing I look at the branch. So, I know the taken target that I can compute, because it is part of the instruction. I book one instruction up, if it is an unconditional jump, then the target of that jump is re-convergence point. I do not have to fetch anything actually. I just get inspect couple of instructions and I have done.

Student: Do we have to expect the all the instructions (( ))

No I do not have to. I compute the target from the instruction. I go to the target. I move one instruction up; that is an unconditional jump, if it is a unconditional jump. Then; that means, if is an if else construct, and the target of the jump is essentially my convergence point.

Student: but then we need to fill up the slots after (( )).

Exactly; that is a very difficult thing to do. you have to create space somewhere in the middle, because see this order is important, because a map table has to maintain these order, because allocation has to happen in this order, so that is very difficult. It is easier to remove instruction from the queue, but filling in something in the middle is not that is. So, the first one easier, but of course, it wastes space in your queue, that you may fill up the queue with many unnecessary instructions; that is a bad thing.

Student: Sir you are fetching the instructions after that, and you are not executed in (( ))

Of course, I cannot, because I do not know which one to execute actually.

Student: No, the instructions after the (( )).

Yes I am executing everything; yes I am executing all these.

Student: So, then you need to figure out the data dependencies.

Wait, that is the last question. So, these are for live in values.

So, values that are needed to execute the re-convergence point. So, what do I do with this; exactly that is what you are pointing out. There may be a value which is produced here and also produced here, which one should I take. First of all I do not even execute any of these, what happens to this instruction. So, the point is that, the easy way to figure this out is that, you do not execute the instructions which are data dependent all anything here. So, you hope you find something that is independent. So, essentially now we are talking about, not only control independence, we are also data independence together. So, looking for instructions here that are controlled independent and data independent, those only the ones we can execute. Anyway, so this one is just. So, this is actually you know a something that the people have looked at, as a research problem. Often it works pretty well actually.

Student: what should (( )) data dependence which we need to (()) all instructions.

You need to. So, yes exactly. So, if you if you have the second strategy that is you skip over your all these things, then figuring on data dependence going to be very difficult. In fact, it may not be possible.