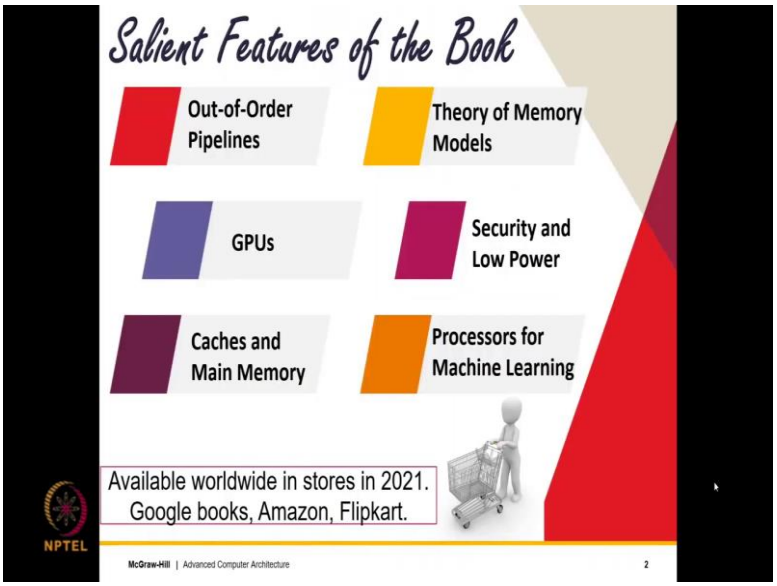
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

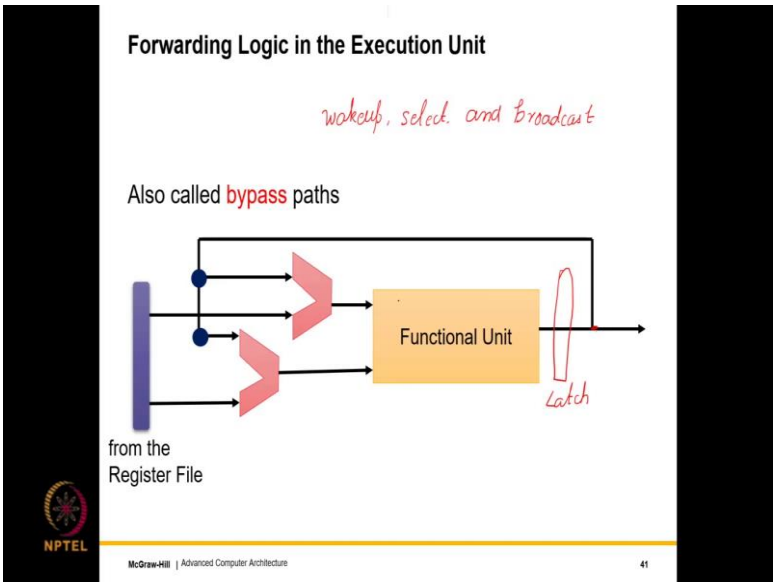
Lecture - 10
The Issue, Execute, and Commit Stages Part-III

(Refer Slide Time: 00:17)

(Refer Slide Time: 00:23)



(Refer Slide Time: 00:36)



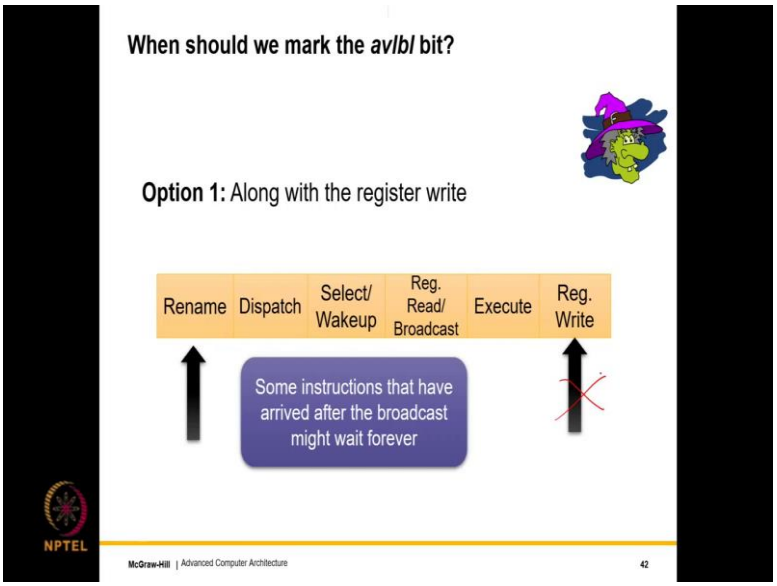
In the last lecture, we had talked all about the scheduling process. So, that included wake up, select and broadcast. So, in this lecture what we will do is, we will start from there. So, we will discuss the basics of the forwarding bypassing logic the basics of executing instructions the ex-stage in an out of order pipeline and then, we will move on to some subtle issues some corner cases in the wakeup, select, broadcast mechanism.

So, first we come to the bypass paths. Now, the bypass paths look almost identical to the forwarding logic and in order processors. So, the logic is very similar as I said, it is almost identical. So, what we have is; we have a forwarding multiplexers here, in this case bypassing multiplexers.

So, from this stage or from the output of this stage, which means after the next latch of course. So, this latch of course, I am not showing here for the sake of simplicity, but we do have a latch. So, from the next stage or from other stages, we can get data and they can be is inputs. In this case, they would be bypassed inputs and the bypass inputs can then be provided to the forwarding unit the execution unit.

So, there is always a choice between the default value that we read from the register file and also values that come via the bypass paths.

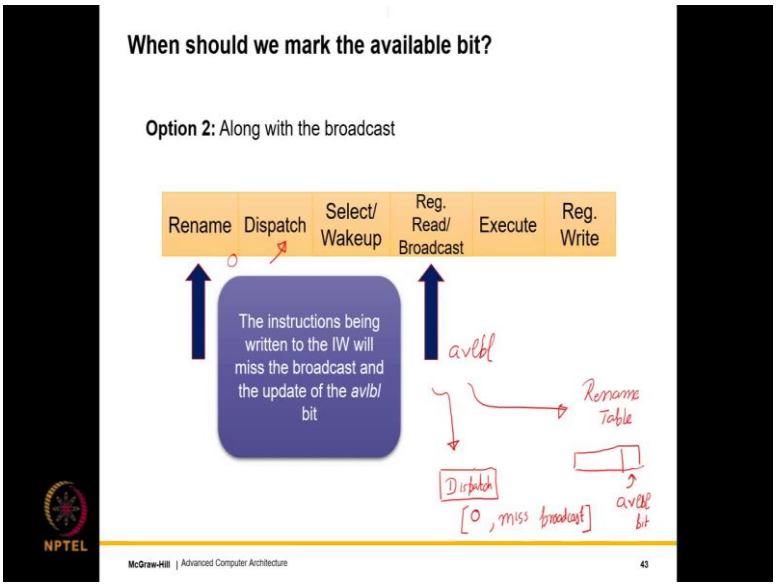
(Refer Slide Time: 02:44)



So, now a question is when should we mark the available bit. So, several options some instructions that have arrived after the broadcast might wait forever, if the available bit is marked in the register write stage. So, this means that after the broadcast stage, some instructions arrive at those instructions would have read the available bit to 0.

So, what will happen is that their available bit will be 0 and they will miss the broadcast. So, they will essentially wait forever; that is the reason writing the available bit with the register write is a bad idea. It will lead to a lead to an incorrect solution. It will lead to instructions indefinitely waiting which of course, is not desirable.

(Refer Slide Time: 03:39)



So, what we need to do is; along with the broadcast, we should set the available bit in the rename table. So, what we do is that we set the available bit along with the broadcast. So, whenever we broadcast, we also send a message to the rename table here, for the entry of the destination. What we do is that we set the value of its.

So, this also is not problem free in the sense that we can have some issues. One issue is of course, that instructions that are being written to the instruction window, while we are setting the available bit will miss the broadcast. And also, the update is available bit. So, let us see what is happening. So, let us assume that we are setting the available bit and the broadcast together at the same time.

So, at that point if an instruction is entering the instruction window. So, if it has been dispatched. So, what will happen if it is in the dispatch stage, this means it has already read the rename table at this stage. What is happening is that it has already read the available bit to be equal to 0. So, that process is already happened that the available bit has been read to be 0. Then, what we are doing now, is that we are also broadcasting.

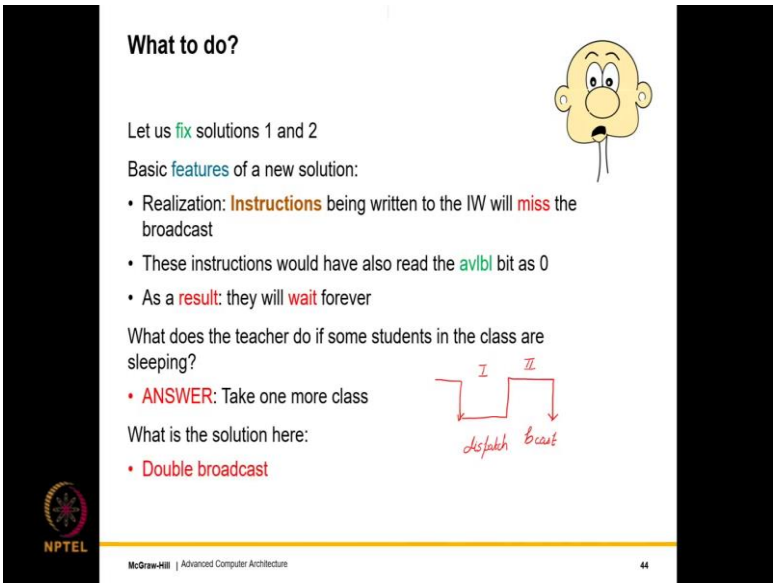
So, this instruction will miss the broadcast; the reason being that it is just being written into the instruction windows. So, it is not fully there yet the entry has not been finalized. So, it will also miss the broadcast. So, two things will happen; the first is that it will miss the broadcast and the second is that it would have already read the available bit to be 0.

