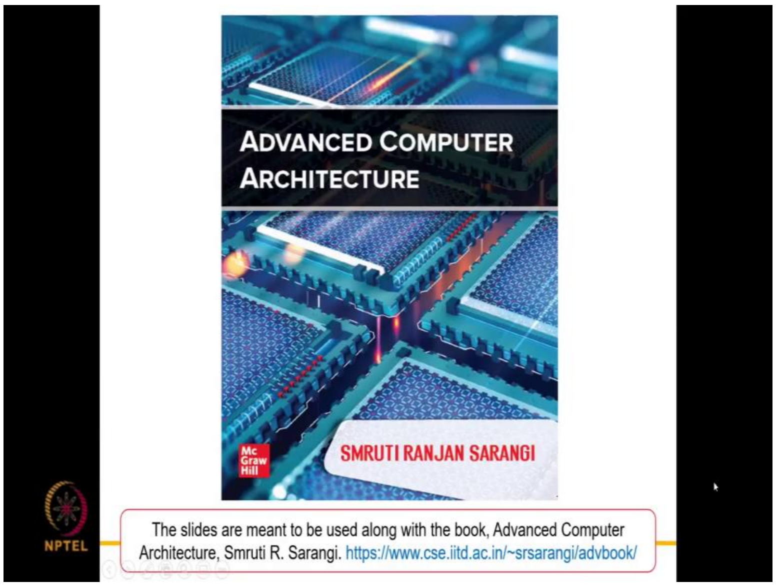
Advanced Computer Architecture

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

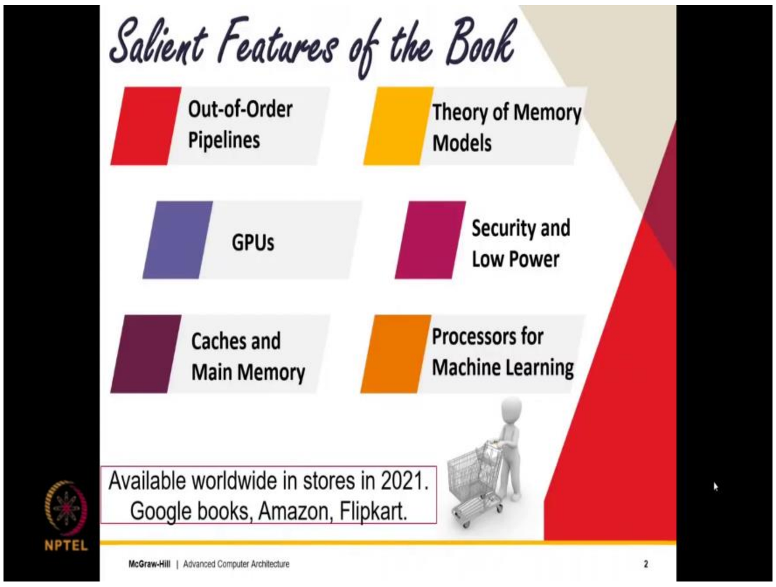
Lecture - 11
The Issue, Execute, and Commit Stages Part- IV

(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)



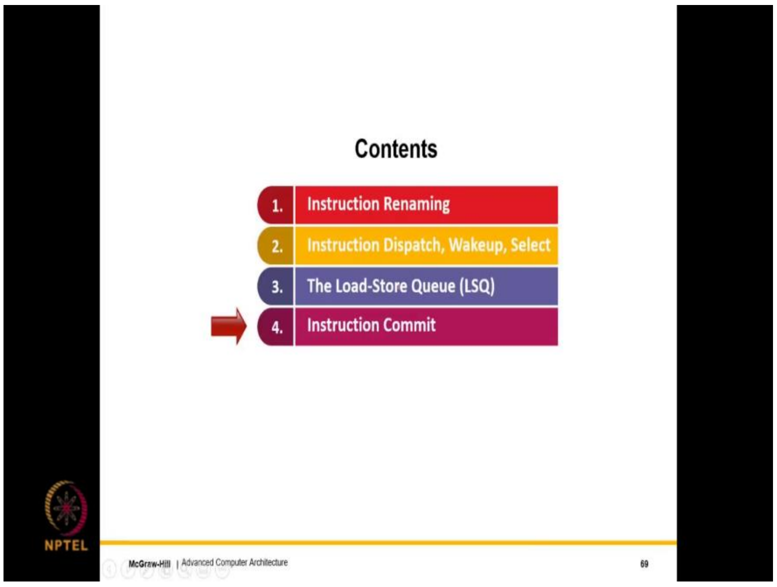
Salient Features of the Book

- 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: 00:39)



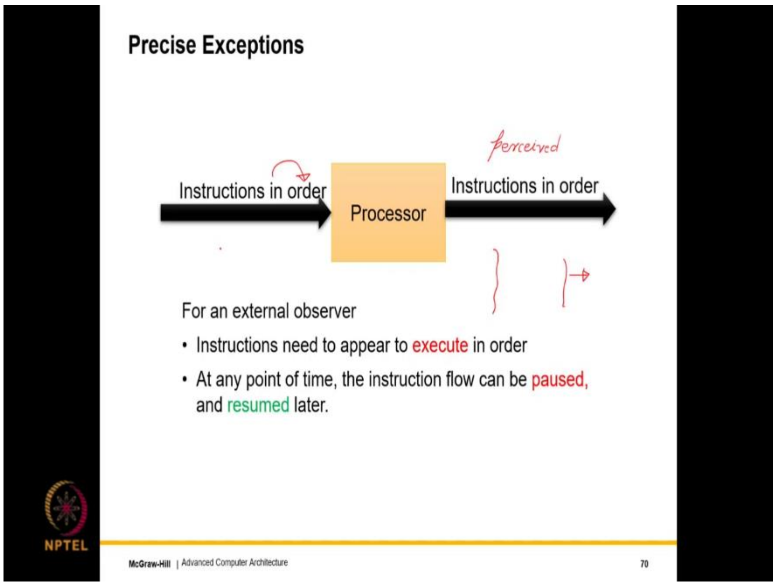
Contents

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

McGraw-Hill | Advanced Computer Architecture 69

Let us now discuss the last phase of instruction processing, which is the instruction commit phase. So, we typically, use this phase to release resources, remove instructions from the pipeline and also, ensure precise exceptions.

(Refer Slide Time: 00:57)



Precise Exceptions

For an external observer

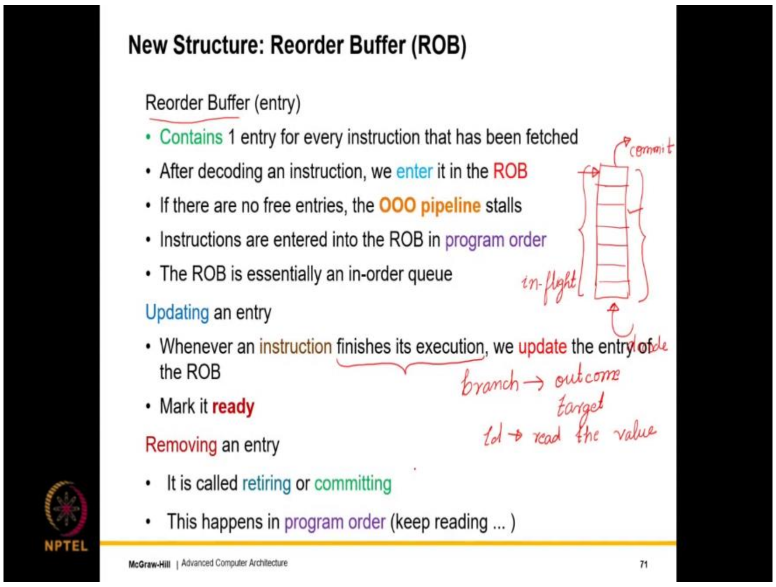
- Instructions need to appear to execute in order
- At any point of time, the instruction flow can be paused, and resumed later.

McGraw-Hill | Advanced Computer Architecture 70

So, what have we have been discussing up till now, what we have been discussing up till now is out of order inside, in order outside say instructions entered in order, mind you they are perceived to be executing or leaving the processor in programme order. So, it is important to note that perceived in a sense that is what an observer perceives, even though the execution is actually happening out of order.

So, for an external observer at any point of time, the instruction flow can be paused. This is a corollary of in order execution, that if instruction execution is in programme order, well we can pause it at some point and resume it at the exactly the same point later on. So, that is the most important aspect of out of order execution.

(Refer Slide Time: 01:57)



New Structure: Reorder Buffer (ROB)

Reorder Buffer (entry)

- Contains 1 entry for every instruction that has been fetched
- After decoding an instruction, we enter it in the ROB
- If there are no free entries, the OOO pipeline stalls
- Instructions are entered into the ROB in program order
- The ROB is essentially an in-order queue

Updating an entry

- Whenever an instruction finishes its execution, we update the entry of the ROB
- Mark it **ready**

Removing an entry

- It is called retiring or committing
- This happens in program order (keep reading ...)

McGraw-Hill | Advanced Computer Architecture 71

So, what we will do? The core structure that we will add over here is the Re-Order Buffer, it is called the ROB; a short form of a reorder buffer is ROB. It contains one entry for every instruction that has been fetched. So, similar to other structures, it is a queue; similar to the instruction window. For example, it is a queue. So, after decoding an instruction, we enter it in the ROB.

So, as soon as we have read the instruction and we have decoded it, it is entered into the ROB and needless to say all queues that we have in our in an architectural setup, they are implemented as circular queues. Otherwise, there is no other way to implement it with using a finite sized array, so they are implemented using circular queues.

So, when we decode an instruction, we enter it into the ROB, that is the important point over here. If there are no free entries right say the ROB is full which means that instructions are not able to leave the pipeline for some reason, we will see what are the reasons later. Then, the ROB will essentially become the bottleneck. The decode process will stall.

This means that the fetch process will stall and the entire pipeline will stall. So, pretty much the ROB maintains a record of all the instructions that are in flight which means within the pipeline. Let us call this in flight instructions and it is pretty much an in order queue, where when we decode we enter and they leave when what we call instruction commit or instruction retirement.

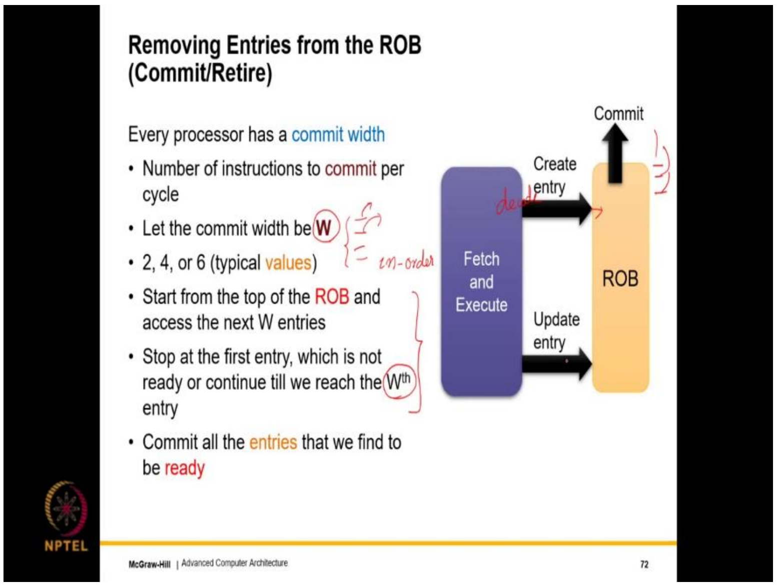
So, when do we update an entry? Well, so, the instruction remains there. After we decode, the instruction remains in the ROB. As soon as an instruction finishes its execution, which means it computes its value and it is done, we update the entry of the ROB, the market to be ready. So, there is a ready bit in the ROB which is market is ready. So, what do we mean by finishing its execution?

Well, an ALU instruction finishes its execution, when it is computed the value of the output register; a branch finishes its execution, when it has computed the outcome and the target; load instruction finishes its execution, when it has read the value from the memory. So, let us call it read the value. So, that is what we mean by finishing.

So, we I am deliberately mute about the store because we will see that the store is a special case and as we have argued earlier, a store is only sent to the memory system, when it is the earliest instruction in the pipeline which would basically mean that it is at the top of the ROB. So, there is no instruction earlier than it. That is when we actually send it to the memory system.

