Advanced Computer Architecture

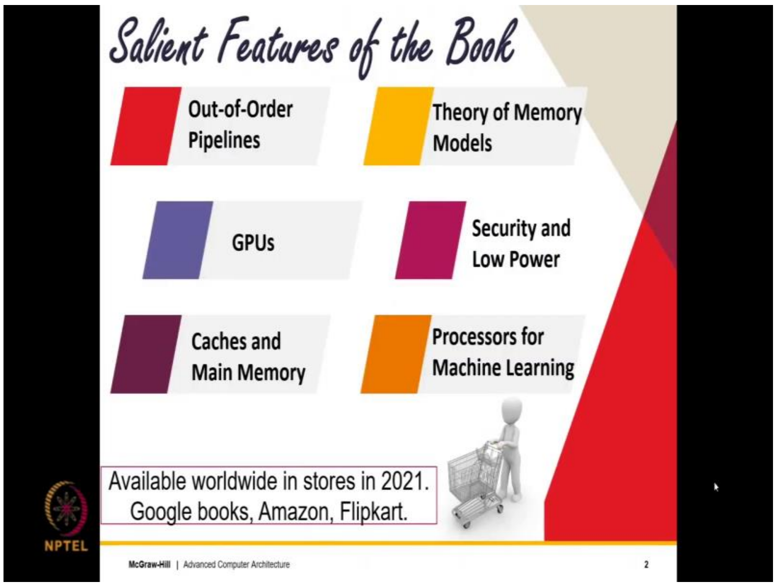
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Module - 03
Lecture - 08
The Issue, Execute, and Commit Stages Part-I

(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

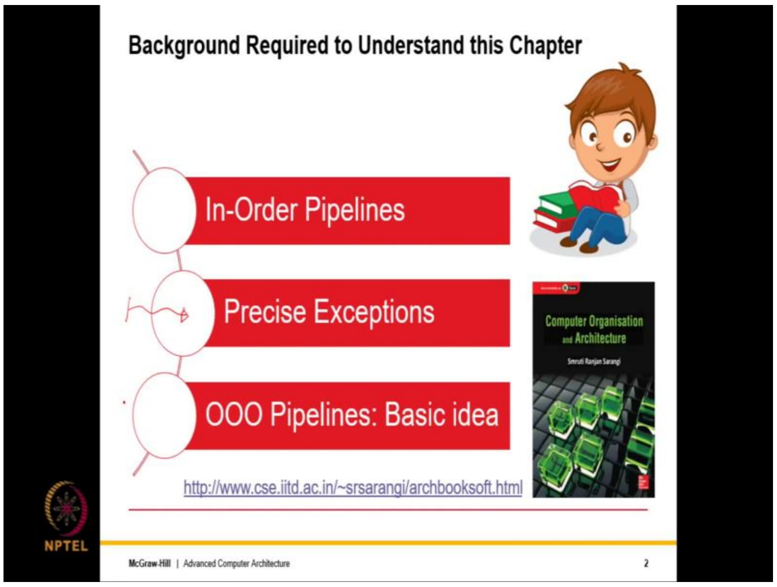
- Out-of-Order Pipelines
- Theory of Memory Models
- GPUs
- Security and Low Power
- Caches and Main Memory
- Processors for Machine Learning

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

McGraw-Hill | Advanced Computer Architecture 2

Welcome to the chapter on The Issue, Execute and Commit Stages. So, this chapter will introduce the core of the out of order pipeline. It will introduce all of its most important structures and how they and how they work.

(Refer Slide Time: 00:59)



Background Required to Understand this Chapter

- In-Order Pipelines
- Precise Exceptions
- OOO Pipelines: Basic idea

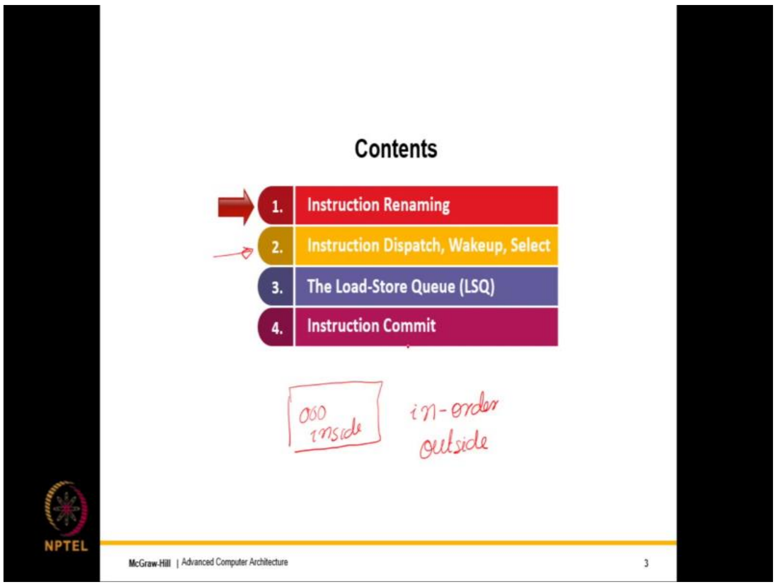
http://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

Computer Organisation and Architecture
Smruti Ranjan Sarangi
NPTEL
McGraw-Hill | Advanced Computer Architecture
2

So, what is the background that we require for this chapter? We require the knowledge of In-order pipelines. So, this was there in a previous chapter and also this is a part of the basic computer architecture course which you can find in my previous book; Computer Organization and Architecture, published by McGraw Hill in 2015, it is available worldwide. So, along with in order pipelines we need to understand what exactly are precise exceptions.

So, this was also there in the lecture for the previous chapter. So, this is another important background concept, and finally, the reader, the viewer in this case should appreciate the basic idea out of order pipelines. In a certain sense the idea that needs to be understood is that we need a large pool of instructions such that we can issue enough independent instructions that can be issued in data dependence order and we need not follow program order.

(Refer Slide Time: 02:08)



Contents

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

ooo inside

in-order outside

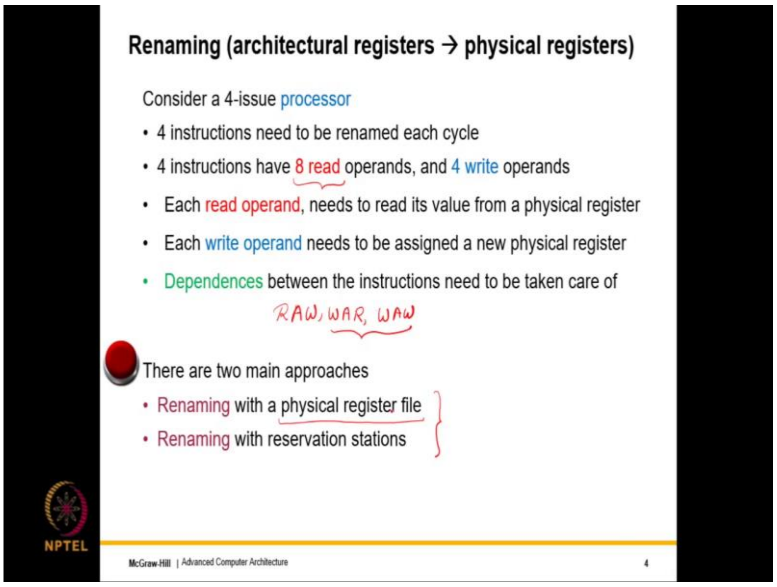
McGraw-Hill | Advanced Computer Architecture 3

So, we will proceed as follows in this set of lecture videos. We will start with instruction renaming. Recall that renaming is a method to get rid of WAR and WAW hazards. So, of course, read after write hazards will remain, nothing can be done about those and we will see what to do about them in a later chapter but not now. Then we will discuss the key features of an out of order pipeline which are instruction dispatch wakeup and select.

The third topic will be discussion of loads and stores. So, we have not discussed memory dependences up till now, we have only discussed register dependencies. So, discussing memory dependencies will be a part of the third part of this slide set and lecture set also and finally, the last a part we will talk about instruction commit.

So, recall that the basic abstraction that we had walls, out of order inside, in-order outside. So, this helps us preserve the notion of intuitive in order execution and also we can guarantee precise exceptions using this. So, this requires a novel mechanism called instruction commit or instruction retirement. So, we will look at this in the last part of the lecture.

(Refer Slide Time: 03:46)



Renaming (architectural registers → physical registers)

Consider a 4-issue processor

- 4 instructions need to be renamed each cycle
- 4 instructions have 8 read operands, and 4 write operands
  - Each read operand, needs to read its value from a physical register
  - Each write operand needs to be assigned a new physical register
- Dependences between the instructions need to be taken care of

RAW, WAR, WAW

There are two main approaches

- Renaming with a physical register file
- Renaming with reservation stations

McGraw-Hill | Advanced Computer Architecture 4

So, let us now start with renaming. So, recall that renaming is a process that converts architectural registers to code with architectural registers, to code with physical registers. So, consider a 4-issue processor. So, in this case 4 instructions need to be renamed each cycle.