So, when both of these things are kind of if you take both of these things together. What would mean that an instruction that is being written into the window the same time that we are broadcasting and updating the available bit; that will essentially end up waiting forever for the destination tag.

Why is this; well two reasons. One is that when it is entering this instruction window at this point, it means that in the previous cycle it was in its renamed stage. In this stage, it read the available bit to be 0. The second important reason is that when it is being written into the instruction window if there is a broadcast in that cycle unless, we architect the entire circuit well, it might miss the broadcast.

If it misses the broadcast, then there is an issue. So, then essentially it has miss the broadcast and it is not going to read from the register file where do we get the data from.

(Refer Slide Time: 07:25)



So, because of that there is a need to fix both of these solutions. Solutions 1 and 2. So, what will be the basic features of a new solution. Well, first we look at this important realization over here, that instructions being written to the instruction window when the concurrent broadcast is going on might miss the broadcast. These instructions as we have argued in the previous side would have read the available bit to be 0.

So, hence they will wait forever. Consequently, a solution is required. So, what does the teacher do? If let us say, some people some students in the class are sleeping. So, in that

case they will miss the instruction, well. So, all that the teacher does is, the teacher teaches one more class.

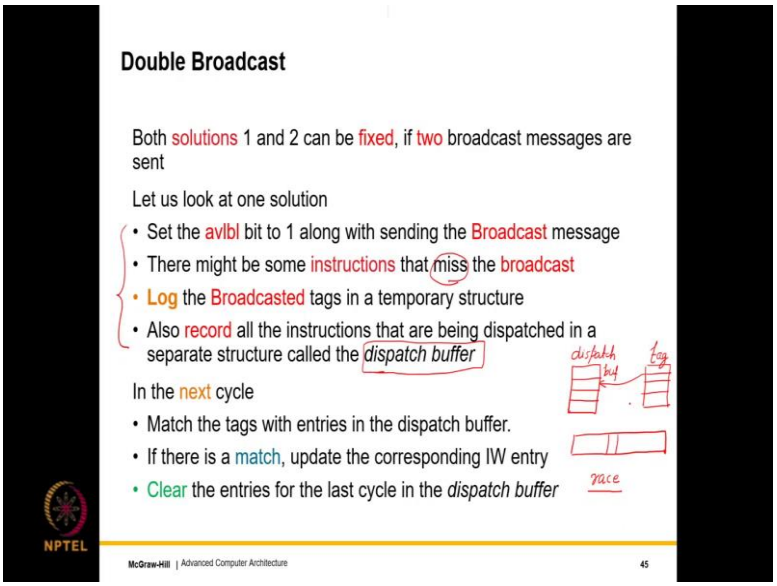
Something very similar can be done over here, for those instructions that are missing that might potentially miss the broadcast we need to do something. So, first let me talk of a trivial solution, which if you look at it is simple; subject to certain technological limitations. So, in this case, what we can do is we can divide a clock cycle into two halves.

So, we divide it into the first half and the second half. This is the first half of the clock cycle this is the second half of the clock cycle. So, what we can do is that we can ensure that we write instructions to the instruction window in the first half of the clock cycle. So, this is the dispatch phase.

So, this of course, requires us to have a mechanism to write instructions very quickly to the instruction window that too in the first half cycle. In the second half cycle, we broadcast. So, this ensures that instructions that are concurrently being written into the instruction window do not miss the broadcast.

So, well given that for back-to-back execution, we want broadcast and select, wakeup of the consumer instruction to happen in the same cycle. This is not very practical; it is not extremely practical to do it. Nevertheless, this is one solution, but this is you know given these issues with squeezing activities into half cycles is very hard to do in practice, this is typically not the method of choice. So, what we instead can do is something called double broadcast.

(Refer Slide Time: 10:14)



So, here we look at a simple solution. So, this is a text heavy slide, but let us go slow. So, let us focus on this part. So, we first follow the default which is that we set the available bit to 1 along with sending the broadcast message. So, along with the broadcast in a tag, we set the available bit to 1.

As we have discussed, instructions that are simultaneously being written to the instruction window might miss the broadcast. So, we log the broadcasted tags in a temporary structure. We also record all the instructions that are being dispatched in another temporary structure called the dispatch buffer. So, dispatch buffer is a very small buffer that might contain two to four instructions that are being written to the instruction window in one cycle.

So, this will be equal to the rename width basically. And also, we keep a small list of tags. Let me write it down this is the dispatch. We also have a small list of tags that would broadcast in a given cycle. So, the next cycle what we do is, that we match the tags with entries in the dispatch buffer. So, we see that if any entry in the dispatch buffer if any operand is getting ready, because of a broadcasted tag. So, essentially, we try to match them.

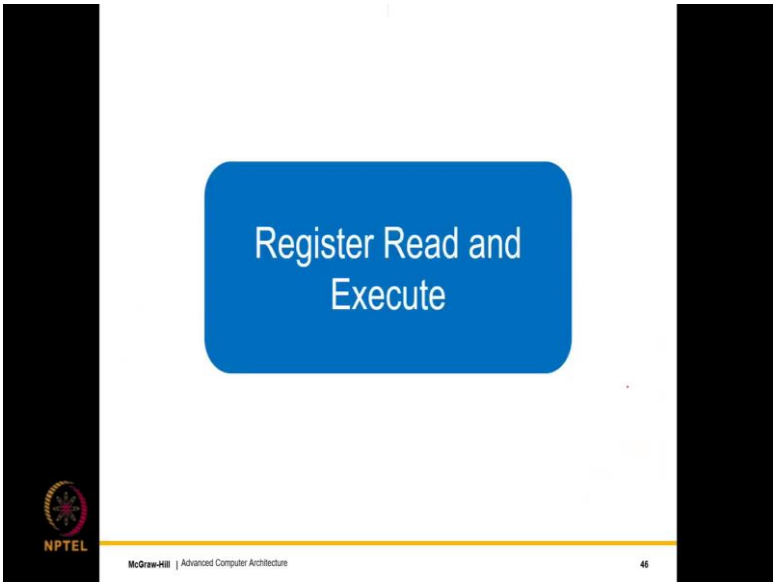
If there is a match, then what we do is; we update the corresponding instruction window entry. So, we figured out all the matches, then we go to the instruction window and of course, this makes it more complex, but something of this nature has to be done. So, we go to that entry and we enable its corresponding source operands. So, this is like a next

cycle fix for instructions that are just being dispatched, but some such small fixing action is required and this is a rather tricky corner case.

So, we do have a course here in IIT Delhi, where we design full out of order processors and while designing such processors, such problems do come up very often, where something is being read something is being written; there is a dependency and they are happening in the same cycle.

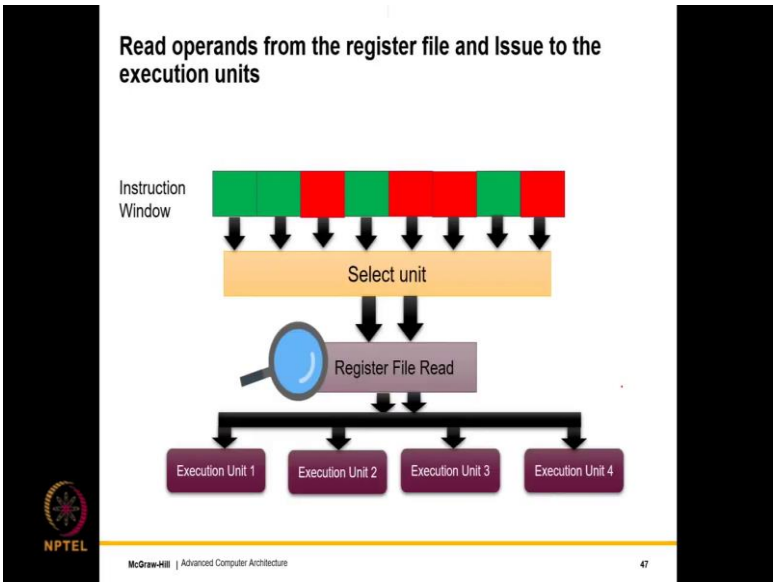
So, there is a possibility of missing the value. So, such conditions are also called race conditions. To avoid that, in this case we can apply a small fix in the next cycle.

(Refer Slide Time: 12:55)



Now, let us look at the simpler parts which are register read and execute.

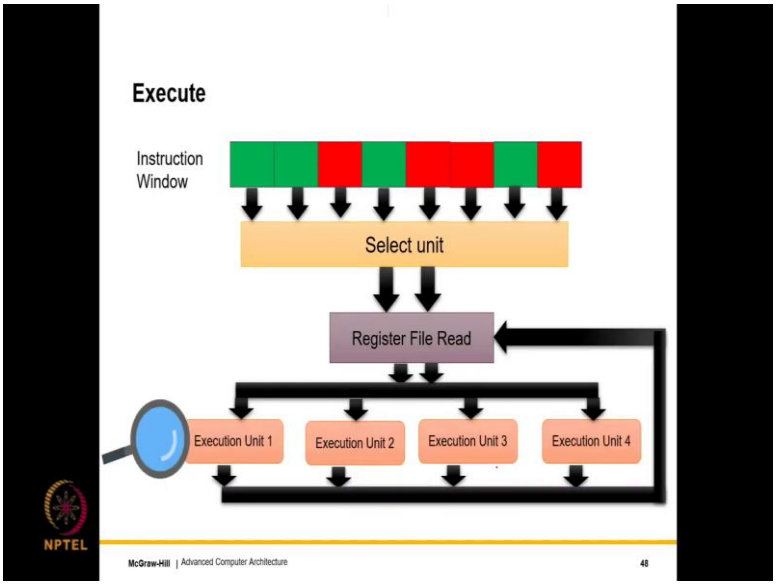
(Refer Slide Time: 13:02)



Well, so we have already discussed the instruction window with green and red entries. We have already discussed the select unit, which uses a variety of heuristics to select the most appropriate instructions that need to be executed. Then, we have the register read stage. So, here we just access the physical register file read the operands.