So, hang in there, we will discuss the case of store, over and over again in the next few slides. Removing an entry; while removing an entry is easy, we just remove it from the top. It is called retiring or committing and of course, instructions are removed in program order. It is in a loaded statement; keep reading, keep watching.

(Refer Slide Time: 06:04)



Removing Entries from the ROB (Commit/Retire)

Every processor has a commit width

- Number of instructions to commit per cycle
- Let the commit width be W
- 2, 4, or 6 (typical values)
- Start from the top of the ROB and access the next W entries
- Stop at the first entry, which is not ready or continue till we reach the W^{th} entry
- Commit all the entries that we find to be ready

Commit
Create entry
Fetch and Execute
ROB
Update entry
in-order
decode

So, what do we exactly mean? So, what we exactly mean is that our fetch and execute stages, so the moment we decode, we create an entry, add it to the ROB. After finishing execution, we update the entry. So, essentially, we set a ready bit. We make it known that the operation associated with the entry has completed. So, the ROB is updated, not the ROB is not updated; the entry in the ROB is updated.

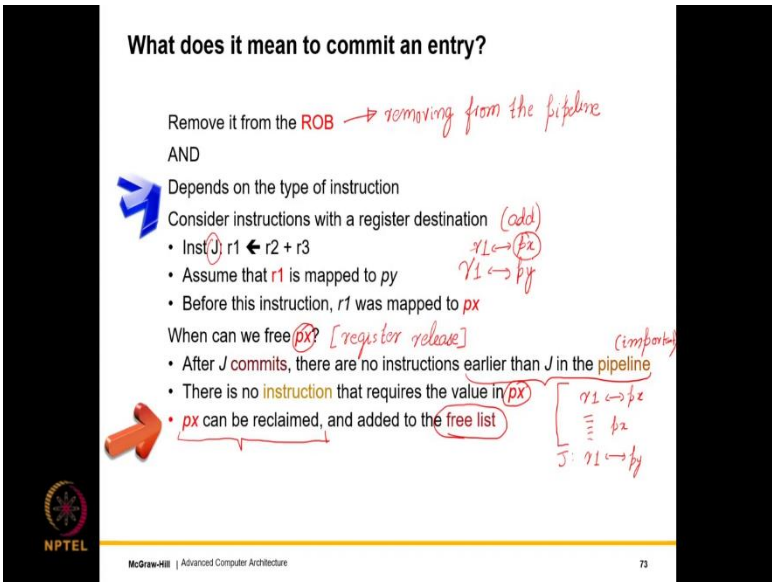
Furthermore, every cycle, we inspect the earliest entries of the ROB. So, maybe inspect the earliest. If that is ready, if it is done this job, we just remove it. Then, we inspect the next; if that is done, we remove it, then the next. So, how many do we inspect at most? We inspect W entries at most, where W is called the commit width. It is typically 2, 4 or 6. So, let us assume it's 4. So, what we do is that we inspect the earliest, if that is ready, well we just remove. If it is not ready, we do not do anything.

We stop there. Then, we inspect the next one, the second earliest, if that is ready, well we remove it; if not we stop there, so on and so forth until we reach the limit which is the commit width. So, the commit width essentially determines the rate of instructions exiting the pipeline.

So, we typically set the fetch width equal to the commit width because if you look at the pipeline as sort of an instruction processor, on an average, if we are committing 4 instructions per cycle, we should also be fetching that much. Otherwise, it will lead to rate mismatch. So, now, coming to the commit, well the commit process is entirely in order which basically means that we keep inspecting the entries earliest first and we stop at the first entry which is not ready.

So, if it is not ready, we just stop there. So, it is never the case that we commit an entry, but an entry earlier then it is still there in the ROB. So, so that never happens and until we reach the Wth entry, so we have a limit on how many instructions, we commit per cycle alright. So, you pretty much commit all the earliest entries that we find to be ready in programme order. So, we never skip any entry.

(Refer Slide Time: 08:37)



What does it mean to commit an entry?

Remove it from the ROB $\rightarrow$ removing from the pipeline

AND

Depends on the type of instruction

Consider instructions with a register destination (add)

- Inst J: r1 $\leftarrow$ r2 + r3 $\quad r1 \leftrightarrow px$
- Assume that r1 is mapped to *py* $\quad r1 \leftrightarrow py$
- Before this instruction, *r1* was mapped to *px*

When can we free px? [register release]

- After *J* commits, there are no instructions earlier than *J* in the pipeline (important)
- There is no instruction that requires the value in px
- *px* can be reclaimed, and added to the free list

$r1 \leftrightarrow px$

$\vdots \quad px$

$J: r1 \leftrightarrow py$

McGraw-Hill | Advanced Computer Architecture 73

What is the advantage of doing this? Well, let us see. What does it mean to commit an entry? Well, one is removing it from the ROB. So, this pretty much removes it, removing it from the ROB is tantamount to removing it from the pipeline. So, that is what this amounts to. Furthermore, removing an entry also might involve additional processing, might entail additional processing.

So, it depends on the type of the instruction. For example, if we have an instruction with a register destination like an add instruction. Let us see. So, here we will answer one of the key important questions of when we release the physical register, put it back into the free list. We will answer it now.

So, consider instruction J. Let it be r1 = r2 + r3. Assume that r1 → py. So, assume that at this point r1 → py. Before this instruction, so before instruction J came and it was renamed let us assume that the mapping just before r1 to py, the mapping was r1 → px.

So, what happens? Well, r1 → px, then instruction J came and it changed px to py because it got py from the free list. Now, the question is when can we free px right. This is a question of register release. So, after J commits, after J is removed from the ROB, from the pipeline, there are no instructions earlier than J in the pipeline.

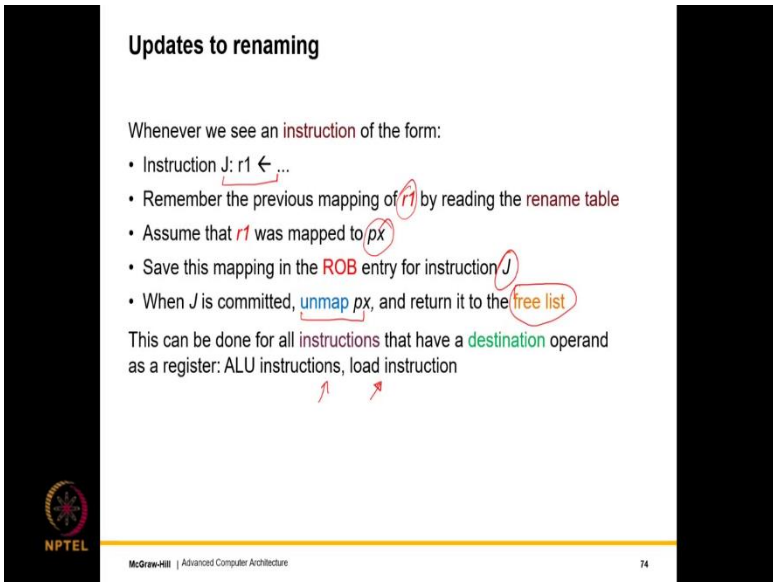
So, this should be rather clear that the reorder buffer ensures that once J is removed, there are no other instructions that are earlier than J that precede J which are still in the pipeline. Important insight; let us keep this in mind. So, this is very important, let us keep it in mind. This means, that there is no instruction that requires the value in px. So, this is correct because r 1 → px, then the couple of instructions maybe came, a few of them might be using px.

Then, r1 got mapped to py and this was a direct consequence of renaming instruction J. So, once J is committed, it implies that all the instructions before it that preceded they are also committed, which means that all the instructions that possibly use px, all of them are committed and all of them have left the pipeline.

Let me repeat the last statement rather important. All of them have committed and all of them have left the pipeline, this means that p x can be reclaimed. So, this means that at this point, px can be released and added to the free list. This is undoubtedly a very important insight for us. So, this essentially says that whenever I am renaming an instruction that has a register based destination operand, I need to remember the previous mapping.

So, at this point, the previous mapping is px. So, when instruction J is committing we cannot release py that is important because consumers of py will still be there in the pipeline. But we can definitely release px because px is uses over; all the instructions that were using px, all of them have been committed, we know that for sure, direct corollary of in order commit. So, what is essentially left is py and its uses. So, px can be released.

(Refer Slide Time: 12:55)



Updates to renaming

Whenever we see an instruction of the form:

- Instruction J: r1 ← ...
- Remember the previous mapping of r1 by reading the rename table
- Assume that *r1* was mapped to *px*
- Save this mapping in the ROB entry for instruction *J*
- When *J* is committed, unmap *px*, and return it to the free list

This can be done for all instructions that have a destination operand as a register: ALU instructions, load instruction

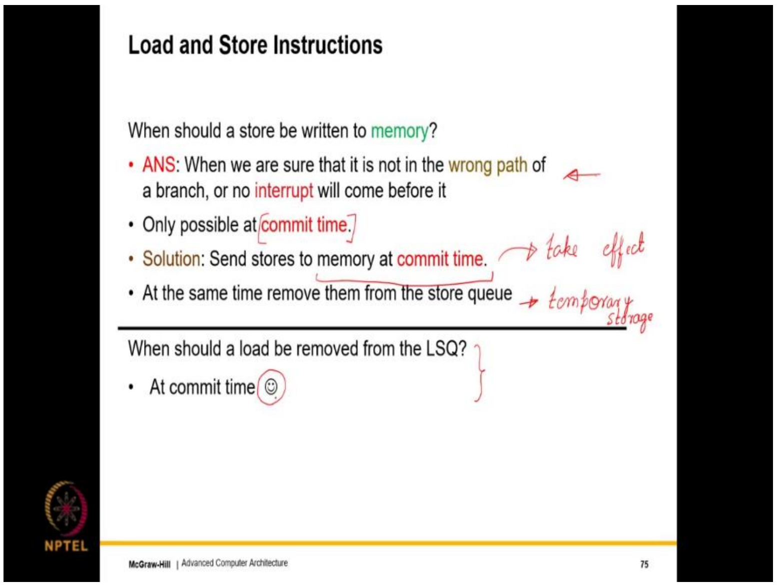
McGraw-Hill | Advanced Computer Architecture 74

So, what is the update we need to do the renaming phase? Well, the moment we see an instruction of this type, where r1 let us say in this case is r1. It can be any rx that is being written to. We remember the previous mapping of r1 by reading the rename table. So, this does complicate the rename table by the way adds additional ports. So, we assume that r1 → px.

So, we save this mapping in ROB entry for instruction J. When J is committed, we unmap px which means we return it to the free list such that the register px can now be given to other instructions, can now be assigned to other instructions. This can be done for all instructions that have a destination operand, that is a register.

So, these will be ALU instructions, load instruction and so on, where the destination operand is the register. So, this should at this point, we easily understood that when we actually release registers will we release when the instruction that overwrites the register when that commits.

(Refer Slide Time: 14:12)



Load and Store Instructions

When should a store be written to memory?

- ANS: When we are sure that it is not in the wrong path of a branch, or no interrupt will come before it
- Only possible at commit time.
- Solution: Send stores to memory at commit time.
- At the same time remove them from the store queue

When should a load be removed from the LSQ?

- At commit time ☺

take effect
temporary storage
NPTEL
McGraw-Hill | Advanced Computer Architecture
75

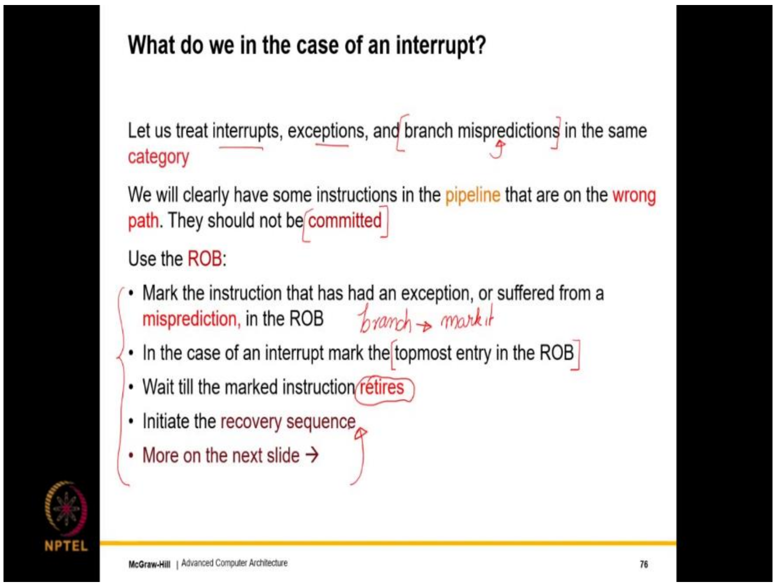
Now, let us come to loads and stores. So, when should a store be written to memory? Well, we have discussed this in the past. Just a quick reminder. When we are sure that it is not on the wrong path of a branch and no interrupt will come before it. So, as long as there is an instruction before it, earlier to it, that precedes it, that instruction might suffer from a fault and we do not want to take the risk.