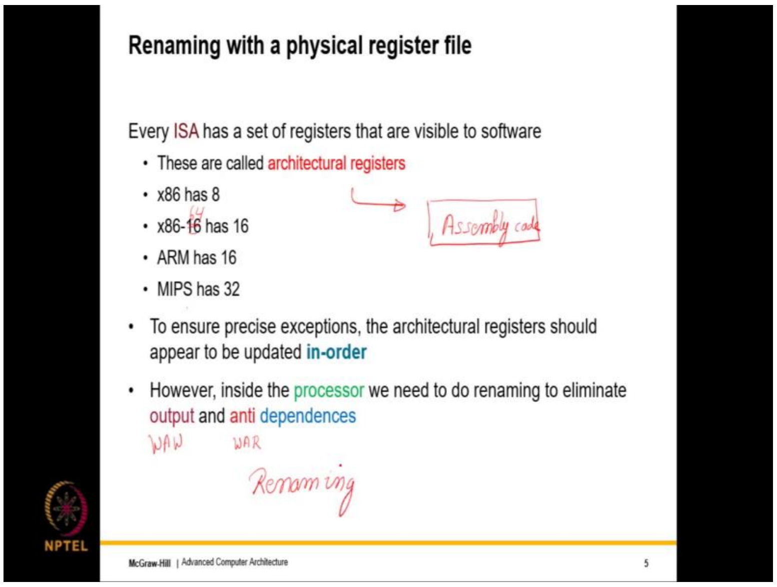
So, we can have a maximum of 8 read operands in our instruction set, which all 8 can be registers and we can have a maximum of 4 write operands where all 4 of them could be registers. Each read operand needs to read its value from a physical register and similarly, each write operand also needs to write its assigned value to a physical register.

And, dependencies will be there, three times of dependencies up till now have broadly been concerning us, where RAW dependencies, WAR dependencies write after read and write after write.

So, the write after read and write after write dependencies are what we are after. So, we will try to solve these issues with renaming. There are two main approaches renaming with a physical register file and renaming with reservation stations.

So, renaming with physical register file is typically what is used in modern high performance processors. So, renaming with reservation stations and other approaches we will look at in the next chapter not here, not now. We will look at the basic idea first and at least describe the basic simple solution first.

(Refer Slide Time: 05:30)



Renaming with a physical register file

Every ISA has a set of registers that are visible to software

- These are called architectural registers
- x86 has 8
- x86-~~16~~ 64 has 16
- ARM has 16
- MIPS has 32

$\longrightarrow$ $\boxed{\text{Assembly code}}$

- To ensure precise exceptions, the architectural registers should appear to be updated **in-order**
- However, inside the processor we need to do renaming to eliminate output and anti dependences

WAW WAR

Renaming

McGraw-Hill | Advanced Computer Architecture 5

So, if I were to consider a physical register file, this is what I need to do. So, every ISA, every instruction set architecture will have a set of registers that are visible to software. These set of registers will be known as architectural registers and these architectural registers are what your assembly code is going to use.

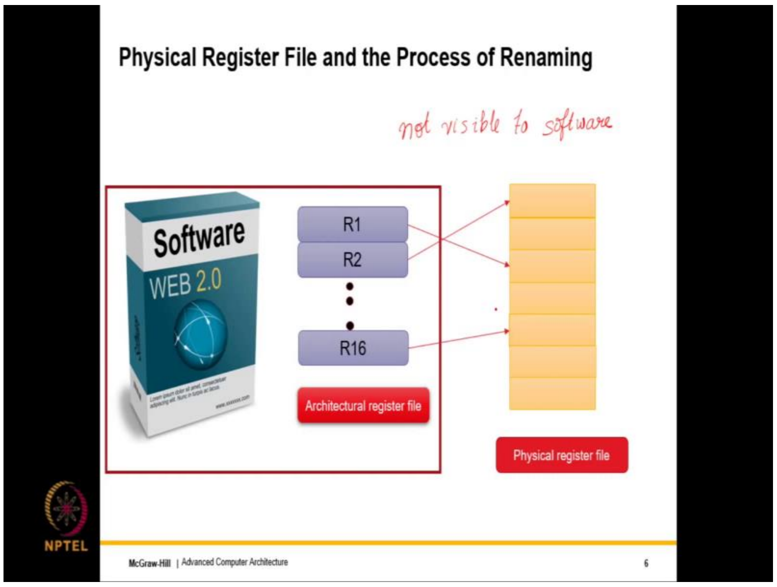
So, these architectural registers are very important in the sense of the compiler maps them to variables in high level programming languages and we have already seen when we write assembly code that these registers have to be dealt with while calling a function and returning from it.

So, the basic x86 instruction set that is used by intel and AMD has 8 such registers. The newer variant the x86, I stand corrected this should be x86 64 has 16 registers. The even the basic version of the arm ISA has 16 registers and MIPS ISA has 32 registers.

So, to ensure precise exceptions as we have discussed earlier the it should appear as if these architectural registers are being updated in program order. Even though inside the processor out of order execution is happening, so that is the basic idea. So, let us now look at the first major out of order construct which is renaming, which is primarily aim towards eliminating output and anti-dependencies.

We call that output dependencies are write after write WAW dependencies and anti dependencies are write after read WAR dependencies. So, let us discuss how exactly we achieve renaming in a practical out of order pipeline.

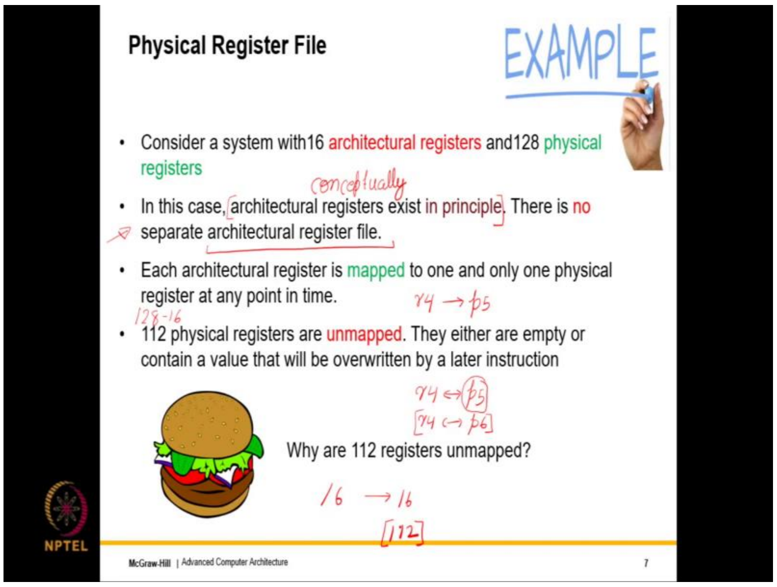
(Refer Slide Time: 07:51)


Physical Register File and the Process of Renaming
not visible to software
Software
WEB 2.0
R1
R2
R16
Architectural register file
Physical register file
NPTEL
McGraw-Hill | Advanced Computer Architecture
6

So, what we actually do is, that any software program that is written with architectural registers, R1 R2 to R16, every single register is automatically mapped by the hardware. So, it is important to note here that this process is not visible.

So, this process is not visible to software, it is only, it is happening totally within the processor and the processor renames the processor as a hardware engine to rename, software is not involved in the processor renaming at all. So, what the hardware does is, then it maps each of this architectural registers to a set of unique and distinct, to a set of distinct physical registers.

(Refer Slide Time: 08:54)


Physical Register File

- Consider a system with16 architectural registers and128 physical registers
- In this case, architectural registers exist in principle. There is no separate architectural register file.
- Each architectural register is mapped to one and only one physical register at any point in time.
- 112 physical registers are unmapped. They either are empty or contain a value that will be overwritten by a later instruction

Why are 112 registers unmapped?

So, let us now explain renaming in the context of an example. So, let us consider a system with 16 architectural registers and 128 physical registers. So, in this case, the architectural registers exist only in principle, in the sense that they are not associated with permanent storage locations. So, they exist cannot conceptually, in the sense the software sees that, but the hardware does not have separate space for the architectural registers, it’s just as a mapping.

So, there is no separate architectural register file that is single register file which is the physical register file, which of course, contains far more registers than the number of architectural registers, 128 in this case. Each architectural register is mapped to one and only one physical register at any point in time.

