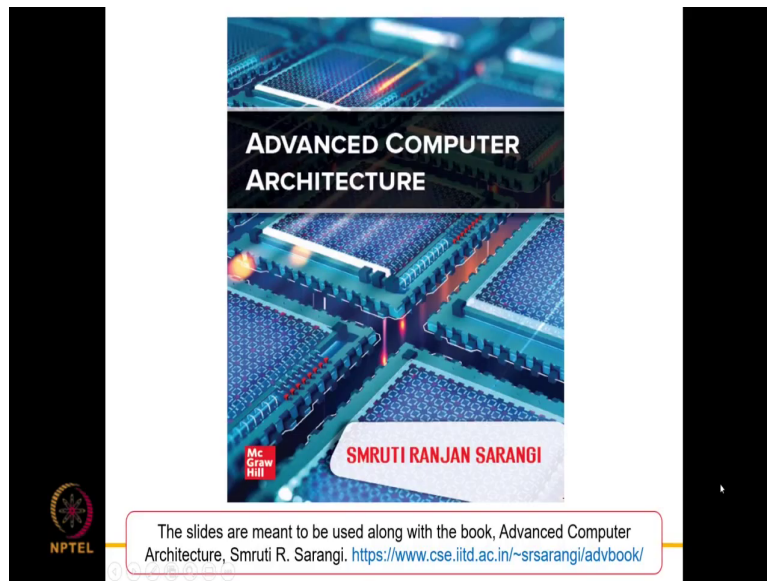


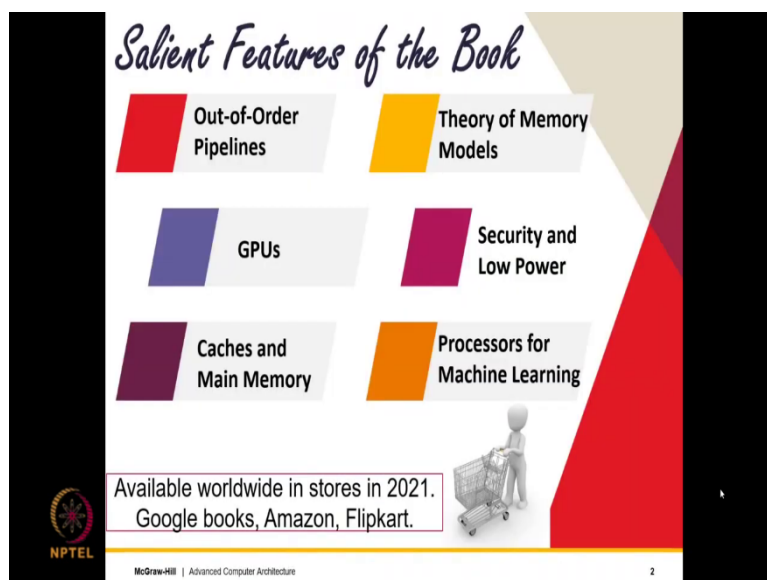
**Advanced Computer Architecture**  
**Prof. Smruti R. Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Lecture - 04**  
**Out-of-Order Pipelines Part - III**

(Refer Slide Time: 00:17)



(Refer Slide Time: 00:23)



(Refer Slide Time: 00:52)

## Nature of Dependences



In the last lecture of this chapter, let us look at the nature of dependencies in an Out-of-Order Pipeline and the problems that these dependencies can cause.

(Refer Slide Time: 01:05)

### Dependences between Instructions

#### Program Order Dependence

```
→ mov r1, 1  
→ mov r2, 2
```

*static* *dynamic*  
*for (i = 1; i <= 500; i++)*

- One instruction appears after the other in program order

*dynamic*

The program order is the order of instructions that is perceived by a single cycle in-order processor executing the program.



So, we have looked at the Program Order Dependence. So, this is rather simple so this is essentially the order of instructions that appear in the original program. So, this is one instruction and this is the instruction after it.

So, this is a program order dependence and we have already distinguished between the terms a static instruction and a dynamic instruction. So, static instruction a classic example would

be let us assume we have a for loop. So, the for loop will have a set of assembly instructions with a jump of course a backward jump.