So, only when we are committing a store which means when it becomes the earliest and it is being removed from the ROB that is when we send the stores to the memory. At commit time, we send the stores to the memory and that is when they actually take effect. So, this is clearly happening in programme order. So, there is no doubt about it and that is when it takes effect.

At the same time, we remove the entry from the store queue as well. It is not required. Because the store queue all that it was doing is that it was essentially providing temporary storage, which by which in itself is an important service. No doubt and loads were supposed to check this temporary storage first, before actually going to the cache. So, this solves the question about stores that when we are about to commit stores, we actually send them to the memory system.

Now, when should a load be removed from the LSQ? Well, this answer is also obvious we have touched upon this. So, the answer for this is that again at commit time, we remove the load from the LSQ and from the ROB as well. So, we just remove it from the pipeline.

(Refer Slide Time: 16:02)



What do we in the case of an interrupt?

Let us treat interrupts, exceptions, and branch mispredictions in the same category

We will clearly have some instructions in the pipeline that are on the wrong path. They should not be committed

Use the ROB:

- Mark the instruction that has had an exception, or suffered from a misprediction, in the ROB branch → mark it
- In the case of an interrupt mark the topmost entry in the ROB
- Wait till the marked instruction retires
- Initiate the recovery sequence
- More on the next slide →

McGraw-Hill | Advanced Computer Architecture 76

Next question, what do we do in the case of an interrupt? So, let us treat interrupts exceptions and mind you, this is very important and this is a question that many people would ask when we talked about branch predictors about recovery. But this question is being answered now that interrupts exception and branch mispredictions, let them be placed in the same category as exceptional events.

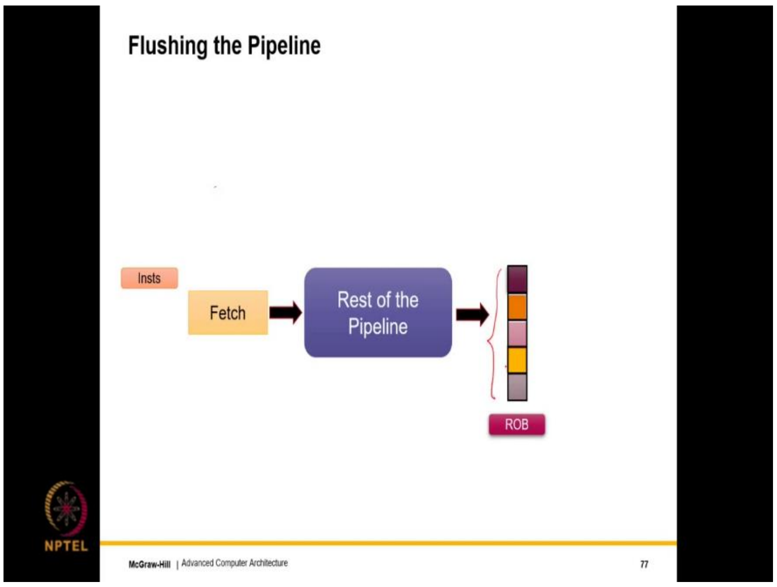
So, whenever we miss predict a branch, this the let this be treated as an exceptional event. So, whenever we receive any of these notifications, we will clearly have some instructions in the pipeline that are on the wrong path and if they are on the wrong path, well they should not be committed. So, they should be removed.

So, let us look at interrupts and exceptions are easy to handle, we just marked one of the instructions, the same way we did in an in order pipeline and we just wait for that instruction to reach the head of the ROB. So, this means that all the instructions, after the interrupt after the marked instruction, they have completed and nothing is completed before that. Because we do not make permanent architectural state changes before commit time. Now, the using the ROB, well, let us look at using the ROB primarily for branch mispredictions; the others are simpler to handle.

So, whenever an instruction has an exception or supple or suffers from a misprediction, we mark it. So, for example, if a certain branch is mispredicted all that we do is we just mark it, we do not do anything else. So, we can either mispredict the outcome or the branch, but all that we do is we just mark it. So, we just mark the instruction which means that we remember that this instruction was mispredicted and we do not do anything else, we wait for the instruction to reach the head of the ROB which means we wait for it to become the earliest instruction.

In the case of an interrupt, well we can mark any instruction that we choose; but if we want the interrupt to be processed as soon as possible, we mark the topmost entry in the ROB which is also the earliest entry. So, that is what we mark. We wait till the marked instruction retires, so retire and commit is something we will use interchangeably. So, we wait till the marked instruction reaches the head of the ROB and is ready to commit, then we initiate what is called a recovery sequence which is described in the next slide.

(Refer Slide Time: 19:03)

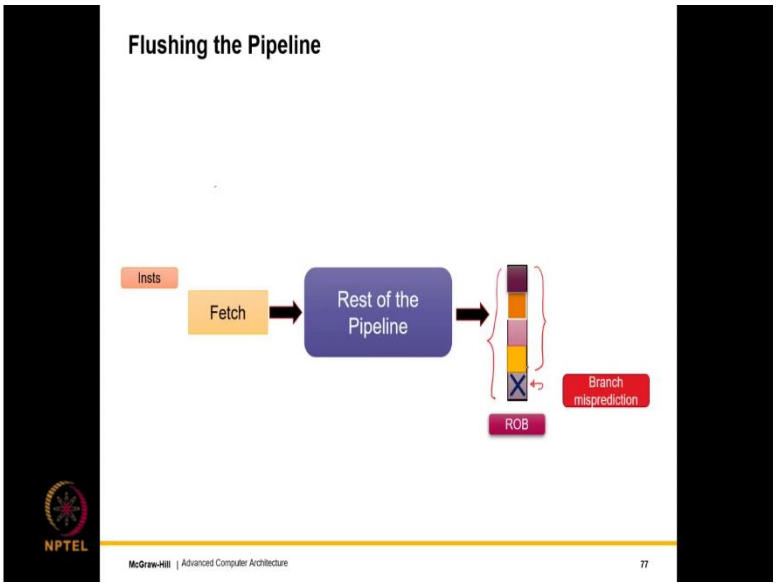


Flushing the Pipeline

Let us now look at a visualisation of the pipeline. So, in the pipeline, we will not show much. What is shown here is the fetch unit, the fetch unit fetches instructions, they pass through the rest of the pipeline and finally, the reorder buffer maintains a record of all the instructions that are in flight. So, this animation will nicely show you all the processes that are involved.

So, initially what we do is we fetch an instruction and a decode time, it is added to the reorder buffer. So, if you fetch 5 instructions as we are doing right here, we are getting added to the reorder buffer and so, we have now fetched and decoded 5 instructions and you can see that all these 5 instructions are present in the reorder buffer over here.

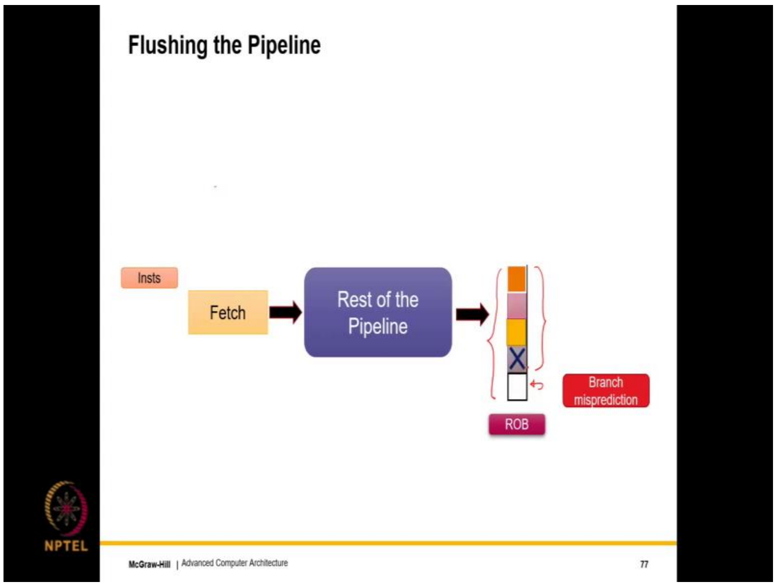
(Refer Slide Time: 20:15)



Flushing the Pipeline

Now, let us assume that the instruction that was fetched that the latest instruction that was fetched that suffers from a branch misprediction. In this case, the 4 earlier instructions which are these instructions, these instructions are safe. They are on the correct path. Hence, they should be committed; whereas, this instruction as well as instructions before it, which means the instruction that mispredicted and the instructions that were fetched after the mispredicted instruction, these instructions should not be committed and they in fact, should be removed from the pipeline.

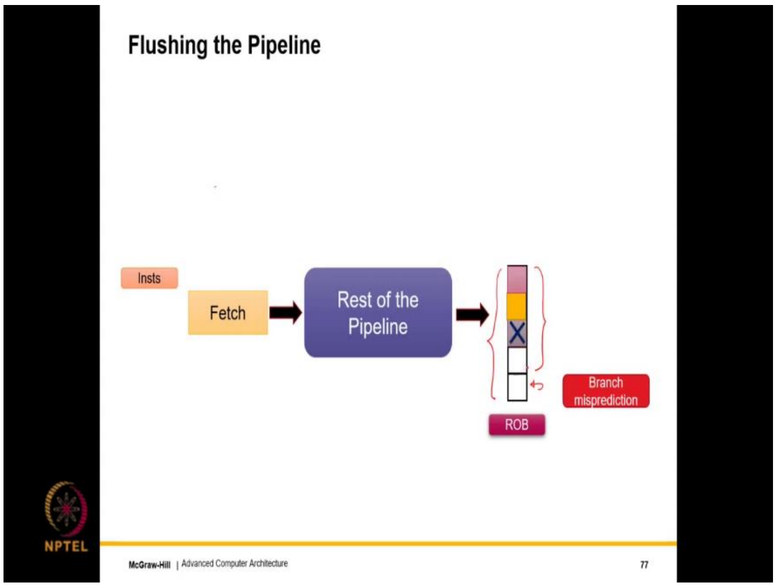
(Refer Slide Time: 21:03)



Flushing the Pipeline

So, what do we do? Well, we just commit the earlier instructions.

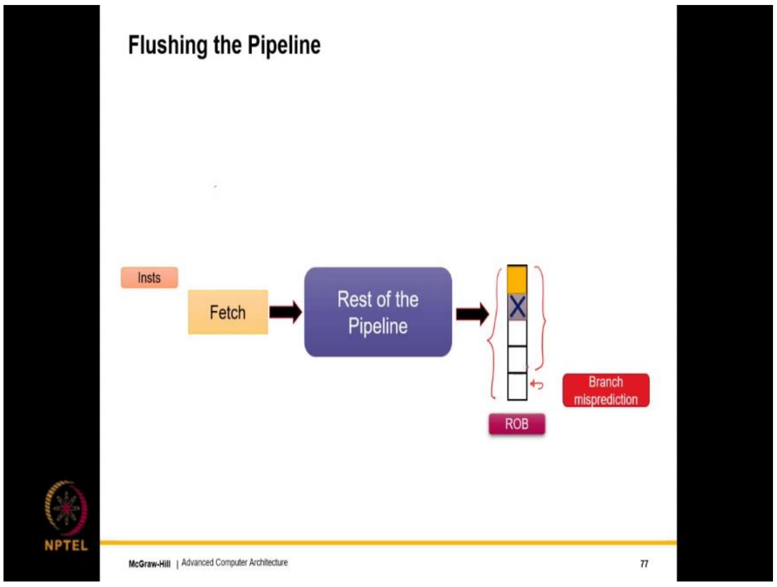
(Refer Slide Time: 21:05)



