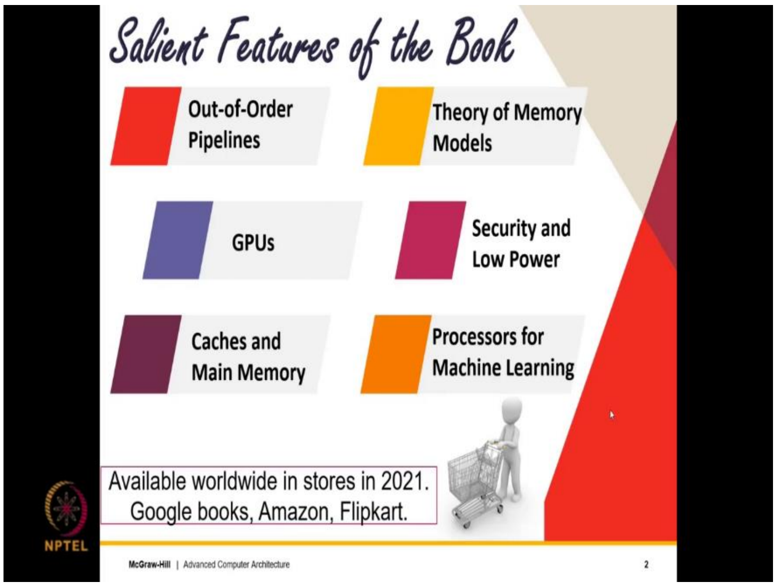
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Module - 03
Lecture - 09
The Issue, Execute, and Commit Stages Part-II

(Refer Slide Time: 00:17)

The slides are meant to be used along with the book, Advanced Computer Architecture, Smruti R. Sarangi. https://www.cse.iitd.ac.in/~srsarangi/advbook/

(Refer Slide Time: 00:23)



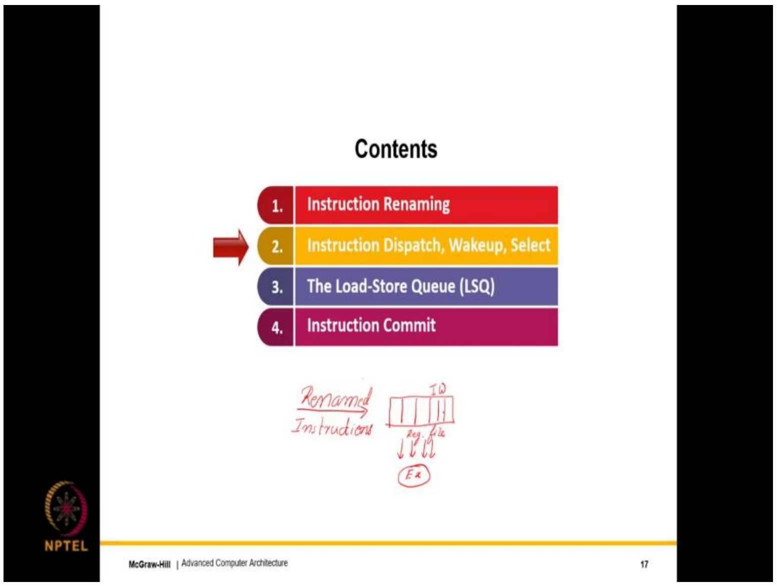
Salient Features of the Book

- Out-of-Order Pipelines
- Theory of Memory Models
- GPUs
- Security and Low Power
- Caches and Main Memory
- Processors for Machine Learning

Available worldwide in stores in 2021. Google books, Amazon, Flipkart.

McGraw-Hill | Advanced Computer Architecture 2

(Refer Slide Time: 00:51)


Contents

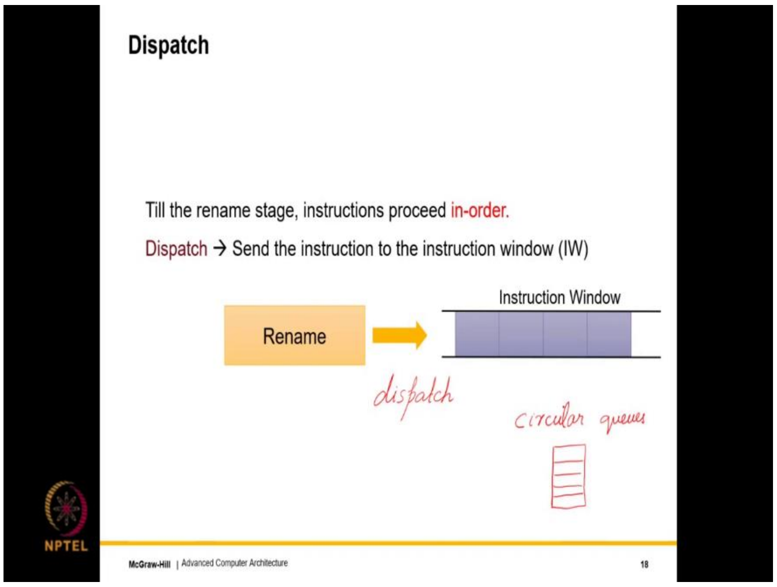
1. Instruction Renaming
2. Instruction Dispatch, Wakeup, Select
3. The Load-Store Queue (LSQ)
4. Instruction Commit

Renamed Instructions
IQ
Reg. file
Ex
NPTEL
McGraw-Hill | Advanced Computer Architecture
17

Let stages. So, their main aim is to taken if I were to draw. So, taken a sequence of renamed instructions and get them added to the instruction window that is what I mean to say.

So, some of these instructions would have both of their source operands ready some more. So, over the once that are not ready, well we wait for the source operands to be ready and for the ones that are ready is send them to the executions. So, of course, if they are ready, we need to read the register file first and then we read the register file and then we send them to the executions.

(Refer Slide Time: 02:04)


Dispatch

Till the rename stage, instructions proceed in-order.

Dispatch → Send the instruction to the instruction window (IW)

So, let us look at the first stage which is the dispatch stage and then we will introduce the other stages first; first high level view and then we shall get into the details. So, until the rename stage instructions proceed in order, in program order; after that they entered into the instruction window.

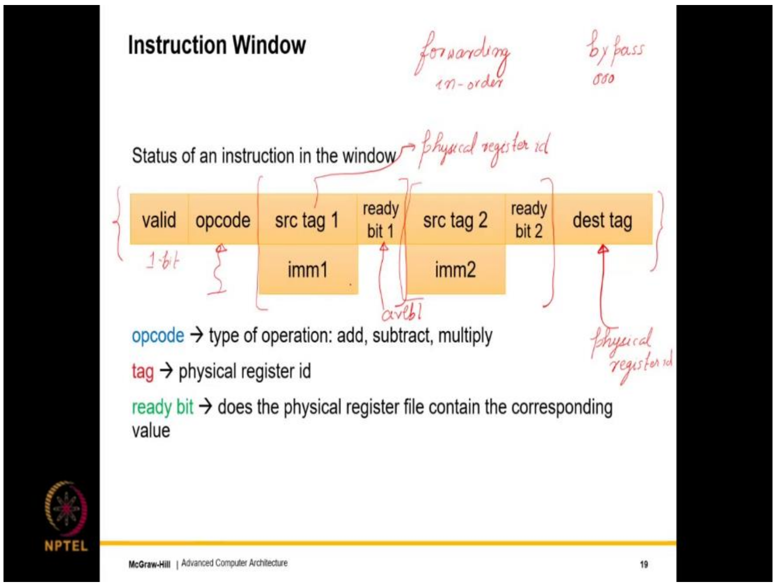
This also happens in program order, because the input of renaming is in program order, the output is also in program order and the instructions are entered into the instruction window from the rename stage.

So, this process is known as dispatch. So, different books or different sources can use different terminology and some might confuse dispatch with a related term called issue; but we will use this convention that, transferring instructions from the renaming stage to the instruction window is the dispatch stage.

So, the instruction window is essentially a set of instructions and it is typically organized in the circulate queues, almost all queues in hardware are organized a circular queue. The reason being that queue is essentially a software concept, that is meant to be implement implemented using a link list; but this is rather hard to do in hardware, because in hardware we cannot implement such a complex thing as a link list.

So, what we have is, we have a single array of entries and we have as discussed before, we have multiple pointers two in this case head and a tail and they wrap around. So, this is a very very standard design that is used in a large number of hardware structures. The instruction window most of the time uses something similar.

(Refer Slide Time: 04:07)



Instruction Window

Status of an instruction in the window

valid	opcode	src tag 1	ready bit 1	src tag 2	ready bit 2	dest tag
		imm1		imm2		

opcode → type of operation: add, subtract, multiply

tag → physical register id

ready bit → does the physical register file contain the corresponding value

McGraw-Hill | Advanced Computer Architecture 19

forwarding
in-order
bypass
ooo
physical register id
1-bit
avlbl
physical register id
NPTEL

So, what is the status of an instruction in the window, what are all the fields? So, it has the lot of information associated with it. So, let us go over them one after the other. So, the first field is that is the entry valid or not. So, almost all queue base structures would have such a single bit entry.

So, this is a single bit, it is a one-bit entry to say if the entry over here is valid or not. The next is opcode. So, the opcode is the operation code of whether we are dealing with the add, subtract, multiply, divide instruction; then we have information for each source operand.

So, we in our risk instruction set, we have a maximum of two source operands. So, if it is an immediate, we store the immediate value over here; because this comes along with the instruction and it needs to be stored somewhere, so it is stored over here. If otherwise if it is not an immediate; then what we do is, we store what is call the source tag and the source tag is nothing, but the physical register id.

So, it is just the idea the physical register, if we are talking of a register operand here and this is the ready bit. So, the available bit that we read in the rename table, the ready bit is definitely related to it; well how? If the operand is an immediate, well then the ready bit is 1; otherwise the ready bit is the same as the available bit of the rename table.

So, what did that bit indicate? Well, that bit indicated if the instruction if the if the first source operand in this instruction, if it is ready or not; if it is ready, then we directly read it from the register file or we read it from the forwarding network. So, forwarding is primarily an in order pipeline concept.

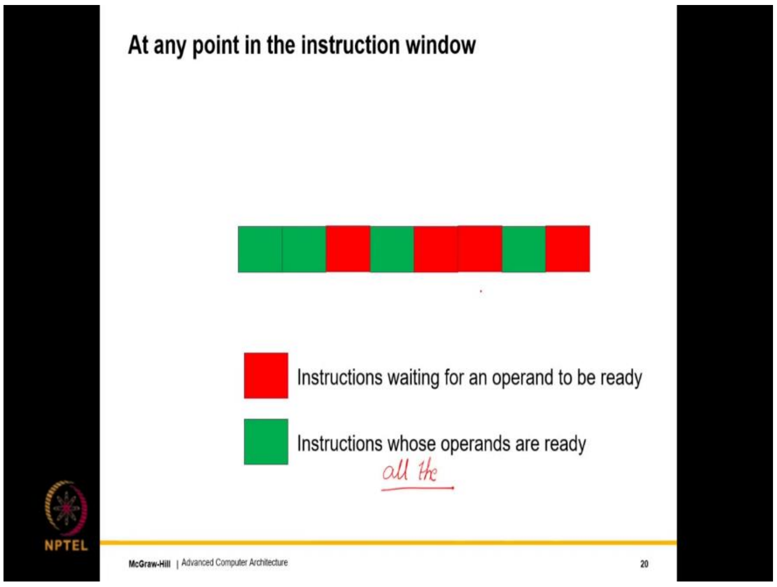
So, we have something very similar in out of order pipelines; because whenever we have multi cycle functional units and we do need forwarding. So, there is no escaping from forwarding, well it has a different name. So, we will henceforth use the different name, even though the mean exactly the same thing in out of order pipelines.