So, how do we detect a loop in hardware? Well, it is simple any jump with a negative offset is essentially well not most of the time it points to a loop, because we go back again we start executing again we go back, it is a loop.

So, if you see here there are 1 2 3 4 5 there are 5 static instructions, but if the loop has 100 iterations it is 500 dynamic instructions. So, here one instruction appears after the other in program order and we always talk about the dynamic sequence in our discussion. So, the program order is simple we have seen that no great no major issues.

(Refer Slide Time: 02:41)

### Data Dependences


*true*  
RAW → Read after Write Dependence (True dependence)

1. mov(r1) 1

2. add r3, (r1), r2

*producer → consumer*

- It is a **producer-consumer** dependence.
- The earlier instruction produces a **value**, and the later instruction reads it.



NPTEL

McGraw-Hill | Advanced Computer Architecture

46

Now let us discuss Read after Write Dependencies. So, these are the dependencies that were causing a lot of issues in in-order pipelines. So, in order pipelines basically this was a classic producer consumer dependency. So, the first one is the producer instruction and the second instruction is the consumer instruction. So, in this producer consumer dependence we found that the consumer always has to execute after the producer.

So, this order you need to maintain both in an in order pipeline as well as in an out-of-order pipeline. So, hence this is called a True dependence, because there is no escaping the producer executes first and the consumer executes next.

(Refer Slide Time: 03:54)

### Data Dependences - II

WAW → Write after Write Dependence (Output dependence) *finite number of registers*

```

1  mov r1, 1
2  add r1, r2
            
```

*reorder*

• Two instructions **write** to the same location

• The later instruction needs to take effect after the former

$r_1 = 3 \times$

$r_1 = 5$

So now, let us move on to two other kinds of dependencies which were not causing an issue in order pipelines, but they will cause an issue in out of order pipelines. So, consider this piece of code whether write to the same register r 1. So, essentially the destination is the same, in this case, since they write to the same location and of course this is the program order it is not possible to reorder these two instructions, we cannot reorder them.

See even if we were to execute them out of order the way the instruction stand we cannot reorder, because let us say this is writing 3 to r 1 and the next instruction is writing 5. And the first is instruction 1 then 2 if you execute instruction 2 first and then instruction 1, then the final value of r 1 = 3 which of course is wrong. The final value of r 1 = 5.

Which means that in all out of order executions we cannot break this write after write order which is also or a write after write dependence which is also called an output dependence; this cannot be violated. So, this of course, is not a true dependence we will see why in the course of the next few slides, it is actually what is called a false dependence and let us look at this closely.

If we look at this closely what we will actually get to see is that, the reason we have such dependencies in assembly code is basically because we have a finite number of registers. See if I had an infinite number of registers somehow then instead of sending the output r 1 I could have send it to r 20. Say every time if I use r 1 whenever I have r 1 if I actually assign it to a new register then I will never have an output dependence.

But since I have a finite number of registers I need to reuse registers this is where the output dependence comes in. So, this is essentially an artifact of the finite number of registers.

(Refer Slide Time: 06:35)

**Data Dependences - III**

*[Write once  
read many times]*

WAR → Write after Read Dependence (Anti dependence)

1: add r1, r2, r3 *R*

2: add r2, r5, r6 *W*

*finite number of registers.*

- Earlier instruction **reads**, later instruction **writes**
- The later instruction needs to execute after the earlier instruction has read its values

NPTEL

McGraw-Hill | Advanced Computer Architecture

48

Let us look at the next kind of dependence which is a write after read dependence it is also called an anti-dependence. So, let us look at the first instruction 1, so in instruction 1 we are reading from register r 2 and in instruction 2 we are writing to register r 2. So, the first is a read and the second is a write.

So, we have a Write after Read. So, for similar reasons we cannot reorder these two instructions, if we reorder them we will end up with the wrong value it is possible where the value that instruction 2 writes instruction 1 will read it.

Which of course is not correct it is wrong and this again arises this is known as anti dependence, this again arises because we have a finite number of registers. This is also an artifact of this problem that we have that, we have a finite number of registers.