Flushing the Pipeline

McGraw-Hill | Advanced Computer Architecture 77

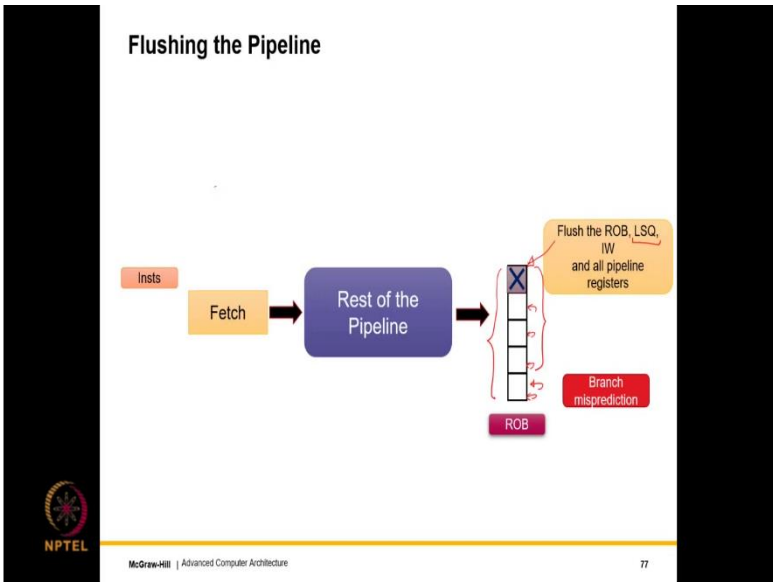
(Refer Slide Time: 21:06)



Flushing the Pipeline

McGraw-Hill | Advanced Computer Architecture 77

(Refer Slide Time: 21:08)

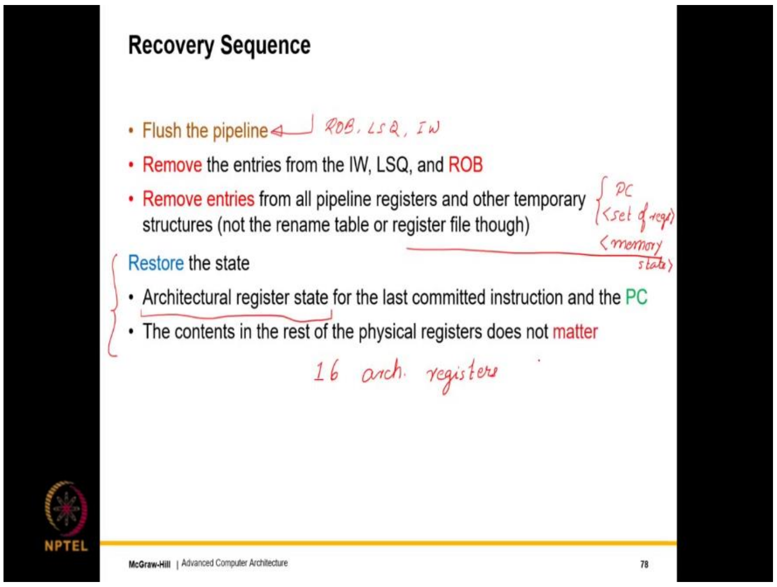

Flushing the Pipeline

McGraw-Hill | Advanced Computer Architecture 77

Keep committing, keep committing, keep committing and wait for the mispredicted instruction to reach the head of the ROB, the top of the ROB. Once it reaches the top of the ROB, well something needs to be done. Also, please bear in mind that even though we are not showing, other instructions might have been decoded and might have entered the ROB, after the mispredicted instructions. So, they will be at these positions alright.

So, it does not matter. So, when the mispredicted instruction or let us say an instruction that suffers from an exception or an instruction which we mark because of an interrupt, the moment it reaches the head of the ROB, which means that it becomes the earliest instruction in the ROB, what we do is that we flush the entire buffer. So, we flush the not only the ROB, but the entire pipeline. This includes the load store queue, the instruction window as well along with all the internal pipeline registers and buffers, we flush everything.

(Refer Slide Time: 22:20)


Recovery Sequence

- Flush the pipeline
  - Remove the entries from the IW, LSQ, and ROB
  - Remove entries from all pipeline registers and other temporary structures (not the rename table or register file though)
- Restore the state
  - Architectural register state for the last committed instruction and the PC
  - The contents in the rest of the physical registers does not matter

So, what is the recovery sequence? The recovery sequence is that we flush the pipeline. What exactly would this mean? This would mean flushing the reorder buffer. This would mean flushing the load store queue, the instruction window and all other internal structures, they are all cleaned up. We remove entries from the instruction window LSQ, ROB. All internal pipeline registers etcetera also we remove the entries and then, the most important part is that we restore the state.

So, before this, I would like to make an important point. So, what exactly was our architectural state? It was the programme counter, the set of architectural registers. So, they here only exist in principle, then we had the memory state. So, since we sent stores to the memory system only when they commit, the memory state is correct. The set of architectural registers, we need to ensure that it is correct; but it has to be correct.

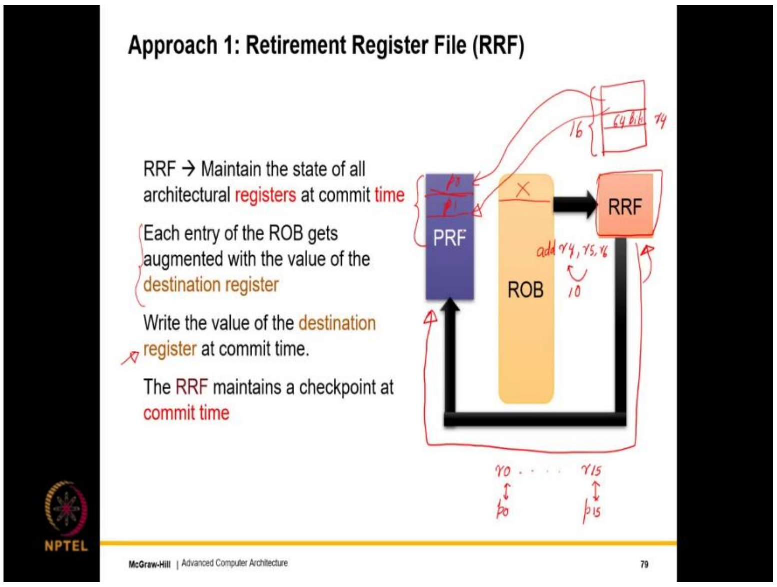
We will see why. The reason is that whatever was the current mapping when the branch was fetched, those physical registers have still not been touched right. To a certain extent, those physical registers still maintain their value, they have not been touched. So, the architectural state is still there, it is still intact and it is still there and the programme counter of course, we can add it into every ROB entry. So, we just need to restart from the programme counter.

So, in a certain sense, the architectural state is still intact. We just need to restore the state, in the sense recover the state that existed just before the branch was fetched. We need to go back to that point and start re-executing everything. Now, that we know the correct target and the outcome of the branch, at least this branch will not suffer from a misprediction.

So, what do we precisely need to do? Well, we need to roll back the architectural register state or recover the architectural register state of the last committed instruction, which is the instruction that was committed just before the branch. The contents in the rest of the physical registers do not matter. In this case, we are only concerned with the contents of the 16 architectural registers.

So, we are only concerned with their contents and so, we somehow need a way to kind of recover and rewind our state to the state that existed just before renaming; you can call renaming, fetching, decoding, does not matter. But just before processing the branch instruction in programme order whatever was the state, we need to roll back to that point.

(Refer Slide Time: 25:39)



Approach 1: Retirement Register File (RRF)

- RRF → Maintain the state of all architectural registers at commit time
- Each entry of the ROB gets augmented with the value of the destination register
- Write the value of the destination register at commit time.
- The RRF maintains a checkpoint at commit time

So, we will discuss four approaches or state recovery. The four approaches have different pros and cons. They are also arranged in an increasing order of complexity. The first two are more commonly used, the rest the next two are not that commonly used. But nevertheless, we will discuss four approaches. Here is approach one, the first approach is rather simple. So, I will not vouch for its efficiency. But in terms of sheer simplicity, it is clearly the most straightforward.

In hardware many a time, we prefer simple designs because it is easy to design and verify many of these protocols and simplicity by itself has its merit and value. So, what we do is that we introduce a new register file. This is an architectural register file mind you. It is called the RRF, the Retirement Register File. Switch if there are 16 architectural registers, it will have 16 entries and let us assume that we are committing an instruction of the form add and then, the value that we compute (r5 + r6) = 10. So, what we do is that when this instruction is committed, we also remember the value.

So, each entry of the ROB is augmented with the value of the destination register. So, this does add extra storage by the way. So, ROB typically has 200 entries. So, this does add extra storage in the sense that if it is a 64 bit machine, it adds 64 bits to the size of each ROB entry. So, this additional storage overhead is there; but we go to register number r4. In r4 we add the 64 bits which is the value that was computed.

So, which means that the RRF contains the retirement state or the state of all the committed instructions. So, what state? The register state. So, at any point of time, it contains the architectural register state of all the committed instructions. So, any point of time an instruction commits, its value is written over here. The destination register at commit time.

So, recovery is very easy because this is sort of a commit time, this called a checkpoint. A checkpoint is nothing but the record of a state. So, anytime, we want to recover, well no problem if the branch is at the head of the ROB and we find it as mispredicted not an issue. We flush the pipeline and the entire state of the RRF is returned to the physical register file.

So, what we do is we need to create a new mapping in this case. So, the rename table also should be modified, I am not showing that. So, what we can do is that we can. So, since we are cleaning up the pipeline, we have the liberty to also clean up all the mappings and start a fresh state. See if let us say our registers are numbered r0 to r15 architectural registers that is, then we can map r0 to p0 in the rename table and in the entry for r0, we can write the contents of r 0 from here; in the entry for r1, we can write the contents of r1 from here.

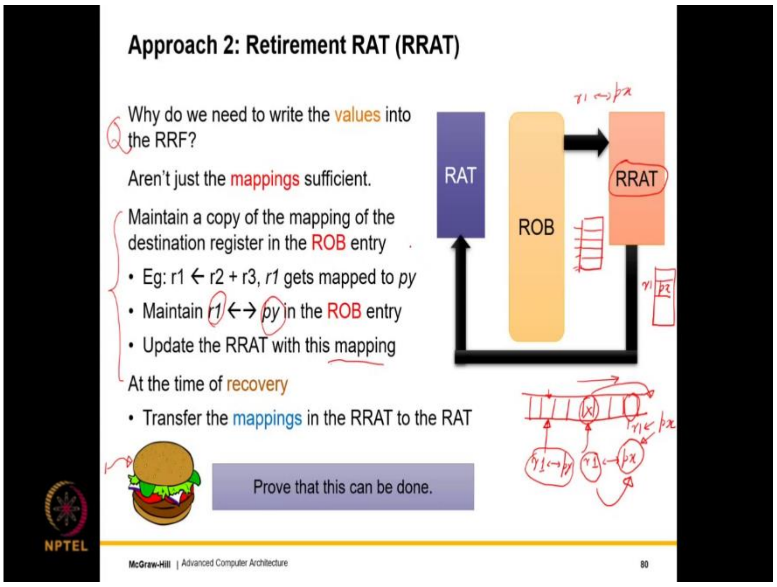
So, we essentially need to transfer this entire state; r0 over here, r1 over here, r0 will come to p0; r1 will come to p1 so on and so forth. But at least the first 16 registers here can just be made a copies can just be made a copy of the architectural registers and the mapping can be set in such a way; r0 to p0, r15 to p15 and this is because we are essentially starting afresh. So, we have the liberty to change the mapping, we do that and we copy the state of the entire architectural register file to the PRF.

So, this is undoubtedly a simple method because what we are doing is we are like doing a fresh start. The entire architectural state is being returned to the physical registers and those physical registers are being mapped to the architectural registers in the rename table. So, the rename table, I should have shown; I am not showing that and then, we just start from here and the programme will not get to know that it actually suffered from a misprediction or an exceptional event of any kind.

So, if we look at this just appreciate the sheer simplicity of this idea. This is like a brute force idea, where it is guaranteed to work because we just are transferring the state of all committed instructions to the PRF and restarting the programme from there. There is no hidden trick involved. What is the problem? Well, the problem is that it is inefficient. What are the inefficiencies?

The inefficiencies are that each ROB entry is being augmented with the value of the destination register that is one; the other is that the process of recovery is slow because physically 64; 16, 64-bit values have to be computed and the rename table has to be adjusted. So, those are the overheads.

(Refer Slide Time: 31:52)



Approach 2: Retirement RAT (RRAT)

Why do we need to write the values into the RRF?

Aren't just the mappings sufficient.