So, in out of order pipelines; instead of using the term forwarding, we will use the term bypassing. So, bypassing forwarding same thing; forwarding in order pipelines, bypassing out of order pipelines and the ready bit basically indicates, if either we have the data with us which means it is an immediate. So, we of course, have another bit to indicate if it is immediate operand or register operand that I am not showing.

And it stores the source tag, source tag is nothing, but the physical register id. Similarly, we have exactly a same copy of this information for the second source operand, if there is one. And also we have the destination tag, where the destination tag is that if this instruction has the register destination, a lot of instructions do not.

For example, the store instruction does not have a register destination; but if there is one, what is the physical register id corresponding to this instruction, corresponding to this destination? So, that is the most important aspect over here that, we basically deal store and deal in terms of these tags. So, we will see that this will become important at a later stage.

(Refer Slide Time: 08:30)



At any point in the instruction window

🟥 Instructions waiting for an operand to be ready

🟩 Instructions whose operands are ready

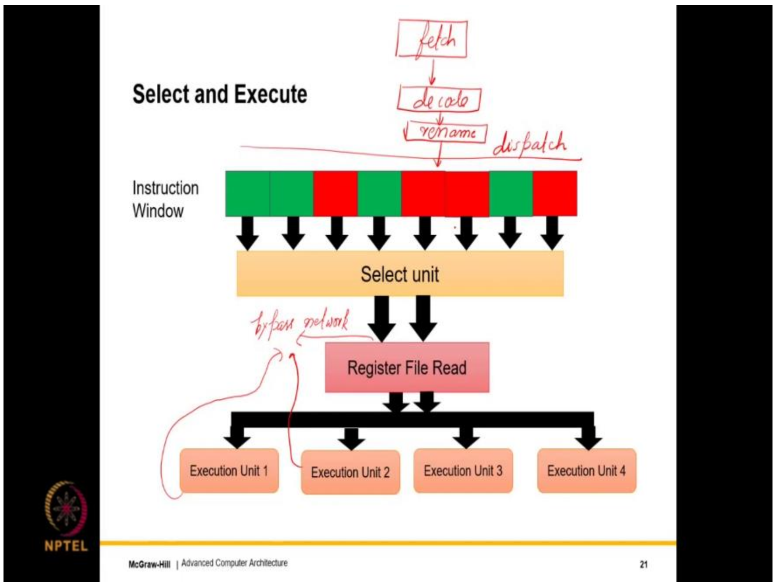
all the

McGraw-Hill | Advanced Computer Architecture 20

So, at any point in the instruction window what will happen is that, we will have a set of green instructions and a set of red instructions. The green instructions will be those instructions or whom all the operands are ready; whose operands means, all the operands right, not one or two, but all the operands are ready.

So, they will be the green instructions. And we will have a set of red instructions, which are; so red means kind of like a traffic lights stop there. So, these are instructions, which are waiting for an operand to be red. So, either one operand or both operands are not ready, so they are waiting.

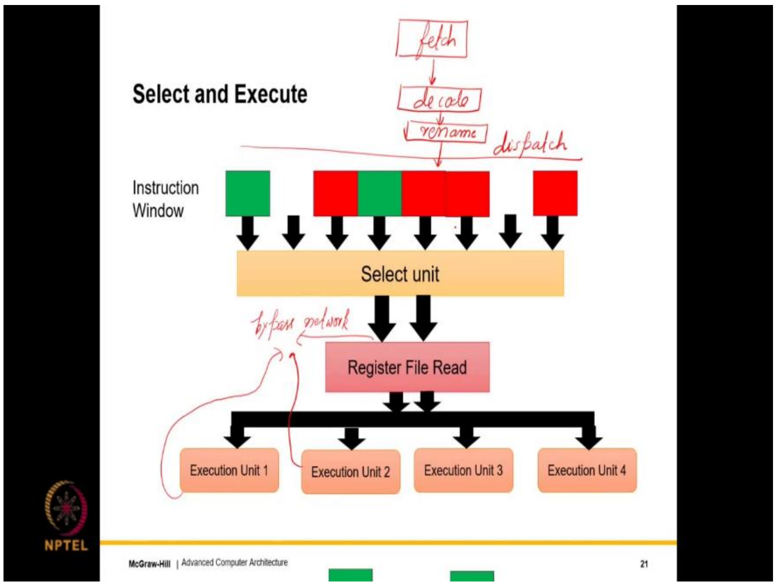
(Refer Slide Time: 09:11)



So, let us look at this part of the pipeline. So, the earlier part what did we have? We of course, had fetch; so fetch had a host of things like branch predictors and so on. Then we had decode, then we had renamed and then we essentially this process is known as dispatch, where we write it to the instruction window.

And at any point of time, we will have a set of green or ready instructions and a set of red instructions that are essentially waiting. So, these instructions are waiting for one or both of their operands to be ready. So, out of these green instructions, a few of them; well all of them we can execute, but let us say that we have two adders and all four of them are add instructions, we can only execute two out of these four.

(Refer Slide Time: 10:17)



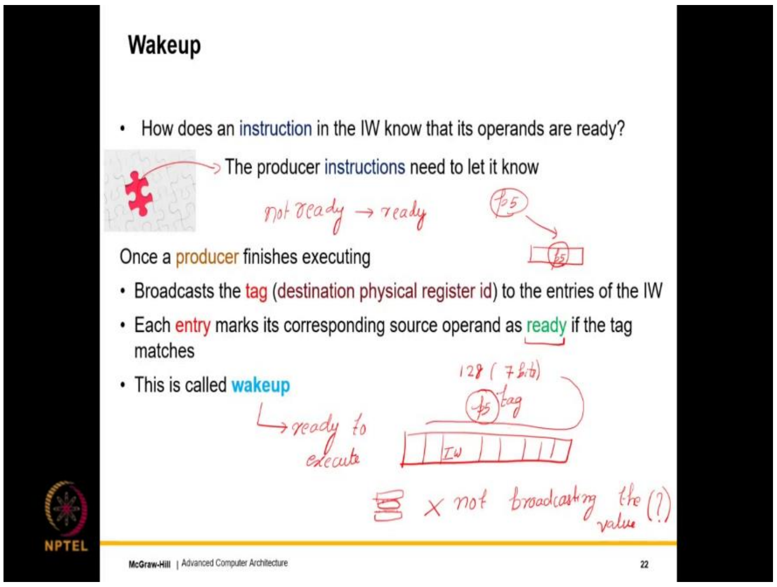
So, let us say they are these two. So, these two instructions essentially get selected. So, we have a select unit. So, it selects these two instructions; they read their value either from the register file or of course, what I am not showing here is that, we have a full bypass network here. We have a full bypass network here ok, from the different values that are being written and we have a full bypass network here at different stages in even from within.

So, we have a host of, is just similar to forwarding; we have a large number of multiplexers, so this is the bypass network. So, it is not being shown for the sake of simplicity. So, it does not matter where they get the value from; they get it either from the registered file or from the execution units, does not matter.

So, I will just show the animation once again, they get selected. So, this point structural hazards can happen; it is an out of order pipeline, so definitely structural hazards can happen. So, as I said we could have only two adders out of this four execution units and all the four green instruction shown away here may be add instructions.

So, we just choose two of them; well how do we choose? we will discuss that in detail when we discuss the select unit. And how do we know who is ready and how is this information propagated? even that will also we discussed; but this is just the high level view. So, in the high level view, we choose two instructions; select them, read their source operand values and execute.

(Refer Slide Time: 11:56)



Wakeup

- How does an instruction in the IW know that its operands are ready?

→ The producer instructions need to let it know

Once a producer finishes executing

- Broadcasts the tag (destination physical register id) to the entries of the IW
- Each entry marks its corresponding source operand as ready if the tag matches
- This is called wakeup

not ready → ready
p5
p5
128 (7 bits)
p5 tag
IW
↳ ready to execute
× not broadcasting the value (?)
McGraw-Hill | Advanced Computer Architecture
22

So, now, that we have seen the overall picture, let us now start looking at each of these individual stages, then great details. Let us can have elaborately look at each of these specific individual stages. So, some of the key questions that we need to answer over here is that, consider an instruction and instruction window that does not have one of its operands ready.

So, how does it know that one of its operands has become ready? So, then. So, the question here is that, how do we make a not ready to a ready transition so; somebody needs to let it know. Who lets it know? Well the solution to the puzzle is that, the producer instructions that are producing the value, they need to let it know; for example, let us say that the physical register is p 5 and p 5s value is being produced and it is being consumed by a later instruction.

So, at the moment p 5 will not be ready; but when the producer produces the value of p 5, it needs to somehow let all of the consumers of p 5 know. So, once a producer finishes executive it broadcast the tag; the tag is nothing, but the destination physical register id as we have seen to an entries of the instruction window.

So, if this is the instruction window, all the entries are kind of listening and once p 5 is done, it is sort of broadcast all of them. All the entries check if p 5 is one of their sources if it is, if the tag matches, if it is one of the sources, they set the corresponding source operand to be ready.

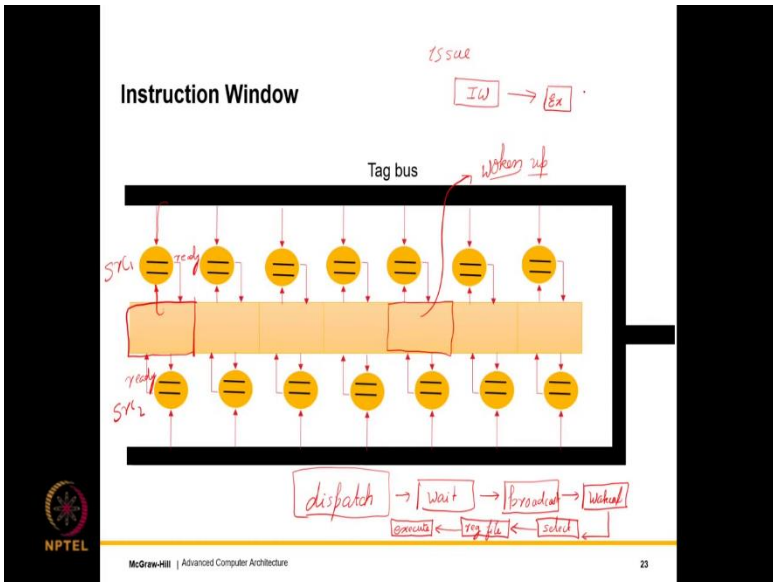
Now, this process is known as wake up. So, in this case we are waking up an operand, it is also known as waking up an instruction; for example, if for an instruction it finds that after a wakeup both its operands are ready, then we say that the instruction has woken up. So, wake up means that, it is ready to execute. What does that mean in this case? Well, it what it means is, that it can go ahead and read the value of its operands.

What is the important point to make here? The important point that we want to make here is that, what is being broadcasted is the tag. So, the tag in this case, if there are 128 physical register, the tag is only 128 log 128 to the base 2, which is 7 bits. So, that is all we are broadcasting, we are not broadcasting the value, ok. So, that is the very important point to keep in mind that, we are not broadcasting the value.

Why are we not doing that? Well, that will become very clear later on, not now, but it is very important. So, keep this in mind, it is an important, it is an important question that, we are not broadcasting the value. But why are we not broadcasting the value, I will not tell you the reason now; this kind of food for thoughts, I am drawing a virtual burger, I am a bad drawer, but this is essentially like a side view of a burger, add your letters and chips, other unhealthy stuff in between.