So, in this case, also if let us say that we have some methods some mechanism, where every time we use we never repeat registers in the sense we write to register once and we keep on reading from it. So, essentially if we have this model write once read many times.

So, we will actually not have both WAW output dependencies and anti dependencies and the reason for that is at every time we write to a register it will actually be a new register. So, this requires an infinite number of registers which of course is not feasible, but we will see that we can do something to get rid of both output dependencies as well as anti dependencies. So, this is something that we need to do and we will see.

(Refer Slide Time: 09:14)

**Control Dependencies**

```
beq .label  
.....  
.label  
    add r1, r2, r3
```

*predict branches*  
*[wrong path]x*

- The *add* instruction is **control dependent** on the **branch**(*beq*) instruction
- If the branch is **taken** then only the *add* instruction will **execute**, not otherwise

NPTEL  
McGraw-Hill | Advanced Computer Architecture  
49

So, now let us come to the next category of dependency that we have seen they are known as control dependencies. So, here well it is rather simple we jump to a label and when we jump to a label we execute instructions after it. So, even an out of order pipelines if we do not predict the branch correctly. out of order pipelines of course we have to predict branches.

If we do not predict branches correctly, we stand to fetch instructions from the wrong path. And wrong path instructions will pollute the instruction window and as we have seen they

have to be discarded they cannot be executed, so they cause a lot of issues. So, that is why we needed a very accurate branch predictor. So, out of order say in out of order issues control hazards are actually even more important much more important than they were in order processors.

(Refer Slide Time: 10:30)

The slide is titled "Basic Results" and contains two orange boxes with text. The first box states: "In-order processors respect all program order dependencies. Thus, they automatically respect all data and control dependencies." The second box states: "OOO processors respect only data and control dependencies." Below the text, there is a handwritten red diagram. It shows a vertical line with an upward arrow on the left and a downward arrow on the right, with the words "data dependences" written in red cursive between them. The slide also features the NPTEL logo in the bottom left corner and the text "McGraw-Hill | Advanced Computer Architecture" and "50" in the bottom right corner.

So, some basic results in order processors respect all program order dependencies, in the sense they execute instructions in program order. Whereas, so this means that they execute both they respect all data and control dependencies. Whereas, out of order processors only respect data and control dependencies. So, let us gradually understand what this means and so it is important for us to kind of gradually see and understand what this means?

So, primarily out of order processors respect data dependencies which are producer consumer relationships, they appear to respect all dependencies including control. So, the way that they appear is that if you have an if statement, we predict the direction of the branch and then we fetch instructions after it. We can very well execute these instructions before we execute the if statement, if they are independent that is write independent from a data perspective.

And then we execute the branch corresponding to the if statement at a later point of time. And if that is fine well we do not do anything because we have anyway predicted it correctly. So, in a certain sense this data dependence is broken, but of course, control dependencies which


basically means that we need to always follow the correct path. That of course, all processors need to do both in order as well as out of order.

(Refer Slide Time: 12:15)

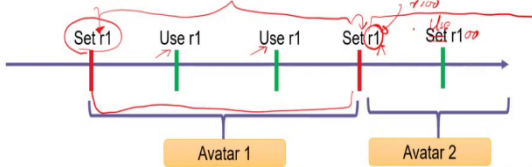
**Can output and anti dependencies be removed?**

```
mov r1, 1
add r5, r6, r7
add r1, r4, r2
add r8, r9, r10
```

```
add r1, r2, r3
add r5, r6, r7
add r2, r5, r6
add r8, r9, r10
```



- Don't you think that these dependencies are there because we have a **finite** number of registers.
- What if we had an infinite number of registers? *Renaming*



NPTEL

McGraw-Hill | Advanced Computer Architecture

51

So, now the question is, can output and anti dependencies be removed? So, what we have been arguing up till now is that they are an artifact of a finite number of registers. So, let us take a look at these 2 examples, in this example, in the first one we are writing to r 1 as you can see. So, there is an output dependence in the second example we have a write after read kind of situation.