But mind you; similar to forwarding an in-order processor, some of these operands might not be useful, because we have a bypass paths in our system and then we have a plurality a multitude of execution units.

(Refer Slide Time: 13:45)



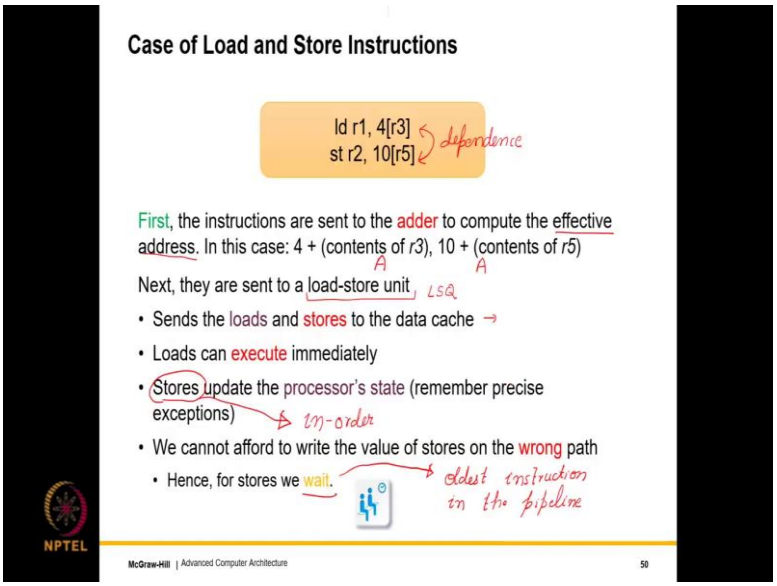
These execution units execute, they produce the final value and the final value is written back to the register file, as we can see over here.

(Refer Slide Time: 14:00)

So, what we have done is, we have reached a very important juncture in our study of out of order pipelines. So, we have completed instruction renaming, instruction dispatch, wake up and select. So, those are critical mechanisms in the out of order pipeline for primarily register based instructions. So, instructions that have register based operands.

So, they help track their dependencies. They help enforce the dependencies and they also help transfer the information from producers to consumers; that a source operand is ready. Now, let us look take a look at memory dependency is a totally different ballgame altogether. And let us propose the load store queue, which whose job main job is to track and enforce memory dependencies.

(Refer Slide Time: 15:06)



So, consider these two instructions load r1 4 (r3) and store r2 10 (r5). So, as far as we are concerned, they appear to be independent in the sense that they access different registers, but well the instructions mind you are sent to the adder to compute the memory address. You also refer to the memory address of the effective address same thing.

So, in this case we are computing 4 + (contents of r3) and 10 + (contents of r5). So, it is possible there is the possibility that they are actually the same address. Is the different addresses no problem, but they are the same? If they are the same addresses, there is a dependency between them. So, there is a dependence between them, if they are the same address.

So, next these instructions are sent to a load store unit. So, the load store unit we will see does something. So, it is also called the load store queue, but since I have not talked about the load store queue yet, I would prefer the term load store unit. So, the load store queue over here, over it does several things; what does it do? It sends the loads and stores to the data cache that is one then the few more things as well.

So, what we need to understand is that loads can execute immediately, they do not have to wait. So, in out of order processing, loads do not have to wait for anybody. They do not have to wait for other loads they do not have to wait for other stores unless, they are to the same address stores update the processor state. So, if you remember precise exceptions;

this means that unless a store is deemed to be on the correct path we cannot send it to the memory system furthermore, store should be sent in program order.

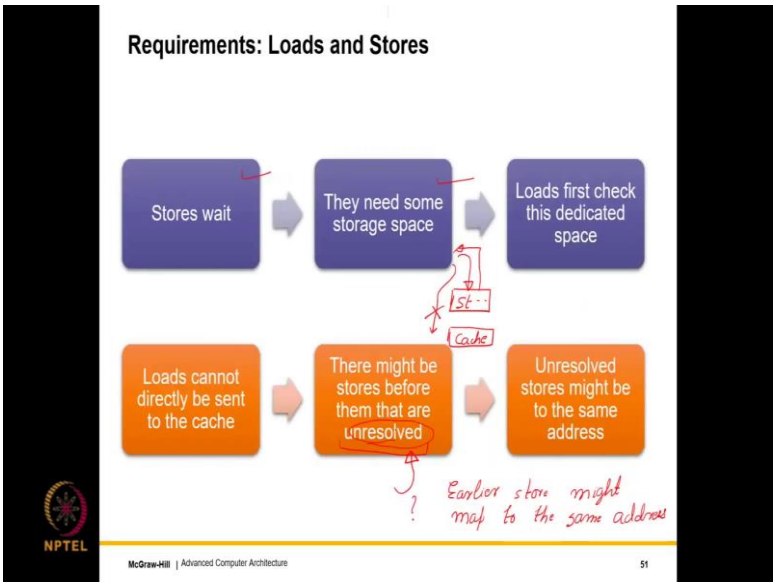
So, handling stores is tricky right. So, the because of precise exceptions, because of the fact that stores actually change the memory state, we have to ensure that they are issued to the memory system the data cache in this case in program order. We cannot afford to write values of stores on the wrong path. So, you have to wait till they become absolutely safe, which means that stores have to wait; for how long?

Well, till they are the oldest instruction in the pipeline, because it is possible that if a store is sent early and an instruction that is older than it has a fault well then we are a trouble, we cannot ensure precise exceptions. So, until stores are the oldest instructions in the pipeline, they have to wait, right. So, until it becomes the oldest; which means that there is no oldest earliest whatever you may call, but until we do not have an instruction earlier than the store in the pipeline, the store has to wait.

This basically means that. Well, number one, we need to have a mechanism of ensuring this and number two, we cannot send a store to the memory system early, because this would break our notion of out of order inside in order outside. So, we will take a look at this in great detail, but the important take home point is loads can go at any point of time immediately. Stores have to wait, because they are changing the external state of the program. So, they have to wait they have to be sent in program order and they also have to be sent when they are the earliest instruction. So, the pipeline in the pipeline.

The reason being that if we send the store and then, we find that there is an instruction earlier than it that has a fault some sort of an exception. Well, then once we do a context switch, we will see that precise exceptions do not hold.

(Refer Slide Time: 19:44)



So, again a quick summary of what I said. I said look stores need to wait; this means that we need some storage space to have the store value with us. And since stores need to wait, we cannot really use bypass this. So, let us say the stores are waiting over here, then a later load comes, load cannot directly go to the cache that is not acceptable to us.

Because, it is possible that there is an earlier store to the same address. So, this does not happen. What happens is that loads check the first check the dedicated space, where stores are waiting. If there is a store to the same address, then the value is forwarded back to the processor. So, this is required.

Loads cannot directly be sent to the cache. Yes, they can execute any time, but they cannot directly be sent to the caches, because there might be stores before them that are either unresolved in the sense that address is not computed or their address has been computed.

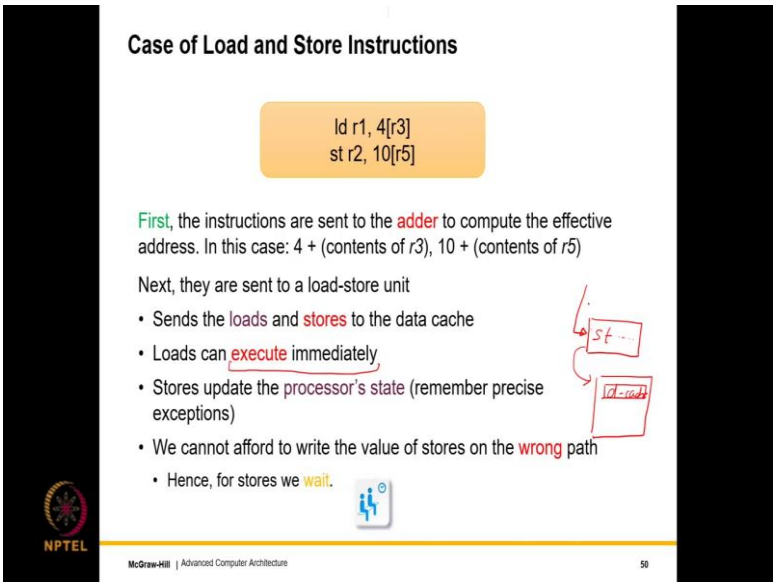
In this case, we need to forward the value from this temporary structure that has store values, but if we consider the more trickier case where we have stores whose address is not resolved called unresolved stores. well then, in this case we do not know, because an earlier store might map to the same address.

If it maps to the same address, then if we bypass it and then read the value of some store that was earlier than it, we will end up with the wrong value. So, we will keep all of these

things in mind. So, I am still presenting a rather vague picture of how to deal with loads and stores.

That is why I just want to go back to the previous slide. On this slide, what we saw is that loads of course, can execute immediately, but stores need to wait for precise exceptions.