So, not broadcasting the value is something we are doing right now, we are just broadcasting the tag, which is 7 bits; 7 bits is not much, a value can be a fact 64-bit value. So, that is not being broadcasted and 7 bits broadcasting is fast and quick and this can be done.

(Refer Slide Time: 15:45)



So, what would be a representation of this? Well, the representation is very simple that, in the simple diagram there is a tag bus. So, the tag bus is being broadcasted to, so the tag is being broadcasted to both sides. So, let us say one side are comparators for the first source operand, and the second side are comparators for the second source operand.

So, tag could match any one of them. So, what we have is that for every entry, we have two comparators. So, for let say this entry; we have two comparators, source 1 and source 2. So, for each of them, well what do we do? We read the contents of the entry, we read the contents of the tag; if they match, we set the ready bit.

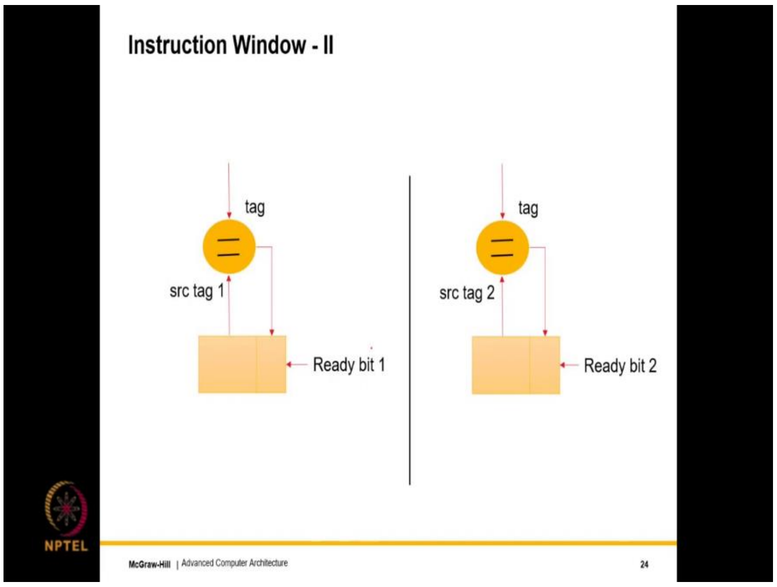
Similarly, here if the match, we set the corresponding ready bit. So, this is done for every single entry and for any instruction, where both the ready bits are set are the instruction is set to have woken up. So, every cycle whenever we broadcast, we will definitely wake up some operands and we will also wake up some instructions, if the rest of the operands are ready.

So, whatever instructions are woken up, they are set to be in a woken up state. So, they subsequently need to be selected, go through the select unit and then get, read the values from the registered file and execute. So, what is this, what is the sequence here? Well, first we dispatch and once we dispatch, they enter the instruction window; then of course they wait, subsequently the tag gets broadcast by the producer, so you broadcast the tag. Then what you do is that, you do your process of waking up, you do a wakeup; for all the instructions that have fully woken up, we select.

Right we select a subset based on structural hazard concentrations and then once you select a subset, we read the operands from the register file. So, it could either be the register file or the bypass network. Once that has been done, what do we do? Once that has been done, well we do not know much, we just execute.

So, this process of waking up the instructions right and sending them to the execute units is known as issue. So, what do we do? Well, issue basically if I were to look at it broadly, is set fair I can basically take the instruction when do and from it I send them to the execution units. So, this is said to be issuing to the execution unit. So, this process is known as issue.

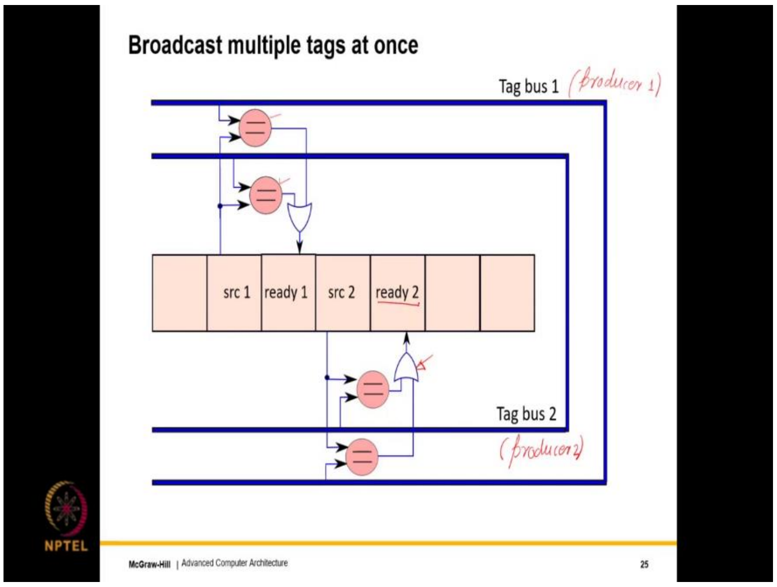
(Refer Slide Time: 19:09)



Instruction Window - II

So, here is the slightly deeper view of it. What we do is; we compare the both the sources source tag 1 with the broadcasted tag and source tag 2 with the broadcasted tag and we set the corresponding ready bits. So, this is happening for each entry.

(Refer Slide Time: 19:32)


Broadcast multiple tags at once

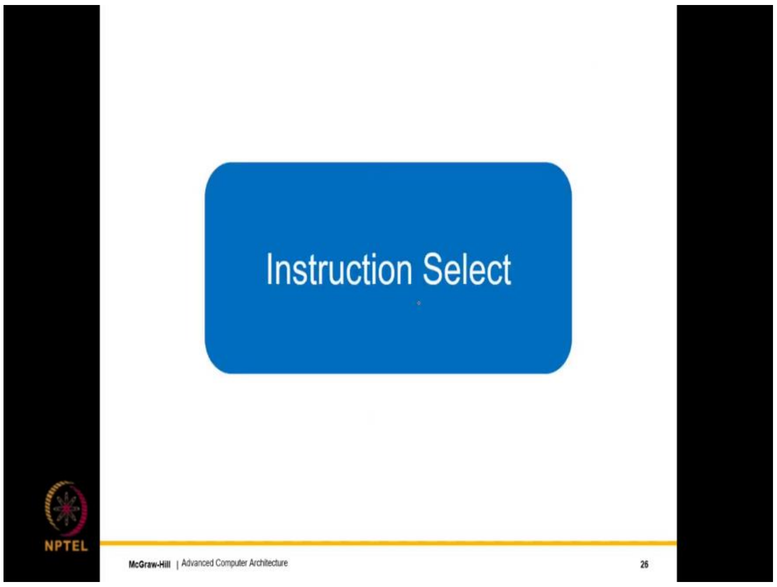
Let us now consider the case, where we have multiple tag buses. So, recall that if we have a multi issue processor, where we send multiple instructions to the execution units every cycle. So, multiple producer instructions will become ready every cycle and they will broadcast their tags.

So, we actually need multiple tag busses one for each producer. So, this can be producer instruction 1 and this can be producer 2. So, the producer instructions can broadcast their tags. So, what will actually happen is that, we need to compare both the tags that are being broadcasted, tag 1 and tag 2.

So, let us say for the first source one, we will actually compare both the tags. So, we need two comparators, we need one over here and we need one over here. So, both the tags are compare and then an or gate. So, just one of the comparisons will be successful. So, we will never broadcast the same tag, the same tag will not be broadcasted on both the buses.

So, there will be at most one match. So, if there is a match the or gate will catch it and it will set the ready bit; same as the case with the ready bit for the second source, where we will broadcast two tags and if there is a match, this will be captured by the or gate and this will set the status of ready 2.

(Refer Slide Time: 21:31)

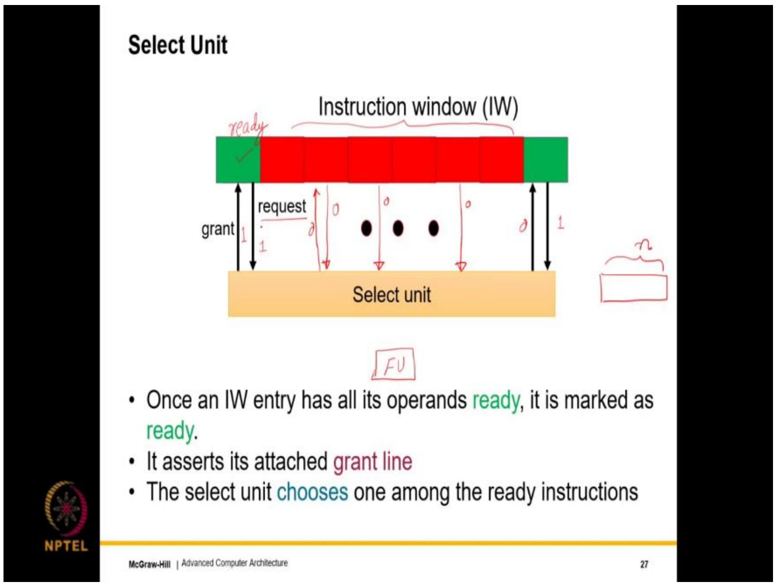


Instruction Select

McGraw-Hill | Advanced Computer Architecture 26

So, this was the basic broadcast and wake up mechanism. So, let us now take a quick look at the instruction select mechanism; which means that once we have chosen, once we have chosen the set of instructions what do we do.

(Refer Slide Time: 21:43)



Select Unit

- Once an IW entry has all its operands ready, it is marked as ready.
- It asserts its attached grant line
- The select unit chooses one among the ready instructions

McGraw-Hill | Advanced Computer Architecture 27

So, what we do is that, let us consider this is the instruction window. So, in the instruction window we will have instructions of two types, red and green. So, green are green instructions are the ready instructions, which are ready to execute and the red instructions are waiting. So, each instruction there is a wire. So, there is a wire essentially for each entry. So, we call it the request line. So, whenever it is ready, it just a search its request line, which means it is set to 1.

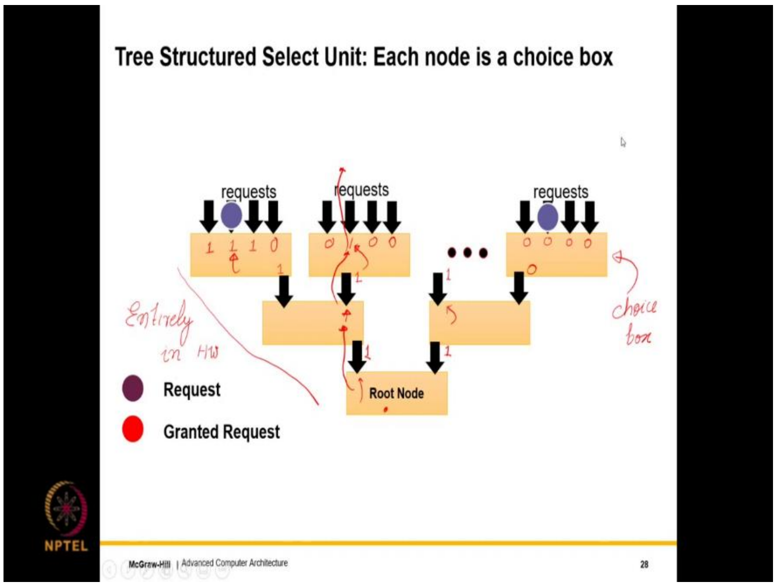
So, there are similar request lines connected from the rest of the entries and all of their status is 0. Well, again for the next green line, it is 1. So, what the select unit does is that, it basically selects one of the requests and grants it. So, well how does it do it and how does it do it for multiple instruction types and takes and take structural hazards into account?