So, we basically read from r 2 and then write to r 2. So, if we look at it if you think about it both of them can be conceptually both the situations can be conceptually represented on this time line where essentially set the value of a register let us say it is r 1, we then have several read instructions that read its value. Again, so this is like the first avatar of the register which is r 1 and then we have the second avatar which sets r 1 again which as you can see it set again and again a couple of instructions read.

So, as far as we are concerned the first avatar over here has starts with a write has a bunch of reads, again starts with the write has a bunch of reads. So of course, it need not have a bunch of reads it can have no read as well, but at least it needs to start with a write. So, the typical pattern would be you write, then you read, then you write, then you read. So, the value that



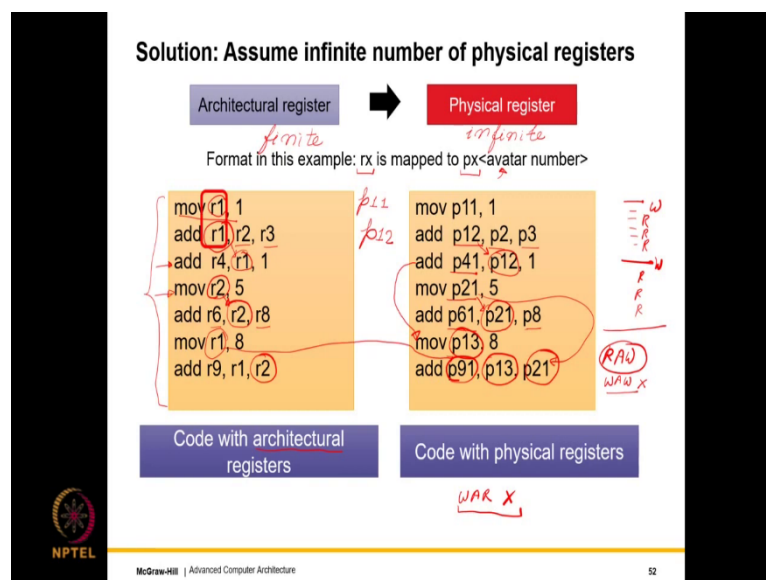
has been set by this instruction the value of r 1 that has been set by this instruction, that will be read over the subsequent instructions.

And so, essentially the lifetime of this avatar over here can be captured is kind of in a sense represented by the value that has been put into r 1. And similarly, we a subsequent avatar starts. So, if let us say automatically we change r 1. So, let us say r 100 and then replace every single use of r 1 with, they should be use.

And then replace every single use of r 1 with use of r 100 it will make more difference at all and the programmer will not be able to see the output of the program will still be the same.

And essentially, the everything will work in exactly the same manner as long as we are consisted, which means that every read that uses this value of r 1 that instead of using r 1 uses r 100. So, this process of actually taking a register and renaming it to another register is known by the word renaming. And as you can see what renaming does is that for each avatar of a register it assigns it a different register. So, we will see the advantages of renaming very soon.

(Refer Slide Time: 15:49)



So, let us see what it does? So, let us take a look at this piece of code where this is a piece of code that the compiler has generated, say it does have architectural registers and which are our normal registers. So, normal registers are known as architectural registers in this

particular context. So, as you can see that because we have a finite numbers of registers, registers are repeated so we write to  $r_1$ , then we write to  $r_1$ , then we write to  $r_1$  again.

So now, let us assume that we have an infinite number of internal registers which are called physical registers. So of course, in practice they are not infinite, but for the time being let us assume it will make our life very easily. So, let us say we have a finite number of architectural registers and we have an infinite number of physical registers. So, what we do is that so this is just a convention for this example that register  $r_x$  is mapped to physical register  $p_x$  with a number of the avatar, 1 2 3 4 5 6 and so on.

So, consider the first instruction  $r_1$ . So,  $r_1$  we map it to  $p_1$  and since it is the first avatar we say it is  $p_{11}$  and so move  $r_1$  internally. So, it is who does this process? Well, the hardware has a renaming engine and the process this conversion from an architectural register to a physical register happens internally. So, the hardware does it internally unbeknownst to the programmer or the compiler it is kind of invisible it happens internally and internally they are mapped to a new set of registers.