(Refer Slide Time: 22:13)



If stores need to wait, we need some storage structure that has a list of stores and their values. So, let us whatever this storage structure is, it just has a list of stores. So, any subsequent load has to check this first. This is like a cache for it, before going to the memory system.

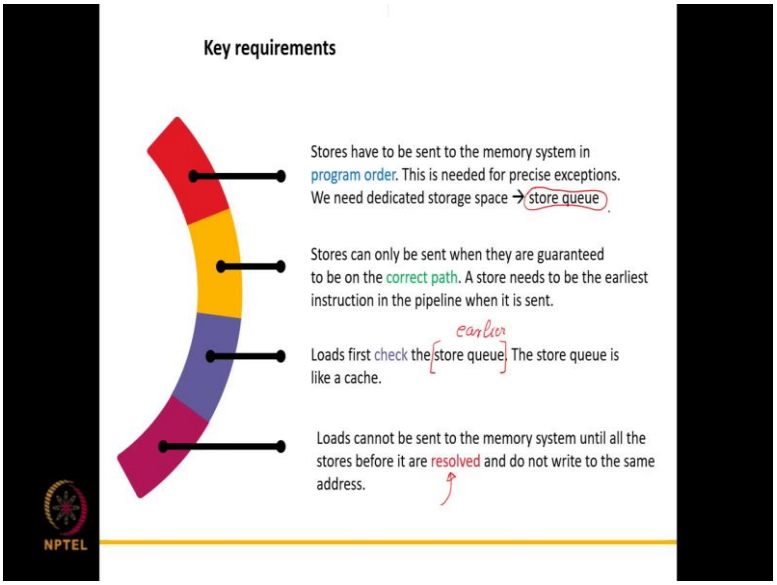
Then the memory system of course, starts with the data cache. So, before going to the data cache, we need to check the temporary stores. When we are checking the temporary stores, it is possible that we might have a store to the same address. Well, then we forward the value, but the trickier case would be when these stores are unresolved.

See the stores are unresolved in the sense the address is not computed, then we in principle do not know if the address will be same as a load or not. So, we need to wait.

(Refer Slide Time: 23:08)

So, such issues have guided us to the design of a structure called the load store queue, as we shall see in the subsequent slides.

(Refer Slide Time: 23:15)



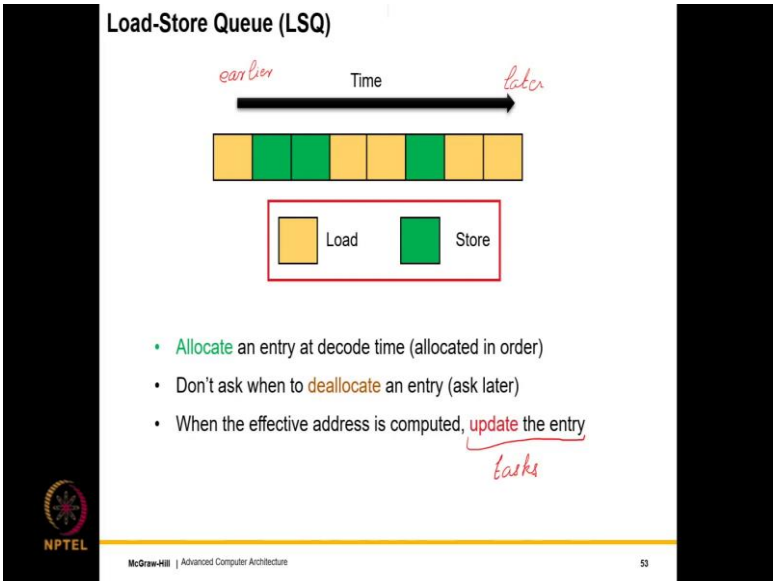
So, key requirements once again. So, I am just going over the requirements several times; such that, it is kind of rather clear in your head. So, stores have to be sent to the memory system and program order, because of precise exceptions. We need some dedicated storage space for keeping the stores that are not being sent. Let us call this the store queue. So, we

did not give it a name, but let us give it a name right here right now. Let us call it the store queue.

So, at the cost of repetition let me say it again, for the last time that stores have to be on the correct path, which means that they need to be the earliest instructions in the pipeline. So, because we need to store the stores in the store queue well load sort of check the store queue first; that is the highest priority that any load has to check all the earlier stores in the store queue. If there is an earlier store to the same address, well then, it needs to use the value that is written by that store instead of going to the cache.

Loads cannot be sent to the memory system until all the stores before the load are resolved in the sense that address is computed and they are found to not write to the same address. So, what we will see in the subsequent slides that this resolving the loads that is an important concept and that does play a very important and rather critical role in the development of the load store queue.

(Refer Slide Time: 24:52)

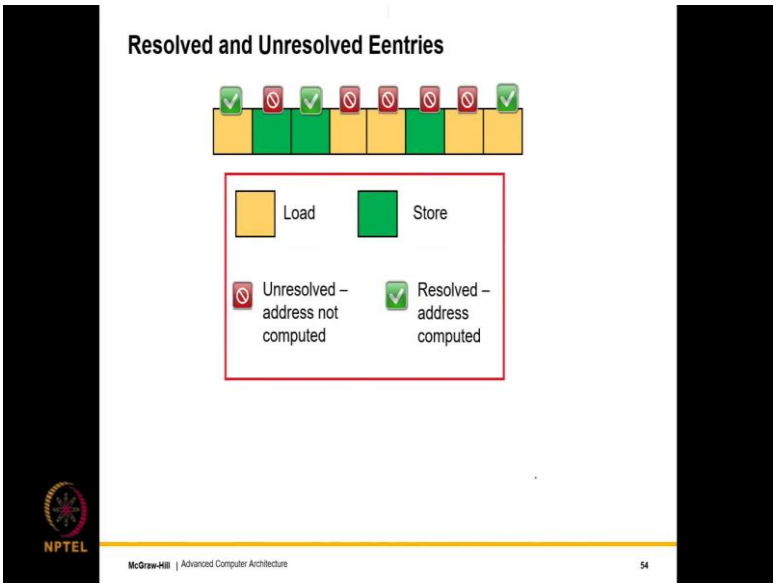


So, let us look at the load-store queue. So, let us assume that time proceeds from left to right. So, these are the earlier entries and these are the later entries. So, that is our time proceeds. Let the yellowish boxes be loads and the greenish boxes be stores. So, we allocate an entry at decode time. So, the allocation of course, is done in program order in the load store queue.

So, the moment that we find that an entry is a load or a store we. So, this is. So, when is this found out; well, this is found out in the decode stage. So, in this stage we allocate the entry to the load-store queue. Do not ask me when to deallocate right now this will come later right at least 10, 15 slides later. So, when the effective address or the memory address is computed, we update the entry with the memory address and this is where something important extremely important needs to be done.

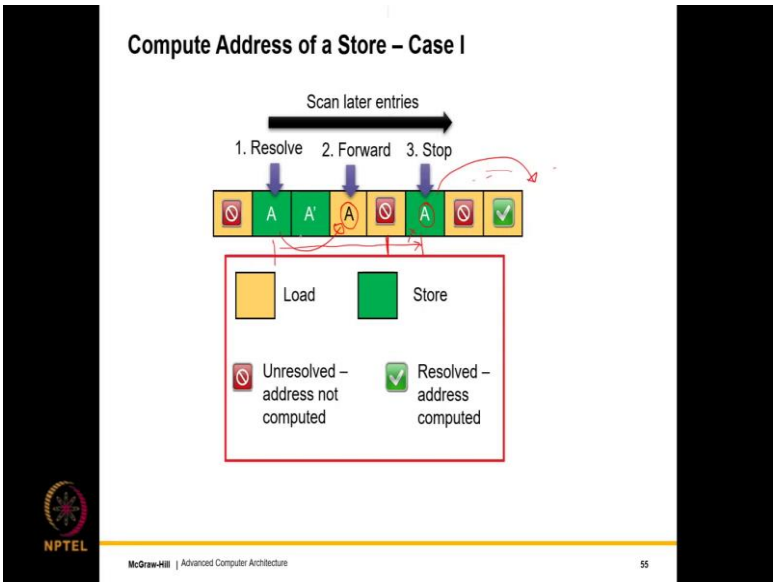
So, this is where most of the tasks are. So, this is where most of the important stuff with regards to a load store queue, needs to be done.

(Refer Slide Time: 26:15)



So, our terminology would look like this that the yellow entries will be a load and the green entries will be a store. So, the no entry sign over here means that it is unresolved the address is not computed and the tick means that the address has been computed. So, we will use this terminology for the subsequent slides.

(Refer Slide Time: 26:46)



So, the other important thing that we see over here in the terminology is that for the loads for which the address has been computed, I am simply writing the address for the loads and stores I am just writing the address in the entry and if I just want to say that the entry has been resolved, but the address does not matter I am putting a tick.

So, the tick indicates that it has been resolved the address, whatever it be it does not matter. And the other is that if an address is written, needless to say the load or store is resolved and this is the address. For example, here the address is A. So, let us look at the first case, where we compute the address of a store this is case 1. So, in this case, let us assume that we resolve the address of this store.