Maintain a copy of the mapping of the destination register in the ROB entry

- Eg: r1 ← r2 + r3, *r1* gets mapped to *py*
- Maintain *r1* ←→ *py* in the ROB entry
- Update the RRAT with this mapping

At the time of recovery

- Transfer the mappings in the RRAT to the RAT

Prove that this can be done.

McGraw-Hill | Advanced Computer Architecture 80

So, let us move to approach two, which does something more efficient and if you look at it, I have you can already see a burger. So, if you can see a burger, it should ring a bell in your head that there is something that needs to be proved. So, unless something is proven, you will not appreciate the complexity of this design and the correctness of this design.

So, as I said, burger means the white and grey matters should start churning in your head and it should tell you that look, there is something coming up which requires some amount of thinking. Well actually it does. So, much of the out of order logic and the rename table logic and all the other things that we have been discussing, we pretty much tossed out all of that in the previous design because it was so simple. It was simple and inefficient.

It turns out that we can make the design far more efficient, if we simply realise a couple of things. First, let us ask a question, why do we need to write the values into the RRF; what is the need? The values are anyway there correct? So, the values are anyway there. Well, why do I say so? So, that is important. If you can understand this sentence of mine that the values are already there, you have probably already guessed what I am getting to; but let us understand the insight.

Let us say the time flows to the right and this is the reorder buffer. This is the mispredicted branch. We have some instructions fetched after it and some fetched before it. So, let us say that when; so, the branch does not write to a register. So, let us say that when we were dealing with the branch, we found that r 1 → px. Now, when this branch instruction reaches the head of the reorder buffer, will r1 still be mapped to px that is question one and question 2 is will px still have the same value or will it have a different value? Well, so this means that we need to think about several things.

The first thing is we need to see when exactly is px release. So, px is released when some instruction, when some instruction writes to r 1 and that instruction commits. So, at the point of the branch if r1 → px, so it is clear that some instruction fetched after the branch would be writing to r 1 which would be py. So, it does not matter, but it is definitely not before.

See if it is not before, then what it means is that when the mispredicted branch actually reaches the head of the ROB, we are guaranteed to still have px and an untouched form because the instruction that maybe modifies r1 would not have committed. So, px would still retain its value and this can be used. So, we do not really have to copy any values right.

So, let me show an let me illustrate this slightly better. So, let us assume that there is one instruction after the mispredicted branch, where r1→ py. So, of course, after this instruction is renamed, all subsequent instructions will see r1 → py. But nevertheless, px will still be alive; it will still return retain its value. The reason is that this instruction over here, this one would not have committed primarily because it would have it would commit only when it reaches the head of the ROB.

But before it reaches the head of the ROB, the branch will reach the head of the ROB and at that point, we will realise that it has been mispredicted. Since, it has been mispredicted, we will then try attempt to recovery and then, we will realise that look when the branch was being renamed r1 → px, I will tell you in a second how we will realise that and then, we will also realise that this is the latest value of r1 from the point of view of when the brand was renamed, at that point the latest value was px.

So, r1 even though it might have been mapped to other registers like py, we should restore the mapping and we should again map r1 to px. Since px has its value intact, restoring the mapping itself is sufficient. So, let us go through the text first and then, I will explain this diagram. So, what we do is we maintain a copy of the mapping of the destination register in the ROB entry. See in this case, in ROB entry for the branch, I am sorry I take the previous sentence back. Let us look at the example.

Let us assume that there is an instruction add r1, r2, r3 where r1 gets mapped to py. Then, we maintain this r1 to py mapping in the ROB entry. So, which means that if let us say there is one instruction over here where r1 gets mapped to px, then we maintain the r1 to px mapping in this in the ROB entry of this instruction and later on, when r1 gets mapped to py, we also maintain that in ROB entry of that instruction.

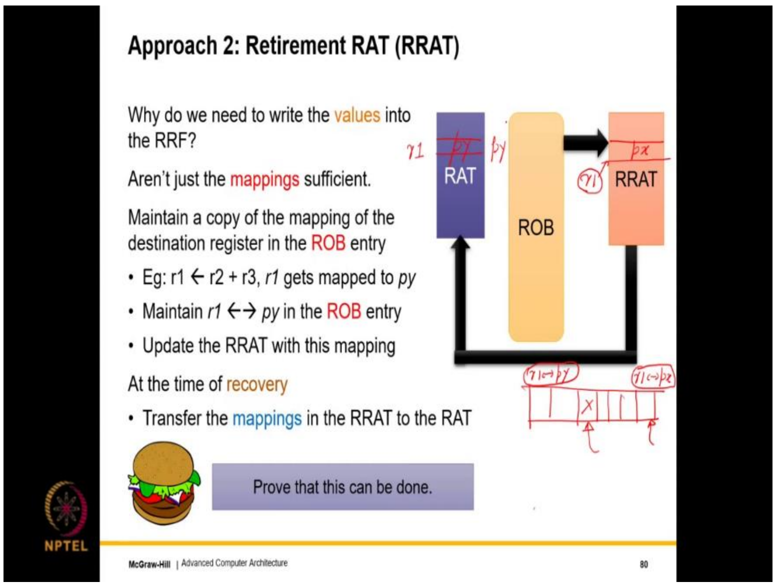
So, whenever an instruction commits. So, let us say at this point an instruction commits where r1 → px, we will update what is called the retirement RAT; retirement register alias table with r1 in the RAT is px. So, which basically means that from the point of view of this instruction r1 and px contains the latest value of r1 and since, this is committing from the commit stream from the point of view of all the committed instructions, the retired instructions, px contains the latest value.

Similarly, if this branch would not have separate from a misprediction, we would have ultimately committed r1 with this instruction, where r1 → py. At that point of time, we would have updated the retirement RAT with r1 and py. So, what is the retirement RAT contain? As it essentially contains a mapping of the 16 architectural registers to the corresponding physical registers in the physical register file.

So, since an architectural register can actually since the mapping of an architectural register to a physical registers changes, we will have several physical registers that contain the value of an architectural register at several points of time. So, given that fact what the RRAT basically contains is, it contains the, it contains the it contains the commit time checkpoint or it contains the fact that what was the mapping when this instruction was being renamed. So, that would be the latest mapping from the point of view of this instruction.

So, we keep on updating the RRAT with this mapping. So, what is the advantage of doing this? Well, the advantage of doing this is like this that when we want to recover.

(Refer Slide Time: 40:22)



Approach 2: Retirement RAT (RRAT)

Why do we need to write the values into the RRF?

Aren't just the mappings sufficient.

Maintain a copy of the mapping of the destination register in the ROB entry

- Eg: r1 ← r2 + r3, *r1* gets mapped to *py*
- Maintain *r1* ←→ *py* in the ROB entry
- Update the RRAT with this mapping

At the time of recovery

- Transfer the mappings in the RRAT to the RAT

Prove that this can be done.

McGraw-Hill | Advanced Computer Architecture 80

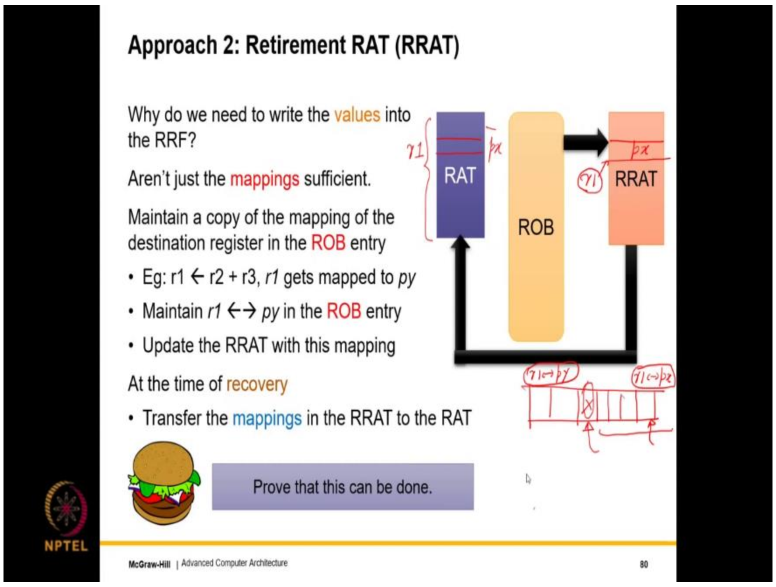
So, let me draw the same diagram again and then, I will talk about what is the advantage. So, this is the mispredicted branch; there are two instructions over here. First, where r1 is getting mapped to px is the earlier instruction; this is the later instruction. So, this instruction is committed.

When this instruction was being committed, we go to the entry corresponding to r1 and we write px over there which means that from the point of view of this instruction and all the instructions fetched before it, px contains the latest value of r1 or in other words, when this instruction was being renamed, just after it was renamed, px contained the latest value of r1.

Next assume, we commit a few more instructions; but nobody writes to r1, then we arrive at the branch. When we arrive at the branch, we will again find that r1. r1's latest value is in px and this value should be written to the actual register alias table RAT. But it is possible that by that time r1 → py, which means we have fetched and renamed this instruction. So, in for r1 actually py is written.

Let me write py over here. But since this instruction is being flushed from the pipeline, it is being removed from the pipeline and it is not being committed, what we need to do is we need to get rid of py.

(Refer Slide Time: 42:49)



Approach 2: Retirement RAT (RRAT)

Why do we need to write the values into the RRF?

Aren't just the mappings sufficient.

Maintain a copy of the mapping of the destination register in the ROB entry

- Eg: r1 ← r2 + r3, *r1* gets mapped to *py*
- Maintain *r1* ←→ *py* in the ROB entry
- Update the RRAT with this mapping

At the time of recovery

- Transfer the mappings in the RRAT to the RAT

Prove that this can be done.

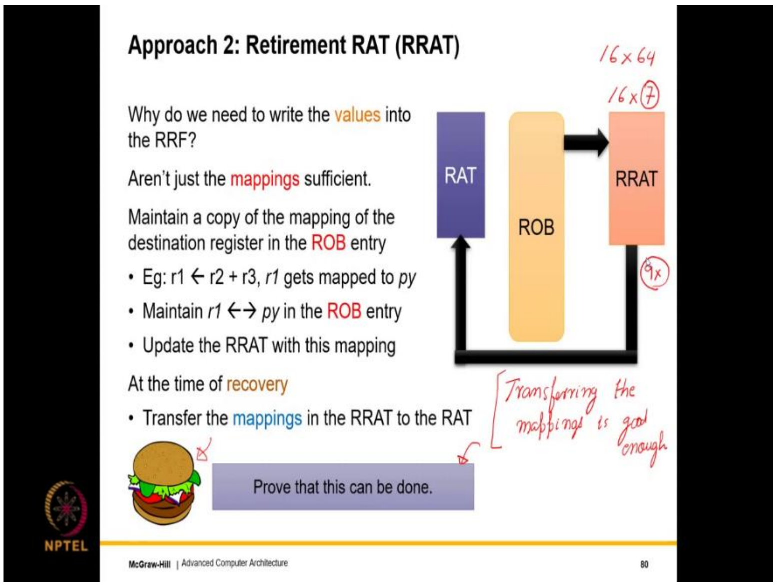
McGraw-Hill | Advanced Computer Architecture 80

So, let me use my eraser to get rid of it. So, we get rid of py over here and instead what we do is that we write px. So, this is equivalent to tantamount to restoring the value of r1 to that value which was the latest when this branch was being renamed. So, which was px not py.

We do the same for the rest of the architectural registers and this can be done. I will tell you why it is efficient in a second; but after doing that, we will have a set of 16 physical registers that contain the value of the committed state when this branch reach reaches the head of the ROB. It holds the architectural state for all of these instructions that are before the branch and if we just restored the mapping instead of the values, it is good enough.

So, what is the if I were to compare this scheme and the previous one, what is the advantage? We will first before discussing the advantage, you need to prove slightly more rigorously that this is actually correct.

(Refer Slide Time: 43:34)



Approach 2: Retirement RAT (RRAT)

Why do we need to write the values into the RRF?

Aren't just the mappings sufficient.

Maintain a copy of the mapping of the destination register in the ROB entry

- Eg: r1 ← r2 + r3, *r1* gets mapped to *py*
- Maintain *r1* ←→ *py* in the ROB entry
- Update the RRAT with this mapping

At the time of recovery

- Transfer the mappings in the RRAT to the RAT

Prove that this can be done.