How that happens? Well, we will see in the next few chapters, currently, I am just introducing the problem conceptually. So, making some assumptions which will later on we will see they were not hold but that does not matter. Now, let us assume that  $r_2$  and  $r_3$  were initially mapped to  $p_2$  and  $p_3$ .

So, you will get the values from there, since we are creating a new avatar of  $r_1$ , we create a new physical register, map it to a new physical register which will be  $p_1$ , second avatar  $p_{12}$ . The third instruction is rather important.

So, here we are using the value of  $r_1$  that is the read after write dependence over here. So,  $r_1$  we will use the latest value that has been written which is essentially the current avatar and this has been  $p_{12}$ . So, we use  $p_{12}$  over here, since we are writing to  $r_4$  we create a new avatar of  $r_4$  which is  $p_{41}$  and we proceed. Now, coming to the next instruction we are writing to  $r_2$  this is the first avatar of  $r_2$ . So, we move  $p_{21}$ , the pattern should be cleared by now.

Furthermore, after a set we have a use so the use will of course, use the current avatar and r 2 the use where it will use the physical register p 2 1, because p 2 1 is using the current value, r 8 is mapped to p 8, r 6 is mapped to p 6 1.

So, that is the first avatar because we are we are not written to r 6 before and finally we write to r 1 again. So, the moment we write to r 1 again we create a fresh avatar of r 1 which is p 1 3. so basically, p 1 3 essentially means that you are creating the third avatar of r 1 and we are writing 8 to it.

So, kindly note that this particular convention is only for this example. So, we will generalize that later, but currently our focus is on understanding this concept. Finally, we will use r 1, which version we will use? We will use p 1 3 and then we will use r 2, which version we will use? The latest version which is the one written over here p 2 1 and we will create a new avatar of p 9 1.

So, this is the code with physical registers. So, let us look at this closely. So, what you see is that whenever we write to an architectural register we actually create a new avatar of it. So, we create a new avatar of it and so then we essentially set. So, we write once we read many times, again we create a new avatar of it which is a write these are all reads, again we create a new avatar and we read it.

So, let us look at each of our dependencies the read after write dependency well that is holding, because we are always reading the latest version. Write after write dependencies well there are none because we are assuming that we have an infinite number of physical registers.

So, we will never actually write to the same register twice, given that we are not doing that there will be no write after write dependencies. Hence, we do not have to bother about them.

What about write after read dependencies? Well, again we will not have to bother because the read is using the previous avatar. So, just take a look at this for r 1 the read is using p 1 2 and after that we are reading we are writing to r 1 again which is this version p 1 3.

So, there is a write after read dependence over here, but this is not causing an issue, because these are actually separate avatars they are in separate registers. So, even if we were to

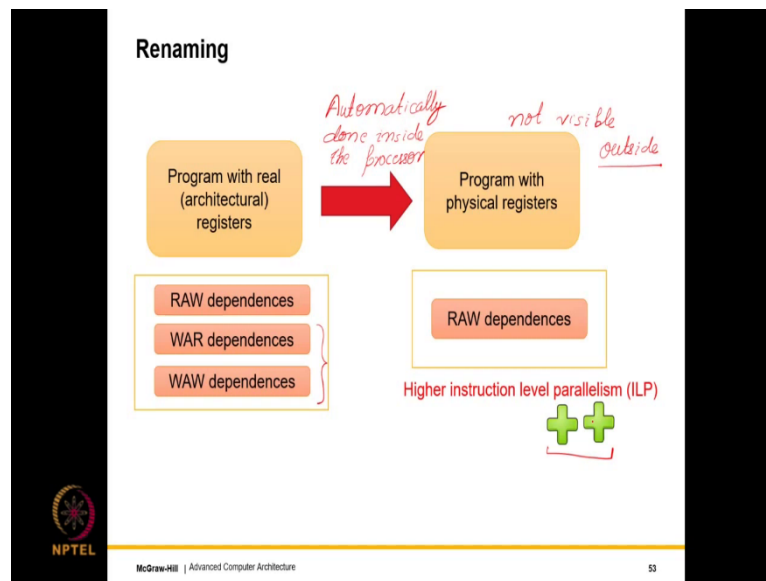
reorder these 2 instructions it will make no difference at all, because they are reading and writing they are accessing different registers. So, the write after read is an issue only when they are accessing the same register. But now since we give a different physical register to each avatar, write after read issues will simply not happen. Because after a read the write will write after a read to an architectural register as you can see the next write will actually write to a different physical register.