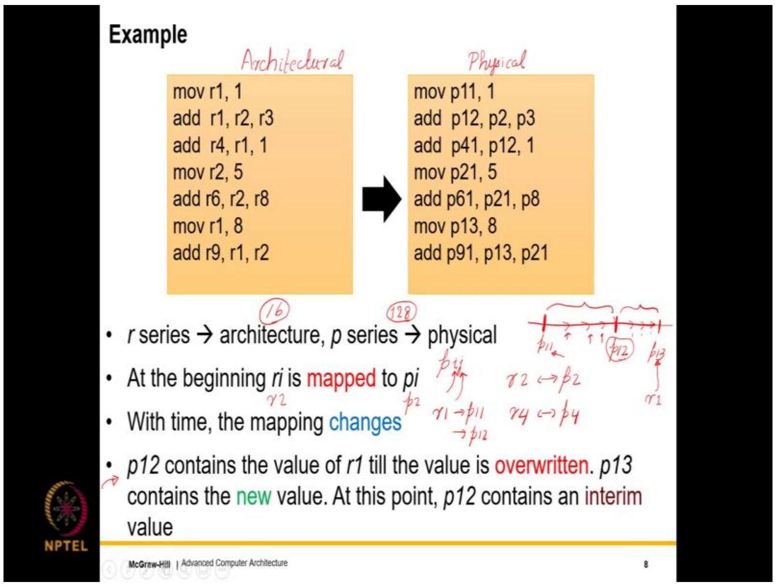
So, let say let us consider register number r4, it is mapped to let say the physical register p5, which means at that point of time r4 is not mapped to any other physical register and also at that point of time no other architectural register maps to physical register p5.

So, this means that at any instant 112 = (128-16), 112 physical registers are unmapped. They so they are simply not mapped. They either are empty in the sense that they do not contain a value or they contain a value in the sense they contain a previous mapping.

So, let us say r4 was mapped to p5, then we have a new instruction that writes to r4. So, in this case r4 might get mapped to p6. So, until p5 is freed, so we will discuss when exactly it is freed. What we actually see is that the value of p5 might still be used by instructions in slight and so p5 the register will still contain valid data till a certain point, but the most recent mapping will still be r4 to p6.

So, food for thought over here are typical burger that keeps on appearing. So, why are 112 registers unmapped? Well, this has been discussed two points at this point. So, they are unmapped primarily because we establish one to one mapping between the architectural registers and a subset of physical registers. See 16 architectural registers are mapped to 16 physical registers. We are left with 112 physical registers that are currently not mapped.

(Refer Slide Time: 11:59)



Example

Architectural

```
mov r1, 1
add  r1, r2, r3
add  r4, r1, 1
mov r2, 5
add r6, r2, r8
mov r1, 8
add r9, r1, r2
```

Physical

```
mov p11, 1
add  p12, p2, p3
add  p41, p12, 1
mov p21, 5
add p61, p21, p8
mov p13, 8
add p91, p13, p21
```

- *r* series → architecture, *p* series → physical
- At the beginning *ri* is mapped to *pi*
- With time, the mapping changes
- *p12* contains the value of *r1* till the value is overwritten. *p13* contains the new value. At this point, *p12* contains an interim value

McGraw-Hill | Advanced Computer Architecture 8

So, let us now discuss renaming in the context of an example. So, this will provide clarity to much of the discussion that we have been having up till now. So, let us look at two pieces of code. The first piece of code is written using architectural registers. The second piece of code is written using physical registers. So, the terminology that we shall adopt is that the r series will be used with architectural registers. So, registers r0 r1 r2 r3 etcetera, will all be architectural and the p series will be physical.

So, recaller assumptions, we are assuming we have 16 architectural registers in our system and 128 physical registers. At the beginning let us assume, so which is the default state that architectural register ri, where ri can be let us say r2, i = 2 is mapped to physical register pi which is let us say p2.

So, r2 is mapped to p2, r4 is mapped to p4 and so on. So, of course, with time the mapping will change. So, with time as we keep on getting new values from the free list and we can keep on updating the value of a register the mapping will change, it will continue to change. So, this of course will get captured in our, in the names of our rename registers. So, it is important to understand that the idea of renaming is basically to split the time line into versions or avatars of a register.

So, the specific convention that we shall use is that for physical registers. We will use a, we will use after it has been written to ones it will essentially we have the form p i j, where i is essentially the number of the architectural register it corresponds to and j is essentially the avatar or version number. So, for example, when you overwrite r1 -- p11 and then it will p12, p13 and so on.

So, it is important to understand that this is just a convention that we are using only for this slide and maybe a few additional slides only for the purpose of explanation right. This is not the scheme used by a hardware renamer, this is not the scheme used in a modem processor this is only being used for better and easier explanation.

So, what we do is that, let say for p11 that this with the instruction that writes to this register. We will then have a few more instructions which I am showing like this on the timeline that will be using p11, the current avatar. And then of course, we will again create one more avatar of r1 which is p12, again a few more instruction that we will use it. So, this can be considered as the lifetime of avatar p11 and then this can be considered the lifetime of avatar p12.

So, clearly we can mix instructions from these avatars, it does not matter. Primarily, because they use different physical registers. Only thing that matters is that these three instructions should execute after this instruction the writing instruction has executed, where this is the only two dependence, the rest can be mixed.

Once this is cleared, let us go to the fourth point. The fourth point basically says, that let us consider p12, it contains the value of r1 the second avatar, I will contain the latest value of r1 till it is overwritten. Once it is p13 will contain the new value, but note that at this point.

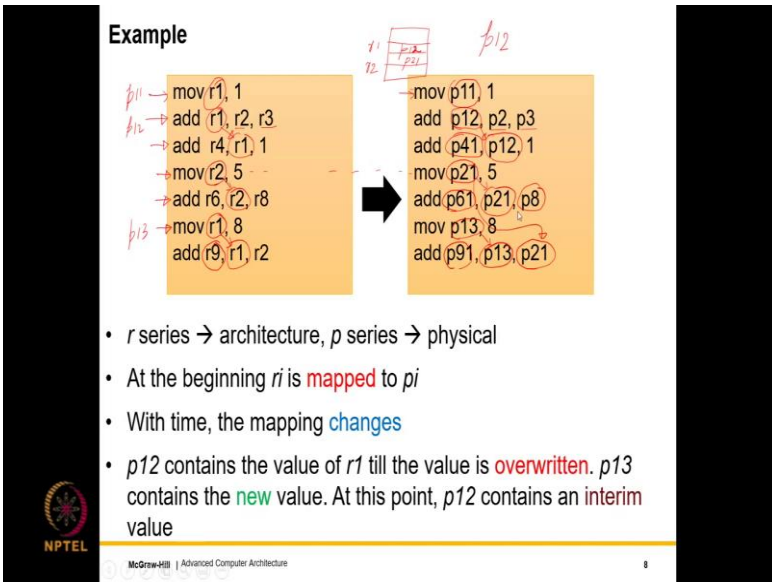
So, let us say p13 is written to over here at this point there will still be, there might still be several instructions that will be using the interim value which is p12 and these instructions might not have completed. So, these instructions are nevertheless in slide and we will have to wait for p12 in all the instructions that use the value of p12 to complete before we release p12. So that is one aspect of this discussion.

The other aspect is that for every architectural register will always have a physical register that contains a latest value and we will also have other interim values like p11, p12 etcetera, but we cannot really remove them, the reason being or release the physical registers because there might be other instructions in the pipeline that are also set to be in slide that would require these values.

And because we are executing out of order there will is very high probability of such instructions being there. So, a given this basic cursory discussion, let us proceed with understanding the example. So, let me remove some of that letter on the slide by clearing of the ink, then we can proceed to understand this example.

So, let us follow this instruction by instruction. So, the first instruction is mov 1 to r1. So, mov 1 to r1 will pretty much create the first avatar of r1 which is p11.

(Refer Slide Time: 18:15)



Example

```
mov r1, 1            mov p11, 1
add r1, r2, r3       add p12, p2, p3
add r4, r1, 1        add p41, p12, 1
mov r2, 5            mov p21, 5
add r6, r2, r8       add p61, p21, p8
mov r1, 8            mov p13, 8
add r9, r1, r2       add p91, p13, p21
```

p11
p12
p13
r1
r2
p12
p21
p12

- *r* series → architecture, *p* series → physical
- At the beginning *ri* is mapped to *pi*
- With time, the mapping changes
- *p12* contains the value of *r1* till the value is overwritten. *p13* contains the new value. At this point, *p12* contains an interim value

McGraw-Hill | Advanced Computer Architecture 8

So, since we are creating a first avatar, what we need to do is, we need to go the entry for r1 in the RAT table and that needs to be overwritten with p11. So, recall that in this case architectural registers only exist conceptually, they only exist in principle, they exist implicitly. They do not have a dedicated storage location. Now, let us come to the second instruction add r1 r2 r3.