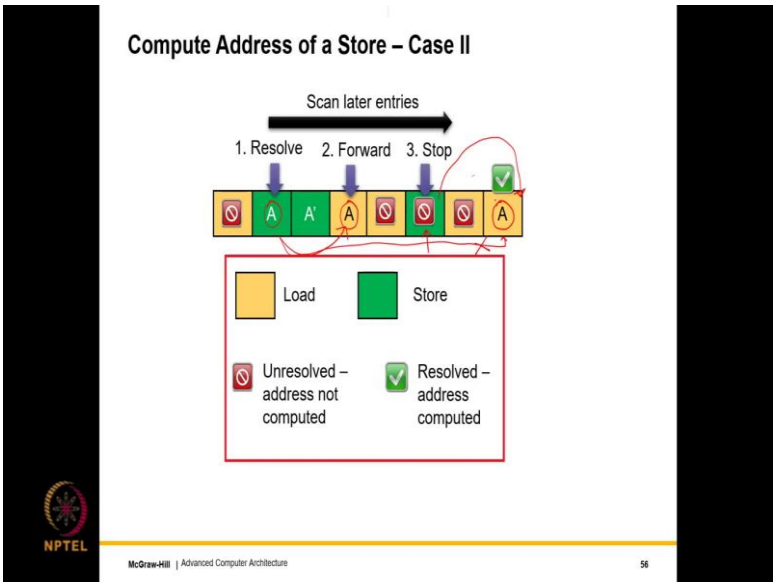
So, this is step 1 that we resolve the address in the sense we find that the address of this store is A, then what we do is that we scan all the entries that have come after the store not before, but after. So, we scan all the later entries. So, we scan all the later entries. So, we just sequentially look at them. We find that we have a load instruction that is reading from the same address A.

So, what we can do is that we can forward the value to the load. So, the value can be forwarded to the load and the load can be said that look your the data is here, it is the same data that the store is writing to memory. So, you can take the data and then, write it to the register file broadcast the tag and do whatever is necessary and essentially, proceed with your execution.

Subsequently, we find another store to the same address. So, we do not scan later entries, because any later loads will actually get their value from this store, if they are writing to the same address which is address A. So, it is important that we understand that the span of this store is essentially from this point until a subsequent store to the same address. So, of course, we do not consider this, but let us say we consider till this point.

And if we find any load is within this window that reads from the same address which in this case is A, then we forward the value. So, that is important we forward the value. So, we do not go to the memory system, because this is the most recent right. So, we take the value and we forward it.

(Refer Slide Time: 29:36)



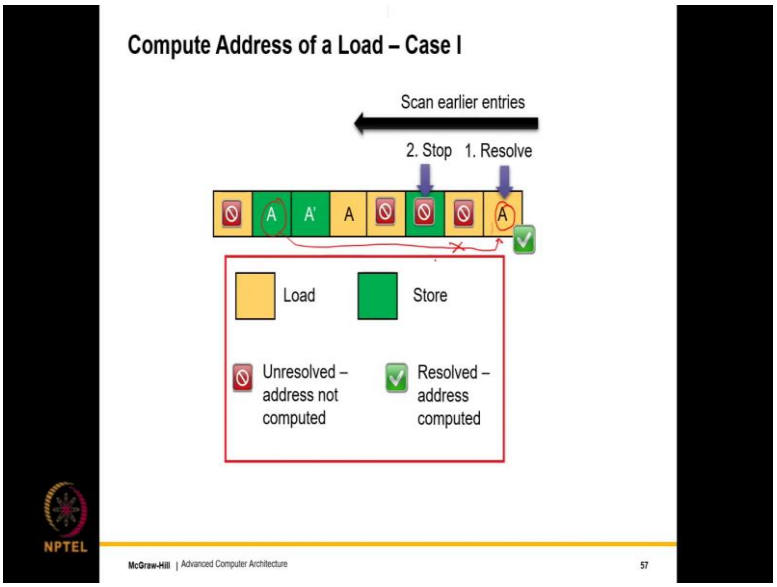
The second case, the we compute the address of a store it is the second case. In this case what do we do? Well, so the first few steps are the same. Let us assume that in step 1, we resolve the address of a store and it is found to be we do the same thing we scan the later entries. As we scan the later entries, we keep going going going going. We find a load with the same address, so we forward the value.

We did it in the last slide as well, but there is a twist in this slide. We come to this entry over here, where we find that the store has been unresolved. Given that the store has been unresolved, we actually do not do anything, because as far as we are concerned, this might be a store to any address. So, we do not know the address.

So, this address can be A or it may not be A it can be any address, but since we do not know we do not proceed henceforth. So, if you proceed, you will see that we have one load again to address A, but we are not doing this forwarding. So, we are not doing this forwarding.

The reason we are not doing it, because we do not know the address of this store. This store might very well be a store to address A. In that case, this forwarding would be wrong and we would instead need to forward the data like this. Hence, any time we see an unresolved store, we basically wait and we do not scan entries after it.

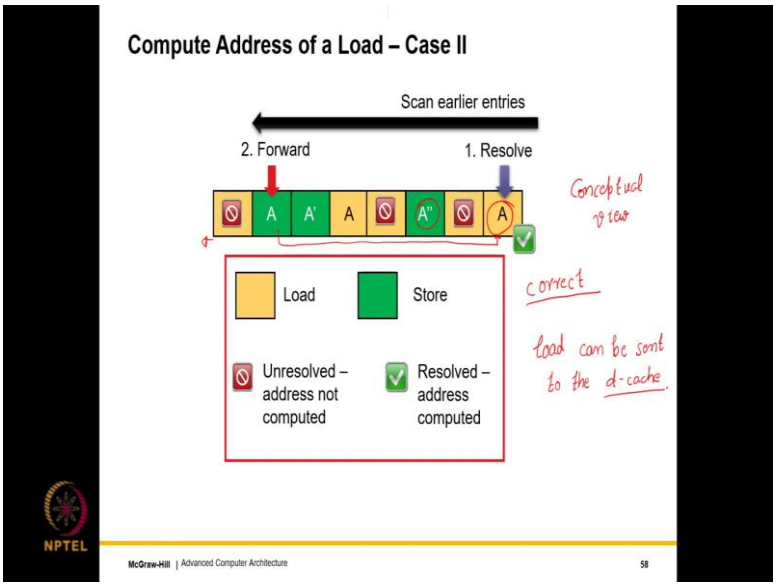
(Refer Slide Time: 31:17)



Now, what do we do when we compute address of a load. Again, we will divide it into two cases case 1 and case 2. So, assume that we compute the address of this load and find it to be A, then we scan earlier entries. In this case, the direction of the scan is reverse. So, we simply keep on scanning the earlier entries, we just go, go, go, go. What we find is, we find a store to an unresolved address. So, this address has not been resolved.

So, even though there are stores before this that right to the same address, we do not do this forwarding. So, we definitely do not do this forwarding. The reason is the same as this reason in the previous slide. We do not know about the address in this entry and this could be writing to A, well, we do not know. Since, we do not know we do not do anything we just wait alright. So, we do not we just stop over here, we do not do anything else.

(Refer Slide Time: 32:25)



Let us now look at case 2. Again, we compute the address of a load. The step 1, we resolve the address of this entry to be A. We scan earlier entries. In this case, this particular store is resolved, it addresses A prime prime. So, we are not bothered with it. So, we find one store that writes to the same address.

Well, since there are no intervening stores that either write to the same address or are unresolved, we can confidently and correctly forward the data from this store to this load and this forwarding will be correct and so, then this will release the load and the load can go ahead and execute.

Fine. So, I hope that this point is well understood that in the case of stores, we scan later entries. In the case of loads when you compute the address, we scan earlier entries. And as we scan earlier entries, we just keep going back, back, back, back, till we either hit a store to the same address or an unresolved store or we reach the beginning of the queue.

If we reach the beginning of the queue and we do not find anything, then the load can be sent to the data cache, because there is no store that can potentially forward its value to it. So, load can be sent, the load can be sent to the data cache, but otherwise, we always give priority to entries in this queue.

In this queue of loads and stores called the load-store queue ok. So, recall that I had mentioned one point and this point I will clarify slightly later. So, what I had mentioned

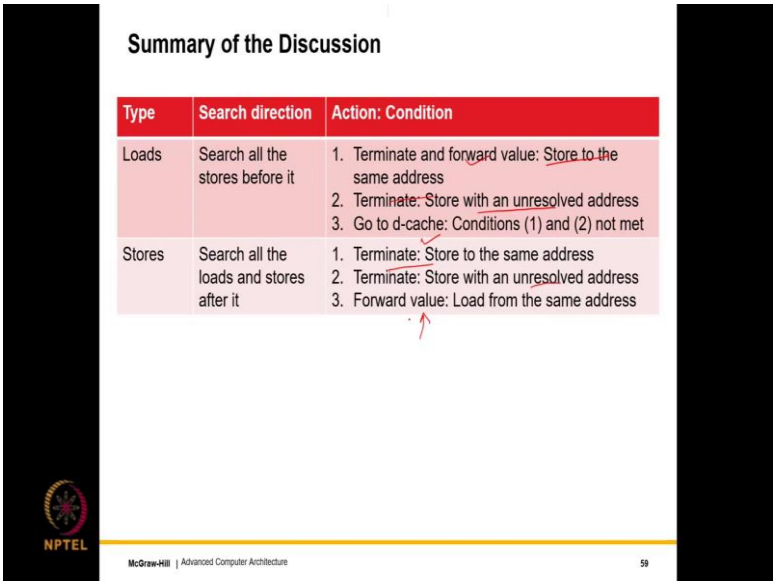
over here on this slide. Let me just go back, because I just remembered it and it has a potential, some potentials of confusion.

So, I said that we will create some space called a dedicated store queue that will store the values of store instructions that have not been sent. Well, yes we are using it and we will see how to use it, but let us assume that this is some structure that just stores the value we have not discussed LSQ yet. We are only looking at the conceptual design. When we will look at the actual design this will become clearer.

So, coming back to the conceptual design, we were over here in case 2. And in case 2 what we had seen is that we simply keep scanning earlier entries and whenever we find either as I said a matching store or an unresolved store, we wait there or if we reach the end of the array, we send the load to the data cache.