Well, we will take a look at all of those issues; but let us look at a simplistic variant of the problem, where we are considering only a single type of instructions and let us also consider only one functional unit, only one execution unit. Let us proceed with that, we will gradually complicate this two.

So, what is the idea over here? The idea over here is that, we have n inputs and. So, let us see the total of n inputs in the select unit. In the n inputs, so out of the n inputs, the few are 1 and few are 0; if it is 1, it clearly means it is interested to be selected. So, one among these request is selected, which one we will see later. And then the one that is selected, let us say that this instruction is selected; then its grant line is asserted to 1 and the grant lines for all the others remains as 0.

So, the one that is asserted, essentially that entry can proceed forward and read its data and execute. So, you read the register file subsequently and you execute. So, the job of the select unit is pretty much to select one among, the one among the n that is interested or whose request line is set to 1.

(Refer Slide Time: 24:33) .


Tree Structured Select Unit: Each node is a choice box
requests
requests
requests
Root Node
Request
Granted Request
Entirely in HW
choice box
McGraw-Hill | Advanced Computer Architecture
28

So, let us now take a look at a select unit. So, select unit takes in a set of requests and its job is to assert one of the grant line, the request that is granted. So, the grant lines are not being shown for the ease of readability. So, in this case what we see is, we see a tree structured select unit as you can see over here; these are the different levels of the tree. So, the tree need not have a fixed arity.

So, I am showing a tree where there are four leafs at the, so sorry there are four request coming in at the lowermost layer and all the internal nodes have two each. So, it does not matter. But what matters is the logic of these rectangles over here. So, a rectangle here is called a choice box. So, what a choice box does is like this, that it first inspects all the request that are coming in, they can be either be 0 or 1.

So, let say if all of them are 0, it is set its output to 0; basically saying that none of its requests are interested. Let us say one of them is a 1 and the rest are 0; well then it is easy, you just remember the fact that this was the request that it shows and it sets its output to 1.

But let us say multiple requests are 1; in that case it needs to choose 1. Well, which one that we will see; but let us assume for the time being that it chooses one kind of arbitrarily randomly, let us say it chooses this one. So, then it remembers its choice internally; so internally it has a set of registers to remember which choice. So, that is not much; if there are four inputs, you just needed 2-bit register. So, with that you remember this choice. Then what you do is that, the output is set to 1.

So, this choice box also does something similar, so let us assume output for this is 1. So, then this choice box would do something similar, it would remember which input it chose. So, let say it chose this one, it will then set its output to 1 and if this choice box does not have a choice, it needs to choose this one, it also sets into 1.

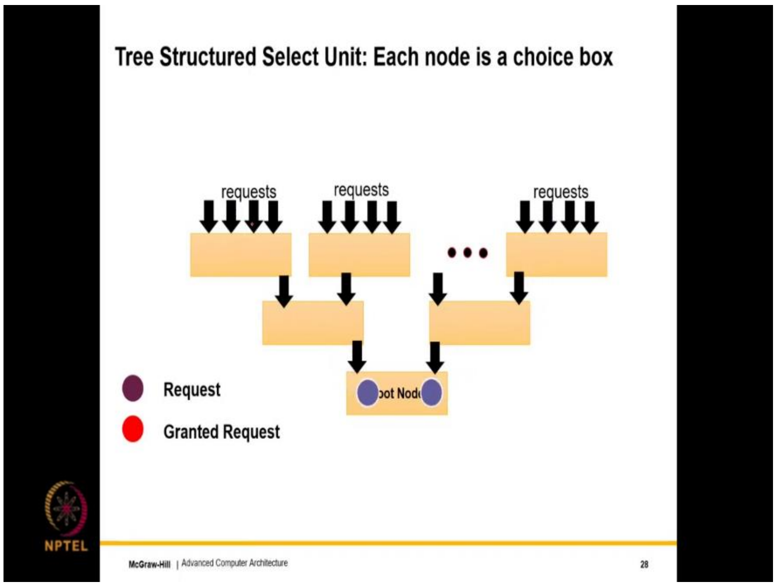
Finally, the root node will make a choice. So, let say it chooses this one. So, then it will decide to grant the request; the grant signal will follow the reverse path, it will grant this, then this choice box will go back to the request that it had remembered. So, then this is this and then the grant signal will go via this root.

So, what we are essentially doing is that, we are aggregating the choices in a tree structured fashion. And so, this entire thing by the way is happening in hardware. So, entirely in hardware. So, whoever thought that trees link list and queues are concepts that are only used in software data structures, that is totally wrong; see you can clearly see an example over here, where a tree is implemented in hardware. So, this tree over here is entirely implemented in hardware.

And so, this is this is of course, the VLSI implementation and all that you require is, we need a little bit of state at each choice box to remember which input was chosen. Once that is done, well we just aggregate the choices, keep going towards the root aggregating the choices and finally, one is chosen and a reverse path is followed.

So, what we need to do now is, we need to create a more a fairer version of the selection mechanism, but I have a small animation that I can show you. So, let us say that we have two requests with these blue circles over here.

(Refer Slide Time: 29:03)



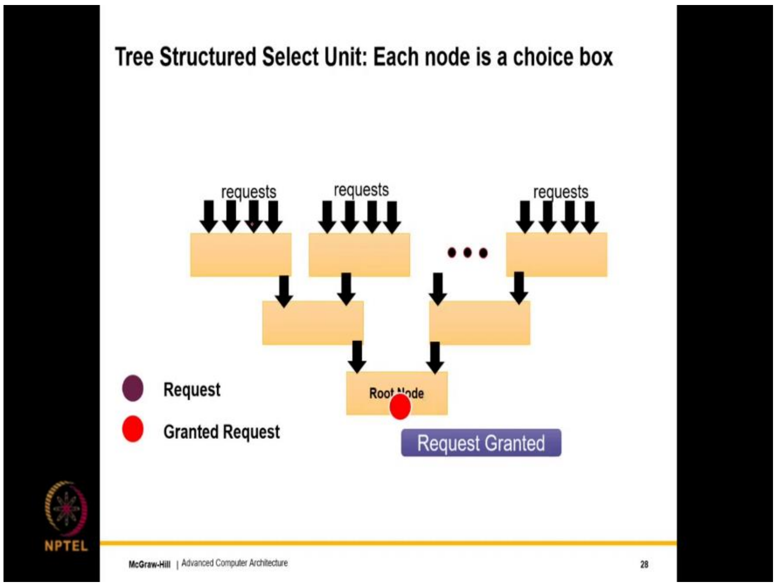
Tree Structured Select Unit: Each node is a choice box

● Request

● Granted Request

So, this is how they flow; they come to the root node, so assume that there are no other requests, the root node chooses 1.

(Refer Slide Time: 29:11)



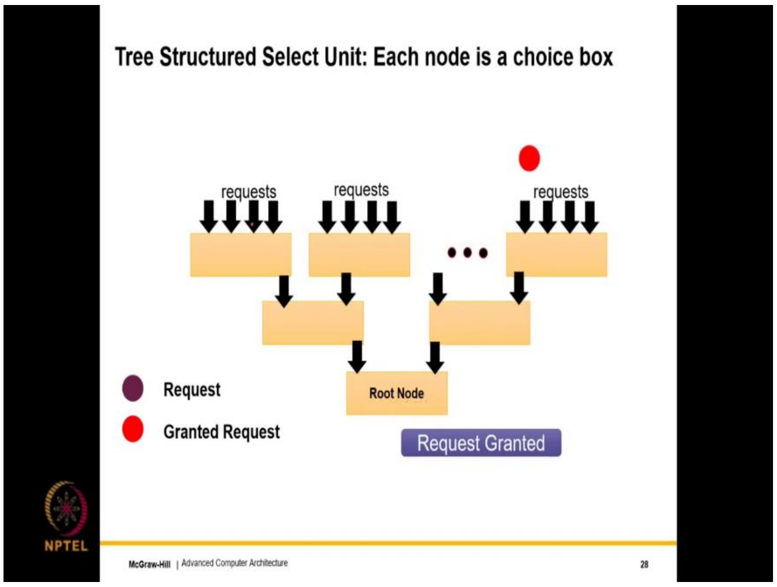
Tree Structured Select Unit: Each node is a choice box

● Request

● Granted Request

Then the grant signal as we have discussed goes back the reverse way.

(Refer Slide Time: 29:16)

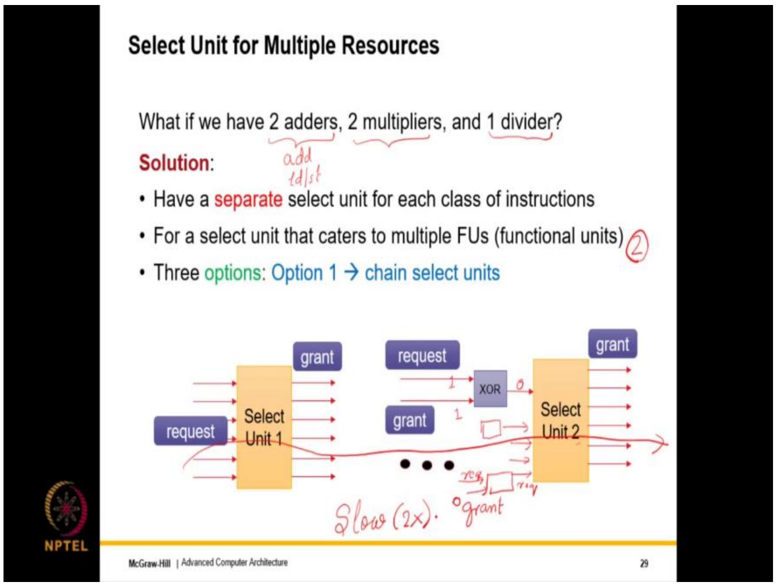


Tree Structured Select Unit: Each node is a choice box

McGraw-Hill | Advanced Computer Architecture 28

It requires some state at the at each choice box.

(Refer Slide Time: 29:20)



Select Unit for Multiple Resources

What if we have 2 adders, 2 multipliers, and 1 divider?

add ld/st

Solution:

- Have a separate select unit for each class of instructions
- For a select unit that caters to multiple FUs (functional units) ②
- Three options: Option 1 → chain select units

McGraw-Hill | Advanced Computer Architecture 29

Let us now make the select unit more complicated and more realistic. So, what if we have 2 adders, 2 multipliers and 1 divider? Well, the simple solution is that, we have a separate select unit for each class of instructions; for example, for the adders, we have a separate select unit.

So, for this select unit, all non add instructions are by default not interested, in or by default sort of not valid for this select unit. So, recall that with an adder, it is kind of an overloaded unit. So, it of course, does do addition, but it does addition as well as subtraction.

And it also computes the addresses of load and stored instructions. So, in that sense, we typically have more adders then what is required; then we can have multipliers and dividers. So, basically for multipliers also we have a separate select unit and the separate 1 for a divider.

So, consider a select unit that caters to multiple FUs or multiple functional units. So, there are separate options for it and also. So, also what I mean by this is, multiple functional units of the same type. So, this basically means that, if let say we have 2 adders, we will have one select unit for adders; but again instead of 1 unit, it will actually have 2 units, because there are 2 adders.