So, in this case, we have write after write hazard, that is not a problem. We simply allocate this version of r1 a new physical register which is p12. Two captures the second avatar of r1 and of course, since r2 and r3 have not been modified before in this piece of code, we assume that they are the come from initial state. Hence, we read them from p2 and p3.

Now, we see our first read after write hazard which is that we use the value of r1. Well, which value do we use? We use the latest value. So, by the way when we modify the value of r1, when we create a new avatar of r1 we replace p11 over here with p12. So, we use the latest value which is p12.

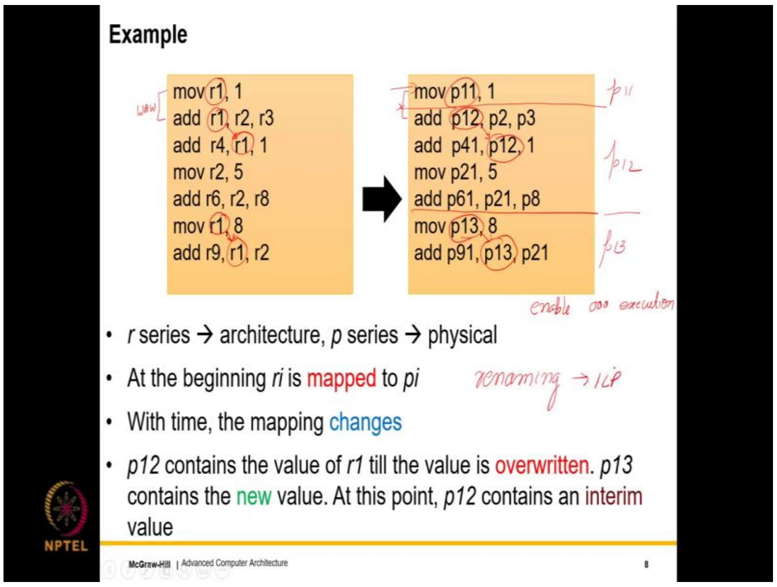
So, we see that there is a read after write dependence over here and there is a read after write dependence over here as well. Then, since we are modifying the value of r4 we are creating a new physical register, a new avatar of it, a new version of it and this becomes physical register p41. And, so this becomes a new version of this and so where does this this come from? This comes from the free list, again we move forward.

So, we write a new value at r2, again we create a new version of it. The time has come to use r2. So, similar to r1 what we do is we look up its latest value in the RAT table entry for r2. So, the RAT table entry for r2 would basically contain p21. So, we read p21 over here. And, since we are reading r8 for the first time we have not written to it turns on p8, we are modifying r6 for the first time, so this would be the physical register p61. Here again we are creating the third avatar of r1, we are writing to it for a third time.

So, what was the first avatar? It was p11 then p12. So, this was p11, this was p12 and this is p13. So, in this case we assign it p13 and then again we read p13 in the next instruction where we have a read after write dependence. And r9 we are writing to the first time, this p91.

See here is a fun part; what do we do about p21? Well, p21 is the mapping for r2. So, you just walk back in the sequence, see the last time that we wrote to r2 and at that point of time we had actually assigned it the physical register p21; hence, we use p21 over here and this becomes our transformed renamed sequence with physical registers. So, let me now erase the ink on the slide and focus on a critical aspect.

(Refer Slide Time: 22:25)



Example

```
WAW  mov r1, 1                    mov p11, 1              p11
     add r1, r2, r3               add p12, p2, p3
     add r4, r1, 1                add p41, p12, 1         p12
     mov r2, 5          ➡         mov p21, 5
     add r6, r2, r8               add p61, p21, p8
     mov r1, 8                    mov p13, 8              p13
     add r9, r1, r2               add p91, p13, p21
```

enable ooo execution

- *r* series → architecture, *p* series → physical
- At the beginning *ri* is mapped to *pi* renaming → ILP
- With time, the mapping changes
- *p12* contains the value of *r1* till the value is overwritten. *p13* contains the new value. At this point, *p12* contains an interim value

McGraw-Hill | Advanced Computer Architecture 8

So, let me just annotate all the accesses to r1 and similarly, annotate them over here in the renamed sequence. So, what do we see? What we see is we have read after write dependencies that we have. We have them over here also because renaming does not remove them, but renaming has very effectively removed write after write dependencies. They are over here or they are not here or there is no write after write.

There is a write after read dependency as you can see over here, between this write of r1 and this read of r1 nothing of that sort is visible over here. So, the write after read problem is also gone. Furthermore, what we have done is, we have divided the timeline into different avatars of r1. So, this is the first avatar, this is second, this is the third.

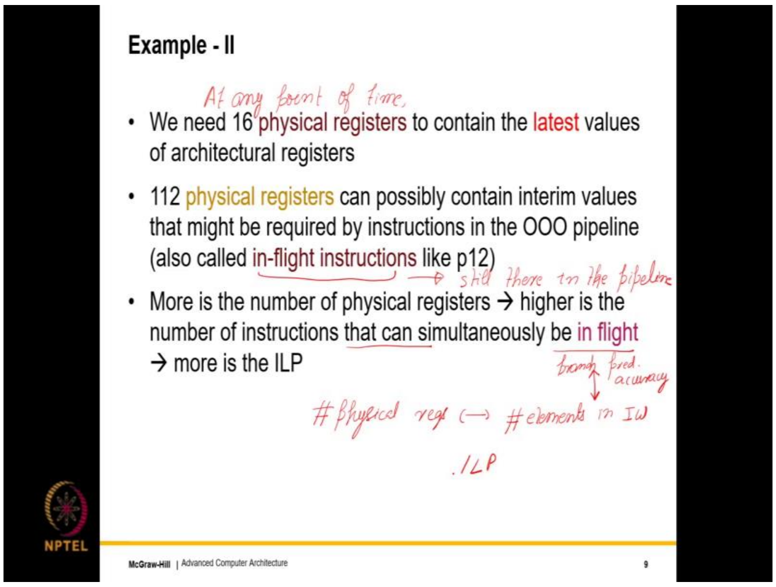
So, as far as we are concerned, these are actually three separate registers. That is the way that we see it in the physical register sequence, that it is actually three separate registers conceptually and nothing stops us from taking instructions in its kind of one avatars instructions and intermingling them with instructions of another avatar subject to data dependence constraints of course, with respect to the other operands. Which is but the other operands are not an issue, then we can happily intermingle them. That is not an issue. Why?

So, this gives us more ILP and why did we do it? Well, the reason we did it is to enable out of order execution, because write after write and write after read WAW and WAR hazards would have been a tremendous problem. So that is the reason to enable out of order execution. What we do is that we have effectively transformed this into a sequence that has only i w has its and then you can clearly see the increase in ILP.

And, you know one example one note worthy example would be that this instruction for example, can be moved anywhere. So, this instructions can be moved anywhere.

So, this is by the way, this is dead code this is dynamically dead code in the sense that in statically also you can see that lot of compilers would remove them or assuming it did not, we can happily move this instruction anywhere we want, it will not cause any correctness issue. So, this is the main aim of renaming that renaming helps us enhance the amount of ILP, enhanced ILP.

(Refer Slide Time: 25:31)



Example - II

At any point of time,

- We need 16 physical registers to contain the latest values of architectural registers
- 112 physical registers can possibly contain interim values that might be required by instructions in the OOO pipeline (also called $\underbrace{\text{in-flight instructions}}$ like p12) → still there in the pipeline
- More is the number of physical registers → higher is the number of instructions that can simultaneously be in flight → more is the ILP

branch pred. accuracy ↕

physical regs ⟷ # elements in IW

. ILP

McGraw-Hill | Advanced Computer Architecture 9

So, what was the entire crux of our discussion? The entire crux of our discussion was that architectural registers exist only in principle at any point of time. So, let us prefix this with at any point in time.

We need 16 physical registers that will contain the latest values of architectural registers, something like latest instantaneous snapshot. The remaining 112 physical registers either maybe empty or can possibly contain interim values, the rather in flight instructions will require. For the moment and in flight instruction is defined as something as defined as instruction that is still there in the pipeline.