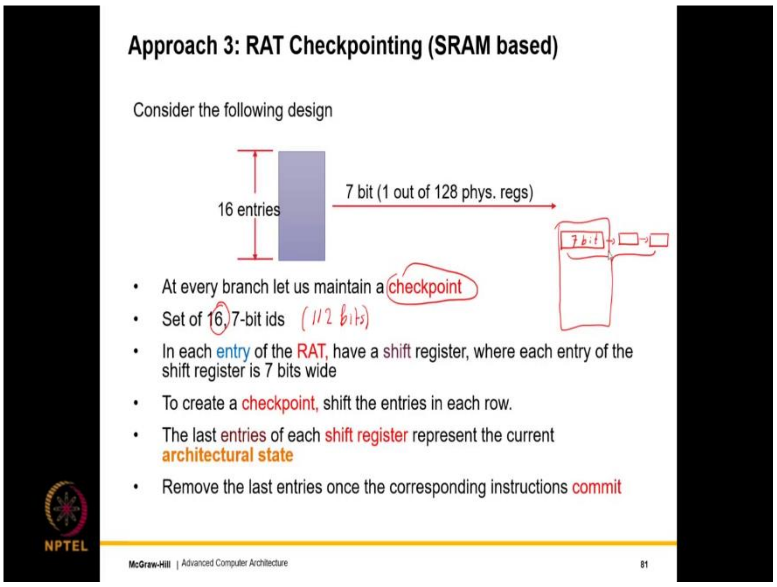
What is correct? That transferring the mappings is sufficient, you do not have to record the values and transfer the values. So, we just remember the mappings and transfer them. So, transferring the mappings is good enough. So, that is the reason instead of storing the value which is a heavy 64 bit number, we actually store a 7 bit number if we have 128 physical registers. So, transferring the mappings is good enough.

So, this again, I have given you sufficient pointers. Furthermore, more rigorous proof should be done. Once we have convinced ourselves of this fact, let us compare the RRF and RRAT. In RRF, we transfer a total of 16 X 64 bits; 16 architectural registers multiplied with 64 bits per register. In this case, we transfer only 16 X 7 bits because 7 represents the size of a mapping.

So, as we can see in terms of the number of bits transferred, this is roughly 9X more efficient, which means it is much faster as well as more power efficient and it also utilises the built in support that we have for out of order execution. In the sense that is the value is already there, all that we need to do is we need to update and adjust the mappings.

Just by adjusting the mappings, we can effectively recover to the state that existed, when the branch instruction was being renamed that was the latest state. So, we can recover to that just by updating the mappings.

(Refer Slide Time: 45:25)



Approach 3: RAT Checkpointing (SRAM based)

Consider the following design

16 entries
7 bit (1 out of 128 phys. regs)
7 bit

- At every branch let us maintain a checkpoint
- Set of 16, 7-bit ids (112 bits)
- In each entry of the RAT, have a shift register, where each entry of the shift register is 7 bits wide
- To create a checkpoint, shift the entries in each row.
- The last entries of each shift register represent the current architectural state
- Remove the last entries once the corresponding instructions commit

McGraw-Hill | Advanced Computer Architecture 81

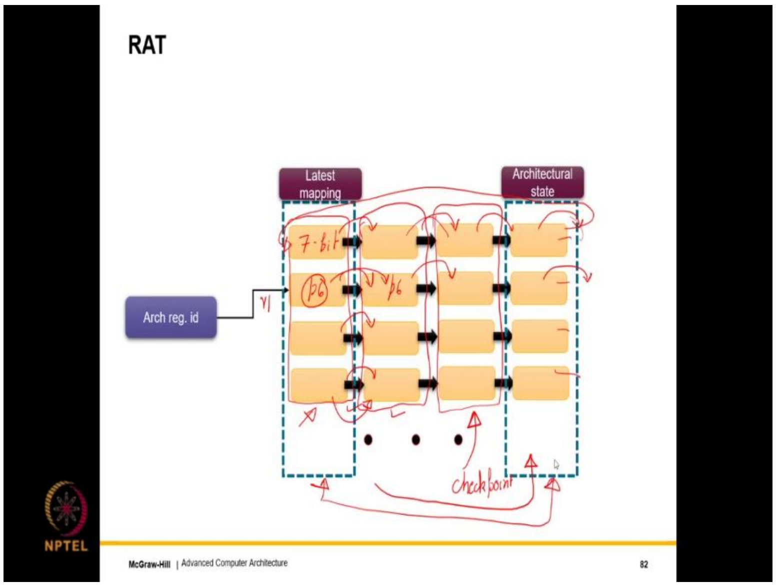
Now, let us look at the third design. So, in the third design, what we do is that instead of recording something or doing some bookkeeping whenever an instruction is retiring, we explicitly consider the fact that most of the exceptions will happen when we actually encounter a branch. So, let us assume that at other points exceptions will not happen. Even if they do, we have some mechanism of ensuring that we pretty much record state only when we encounter a branch or at other periodic points; but not on not for every instruction.

So, for every instruction, we do not read or write a register at commit time. So, we do not we do not do reads or writes at commit time. Because that is the biggest disadvantage of the previous approaches that whenever an instruction is committed, we need to read something, we need to write something. This is not power efficient. So, what we can do is if we are considering only branches and the only source of flushing and recovery is a branch misprediction.

Whenever we encounter a branch we should maintain a checkpoint. So, what we can do is that whenever we see a branch, we simply take a checkpoint of the rename table which is essentially a set of 16, 7-bit ids; 112 bits total. So, what we can do is we can architect the rename table differently. So, in the rename table previously, we just had one 7-bit field along with the available entry; of course, I am not discussing that.

So, we just had pretty much one 7-bit field to discuss the to record the physical register id. So, what we can have is we can have a circular shift register, where instead of one, we can think of this as a linked list. Just think of it as a linked list. So, to create a checkpoint all that we need to do is that we need to record a column of these mappings which can be seen better on the next slide.

(Refer Slide Time: 47:56)



RAT

So, what we do is that we update the rename table such that each entry is a linked list. So, what we do is that each of these entries are 7 bits. We create a linked list of such 7 bit entries and then whenever we want to take a checkpoint, this is what we do, that we shift all of these entries one step to the right. So, we shift all of them one step to the right.

Assuming we have enough space that is we shift all of them one step to the right. So, this essentially, becomes an instantaneous snapshot of the rename table. This becomes an instantaneous snapshot of the rename table for a different branch and finally, the last one essentially becomes the architectural state for the branch that is at the head of the rename table right. So, this becomes architectural state why?.

Well, because this captures the state of the rename table for the last branch and if this branch reaches becomes the earliest, then this mapping is essentially what we need to restore and this would be very simple. We just need to move the latest mapping pointer over here. So, recovery is very fast and to create a checkpoint; a checkpoint is essentially a snapshot of rename values, all that we need to do is we just need to transfer the values in these registers one step to the right.

The same way that we operate a shift register or a linked list, we just take all of these values, transfer them one step to the right and this is our latest rename table which we can keep on updating. There is no problem, this we can keep on updating. But whenever we want to freeze it, essentially take a photograph instantaneous photograph of it, all that we do is we just shift it one step to the right.

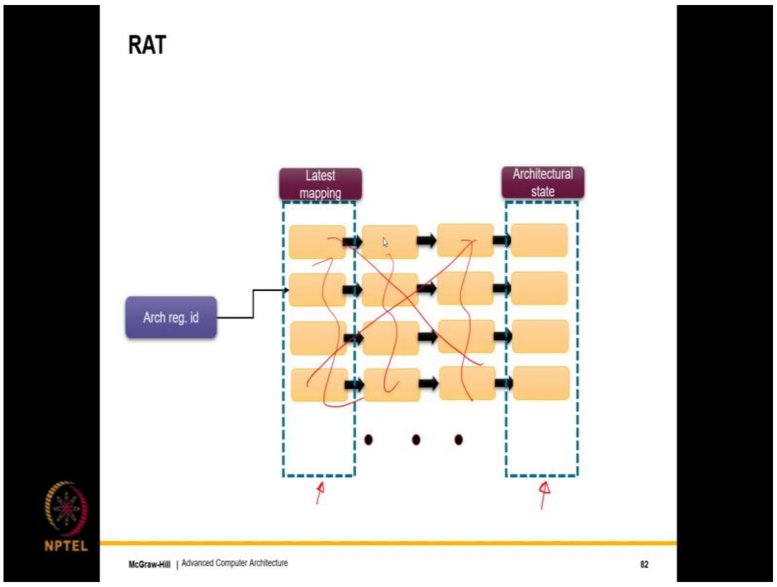
So, once it has been shifted, it cannot be modified again. It is like immutable photograph which cannot be changed and so, we it is basically a copy and a shift. So, let us say for register r1, p6 is written. So, what we do is when we shift it we maintain p6 here, then we shift I am assuming there are other columns. So, we just shift them here, shift this one here, shift this one here and ultimately, when this becomes empty, we write p6 over here.

So, recall that when we are creating the checkpoint, this column and this column will be the same. But gradually, since this column is meant to contain the latest mapping, it will keep on changing as new instructions keep getting renamed. Once if you want to take a checkpoint, we follow the same strategy, where we right shift all the previous checkpoints.

Because each one of them contains a checkpoint, each column in a sense contains a checkpoint. So, all that we need to do is we need to right shift all the checkpoints, create space for this and transfer the contents of all of these over here to this column. So, this is advantageous no doubt; one advantage is that recovery is quick.

So, while recovering, we do not really have to do much. In the sense, all that we need to do if I were to consider a physical implementation. So, these are not really linear linked lists; but these are essentially circular linked lists with one pointer to the latest mapping. So, all that I can do is I can move the pointer of my latest mapping from here to actually here. So, this will give me the one sec, I will probably clean the screen and show you.

(Refer Slide Time: 52:01)



RAT

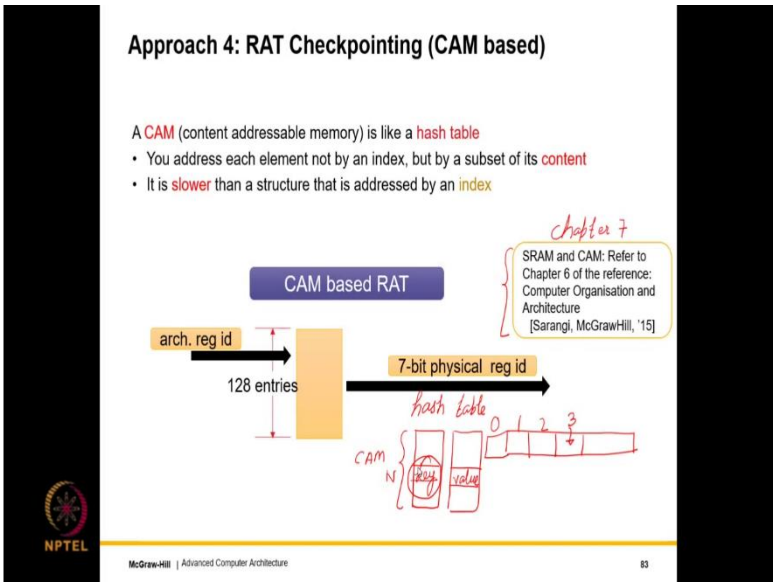
McGraw-Hill | Advanced Computer Architecture 82

So, the pointer the latest mapping can be moved from here to actually here which will tell me that look, we have discarded all of these checkpoints, right all of these checkpoints to contain irrelevant data now. This is the only checkpoint that we need to use and for us, this is the latest mapping. So, the execution can begin from here. What is the disadvantage? Well, the disadvantage is that each entry or the rename table is now a linked list.

Essentially, it is a circular linked list we can also call it a; so, physically you will implement it using a shift register that is one disadvantage. And the other is that if let us say an arbitrary instruction suffers from a fault, we can only recover to the nearest branch.

So, that will complicate some of the logic for maintaining precise state; but that of course, can be fixed in the sense that if you are recovering to the nearest branch, then the state difference between the correct and current instruction and the branch that difference has to be recorded. But that is not a major issue, it is something which is solvable.

(Refer Slide Time: 53:18)



Approach 4: RAT Checkpointing (CAM based)

A CAM (content addressable memory) is like a hash table

- You address each element not by an index, but by a subset of its content
- It is slower than a structure that is addressed by an index

SRAM and CAM: Refer to Chapter 6 of the reference: Computer Organisation and Architecture [Sarangi, McGrawHill, '15]

Let us now come to the fourth scheme which is more complicated. Many of you also might not have the adequate background to understand this scheme. So, in this case, we architect the rename table differently. So, a traditional rename table actually uses a structure called an SRAM.