So, in this case one option which is the simple one is the chain select units; in the sense we have a set of request coming in, one of them is granted and for the line that is granted, if it is 1 and we just XOR it with its request. So, we do it for all and I am only showing the circuit for the request that was granted.

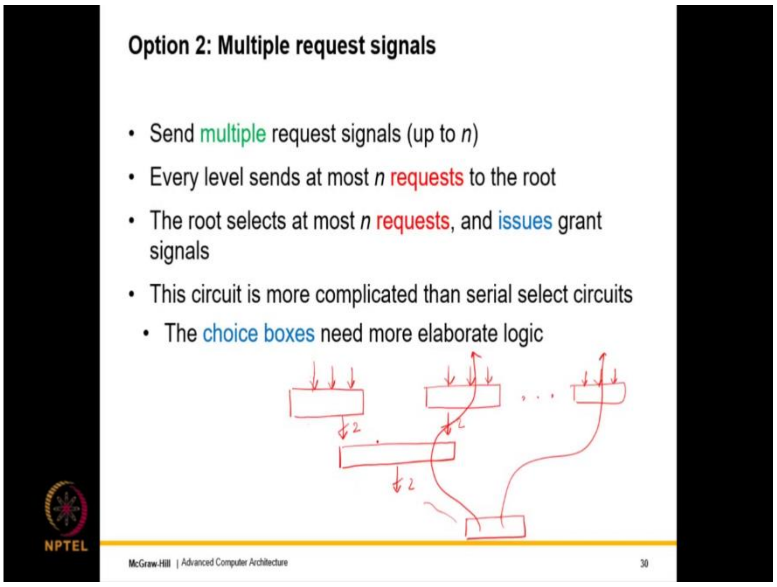
So, for the next select unit, essentially the request for it is 0; that is because it was granted by the earlier one, so it cannot be granted by the second select unit and it is 0. But for all, but for all the other inputs; so there are many other inputs and I am not showing them. So, all of these inputs. So, all of these inputs are pretty much connected to others XOR units; for each one of those, we will have a request unit, request line and a grant line which is coming from the previous select unit.

So, in this case, the grant line for 1 only one of them will be 1. So, the rest will be 0. So, pretty much the request will show up over here. So, the request is 1, it will show up over here; if it is 0, it will show up over here. The job of the second select unit is a to basically grant one more request.

So, that is how even if we have 2 adders, by chaining the select units, we can grant two requests. What is the disadvantage of this method? The disadvantage of this method is that, the time that is required is double, right. So, it requires twice the time, so it is slow.

So, the signal has to pretty much pass through the first select unit and then pass through the second select unit; say it is a roughly 2 times the latency, which is not optimal, which is clearly not what we wanted. So, we need to search for a more efficient method.

(Refer Slide Time: 33:11)



Option 2: Multiple request signals

- Send multiple request signals (up to *n*)
- Every level sends at most *n* requests to the root
- The root selects at most *n* requests, and issues grant signals
- This circuit is more complicated than serial select circuits
  - The choice boxes need more elaborate logic

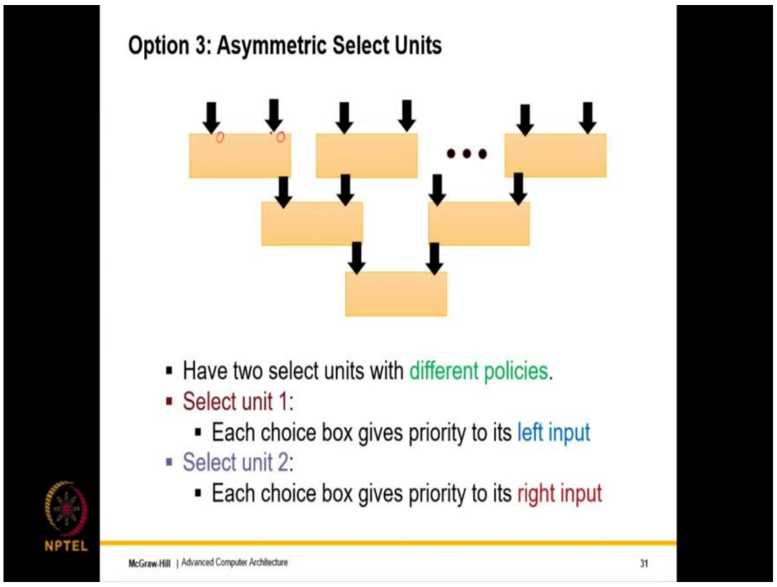
McGraw-Hill | Advanced Computer Architecture 30

The second method is that; we complicate the choice box. So, what we do is that, we send multiple request signals up to the root. So, we have the lowest layer of choice boxes looks the same. So, pretty much here we have request lines that are coming in. But each choice box instead of selecting one request, selects a maximum of two requests. So, each one of them instead of one selects a maximum of 2.

So, again they are send to one more choice box, which again selects a maximum of 2 so on and so forth until you reach the root. So, what I say is that, we can generalize this say every level sends at most n request to the root; the root will ultimately select at most n requests and issue grant signals, where one grant signal might go this way, another might go that way it does not matter, but at the most n signals at the most n request will be granted.

So, this you know certain sense is faster than the design shown in the previous slide, no doubt it is faster; however, the choice boxes need more elaborate logic and they become more complicated. So, it is not, it is a better solution at the cost of complicating the choice box.

(Refer Slide Time: 35:00)



Option 3: Asymmetric Select Units

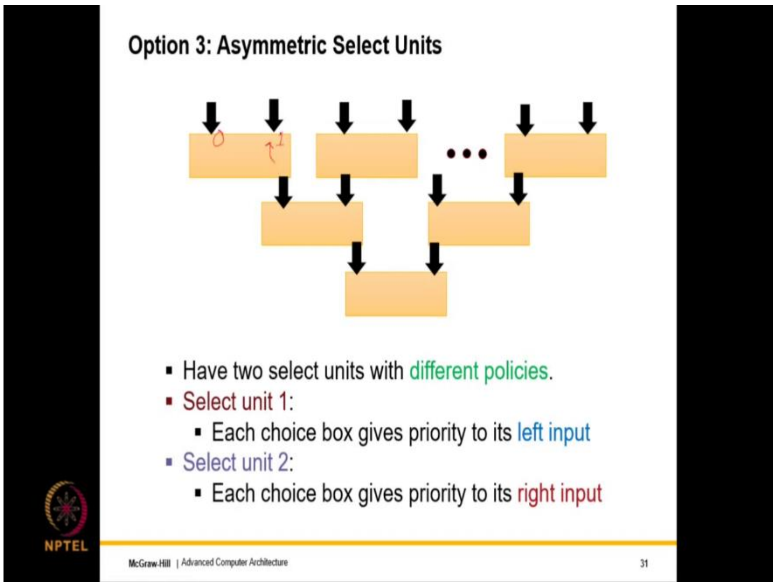
- Have two select units with different policies.
- Select unit 1:
  - Each choice box gives priority to its left input
- Select unit 2:
  - Each choice box gives priority to its right input

McGraw-Hill | Advanced Computer Architecture 31

Let us consider a different design that has asymmetric select units. See in this case what we do is, I have drawn a regular tree structured select unit, where there are two copies of the tree and both run side by side in parallel. So, let us consider the first select unit, which is select unit 1.

So, there will be four cases. So, if you consider this choice box, either both the request lines are 0; well then there is no problem at all, no choice needs to be made. So, we can happily erase the ink on the slide.

(Refer Slide Time: 35:38)



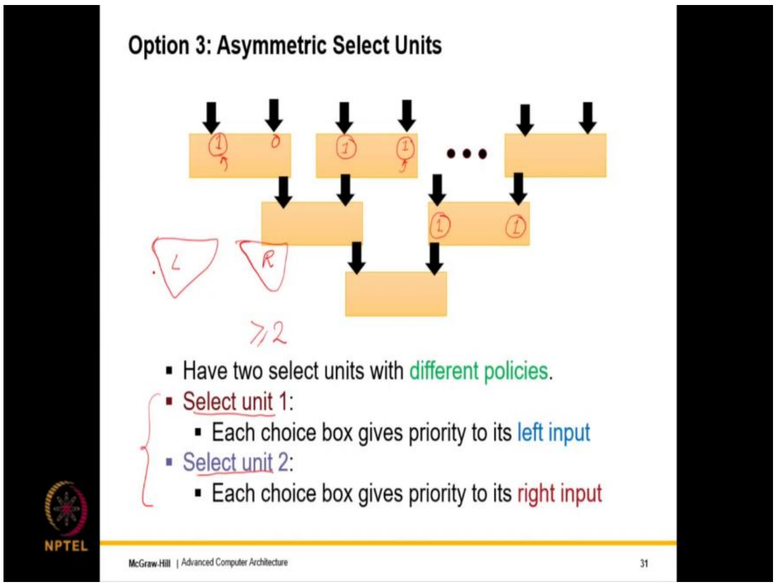
Option 3: Asymmetric Select Units

- Have two select units with different policies.
- Select unit 1:
  - Each choice box gives priority to its left input
- Select unit 2:
  - Each choice box gives priority to its right input

McGraw-Hill | Advanced Computer Architecture 31

If one of the request lines is 1, then also there is no problem at all; because by default this request is chosen. So, we can again erase the ink on the slide and likewise is the case for the other case.

(Refer Slide Time: 35:54)



Option 3: Asymmetric Select Units

- Have two select units with different policies.
- Select unit 1:
  - Each choice box gives priority to its left input
- Select unit 2:
  - Each choice box gives priority to its right input

If this is 1 and 0, then by default this input is chosen. However, if both the inputs are 1, then the fun lies here. See if both of them are 1 at any level, regardless of the level; the first select unit, select unit 1, the choice box gives priority to its left input. What this basically means is that, it chooses its left input. So, the left input is chosen. In the second select unit, each choice box gives priority to its right input; which means that the right input in this case, this input will be chosen.

So, it is possible to prove and in fact I would like to invite you to prove the following that; if I were to design a select unit in this fashion and if at least ≥ 2 request lines are positive, both the select units will never make the same choice. So, this is easy to prove, it is not hard at all.

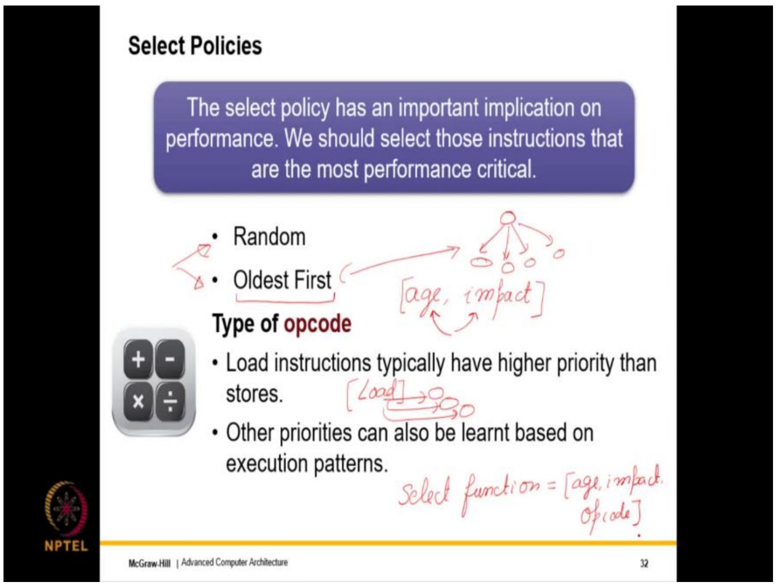
So, let us consider the case when only two request lines are 1. So, of course, they will be at different places. So, ultimately they will collide in some choice box whenever; so wherever they collide, let say they collide over here. One the select unit 1 will chose a left input, while the select unit 2 will chose the right input. So, automatically the choices will be different.