So, we will actually not have any write after read dependencies. So, given the fact that we will not have these dependencies and they have been removed with this renaming mechanism that I am proposing, they are false dependencies in the sense they can be automatically removed. It is only the read after write which is a true dependence.

So, now this is another so let me erase the ink on this slide and I can take you through this through the animation. So, as you can see there is a write after write dependence over here, but there is no such dependence on the right side. The read after write dependence is clearly visible their mapped to registers of the same avatar, again another read after write is clearly visible.

And another as you can see read after write is clearly visible, but nothing else is visible. So, there are no other dependency, so we did draw your attention to a write after write problem, but as you can see it is not over here.

(Refer Slide Time: 24:29)

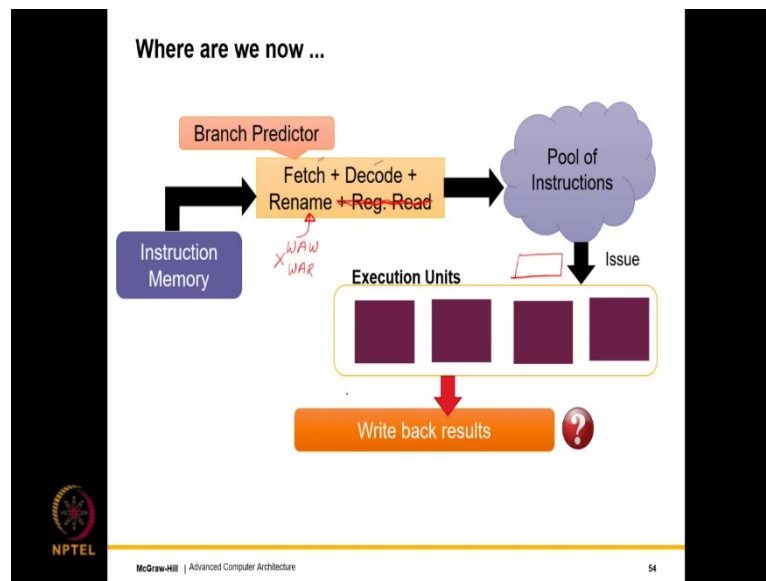


So, what does renaming actually do? Well, it takes a program with real registers, architectural registers that has 3 kinds of dependencies read after write dependencies, write after read dependencies and write after write dependencies. So, they have 3 kinds of dependency. And so then, the process of renaming which is automatically done inside the processor and nobody can see what is happening neither the programmer nor the compiler.

So, this converts the program internally of course not visible outside, internally of course so it is not visible outside it converts it to a program with physical registers. So, a program of with physical registers has only RAW dependencies read after write dependencies and of course as you can see fewer is the number of dependencies fewer are the dependencies higher is the instruction level parallelism the ILP.

So, since we reduce the number of dependencies we have exposed more instruction level parallelism, which is something that the program can use to find more independent and parallel instructions that can be issued at the same time.

(Refer Slide Time: 26:27)