So, Static RAM and a CAM can be picked up from two sources; the first source is chapter 7 of this book, where this concept is explained rather briefly or you can look it up in chapter 6 of the older book, Computer Organization and Architecture, McGraw-Hill 2015. So, in that book, if you go to chapter number 10, then SRAM and CAM arrays are described in great detail.

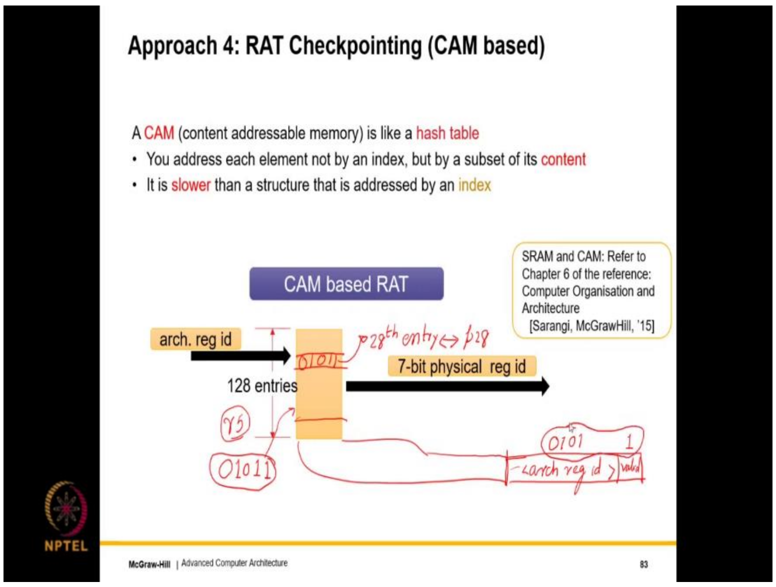
So, an SRAM array is like a regular array, where we essentially have entries and we have indices. So, essentially, if we are talking of entry number 3, it points to this point; the third index; if you start counting from 0, well no problems 0, 1, 2 and 3. A CAM array on the other hand is like a hash table. So, for a CAM array, we basically have an array of keys and an array of values.

So, both contain the same number of entries. So, what we can do is that we do not search by index, we search by the key and what we return is the value. In many cases, the value field can be empty, we just search by the key and the index in which the key matches that is the value.

So, in many cases, that is what we want. We want to find out, where the key is in a large array. If it is at the 59th entry, the value is 59 that is a legitimate variation of a CAM array; but regardless of which flavour we are using, we need to think of a CAM array pretty much as a hash table, where we search not by the index, but by a multi bit field. The multi bit field in this case is being referred to as the key.

So, I mean the basic abstraction is that an SRAM array is like a regular array, but a CAM array is like a hash table. So, it is possible to implement the RAT, the Register Alias Table using CAM arrays. So, what we can do is we can have 128 entries each of these entries will contain two things.

(Refer Slide Time: 56:16)



Approach 4: RAT Checkpointing (CAM based)

A CAM (content addressable memory) is like a hash table
- You address each element not by an index, but by a subset of its content
- It is slower than a structure that is addressed by an index

SRAM and CAM: Refer to Chapter 6 of the reference: Computer Organisation and Architecture [Sarangi, McGrawHill, '15]

McGraw-Hill | Advanced Computer Architecture

83

If I were to see one entry, so each entry I am just blowing this thing up. So, each entry will contain two things; it will contain the architectural register id that the given physical register is mapping to. So, why 128? Because we have 128 physical registers and let say I want to find out which physical register is r5 currently mapped to.

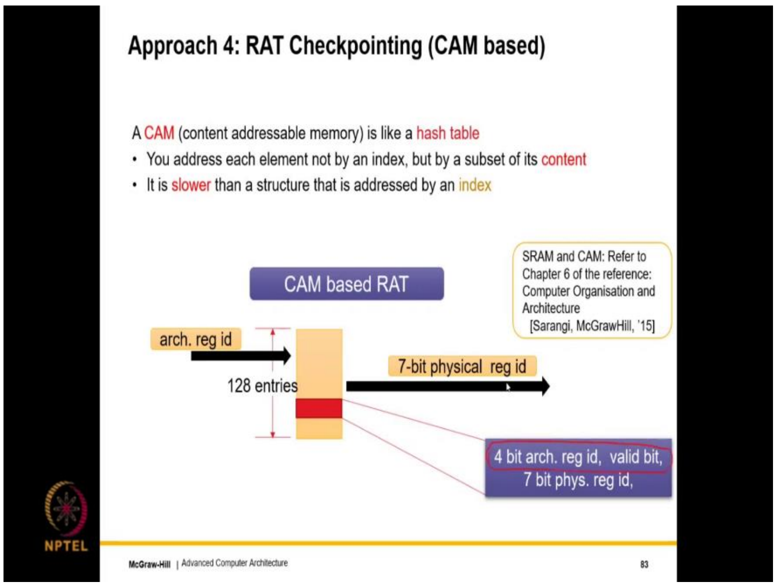
So, in the previous approach, I will just index the 5th entry; but in this case, I will not do that. In this case, what I will search for is I will search for 5 which is 0 1 0 1 in binary and also, the mapping should be valid in the sense that what we have discussed in the past is that there might be multiple physical registers that map to r5 in the pipeline at the same time. One of them will be the most recent, let us call it valid and the rest invalid.

So, we want the valid bit to be one which means it is the most recent mapping and also that it is mapping to r5. So, what we need to search for, what the key would be in this case is 0 1 0 1 corresponding to r 5 and the fact that its valid mapping, the most recent mapping. I search for this within the CAM array. I am guaranteed to find only one entry that matches and the index of that can give me the physical register id.

So, let us see that this maps to the 28th entry and here we find out that it contains 0 1 0 1 1, then I say that r5 → p28. So, if you think about it, this is this in itself is a valid way; I would not say it is a common way, but it is undoubtedly an extremely plausible way of implementing a register alias table, where instead of implementing it as an array, we implemented as a hash table.

So, in this case, we will have a CAM array and the CAM array will pretty much have two columns; the architectural register id and the valid bit. The valid bit indicates that it is the most recent mapping. So, when we searc, we concatenate the architectural register id with one, which indicates we need the most recent mapping for a given architectural register.

(Refer Slide Time: 58:56)



Approach 4: RAT Checkpointing (CAM based)

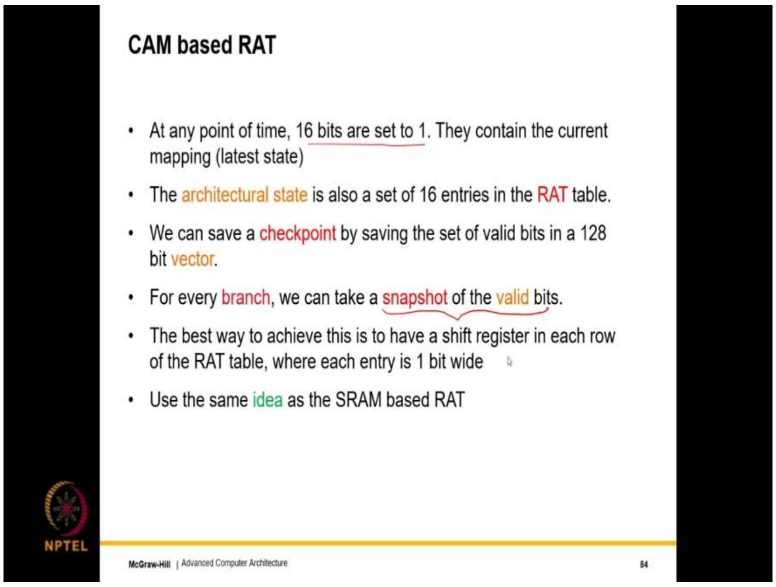
A CAM (content addressable memory) is like a hash table
- You address each element not by an index, but by a subset of its content
- It is slower than a structure that is addressed by an index

SRAM and CAM: Refer to Chapter 6 of the reference: Computer Organisation and Architecture [Sarangi, McGrawHill, '15]

CAM based RAT
arch. reg id
128 entries
7-bit physical reg id
4 bit arch. reg id, valid bit,
7 bit phys. reg id,
NPTEL
McGraw-Hill | Advanced Computer Architecture
83

So, it is a 4-bit architecture register id and a valid bit is what you get for a 7-bit physical register id. So, 7-bit physical register id need not be a part of the entry, the index in which this information is found can represent the physical register id.

(Refer Slide Time: 59:28)



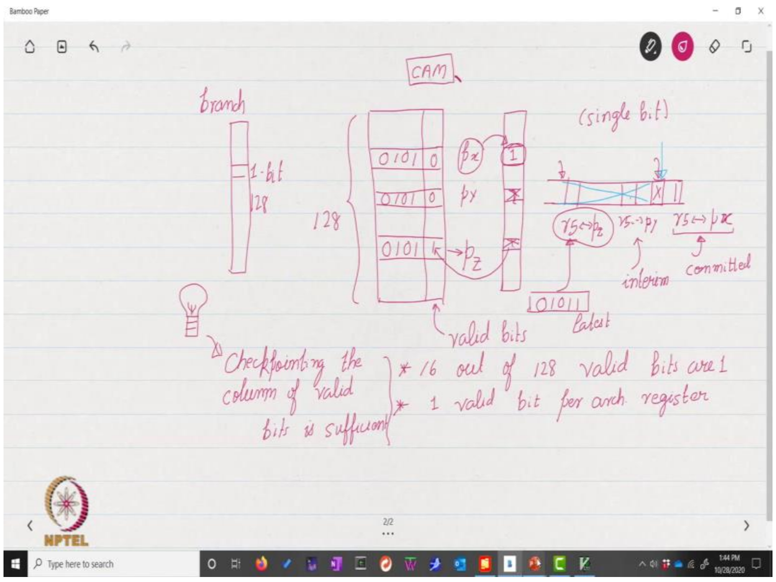
CAM based RAT

- At any point of time, 16 bits are set to 1. They contain the current mapping (latest state)
- The architectural state is also a set of 16 entries in the RAT table.
- We can save a checkpoint by saving the set of valid bits in a 128 bit vector.
- For every branch, we can take a snapshot of the valid bits.
- The best way to achieve this is to have a shift register in each row of the RAT table, where each entry is 1 bit wide
- Use the same idea as the SRAM based RAT

McGraw-Hill | Advanced Computer Architecture 84

So, now let us see how a CAM based RAT actually works. So, the key points over here are that at any point of time 16 bits are set to 1 because only 16 architecture registers are what we have. So, this is the set of 16 entries in the RAT table, we can save a snapshot of these. So, this is an important point in a graphical form. So, there is an important thing that needs to be understood.

(Refer Slide Time: 60:07)



CAM
branch
1-bit
128
128
0101 0
0101 0
0101 1
px
py
pz
1
(single bit)
r5↔pz
r5→py
r5↔px
interim
committed
101011
latest
valid bits
Checkpointing the column of valid bits is sufficient
* 16 out of 128 valid bits are 1
* 1 valid bit per arch. register

So, a lot of people find it very very hard to understand this scheme and the entire crux of the problem stems from the last statement that I made. So, you need to understand that there are 128 entries in this structure. This structure is a hash table which we are calling a content addressable memory or a CAM. Each entry of this CAM contains an architectural register id. Let us see if we are talking about r5, then it is 0101 and a valid bit. So, the valid bit is an important concept and in fact, it is the most important concept when we are discussing this.

So, it is possible that in our pipeline, we might have four instructions fetched at different points of time that pretty much right to register r5. So, the earliest one might be right r5, if you would have mapped it to px; the next one might have been mapped to py and the next one might have been mapped to pz. So, all three of these mappings are will exist in our pipeline. But clearly the r5 to pz mapping will be the most recent. So, the valid bit for the px and py entries will be 0; the valid bit for this will be 1.

So, whenever any subsequent instruction comes and it tries to see which physical register is mapped to r5, it will essentially search for this value in the CAM which is 0101 and 1, 1 because of validity and that would point us to this entry which is the entry for pz. So, this is the crux. So, regardless of the fact that many instructions might be mapped to r5; of course, capturing different points of the execution, only one will be valid.

The same holds true for other registers. So, it can be proven that this column; the column of valid bits will have these properties. So, I am I have provided an informal proof, but a formal proof is something that you can work out on your own. It is easy, it is not that hard. So, the first is that only 16 out of 128 valid bits are 1 that should be obvious because we will have 1 valid bit per each architectural register and there are 16 such architectural registers.