So, this can of course be proved for more than two inputs that are set to 1. So, it does not make a difference; this is a general result it can be proven. So, this is a rather fast method of designing parallel select unit. So, we will essentially have two, one like this and one more like this.

So, this gives priority to the left input and this gives priority to the right. And as we see, if we have 2 adders; then we can make two choices and they will be different. And in terms of latency, well the latency is roughly the same and we get twice the throughput. So, this is one of the examples of using a smart design method, with little bit of math to actually speed up hardware significantly.

I would like to repeat what I said that, it is normally constitute that links, queues, trees etcetera are software concepts or computer science concepts; however, that strictly speaking is not true, you are seeing an application of it in hardware.

(Refer Slide Time: 38:44)



Select Policies

The select policy has an important implication on performance. We should select those instructions that are the most performance critical.

- Random
- Oldest First

Type of opcode

- Load instructions typically have higher priority than stores.
- Other priorities can also be learnt based on execution patterns.

[age, impact]

[Load]

Select function = [age, impact, opcode]

McGraw-Hill | Advanced Computer Architecture 32

So, the select policy has an important implication on performance. So, we have till now kind of elided the question on which instruction we should select, it is a clearly all instructions are not the same. So, there might be one instruction and no other instruction is dependent upon it, so that should have a lower priority.

But it is possible that there is one instruction, but a lot of instructions are dependent on the output that it produces. Say if let us say this instruction is executed and in a certain sense sent forward; then all of the instructions are depend on it, all of them can wake up and the amount of ILP will increase significantly.

So, the main point of designing a good select unit is to figure out those instructions on which a large number of other instructions are dependent, these instructions should be selected. So, of course, how to do it in hardware and back to quickly? Well, that is difficult, but lot of modern select units actually have some estimate of how to do it. So, let us look at this.

So, one of course, the most simplest method and yet the most impractical is random, that which is select any input at random. Another option which is fair is oldest first, in the sense we give priority to the requests that are that have been in the instruction window for the maximum amount of time.

And given that the instruction window is a queue, from that position we can roughly a figure out how old an entry is. But ideally it should be a combination of the age and the impact; say any fairness by definition should be a combination of age and impact. So, this is kind of a natural rule in the sense that, even if a leader is to be chosen like a village headman or a headwoman or a head person.

Well, in general people give preference to age, because everybody has a natural respect for age; but along with that, you would also look at the most eligible person, the person has the maximum amount of caliber for doing a certain job. So, in this case, the caliber is the number of instructions that are dependent on it, but if we always start choosing this; then of course we will have starvation and that will lead to other problems, which we will see few slides later.

So, it should be some combination of basically age and impact and as I said it is a general rule in life as well. There are a few other important points to keep in mind; one is that load instructions typically have slightly higher priority than stores. The reason is that if you think about it whose dependent on a store, actually it is a later load, but there are other ways to take care of them.

So, stores pretty much write to memory and it is rather unlikely that you would write to memory and you would read it; but we have seen such a pattern in assembly code when you store a register and you read it back. But as I said, we will talk about another hardware structure called a load store queue that precisely capture such patterns.

Otherwise it is kind of rare that we would write the memory and read it back in general code. But in general for a load instruction it is on the critical path; because the sooner its value comes back, many many more instructions will be dependent, might be dependent on the value that the load brings on memory. So, many a time it is seen at load instructions are way more critical and thus if we have to give a priority, we should give a priority to a load as compared to the store.

So, a most modern select unit, sort of design a select function which is rather complicated function and also the function can change over time. So, a lot of learning is involved over here, which is sort of a function or the age the impact of the instruction; as we discuss the number of instructions dependent on it and also the opcode.

So, the opcode can for example, loads have higher priority than stores and so on. So, based on this you derive some sort of its code and this is kind of fed to the choice boxes and a choice boxes make choice accordingly. So, of course, they do become complicated; but well this complexity is, it is not a bad complexity, in the sense that this is a performance enhancing complexity. So, there is nothing to be offerly concerned about.

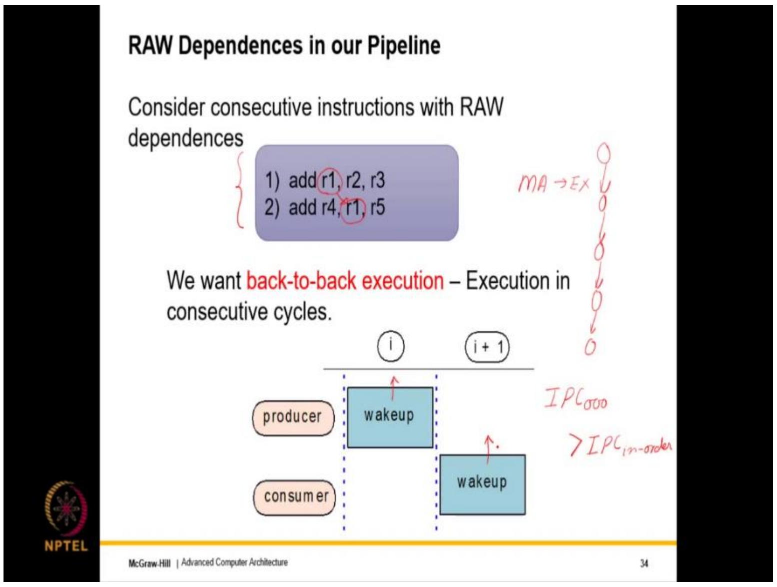
(Refer Slide Time: 43:57)

Tag Broadcast

McGraw-Hill | Advanced Computer Architecture 33

Now, let us come to another important and rather critical issue, which is how do we broadcast the tag and when do we broadcast the tag?

(Refer Slide Time: 44:10)



RAW Dependences in our Pipeline

Consider consecutive instructions with RAW dependences

1) add r1, r2, r3
2) add r4, r1, r5

We want back-to-back execution – Execution in consecutive cycles.

MA → EX
i
i + 1
producer
wakeup
consumer
wakeup
IPC_{ooo}
$> IPC_{in\text{-}order}$
NPTEL
McGraw-Hill | Advanced Computer Architecture
34

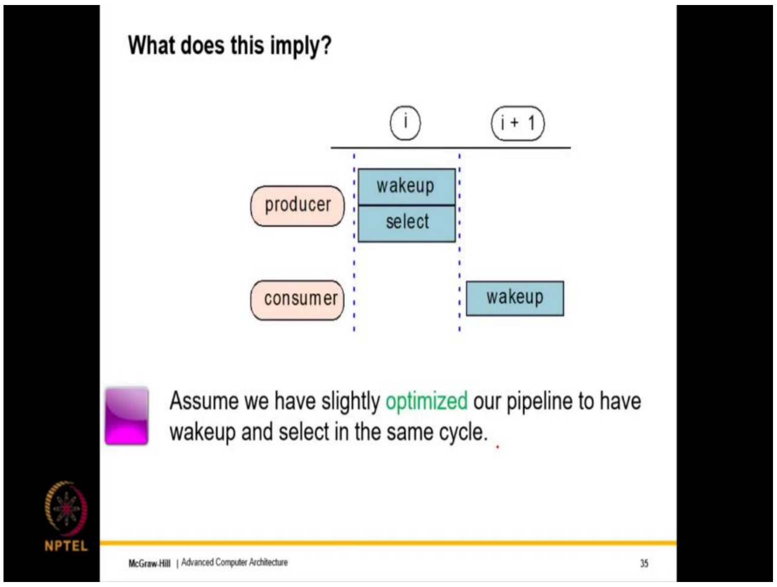
So, let us quickly look at this short snippet of assembly code. So, these are consecutive instructions with a raw dependency and so, this is writing to r 1, reading from r 1. So, even after renaming the read after write dependence would remain; so this would of course be in physical registers, but nevertheless there would be a dependence.

So, what we ideally want is, we want back to back execution; because after all in an in order pipeline with forwarding, they would execute in consecutive cycles. So, we would have forwarding pretty much MA to EX forwarding, memory access to execute forwarding, we would have that; but in order pipeline you would ensure what is called back to back execution or execution in consecutive cycles.

If we do not get that in an out of order pipeline, then for a long piece of code which has kind of a chain of dependencies and in order pipeline would give us a higher IPC, as opposed to an out of order pipeline and this is not acceptable to us. So, the reason that we went for an out of order pipeline is, because what we expect is that the IPC of an out of order pipeline is > IPC of an in order pipeline; but programs with the chain of dependencies have a potential or spoiling the party.

So, they have a certain potential of actually ensuring that IP the IPC of out of order pipelines is lower, which is something that we should not allow. And what do we need to do? Well, if the producer instruction wakes up in cycle i and proceeds to execution; the consumer instruction should also wake up in cycle (i + 1), which is the cycle just after it, such that there are no stalls in between. Because there are no stalls in between in an in order pipeline, so the same should hold in an out of order pipeline as well.

(Refer Slide Time: 46:19)



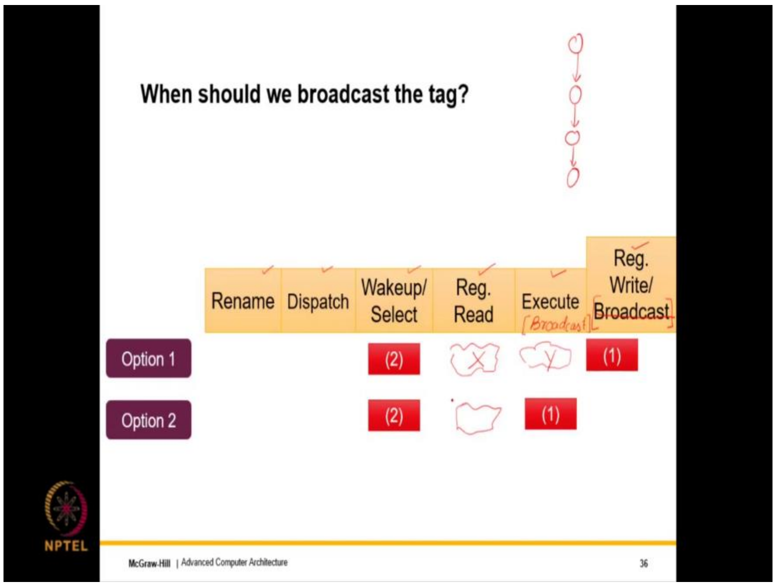
What does this imply?

Assume we have slightly optimized our pipeline to have wakeup and select in the same cycle.

So, what was this imply? Well this implies that, we sort of need to design these stages very carefully. So, let me look at one design point. So, in this design point let us assume that, we are able to optimize our pipeline, such that wake up and select happen at the same cycle.

So, if they happen in different cycles, it will introduce more complexities and it is also there is an exercise at the end of the book; but let us look at simple case first and we will then gradually slowly wrap up the complexity. So, in cycle I let us assume we can do both wakeup as well as select. So, that would be like packing two things, two very performance critical things together; but let us assume we have an optimized pipeline and good designers to do it. So, then what essentially happens?

(Refer Slide Time: 47:22)


When should we broadcast the tag?