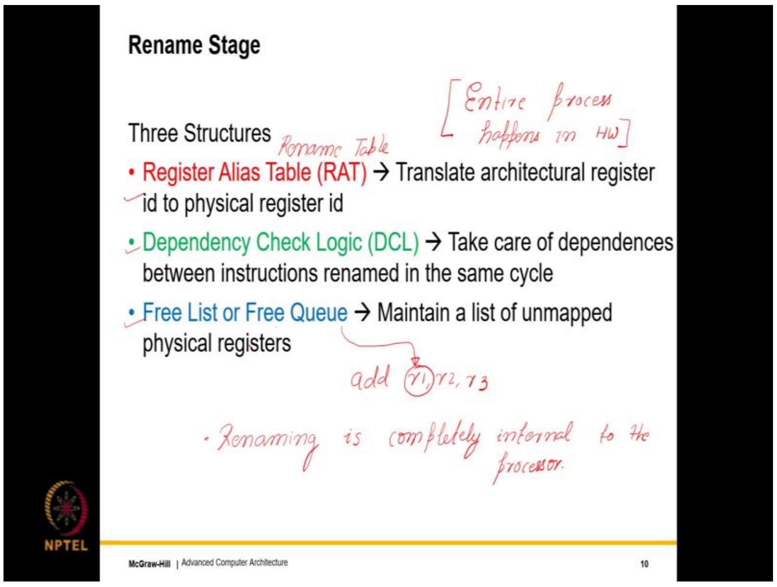
So, we will kind of enhance this definition later, but let us say that this is something there in the pipeline, and clearly more is the number of physical registers well higher is the number of instructions that can simultaneously be in flight.

Because, the number of physical registers is in a sense connected with how many elements we can have in our instruction window, because that one time if we run out of physical registers, we simply cannot rename new instructions or rename will stall or decode will stall or fetch will stall. So, the front part of the pipeline.

So, this part is actually called the front end. So, the front end of the pipeline will actually stall. So, and we have also seen that the number of elements in the instruction window is related to the branch predictor accuracies. See, nicely see how everything is kind of connecting together and falling in place. So, they are all kind of intertwine interconnected and number of physical registers also important.

So, any designer has to look at all three of these factors before sizing up the processor or the pipeline. So, in this case, clearly all of them together determine the amount of visible and exploitable parallelism, the amount of ILP and this is something that we will use in the subsequent slides to analyze the performance of an out of order machine.

(Refer Slide Time: 28:17)



Rename Stage

[Entire process happens in HW]

Three Structures

Rename Table

- Register Alias Table (RAT) → Translate architectural register id to physical register id
- Dependency Check Logic (DCL) → Take care of dependences between instructions renamed in the same cycle
- Free List or Free Queue → Maintain a list of unmapped physical registers

add $\boxed{r1}$, r2, r3

- Renaming is completely internal to the processor.

McGraw-Hill | Advanced Computer Architecture 10

So, for the purpose of doing renaming, we need three hardware structures which are as follows. So, the first is the register alias table also called the RAT table. So, what is its job? Its job is to translate architectural register ids to physical register ids. Then we have another hardware structure. So, it is called a dependency check logic. So, the idea here is like this that we might be renaming multiple instructions per cycle.

So, let us say we are renaming four instructions per cycle. So, what can happen is, so there might be dependencies between these instructions and if there are dependencies between these instructions, it will be rather difficult to take care of them. So, clearly since they are in since they are getting renamed in parallel, the rename table or also called the register alias table. So, we will also called as the rename table by the way, is something that cannot be used. We will see why it cannot be used. So, we will need this piece of hardware.

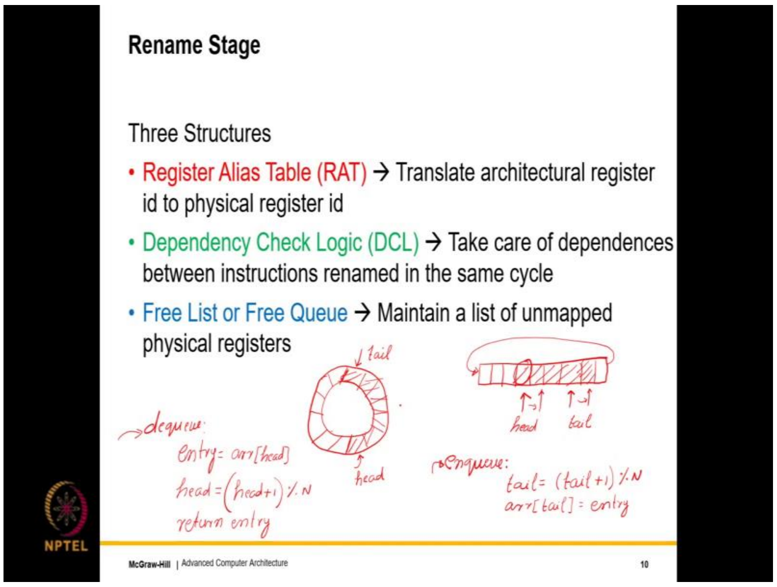
When we are renaming an instruction of this type, add r1 r2 r3, what will happen is that for register r1, we need to access a free list or a free queue and get one of the unassigned physical register ids and assign it to r1. So, of course, we had a very simple convention in the last slide, in this I think slide before the last one, but this convention is only for explanation, this is not what hardware would actually use.

What hardware would actually do is, that you would have a free list or a free queue and this would maintain a list of unmapped physical registers. So, whenever we need one, we just lock one from there. We just stretch one from there.

So, I will repeat this several times that the entire process happens in hardware, entire process happens in hardware completely without the knowledge of software. So, software is not aware that renaming is happening. Renaming is completely internal to the processor.

So let me write it, renaming. And so, how is renaming achieved? While renaming is achieved in hardware by hardware using these three structures. So, we will use circular queue for doing many things. So, the free list is one example of using a circular queue. So, circular queue is a data structure which all of you should brush up.

(Refer Slide Time: 32:00)



tail
head
head
tail
NPTEL

Rename Stage

Three Structures

- Register Alias Table (RAT) → Translate architectural register id to physical register id
- Dependency Check Logic (DCL) → Take care of dependences between instructions renamed in the same cycle
- Free List or Free Queue → Maintain a list of unmapped physical registers

```
dequeue:
    entry = arr[head]
    head = (head+1) % N
    return entry

enqueue:
    tail = (tail+1) % N
    arr[tail] = entry
```

McGraw-Hill | Advanced Computer Architecture 10

So essentially, what is it we will circular queue conceptually looks something like this where we basically have a circle and we have entries. So, some of these will be filled, some of these will be unfilled, but let say these entries are filled. So, these are all the filled entries.

So then, every queue will have a head and a tail. So, we always enqueue at the tail and all we always dequeue at the head. So, let us say this can be the tail and this can be the head.

So, whenever we want to add new entries, what we do is that we keep adding here and whenever we want to detriment what do we do? We keep removing them from here. So, how is the, so what we can think of is a circular queue can be thought of as something conceptually like this. So, well how is this actually implemented? Well, the way that this is actually implemented is something like this. That we take an array or regular array all.

So, it so arrays of course, software concept we have a similar hardware concept where we can just think of it as I of flip flops and the array of registers. So, then what we do is that we have two pointers, two instantaneous pointers. So, let us say one of them can be head, one of them can be tail. So, let us say that the entire this part of the array is full. So, which means that this is the queue.

So, this index can be head, this index can be tail. So, when we add a new entry, we will add it over here. And of course, a tail pointer will move to here, and when we dequeue an entry then the head pointer will move here and this must be the entry that is dequeue.

So, now the question comes up is what if we reach the end of the array? Well, if we reach the end of the array, we kind of wrap around and this is why this functions have a circular queue because of the motion of wrap around. So, the enqueue operation where essentially we add an entry can simply we say said as.

So, we of course add an entry. So, when we add an entry, this is what we do. I am assuming that the size of the array is N. So, increment in, but then I also take care of the wrap around issue. See, if I add 1 I also compute this modulo N, the size of the array and then let say if the name of the array is array, then at this point I add the entry. Whatever entry I want to add; I add it over here.

So, this is the simple pseudocode for enqueue. So, let me also write the pseudocode for dequeue. So, dequeue I am assuming that there is at least one entry in the array. So, if there is no entry if it is empty that is easy to check, I can always have a small counter that maintains a count of the number of entries in the array.

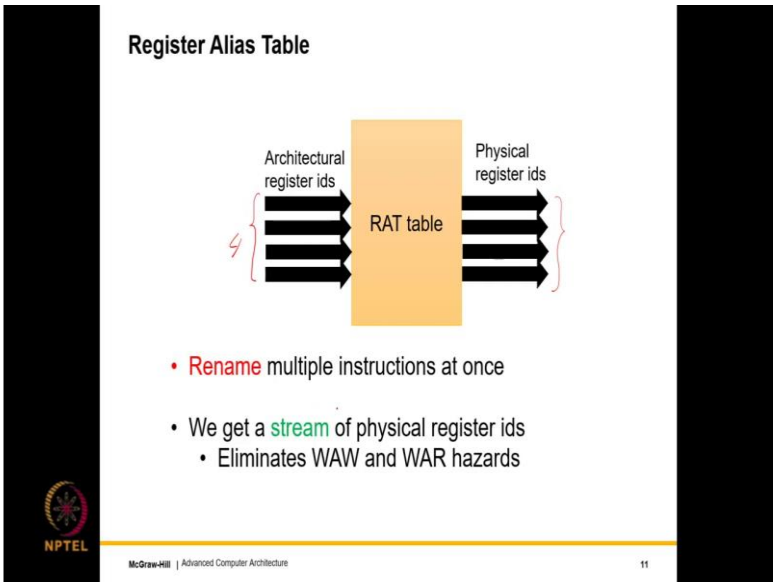
So, in this case what I do, so I do this exactly this this logic in hardware. It is not a software logic; it is a hardware logic. So, I take a look at the current head. So, the current head is an entry that I would like to dequeue. So, I store it over here and then this is what I do. And then of course, I return the entry. So, there is no return in hardware, but this is essentially the entry that I use.