So, this is the last four slides, which have the same diagram with green and yellow entries essentially represent a conceptual view of the load-store queue. It is kind of a conceptual view and the actual implementation well we will see ok.

(Refer Slide Time: 35:52)



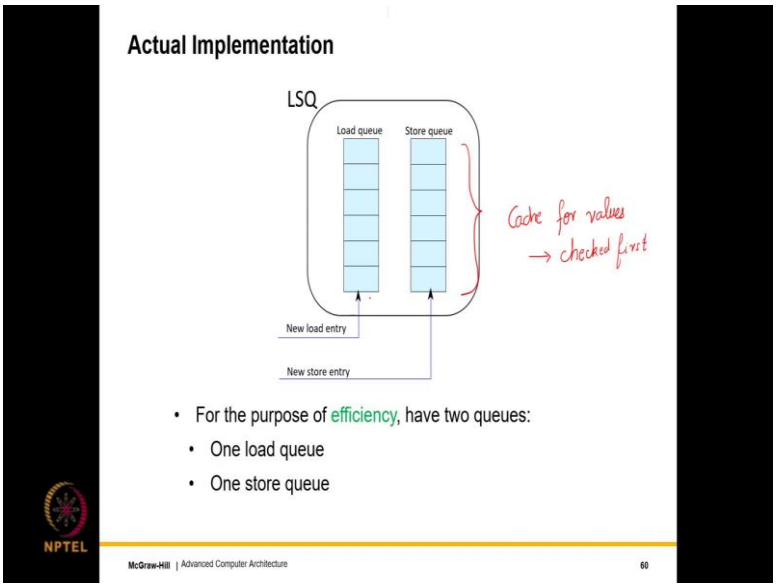
So, let me now kind of summarize whatever we discussed in the conceptual implementation of the LSQ and what are the important concepts that we are taking along with us to an actual implementation. Well, loads well we search all the stores before it. Since, the action condition pairs, if you find a store to the same address, we terminate and

forward the value. If it is a store with an unresolved address well, we just terminate the search process we do not do anything. Else, if both of these do not hold in a sense. We reach the end of the array, then we go to the data cache.

Similarly, for stores when we search all the loads and stores after it writes in loads is just stores, but in stores has loads and stores after it. So, if you find a store to the same address well, we terminate the process. If you find the store with an unresolved address then also, we terminate the address, because we do not know the address that the store is going to resolve to.

Else, before termination if we find a load that also loads from the same address, we just forward the value as we have seen. So, this table is rather important, because it tells us what all we need to implement to actually realize the physical load-store queue.

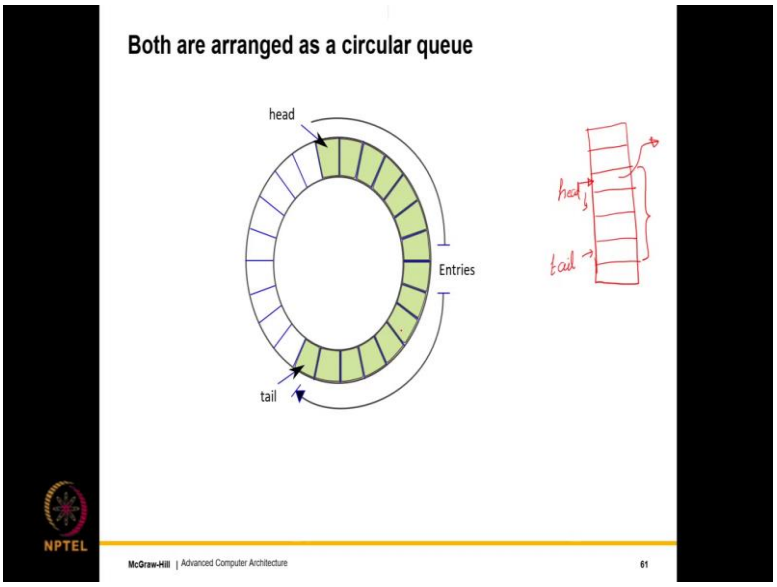
(Refer Slide Time: 37:18)



So, this is where the notion of that store queue comes up. This is something that we had alluded to very briefly, but because we are not discussed the design of the load-store queue, this part I would presume would not have been that clear. So, the load-store queue for efficiency reasons actually has two queues called a load-queue and a store queue. So, whenever we decode when we find a new load entry, we will be enter it into the load-queue and whenever we decode and we find a new store entry, we enter it into the store queue.

So, store queue additionally also acts as a cache for values as we have seen, for store values basically, as we have seen and essentially this is checked first. Before going to the d cache. Before accessing the caches, we access the store queue first. To see if we can forward data from a matching store. If we are able to forward data well and good the load can execute quickly else, we access the data cache which of course, is slower and which is also outside the pipeline.

(Refer Slide Time: 38:39)



So, before we proceed it is important. Before we proceed, we need to understand how many of these structures bit an instruction window free list and load-store queue are implemented. So, we have already discussed this earlier in one of the previous lectures; all of them use a circular queue.

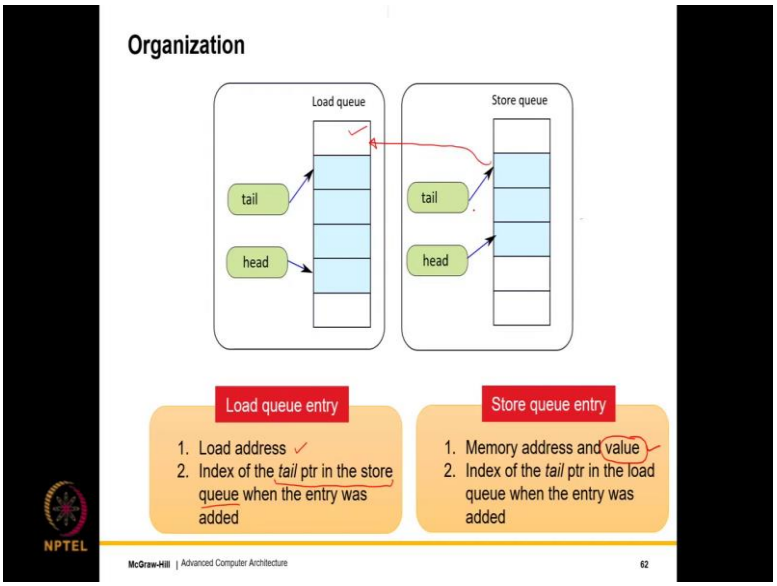
So, circular queue is a conceptual structure, which was of course, you cannot have such a circle in hardware, but what you can have in hardware is that you can have a regular array. And the regular array converting that to a circular queue would implement a degree of wraparound, but you will essentially have two pointers. You will have a head pointer and you will have a tail pointer.

So, whenever we want to n queue well, all that we need to do is we need to increment the tail pointer and if we reach the end of the array, we will wrap around. Whenever we want to d queue well, again we increment the head pointer and if let us say I were to d queue I will return the value over here. So, this is again a conceptual view of a circular queue. So,

both the load queue as well as the store queue are internally implemented as circular queues.

So, this is kind of a universal statement across all of architecture that whenever we need to implement a queue like structure, it is almost always invariably a circular queue.

(Refer Slide Time: 40:12)



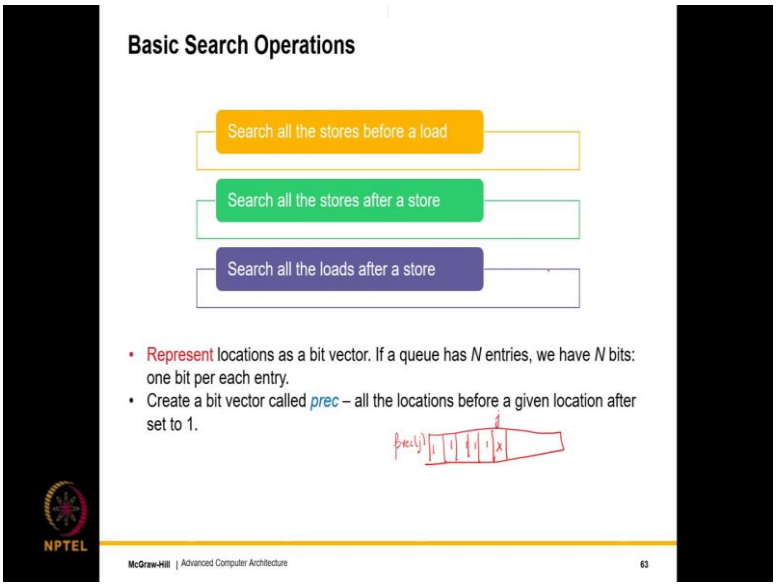
So, the organization well, we will have a load queue and a store queue. Each of them are implemented as circular queues. They will have a head pointer and a tail pointer as was discussed. So, each entry of the load queue will have the load address and similarly, each entry of the store queue will also have the memory address and the value.

So, the important point is that the store queue is also storing the store value; such that it can be forwarded. We will also need some more information for doing the kind of checks and searches that we have been proposing. So, these new pieces of information that we need to add for the purposes of searching the way that we have described like this that in every load queue entry we also add an index of the tail pointer in the store queue; whatever was the value of the tail pointer in the store queue. When the entry was added? That is important.

That when I am adding a new entry in the load queue. So, I am adding a new entry, I will increment tail. So, I will add an entry over here. So, when this entry is being added, I just also add the tail pointer of the store queue. So, that is important we will see why.

So, this essentially synchronizes the load queue entry and the store queue entries points; such that if let us see for a given load, I want to find all the earlier stores, I can use this information at this point. That is kind of important. The other is that for a store queue entry what I do is, I also add the index of the tail pointer in the load queue when the entry was added. So, this is also giving it a point that you can find earlier loads and later loads based on this information.