So, what happens is that, this part of the pipeline would look like this a rename, dispatch, wake up and select, register read; we read the values from the register pipe, then we execute, and then we write to the registers. So, one of the simple options that we are looking at here is that, we actually broadcast over here. And if you broadcast over here, let us see what are the problems. So, note that in this pipeline, we have full bypassing and forwarding enabled; it is a forwarding in an out of order pipeline is called bypassing.

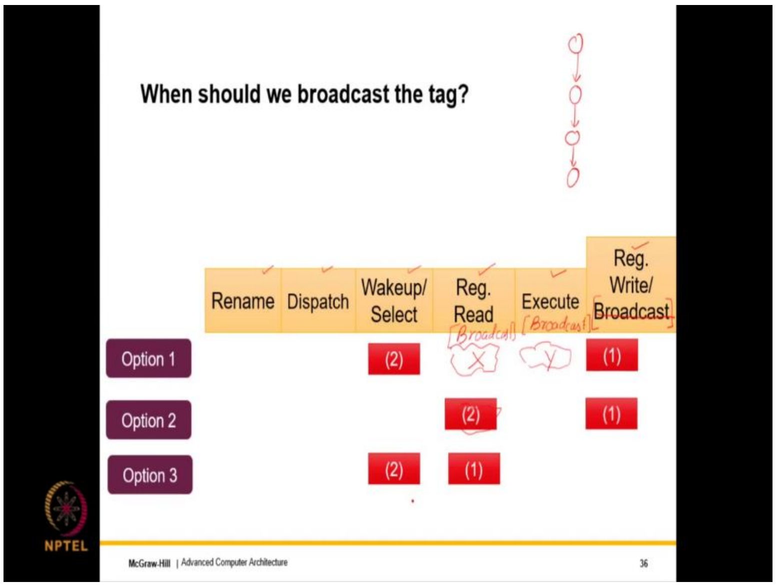
So, in this case what will happen is that, if you broadcast in this stage; we will have to have a two cycle delay as we can see over here, between the producer and the consumer instruction, this delay is clearly not acceptable to us. So, this slice in the face of pretty much everything that we have been talking about. So, what this basically says that, look in these two cycles; we either insert bubbles, which means we do not do anything or we execute some other instruction.

So, that possibilities always there; because it is an out of order pipeline, we can bring some other instructions, instruction X and instruction Y and they can be executed in this stage. So, in a certain sense we can still have a reasonably high instruction throughput; but if you are unfortunate enough to have a code with low ILP, which means that we have a code lot of code with these sequential dependencies, the dependencies are like a chain, clearly we need to insert a two cycle bubble.

And there was no need to do it in an in order pipeline; but in an out of order pipeline, we have to do it, which is not something that makes us happy. Let us assume that somehow we are able to. So, one thing is that, along with writing to the register; we cannot broadcast the tag, which incidentally appears to us to be to be an intuitive solution intuitive way of doing it, but this is clearly not allowed, this is not acceptable. Another approach could be that, we broadcast the tag along with execute.

If you broadcast the tag along with execute, what will happen; is that we can reduce this delay a little bit. So, we are of course assuming that we broadcast at the beginning of the cycle and then it immediately catches it, it wakes up and selects; that is what you are assuming all happens in the same cycle. So, even if this does happen, there is still a one cycle bubble. So, this is the again not acceptable to us.

(Refer Slide Time: 50:34)



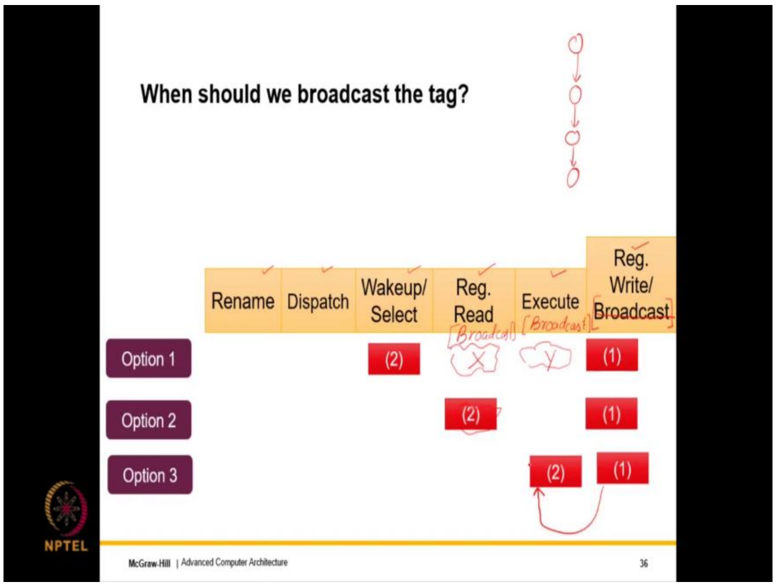
When should we broadcast the tag?

	Rename	Dispatch	Wakeup/ Select	Reg. Read [Broadcast]	Execute [Broadcast]	Reg. Write/ Broadcast
Option 1			(2)			(1)
Option 2				(2)		(1)
Option 3			(2)	(1)		

McGraw-Hill | Advanced Computer Architecture 36

What is possibly acceptable to us? So, it just showing the way that instructions move; but what is possibly acceptable to us is option 3. Well when instruction 1 is in the register read stage; then in the register read stage itself, we broadcast the tag. So, this is clearly an early broadcast. So, we will discuss the implications of an early broadcast. But if we are able to do it and the instruction 2 catches the broadcast in the cycle itself; when we broadcast in a same cycle it catches it, then it can get, then it can wake up and select.

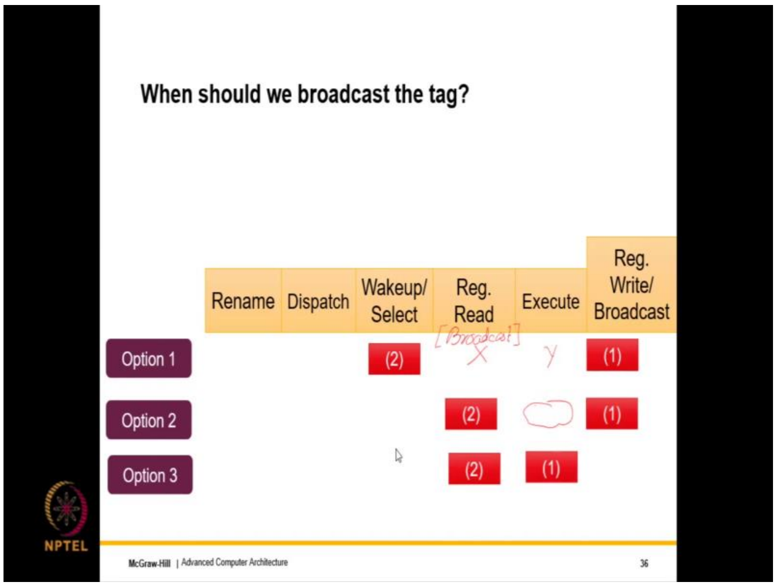
(Refer Slide Time: 51:25)



And in this case, the execution will be back to back; because then when 1 is executing, 2 is reading the registers, which of course it will not get the right value. But then it will get it from the bypass path, which is in a next cycle; because 1 would have computed its result. So, it can bypass the results to instruction 2. So, it is important that we see this animation once again and I will clear of the ink and it is important that we see this once again to appreciate what exactly we are trying to say.

So, what we are trying to say over here is that, look if you broadcast along with a register write, it is not a good thing; the reason that it is not a good thing is because, we will have to introduce the distance between instruction 1 and 2 has to be two instructions.

(Refer Slide Time: 52:20)



When should we broadcast the tag?

	Rename	Dispatch	Wakeup/ Select	Reg. Read	Execute	Reg. Write/ Broadcast
Option 1			(2)			(1)
Option 2				(2)		(1)
Option 3				(2)	(1)	

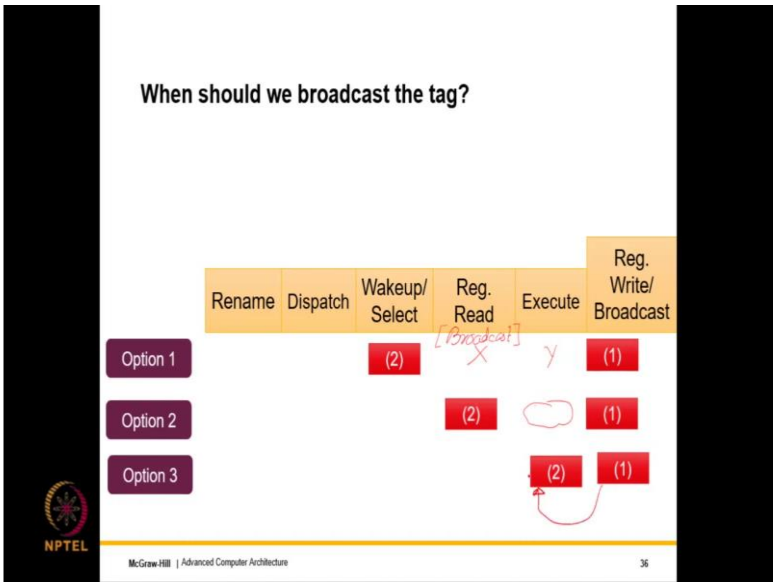
McGraw-Hill | Advanced Computer Architecture 36

Since it is an out of order pipeline, these two instructions need not be stalls, they can be other instructions; nevertheless, we will not have back to back execution, which does matter in a code with very low ILP, we will be have a chain of dependencies. Even if I were to broadcast one cycle earlier that is still not work, because we have this.

Hence what I need to do is that, when instruction 1 goes to the register read stage, just after getting selected; it kind of speculatively broadcast the tag, I am using the word speculative. Well, so in this case it makes an assumption that execution will take one cycle. So, we will come to that; but, so we will come to the implications of this.

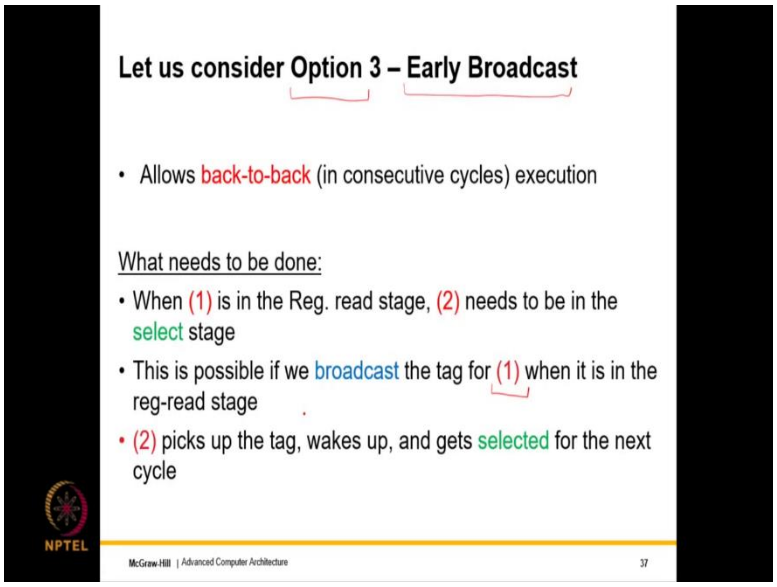
But what we are essentially trying to say is that, along with the register read it broadcast the tag; at the same cycle instruction 2 wakes up, then both proceed. So, naturally instruction 2 will not get the right value from the register, so that is ok.

(Refer Slide Time: 53:23)


When should we broadcast the tag?