So, this is a very simple logic in hardware that can easily be done and typically the size of the array here is the power of 2, so, computing this operation, the modulo operation is very easy and also incrementing is easy. So, this is a simple way of implementing a circular queue in hardware.

So, we will see that a lot of structures, a lot of hardware structures in architecture particularly are made of circular queues and the logic for a circular queue is like this which is rather simple alright. And, it suppose two simple operations enqueue and dequeue and of course, we can add another operation if we wish to say if the circular queue is empty or not.

We can either one option is we can compare the head and tail pointers, but a far simpler option is we have some simple counter that maintains a count of the number of entries in the queue. So, that is a far easier far simple, far most straight forward option.

(Refer Slide Time: 37:15)



Register Alias Table

Architectural register ids
RAT table
Physical register ids
4

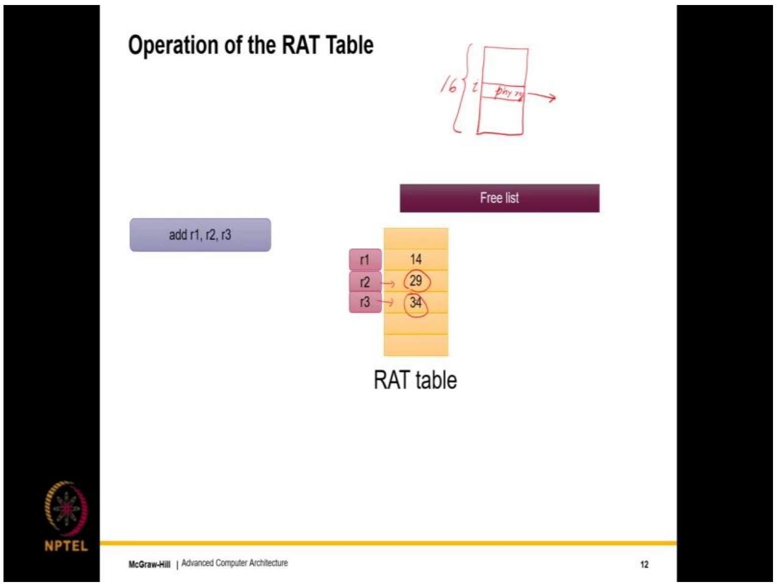
- Rename multiple instructions at once
- We get a stream of physical register ids
  - Eliminates WAW and WAR hazards

McGraw-Hill | Advanced Computer Architecture 11

So, the register alias table or the RAT table essentially takes in a set of architectural register ids. So, if we are renaming 4 instructions per cycle, so they can have a maximum of 8 register source operands. So, we will actually have 8. So, here I am assuming that we have a total of 4 register source operands.

So, which means that we are renaming two instructions per cycle. So, then the number of an inputs is equal to the number of outputs. So, outcome 4 physical registers ids. So essentially, what are we doing? We are getting a stream of physical registers ids with implicitly the WAW and WAR hazards removed.

(Refer Slide Time: 38:02)

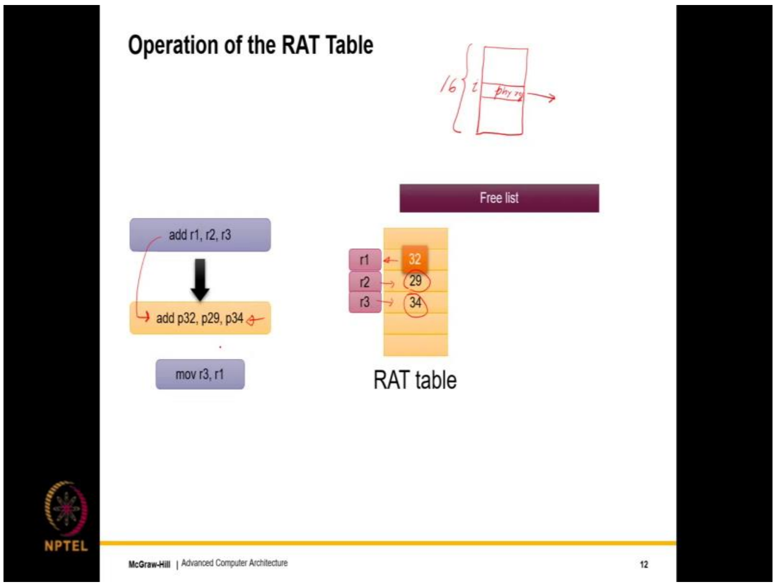

Operation of the RAT Table

How does this work? Well, the way that this works is that this is a simple table the RAT table, is a simple table. So, the structure is like this, it is a simple table, where we basically have if there are 16 architectural registers we have 16 rows, for the ith architectural register, we just access the ith row. And here, we will have the id of a physical register the one it is mapped to we just need to read that and that is done. So, let us explain with an example.

So, let us say that I want to rename the instruction add r1 r2 r3. So, what I do is, that I go to the RAT table. So of course, r1 is a destination I am not concerned, but what I am concerned is about r2 and r3 say, I go to the RAT table, so RAT table by the way is also called the rename table. So, then I access the entries for r2 and r3. And how many entries are there?

If there are 16 architectural registers, then there will be 16 entries one for each architectural register. So, as we have discussed earlier also, architectural registers in this case exist only in principle, only implicitly, only conceptually. So, the mapping is to 29 and 34.

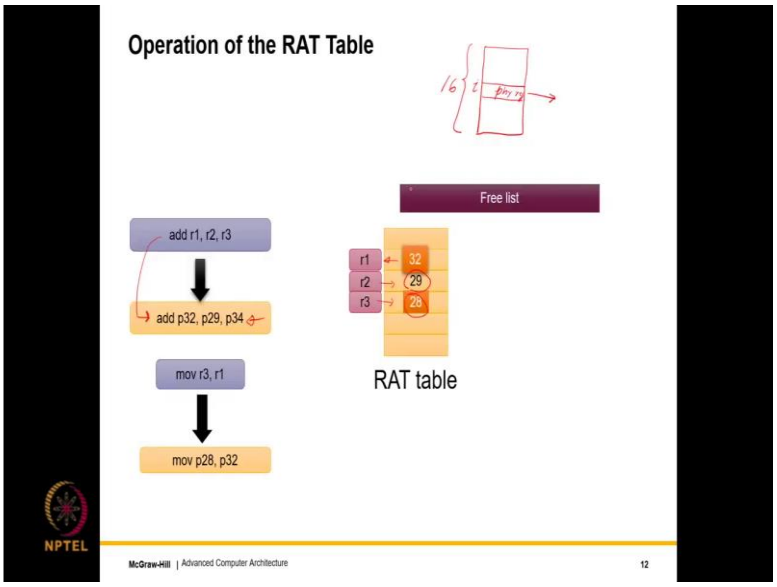
(Refer Slide Time: 39:33)



So, what I do is that I note these two and in addition I fetch from the free list register number 32, which let us say was the head of the queue and I map it to r1. So, this register 32 was unmapped, it was free. So, the renamed instruction becomes add p32, physical register 32, where this was just assigned to the instruction. It came from the free list and then p29 and p34. Why p29?

Well, that is because my rename table says that r2 is mapped to p29 and my RAT table says that r3 is mapped to p34. Hence, this instruction gets renamed internally, internally within the code done by hardware unbelongs to software, this is what the instruction becomes. So now, let us keep, let us proceed. So, let us say the next instruction is mov r1 into r3.