(Refer Slide Time: 42:24)



So, what are the basic search operations. Well, the basic search operations if you take a look at the table that we had shown a couple of slides earlier. The basic search operations are we search all the stores before a load. For a store, we search all the stores after it and we also search all the loads after it.

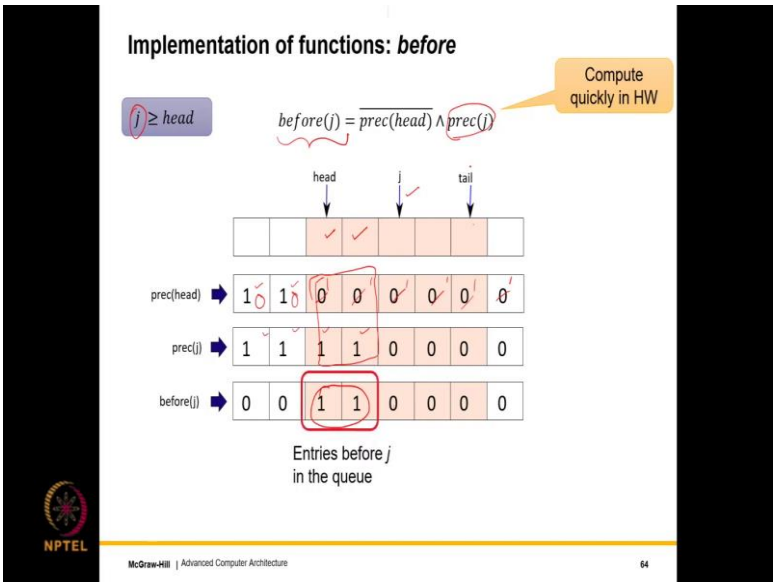
Such that. So, we search and then, we look for the conditions that we have mentioned in the table like the earliest address of the mat store or an unresolved store. So, pretty much we need to find the indexes of all entries that are after a given load of store and that meet a certain criteria. After that we need to find the index of the earliest search entry or latest search entry depending upon whether it is a load or a store.

So, what we can do is we can represent store queue location is a bit vector. See, if a queue has N entries, we can have N bit vector with one bit for each entry; that is also create a bit vector called prec, where all the locations before a given location are set to 1. So, in the sense if this is the array and we have one location over here, then essentially let this index

be j then, prec j would basically contain 1 in all the locations other than this the 1's before it.

So, it has nothing to do with the fact that whether it is a circular queue or not it just adds 1's in all locations in the array that are before that given location.

(Refer Slide Time: 44:20)



So, the implementations of the function before. So, we will implement two functions before and after. So, I will just show you that how they are done in hardware. So, these are representative designs. So, actual LSQ implementations might be doing something else, but not very different.

So, let us see I am given a certain index j and this there is no wrap around. So, j ≥ head. So, I want to find all the locations within the circular queue that are before j. So, let us see what we have is we have an array without wrap around. So, we have the head pointer the tail pointer and j.

I basically want to find the indexes or indices of these two locations that are before j and that are essentially there in the queue. So, those are our two requirements. So, the computed what I can do is, I can use some Boolean expressions. Hardware is in general very fast with Boolean expressions.

Secondly, what happens is that hardware is nice and parallel. So, given the inherent parallel processing that hardware enjoys, if we sort of create a lot of Boolean expressions, hardware

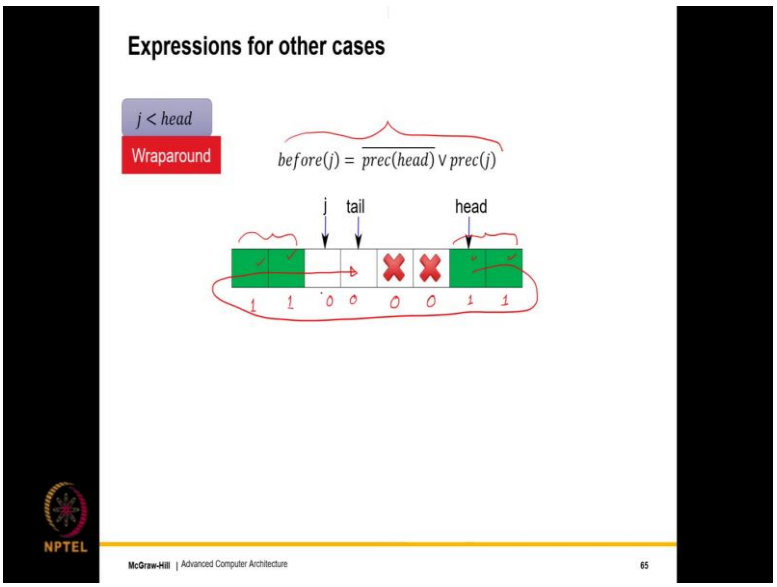
can process them very easily and very quickly. So, let us look at prec head. Well, prec head will have a 1's will have 1's in these two positions prec j will have 1's in these four positions. And before j will essentially be well, we need to see.

So, we need to see how these two are related. So, we can compute NOT of prec head. So, it will become 0 0 and all of these will become 1. And prec j. So, a prec j will again have 1's in these positions. So, if I just compute a logical AND of them, what I am left with is this block. So, this is exactly what I am left with and I compute a logical AND then I arrive at the two positions that I am interested in. These are the positions of the two entries that are before the jth entry.

So, how will it actually translate to a real LSQ. Well, typically given a load we want to find all the stores before it. So, we can go to that tail pointer, which that load entry stored and just find the addresses of all the stores the indices in this case in the queue of all the stores that are before it and let us still a part of the queue.

So, that will be given by this before function, which as you can see is a simple Boolean combination of NOT of prec head and prec j. And this can be computed; such functions that deal with Boolean vectors can be computed very very easily and quickly in hardware.

(Refer Slide Time: 47:28)



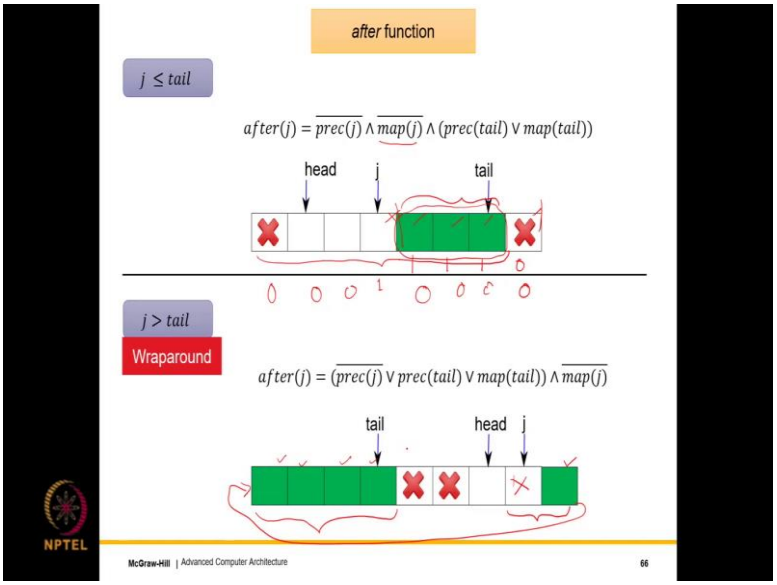
So, let us now consider the other case, where we have a wrap around. So, where we have a wrap around what will happen is that the head pointers will be over here, towards the

end of the queue and then, the queue the logical conceptually logically and conceptually the queue will wrap around like this. So, in this case, all the entries that qualify to be before j will essentially be this entry, this entry, this one and this one.

So, What we can do is, we can create a Boolean expression that will have 1's for these entries in the bit vector and 0 for the rest of the entries. So, mind you here across is basically those entries that are not a part of the queue. So, in this entry what we will have is that if we see before j will basically be all of these entries that are between head and the end. So, all the entries between the head and the end of the queue will be NOT of prec head. So, NOT of prec head will be these two entries.

And then, we will have a logical or with prec j, which are all the entries between j and the start which are these two entries. So, this Boolean expression provides the expression for before j, when we have a wraparound. And wraparound is when j < head and that is when we will see that we will have a wraparound in the sense that the queue the part of the queue we are interested in, the circular queue is wrapping around the end of the array.

(Refer Slide Time: 49:42)



We can create similar looking expressions for the after function. So, the after function is shown over here. So, the first case is the case, where we do not have a wrap around. So, j ≤ tail. So, in this case after j would be these three entries, the cross mark shows those entries of the array that are not a part of the queue.

So, of course, we will not consider these two. So, we are only interested in the three green cells. So, what we do is for that, this Boolean expression the claim is that this Boolean expression represents these three cells where these three cells are 1 and the rest all are 0.

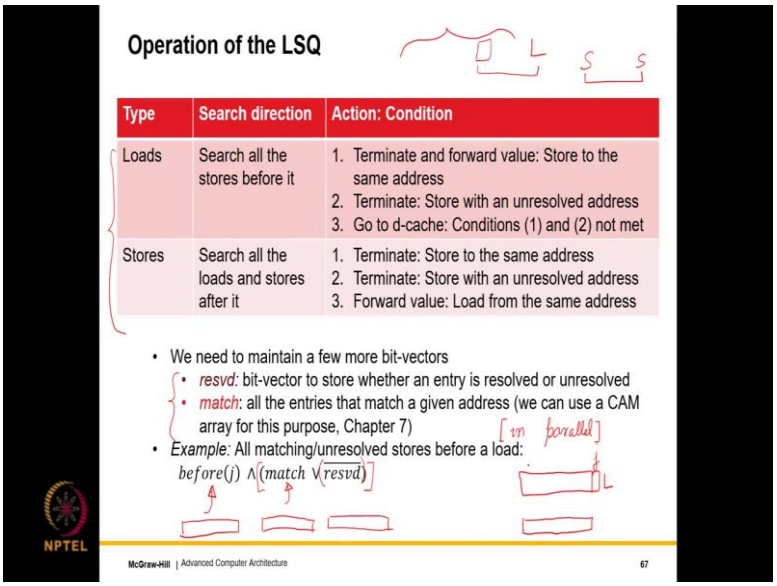
So, we are introducing a new function here called the map function. So, the map function has 1 for j map j has 1 for j and 0 for all the other positions. So, it is rather obvious to see that why this expression would provide would lead to these three cells being selected as one. So, we can quickly see NOT of prec j, they are essentially all of these cells ANDed with NOT of map j.