The other would be that essentially it is 1 valid bit per architectural register which is just a rehash of the previous statement, the rest remains the same. So, here is the key insight. So, let me draw a small bulb. I am not a good drawer and I am not a good artist that too with a tablet; still I am trying my best. I seem to have done an ok job.

So, what I am trying to say over here is the key insight is that since registers tend to maintain their value until another register that writes to the same architectural register gets committed. So, it is the same insight as the previous approach, approach 3. What I am saying or rather I am claiming is that check pointing, the check pointing the column of valid bits is sufficient; check pointing essentially means storing.

So, this is sufficient. All that I need to do is that this column that essentially contains these one bit numbers, this column of valid bits. If I just check point it, if I just store it, then I can easily restore it. So, which basically means that for a branch instead of having to deal with 7 bit numbers; so, instead of having to deal with 16, 7-bit numbers, for of course moving them in principle, we will take 7 clock cycles, I can have one thin and narrow column of 128, 1-bit numbers and as far as I am concerned, these 1 bit numbers, a sequence of 128 of them, they represent a checkpoint and what I can do is that I can just roll back and restore a checkpoint rather easily.

So, if we think about it, if we look at it in the context of this example, I can do this for r 5 and you will see that it will be correct. So, let us consider the execution at this point at this point clearly pz is the latest mapping and what we see is that the rest of the mappings to py and px are outdated and old. Now, let us assume that branch misprediction happens over here. So, one branch misprediction over here happens.

So, which means that at this point, we do not do anything; we wait for the branch to become the head of the ROB. When that becomes the head of the ROB then the mapping of r5 →px is a part of the committed state correct and r5 to py is an interim value and r5 to pz is the latest.

So, let me name them. So, this is a part of the committed state right. So, because this has been written, this instruction has been committed. This instruction is interim which means that write has been done, but the instruction has not committed. Consequently, we have not released px because this instruction has not committed, r5 to py and this is the latest. So, what do we do?

What we need to do over here is we need to what we need to do is well the first thing that we need to do is essentially change the colour and I am changing the colour of my ink. So, we pretty much need to cancel this part of the execution because this is potentially on the wrong path and we need to rollback our state to this point.

If we need to roll back our state to this point, we pretty much need to restore the values of all the registers that existed at this point. So, we have already seen an approach 2 that restoring the values is a slow operation. What we should instead do is that we should restore the mappings. So, for restoring the mappings, what we need to do is that we will have a bit vector that would have stored the mapping for the current for the branch corresponding to this.

So, there will be a mapping corresponding to this branch? So, the mapping that would correspond to this branch would have a one over here for px because for it, px is the latest mapping of r5 and for the other two, well we do not really care what it contains because those are not relevant. So, let me put a question mark because we do not care.

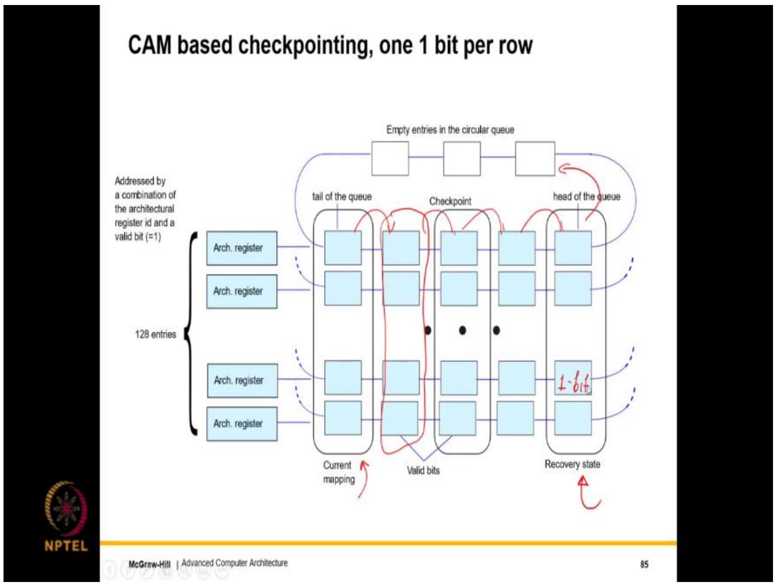
So, when we restore the mapping for this branch and we started execution from here, we will pretty much start sorry this does not look like a question mark. So, let me put a cross because the mapping does not really matter. So, what we would do is we will restore this entry as the valid bit for register r 5 and then, this is effectively the same as changing the mapping in the previous approach.

But in the previous approach, we were actually copying the mapping a 7-bit physical register id; in this case, all that we do is in a px'th entry of this CAM based RAT instead of 0. We just convert it to 1 and naturally over here, since we are this entry is replacing this. If 5 is written over here for pz, then of course, this gets converted to 0; but if this is mapping to some other architectural register, then we do not care.

But the most important point is that for px, the mapping becomes one which means that px now contains the most recent value which actually it should because the rest of the instructions state anyway we are getting rid-off. So, this actually it should. So, this basically tells us that the process of our checkpoint creation is much faster because in this case, we are dealing in with single bits, instead of larger 7 bit quantities. So, dealing with them is much faster.

But of course, a CAM itself is slower than a basic SRAM. In a sense, a hash table is always slower than an array. So, that issue notwithstanding. Since, we are dealing with single bits we can look at this as a faster alternative subject to other constraints. So, now let us go back to the slideshow. So, essentially for every branch, we take a snapshot of the valid bits. The best way to achieve this is to have a shift register with each row similar to what we had in the previous design and we use the same idea as the SRAM based RAT.

(Refer Slide Time: 71:54)

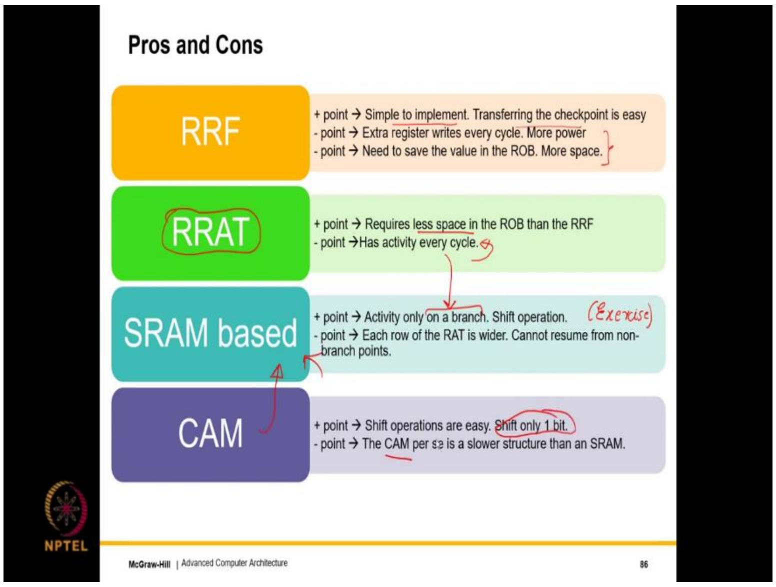


CAM based checkpointing, one 1 bit per row

So, what do we do? Well, in this case, what we do is we have a list of mappings, where this is of course, the current mapping; then, we have a field of architectural registers. So, this will remain the same, only thing that will get check pointed are these 1 bit columns. The check pointing logic also happens in the same way. So, how does it happen? Well, the way it happens is whenever we want to take a checkpoint, we just right shift everything. So, these are internally organized as circular queues which is a recurring pattern that we are seeing in hardware.

So, pretty much this moves over here, this here, this here, this here, then this here and whenever we want to recover we pretty much take the rightmost entry or rather the head of the queue, the earliest entry of the queue, the head of the queue and then we recover our state to that point. So, this mind view is a much faster option because each of these entries is 1 bit instead of 7 bits. Hope it is clear.

(Refer Slide Time: 73:06)



Pros and Cons

RRF	+ point → Simple to implement. Transferring the checkpoint is easy - point → Extra register writes every cycle. More power - point → Need to save the value in the ROB. More space.
RRAT	+ point → Requires less space in the ROB than the RRF - point →Has activity every cycle.
SRAM based	+ point → Activity only on a branch. Shift operation. (Exercise) - point → Each row of the RAT is wider. Cannot resume from non-branch points.
CAM	+ point → Shift operations are easy. Shift only 1 bit. - point → The CAM per se is a slower structure than an SRAM.

McGraw-Hill | Advanced Computer Architecture 86

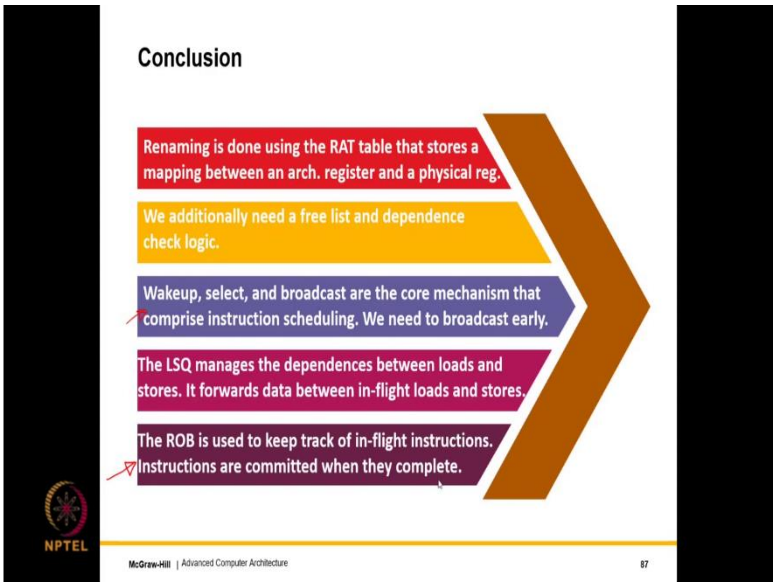
So, now, we are in a position to look at the pros and cons of all four schemes. So, the RRF is clearly simple to implement. Transferring the checkpoint is easy. It does involve extra register writes every cycle, we need to save fat values and that too in the ROB. So, space is a concern, power efficiency is a concern. So, I would say RRF is simple. It is not something that you would always want to use, if you want to create high performance processor.

What you should instead use is an RRAT? where you require less space and, but the point is that transferring the checkpoint etcetera is much faster because you are dealing with mappings and not with values. It still has activity every cycle. If you are not happy with activity every cycle you just want to have it on a branch, well then use an SRAM based RAT check pointing scheme.

But well, what do you do if a normal instruction has an exception? Well, you need to recover to the nearest branch. What about the instructions between the branch and that instruction? Well, then you need a smaller state buffer for that. What if a code that does not have branches for a long time? So, for all of these, I would there are simple solutions that require some amount of book keeping or some amount of forced check pointing.

So, this is left as an exercise for the viewer to figure out a solution of creating a practical form of this. CAM based approach is needless to say much more efficient than SRAM based approach because you are only shifting 1 bit. But the overhead is that you are using a CAM which per se by itself is a slower structure than an SRAM.

(Refer Slide Time: 75:02)



Conclusion

Renaming is done using the RAT table that stores a mapping between an arch. register and a physical reg.

We additionally need a free list and dependence check logic.

Wakeup, select, and broadcast are the core mechanism that comprise instruction scheduling. We need to broadcast early.

The LSQ manages the dependences between loads and stores. It forwards data between in-flight loads and stores.

The ROB is used to keep track of in-flight instructions. Instructions are committed when they complete.

McGraw-Hill | Advanced Computer Architecture 87

So, we have successfully ended our journey of this very long chapter on the rather intricate aspects of an out of order pipeline; but at least you will be happy to know that the out of order pipeline is complete. So, key some of the key points renaming and done using the RAT table between architectural physical registers. We do need a free list and dependence check logic. Wakeup, select and broadcasts are the core mechanisms that comprise instruction sheet scheduling.

The LSQ manages the dependencies between loads and stores, forwards data between in-flight loads and stores and the reorder buffer is a very important buffer, that keeps track of in-flight instructions and when they complete, they are said to be committed and of course, we need a method of state recovery to move back in time to restore the state that existed when a given branch was being renamed.

So, in the next chapter, chapter 5, we will study more complex mechanisms and out of order processors and we will also look at how the compiler helps make processors more efficient. So, what we can do at a compiler level to make the process of dealing with instructions, to make processors essentially way more efficient. the end.

Thank you very much. See you in chapter 5.