(Refer Slide Time: 40:43)

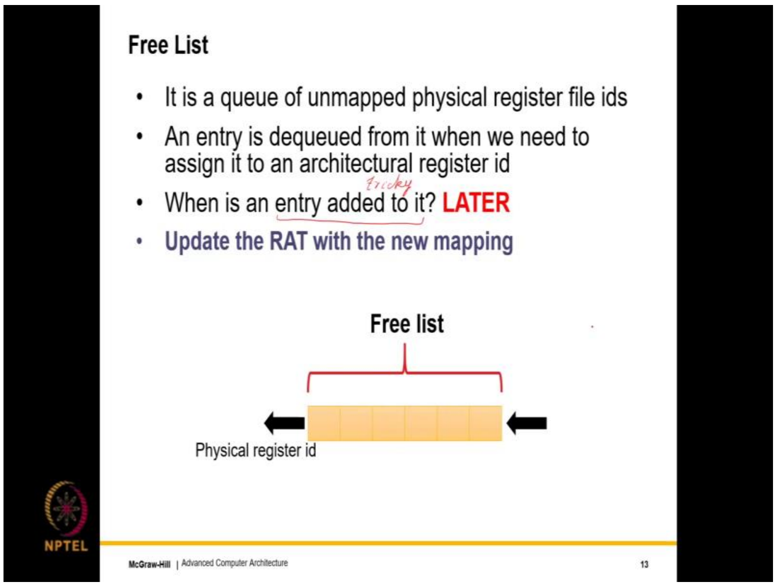


Operation of the RAT Table

McGraw-Hill | Advanced Computer Architecture 12

So, first for r3, we try to fetch free register from the free list. So, let us say this is 28, for r1 all that we need to do is we need to read its value and its value would already be return to the RAT table. Recall that in the, for the previous instruction we got it from the free list. So, this becomes the renamed instruction p mov p28, p32.

(Refer Slide Time: 41:16)



Free List

- It is a queue of unmapped physical register file ids
- An entry is dequeued from it when we need to assign it to an architectural register id
- When is an entry added to it? **LATER**
- **Update the RAT with the new mapping**

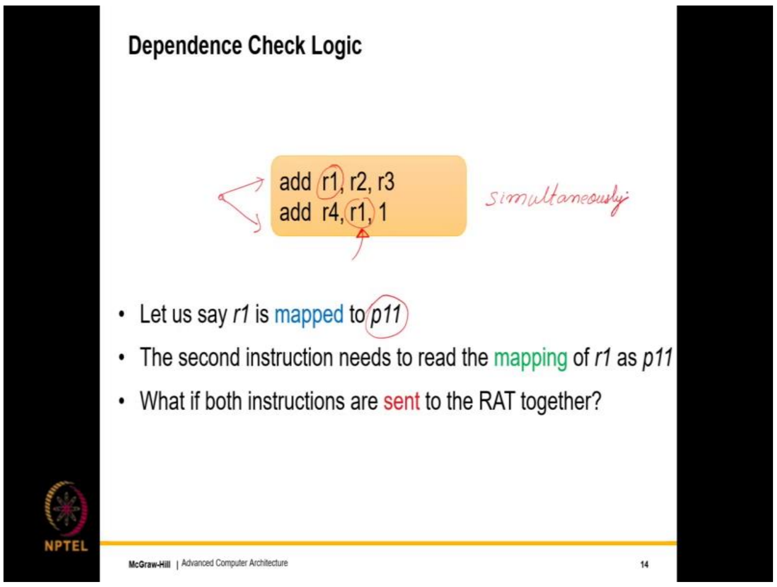
tricky
Free list
Physical register id

McGraw-Hill | Advanced Computer Architecture 13

So, what again is if free list? Well, if free list is essentially a queue as we have seen it is not a normal queue, is a circular queue of unmapped physical register ids, when we need to assign it we dequeue an entry. When is an entry added to it? Well, this is a very tricky question. It is an important question yet at the same time it’s a tricky question. So, when do we add it? Well, we will see later.

And then of course, we update the RAT with new mapping as we did over here, that when we got the new mapping for r1, we updated the RAT with 32 and when we got the mapping for r3 we updated it with 28. So, the free list conceptually is essentially like a queue, but internally its implemented like a circular queue to take, because we have to take into account issue of the wrap around.

(Refer Slide Time: 42:11)



Dependence Check Logic

add r1, r2, r3
add r4, r1, 1

simultaneously

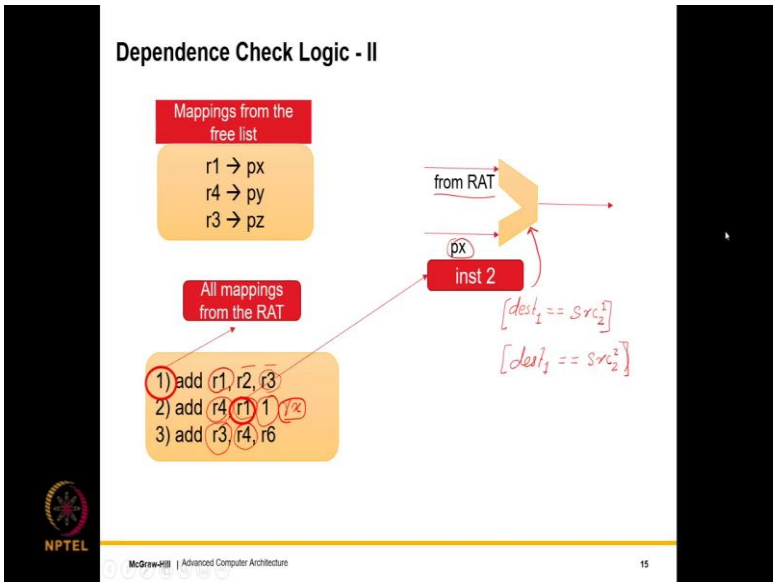
- Let us say *r1* is mapped to *p11*
- The second instruction needs to read the mapping of *r1* as *p11*
- What if both instructions are sent to the RAT together?

McGraw-Hill | Advanced Computer Architecture 14

So, let us now take a look at the dependence check logic. So, assume that we are renaming these two instructions together at the same time. So, we have we are renaming add r1, r2, r3 and add r4, r1, 1. So, in this case, what we have is, we have a read after write dependency bit on the register r1. So, in register r1 is mapped to register p11 then the second instruction will have to read the mapping of r1 as p11, but what is the problem? That the problem is both are being sent to the renaming process, the renaming stage the RAT table, at the same time.

So, when we will actually read the value of r1, it will not be the value the latest value that is being written in r1s entry which is p11, it might be something else. So, there is a chance or probability that we might actually miss the value which means that instead of p11 the second instruction might actually get some other physical register, which r1 was mapped to in some previous cycle. So, this is actually going to be a problem, because both of them have to be renamed simultaneously. Hence, we need what is called dependence check logic to take care of this.

(Refer Slide Time: 43:44)



So, let us look at the slightly more complicated example, where we are actually renaming three instructions at the same time. So, we have a read after write issue over here and we also have a write after read renaming issue over here. So, given all of this dependencies they need to be handled correctly.

So first, let us assume that for all the destination. So, we have three destinations here r1, r4 and r3. So, we accessed the free list, so of course, we access the free list in parallel. So, let us assume that the entry is that we get from the free list are px, py and pz. So, if they are px, py and pz, let r1 be mapped to px, r4 be mapped to py, r3 to pz.

So, all the mappings from the RAT which is the mappings for r2 and r3 will be used by instruction 1, because it is the first instruction in this three instruction bundle. So, renaming the first instruction in that sense is very easy, because we just read the values that we get from the RAT table, but renaming the second instruction is hard, the reason being that is source operands can come from the output of renaming the first instruction.

So, let us consider the first source operand of instruction 2 which is r1. So, we actually have two choices. So, the first choice is that we can taken the value from the RAT table entry of r1 which we know that for at least this case will give us a tail value, it will give us a wrong value.

Because that has not been populated as of now, which is basically because the first instruction; instruction 1 are r1, r2, r3 is still being renamed is being renamed in parallel simultaneously. When nevertheless this is one of the options that we have. The other option that we have is we actually take the value px which has been given to us by the free list. So that option we also have.

So, what we need to do is, we need to choose between px and the value from the RAT. So, whenever we have a choice of two values, we need a multiplexer in hardware. And how do we exercise this choice? Well, what we do is, that we essentially compare if the

des1= src$\frac{1}{2}$.

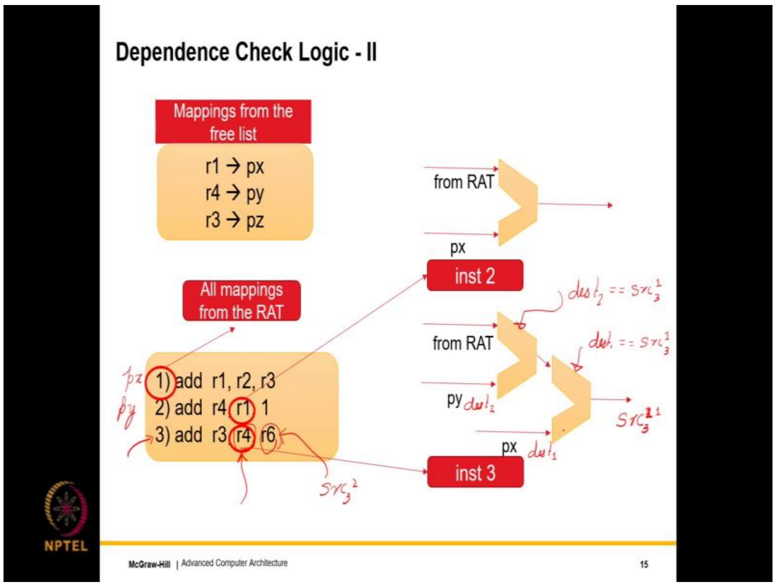
So, let us say the source, so in this case it is the first source, if this is a first source of instruction 2. If this comparison is successful, which means they are actually the same register, then we use the value that was assigned to the destination of instruction 1. So, this is what we use.