We can use the forwarding mechanism called bypassing here. So, we can bypass the results from instruction 1 to 2 and this is work in the exactly the same fashion as the in order pipeline.

(Refer Slide Time: 53:35)


Let us consider Option 3 – Early Broadcast

- Allows back-to-back (in consecutive cycles) execution

What needs to be done:

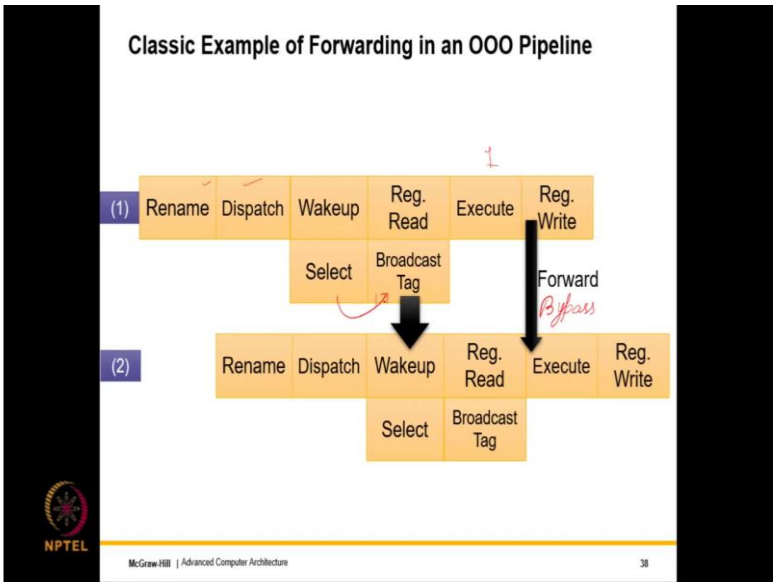
- When (1) is in the Reg. read stage, (2) needs to be in the select stage
- This is possible if we broadcast the tag for (1) when it is in the reg-read stage
- (2) picks up the tag, wakes up, and gets selected for the next cycle

McGraw-Hill | Advanced Computer Architecture 37

So, what is the option that actually worked; it is early broadcast. This allows back to back execution, which means execution of dependent instructions in consecutive cycles. So, it does allow this to a certain extent. Well, we will see; so when we discuss later ideas, we will see why to a certain extent.

So, what needs to be done? Well, when instruction 1 is in the register read stage, 2 needs to be in the select stage. This is possible if you broadcast the tag for 1 when it is in the register read stage; 2 will simply pick up the tag, it is going to wake up, and get selected for the next cycle. So, that is all that it is going to do; that it is going to wake up, and simply get selected in a next cycle, which is anyway what we want.

(Refer Slide Time: 54:34)

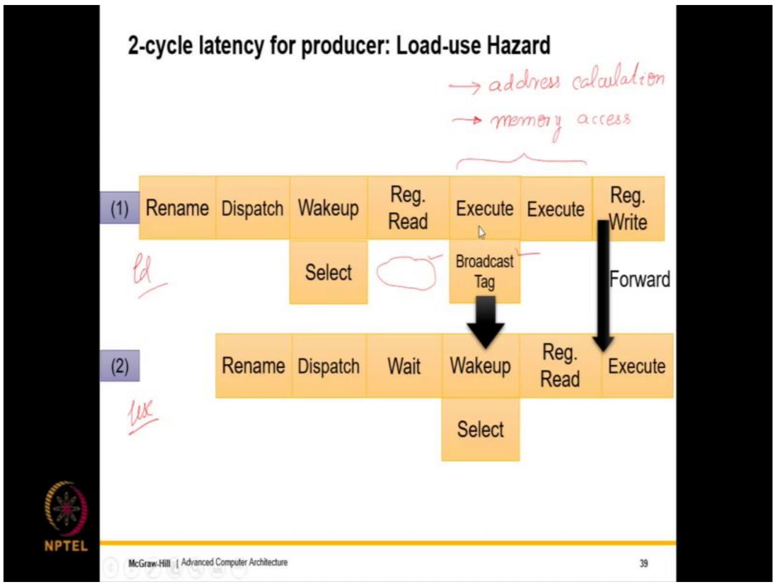


So, the classic example of forwarding in an out of order pipeline would look like this. So, by the way our out of order pipeline now has change. So, I am not showing the fetch and decode stages, so they are of course there. So, I am showing rename, dispatch, and then wake up select.

So, wakeup select is happening in the same cycle and then we have register read and broadcast. So, we are of course assuming here, it is kind of a bold assumption; but this needs to be made at least in this simplistic setting that, we broadcast at the beginning of a cycle and roughly in the same cycle. So, not roughly, in the same cycle the consumer instruction wakes up and of course if no other instruction is being selected, it get selected.

So, the select is again the, again a best case selects. So, it is possibly does not get selected in this cycle and a later cycle; but we are just assuming a tight case over here. So, after this, the consumer proceeds along the same path and then of course when it needs the value, there is a forwarding that we can see over here; I should also write bypass, the word bypassing is more appropriate over here, the value is bypassed and the execution proceeds.

(Refer Slide Time: 55:59)



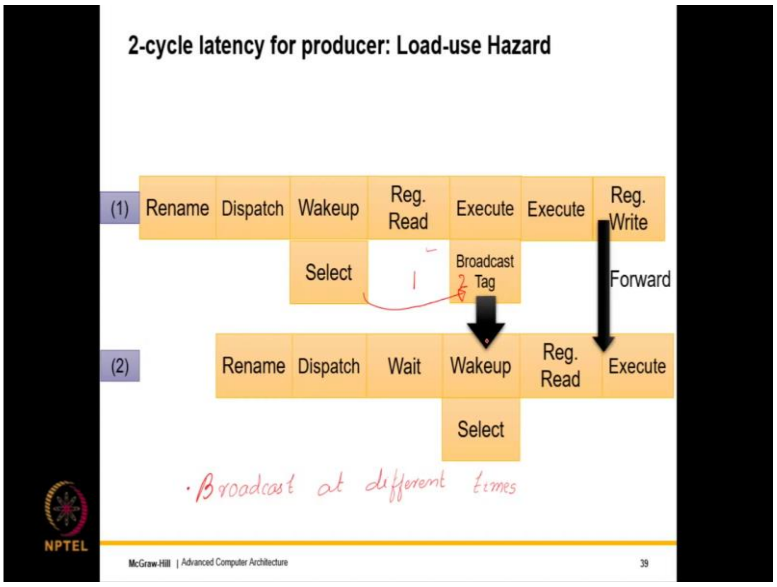
So, what do we get to see here? Well, what we get to see here is assume that the duration of execution instead of one cycle, it is two cycles. When can this be the case? Well, this can be the case for a load instruction; in the load instruction what happens is that, we actually spend two cycles executing it. Why? The first cycle is address calculation, that is the first cycle which is the address calculation. The second is memory access.

So, that is the reason what I have done is that, I have pretty much created a two cycle execute over here. So, in this case of course back to back execution is not possible; it was not possible in order pipeline as well, so there was a one cycle delay or a one cycle stall. So, in this case the advantage of an out of order pipeline is that, we do the same; but instead of actually stalling, you can execute some other instruction in the middle, in this middle cycle right we can execute some other instruction.

But that surrender what we do is, we in this case we do not broadcast immediately, that is important, that needs to be kept in mind; we do not broadcast immediately after getting selected. So, instruction 1 is the load and instruction 2 is the use. So, what we do is that, we broadcast one cycle later; because in this case it is not a one cycle execution, so we broadcast one cycle later. And so, we instead of broadcasting the tag here, we broadcasted over here.

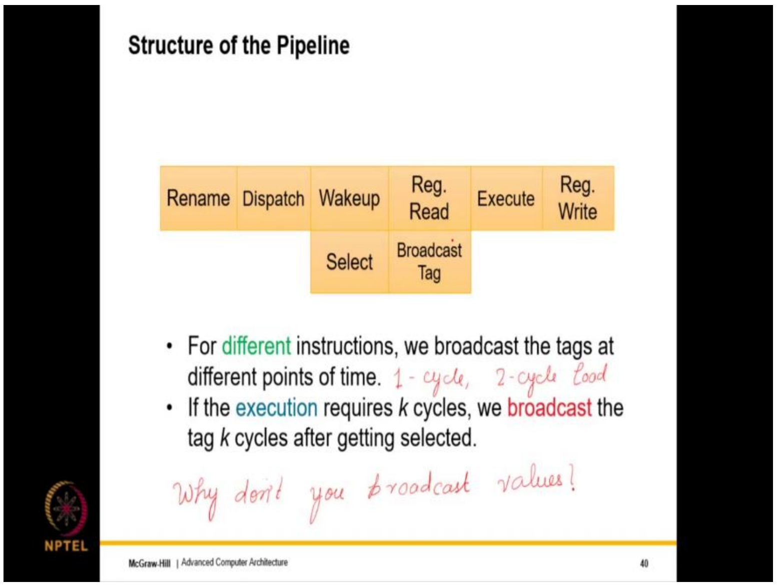
After this we have wake up select, we have read and execute; if bypassing is required, then we forward our bypass in this cycle. And as you can see it happens seamlessly; the same way it would happen in a regular in order pipeline. So, what is the key learning from this figure? Well, I am erasing the ink and may be give you five seconds to meditate on this figure before I tell you the answer. So, your meditation time starts now ok; 3, 4, 5.

(Refer Slide Time: 58:50)


2-cycle latency for producer: Load-use Hazard

So, the important point is that, for different kinds of instructions, we just broadcast a different times. So, what we do is, we just broadcast a different times; we do not broadcast with the same time, broadcast a different times. So, for an add instruction, we broadcast in the next cycle in a registered read cycle and for a load instruction, we broadcast one cycle later, which is in the execute, in the first cycle of execute we do that.

(Refer Slide Time: 59:35)



Structure of the Pipeline

Rename	Dispatch	Wakeup	Reg. Read	Execute	Reg. Write
		Select	Broadcast Tag		

- For different instructions, we broadcast the tags at different points of time. 1-cycle, 2-cycle load
- If the execution requires *k* cycles, we broadcast the tag *k* cycles after getting selected.

Why don't you broadcast values!

McGraw-Hill | Advanced Computer Architecture 40

See if I were to kind of summarize everything that I said up till this point. So, what would it be? Well, what would what it would be is that, well we have rename, dispatch, wake up select, read broadcast, execute, assuming one cycle execute of course. So, for different instructions, we broadcast the tag at different points of time. So, we clearly saw that for a regular one cycle add and for a load which in a by definition takes multiple cycles, because of address calculation, we broadcast at different times.

If I were to generalize this I can say that, if the execution requires k cycles; we broadcast the tag k cycles after getting selected. Is this true? Well, all that we need to do is, we need to go back. So, in this case the execution requires one cycle. So, we get, we select in this cycle, one cycle later we broadcast. In this case, the execution requires two cycles. So, after selection we wake; after two cycles, we broadcast cycle 1, cycle 2, we broadcast.

So, this is the general idea of broadcasting and so, if you would recall, I had asked a question, several slides earlier; what I had said is, why do not you actually broadcast values, but you broadcast tags. So, the answer should be clear at this point, let me just write this down.

So, why do not you broadcast values, why do we broadcast tags? Well, the reason we broadcast tags instead of values is, because we do an early broadcast and in an early broadcast, we do not have the value with us, we cannot broadcast it. So, we need to broadcast the tag.

So, this pretty much completes our discussion of the basic mechanism. So, we will now look at a more complicated; we will look at a complicated version of broadcast, slightly more and we look at some corner cases.