So, if I do that, then this cell will go away. So, we are only left with the cells from here and here. Prec tail or map tail are essentially all the entries from the beginning to this point and then, we can see that a logical AND between them will provide us pointers to these three entries.

Similarly, we can also consider the next case fare a j > tail. So, in this case this is head. The array as you can see has wrapped around and j > tail. So, in this case we would like all of these four entries to be selected. This is not very hard to see. So, let us consider each of these individual terms are NOT of prec j are essentially these two terms.

prec tail or map tail are these four sorry not terms, but entries and if I take the union of them of course, I have these four and these two, but out of this I want to discard this. So, I just compute a logical AND with NOT of map j. So, that removes this cell. So, what I am left with that these five which are the ones that I am interested in. So, given the fact that I have computed the before and after functions, I can proceed.

(Refer Slide Time: 52:42)



So, for the sake of readability, I am just reproducing the table that we had shown on a previous slide that talks about what we do for loads as well as for stores. So, for loads; what we do is we search all the stores before it . And this is easy, because when we add a load to a load queue, we are essentially adding a pointer; the value of the tail pointer that existed in the store queue.

So, that can be used to check for all the stores before it and. So, then of course, we need to find see if there. Let us say, this is the load we find all the stores before it considering wraparounds and everything. And in this, we need to find the latest store that is either a store to the same address or an unresolved store then, we need to consider this time window sorry this window of entries. And any entry over here, can potentially forward a value to the load of course, if the address is matching.

Then let us consider stores. So, for stores it is the reverse. We essentially look at later entries and so entries that came after the store. Here, also we terminate, if you have a store to the same address or an unresolved address. So, here also we have a window between one store and another store.

For all the loads in this window that map to the same address, we can forward the value. So, that is what we do for stores. So, we have up till now, only discussed the after and before functions. We need to maintain a few more bit vectors. So, resolved. Let us say it

is a bit vector to store whether an entry is resolved or unresolved. So, we can say it will store a 0, if it is unresolved one if it is resolved.

Another bit vector called match, which shows all the entries that match a given address right it shows an address match. So, how will we implement this quickly in hardware without doing a sequential search, it turns out that this process can be done in parallel. So, we will only have to use a content addressable array called a CAM array.

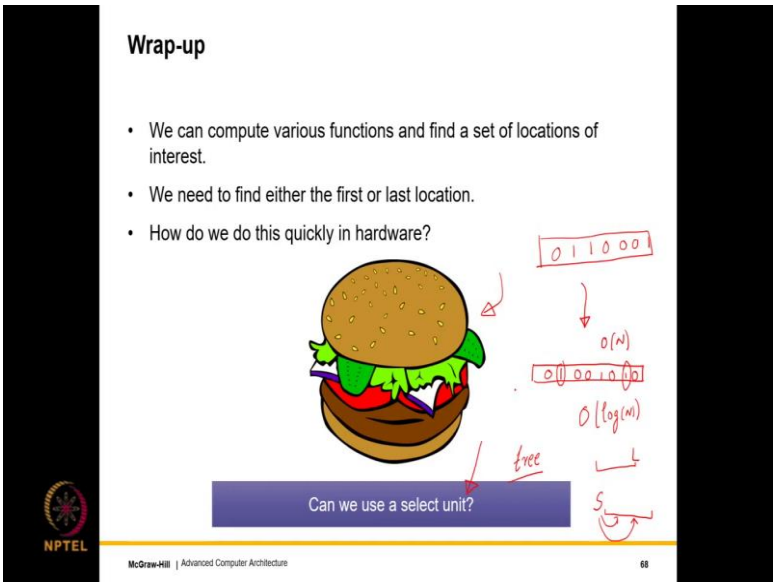
We are not in a position right now to appreciate what a CAM array is, but we will gradually do that. So, we will do it in chapter 7. We will use CAM arrays to some extent in the next lecture on come on instruction retirement. But in chapter 7, we will discuss a CAM array in great detail, but till that point let us assume that we have a mechanism to find in parallel all the entries that match a given address. See, if I were to find all the matching and unresolved stores before a load. So, if this is a load and what I want to do is that before it.

I want to find all matching and unresolved stores. So, what I need to do is that first I compute the bit vector before j and then, I compute. See, either then or in parallel, I compute a match vector, which is essentially a bit vector that points to all the entries that match the address; 1 if there is a match 0 if there is no match.

So, before j produces a bit vector, match produces one more. And resolve produces one more. So, since we are interested in unresolved entries. So, we just compute a logical NOT of that. So, match or the complement of resolved. This is essentially a bit vector that represents all the matching entries the all the entries that are either matching or unresolved. Matching means same address and before j and this quantity basically means that these are all the entries that are before the entry in consideration.

So, if this is j. So, before it, all the entries that additionally have these properties. So, what is the final outcome? Well, the final output over here, is essentially a bit vector that has 0’s and 1’s. If it has a 1, it means it is before j. Along with that, it either matches or it is unresolved and that would essentially indicate that this searched terminates over there.

(Refer Slide Time: 57:40)



So, what do we do; what we have learned here is that we can compute a wide variety of functions and find a set of locations of interest. So, once we have done that. What is the output of all that we do; the output is a bit vector of let us say that contains 0 1 0 0 0 1. In this bit vector, we often need to find the position of either the latest one or the earliest one or let us say the leftmost one or the rightmost one.

So, the question that we would like to now, ask ourselves. So, this is a very stock question for hardware designers is that given a bit vector, which of course, we have computed with a combination of our before or after functions and the match and the resolved arrays. So, this I claim is sufficient to handle all cases of loads and stores.

How can we do this quickly in hardware fair? What is the problem? The problem is either find the position of the leftmost one or the rightmost one. So, left or right depends upon the problem on what we are trying to do.

And how do we do this quickly? Well, we do not want to scans. See, if there are N entries, scanning will take order and time, which is inefficient. So, we do not want to do it. We want to use some solution that uses order of log N time to quickly find out the index the position of the leftmost one or the rightmost one well. So, given the fact that I have a burger over here, all of you are expected encouraged to think about this for 5 seconds.

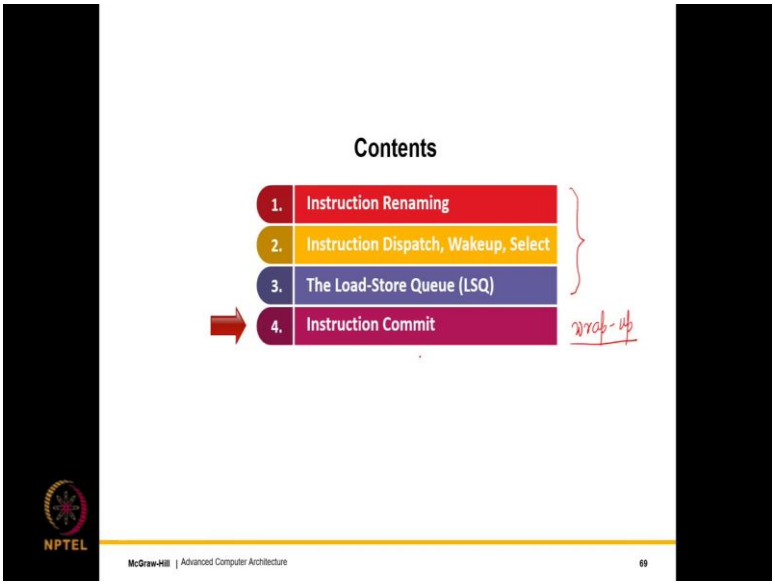
So, I am giving you 5 seconds now. Can we use a select unit? Well, this is something that you need to think I do not want to answer this question, but can we use a tree shaped select unit to quickly find the position of. So, of course, it will not be a regular select unit, it will be a select unit with modifications, but can we use it to find the position of the leftmost one or the rightmost one in an array of 0's and 1's.

If we can, then we can find the index to tell which we need to search the, till which we need to scan for loads and stores and also, a host of other information. So, it will essentially tell us that for a load what is the valid window within which we need to look or whether there are matches or not.

For a store again, what is the valid window within which we need to look. And furthermore, which entries within this valid window have a matching address and we need to forward. So, all of these things can sort of be computed in parallel in hardware. You have already shown the mechanism, how of computing the before and after functions, but again that is not enough, because the final result is a bit vector.

So, the bit vector finding out the positions of the leftmost one and the rightmost one are crucial; that is where I suggest that is simple modification to a tree structured select unit is all that you require. How to do it? Well, think about it.

(Refer Slide Time: 61:02)



Good. We are now in a good position in the sense, we have completed roughly 75% of the work. So, most of the heavy work within the pipeline is done. Now, essentially what we need to do is, we need to wrap up. So, this process of wrapping up, which is the process of removing instructions from the pipeline; this is known as instruction commit or instruction retirement. This we shall see next in the next video.