So, this is the kind of simple logic. So, when we are actually renaming these three instructions at the same time, at run time we do not know what those instructions will actually be, but that is the reason we need to fabricate these multiplexers in hardware and keep them ready, and this multiplexer clearly has a choice between px. So, what is px? In this case px is the physical register that has been assigned to the destination of instruction 1.

So, that is one option and the other is of course, what we read from the RAT. So, we need to keep the multiplexer ready. Which one we actually take in run time depends on the result of this comparison. And, the this will be the one that is, that r1 is mapped to. So, in this case of course, other entry over here is an immediate we do not have to rename it, but let us assume this was some other register r x.

So, we would need two fabricators similar multiplexer based design for the second source operand, and here also the comparison function will be something like this source. The second source of the second instruction. So, this was a simple case. So, let us now look at instruction number 3, where it gets more complicated.

(Refer Slide Time: 48:41)


Dependence Check Logic - II
Mappings from the free list
r1 → px
r4 → py
r3 → pz
All mappings from the RAT
1) add r1, r2, r3
2) add r4, r1, 1
3) add r3, r4, r6
from RAT
px
inst 2
from RAT
py
px
inst 3
NPTEL
McGraw-Hill | Advanced Computer Architecture
15

So, for instruction number 3, which is this instruction, for let us say for r4 we have a choice between the values that were assigned in the previous 2 instructions of the bundle px and py. So, note that we have still so they are also getting renamed. So, when nevertheless we have a choice and of course we have a choice from the rename table, so we have three options. This is exactly what you get to see.

So, what we have is, we have three options; one is the entry from the renamed table, one is py which is the destination of instruction 2, and we have px which is the destination of instruction 1 alright. So, for so we will have two multiplexers, we can have a multiplexer with three inputs, but I have drawn two multiplexers here.

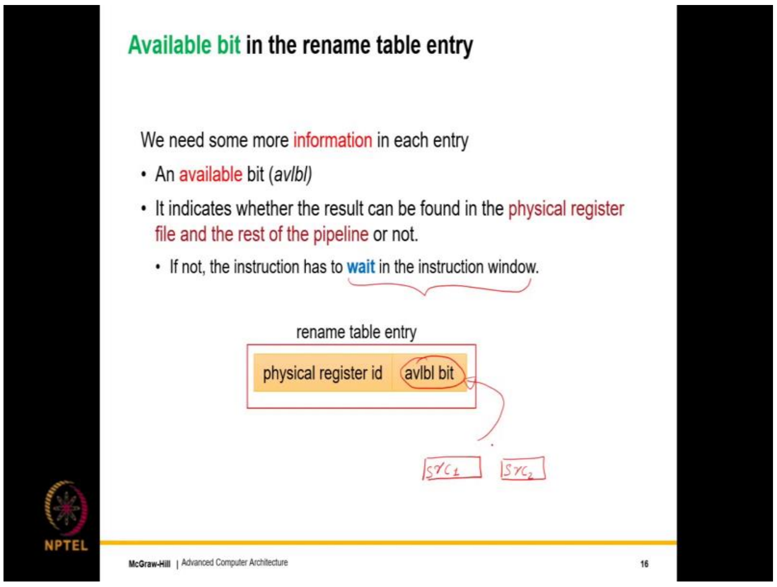
So, we will have this for of course, the second source of instruction 3. So, what we are doing is if this is instruction 3 and this is the second source. So, this is what we get. And, so what is py? Well, the py is the destination of instruction 2 and px is the destination of instruction 1.

So, the multiplexers will have control signals and the control signal in this case is rather simple. All that we need to do is, we need to check if this two are equal, which means if the des1= src$\frac{1}{3}$. this is the first source the first source of instruction 3. Similarly, here, what we need to do is we need to check if the des1= src$\frac{1}{3}$. That is what we need to check.

Once we have done this checks and of course, these check these checks can happen in parallel, then the appropriate value from the multiplexers will kind of flow out and that will be the final output of the dependency check stage which will either take the entry from the rename table or one of the destinations of the previous two instructions, if there is a match that is and that will be the final rename value for r4. We will have a similar circuit for the second source of instruction 3, which is for r6.

So, what we have to do? Well, we will have to create such multiplexers for each of the sources other than the sources for the first instruction in the bundle and then make a choice based on these comparisons. Of course, if we are renaming 4 5 or 6 instructions then we will have more multiplexers. And of course, passing a signal through so many multiplexers is a slow operation this places a natural limit on the number of simultaneous instructions that we can rename per cycle.

(Refer Slide Time: 51:51)



Available bit in the rename table entry

We need some more information in each entry

- An available bit (*avlbl)*
- It indicates whether the result can be found in the physical register file and the rest of the pipeline or not.
  - If not, the instruction has to wait in the instruction window.

rename table entry

physical register id	avlbl bit

src_1 src_2

McGraw-Hill | Advanced Computer Architecture 16

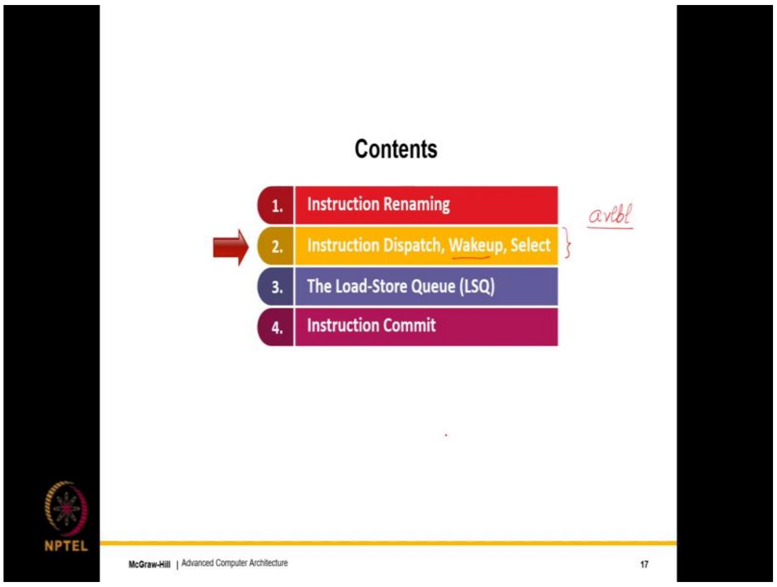
We need something more. At this point we are done with the rename stage, we have talked about the rename table also called the RAT table, we have talked about the free list and we have talked about the dependence check logic. So, we also need to add one bit to each entry called the available bit. This indicates whether the result can be found in the physical register file and the rest of the pipeline. Whether the result is ready or it is not ready.

So, this is a single bit which is there along with the physical register id which you read from the rename table entry. And, essentially what this says is, that if let say I proceed through the pipeline, the instruction proceeds through the pipeline can it just go to the register file and read the value or does it have to wait? If it has to wait for this particular source operand, it is going to wait for it in the instruction window.

So, we will see more uses of this available bit in the subsequent slides, but what is clear is that this information needs to be available at this very point, this very moment. Because it will tell the instruction that look, if an instruction has two source operands; source 1 and source 2 are they ready or not. In the sense have their values been computed or not.

If they have been well and good, this instruction can just enter the pipeline and execute the pipeline means the rest of the pipeline and execute. If it is not ready then instruction needs to wait in instruction window and it needs to wait for its operands to become available; which means, whoever is producing those operand registers needs to complete.

(Refer Slide Time: 53:48)



Contents

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

avlbl
NPTEL
McGraw-Hill | Advanced Computer Architecture
17

So, we have reached a very interesting juncture over here. So, we have finished renaming. So, we have finished renaming with a very important concept which current is like a thought in the I that is the available bit with every entry. This essentially tells the entry tells the instruction whether its source operands are ready or not, which will be used by the subsequent stage bit time for making instructions wait.

So, they will wait once they get all their operands, they will wake up, which means they are ready to execute. If multiple instructions wake up at the same time, but let us say we have finite number of resources, we need to select a subset and they need to be executed.