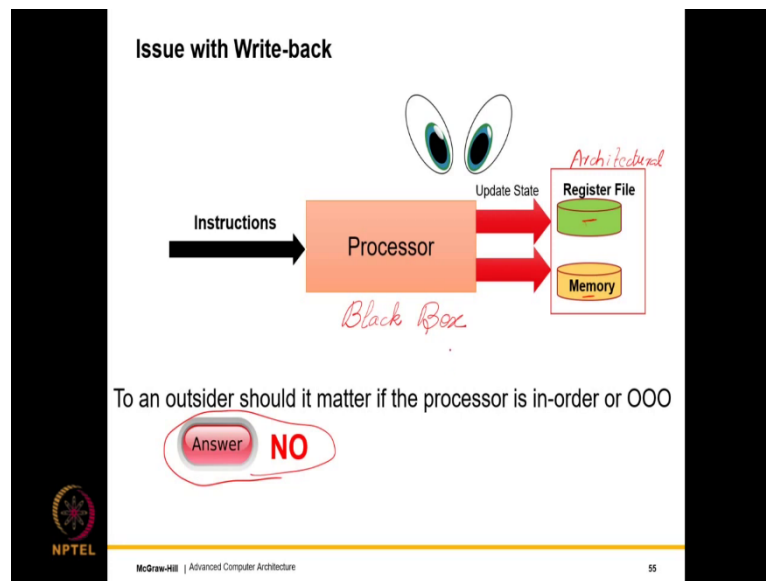


So, where are we now? Well, we have the design of a proto out of order processor there we read instructions from the I cache. So, we are generically generally referring to this as the instruction memory in this slide set. Then we of course we need a branch predictor we all have seen why? such that we can quickly fill the instruction window then we will have a regular stages of fetch and decode.

After decoding an instruction, we have to rename the instruction such that why that is essentially to removed these two kind of dependencies. So, they need to be removed then we need to read the value of registers. So, basically this process can be done after it. So, I it is better you know I deleted its small error over here in this slide. So, this is actually done slightly later the register read is actually done slightly later at this stage.

So, we will discuss that later, so not now. And so then we create a large pool of instructions. So, every cycle we find a set of ready instructions. So, operands are ready and then these independent ready instructions are sent to the execution units not one but several in parallel, the results are computed and then we write back the results to the register file. So, this is kind of the design of a simple out of order processor as much as we have seen till now.

(Refer Slide Time: 28:18)



So, now, let us look at the Write back process. So, the write back process has some issues these issues have to be addressed, these issues have to be fixed. So, let us look at the processor from outside, let us look at it as a; let us look at it as a Black Box.

So, in the black box instructions coming and then the processor updates the state which is the state of the register. So of course, outside the processor we do not get to see the physical registers these are the regular architectural registers which the compiler and the rest of the world sees or regular registers basically.

So, this architectural register file gets updated and the memory gets updated, data memory gets updated and the register file gets updated. So, should an out to an outsider should it matter in the processor is in order or out of order. Well, the answer is absolutely no, that the outsider should simply not care whether we are using an in order pipeline or an out of order pipeline.

So, it should not concern the outsider at all, it should not concern the outsider to the least bit what kind of processor is actually being used. So, given the fact that externally we want the behavior to be the same, how do we ensure that? So, let us elaborate on this topic.

(Refer Slide Time: 30:00)

Assume that there is an exception or interrupt

```
mov r4, 10
mov r2, 0
div r3, r1, r2
sub r4, r4, 1
```

Languages like C or Java have dedicated functions that are called if there is a divide-by-zero in the code.

The question is:

- What if the **sub** instruction has executed when we enter the exception handler?
- An in-order processor will never do this.

NPTEL

McGraw-Hill | Advanced Computer Architecture

56

So, assume that there is an exception or interrupt so, why? So, we are we are executing, so assume here we divide by 0. So, whenever we have a division instruction we can have a divide by zero problem. So, then there are several options one is that the program can crash.

So, this is true for a simple program, but consider that you are running something on excel and you divide by 0 does the excel crash it does not. What it does is, it tells you that look you have divided by 0 that is an illegal operation and then excel continues its operation, continues its working.

So, in most mature software we actually have an exception handler. The exception handler is another small function which is called it shows you a message on the screen that, look you have divided by 0 that is an illegal operation. And then you we either continue executing from there or we jump to some other function and start executing that. So, this is the typical flow of a program whenever an exceptional condition of this nature is encounter.

So, almost all high level languages such as C or Java have features to write your own exception handlers if you know programmer somehow messes up when you still do not want the program to crash. The important question over here is that should the sub instruction. So, let us say our exception handler is called over here and let us assume it does something and then again it comes back.



So, at this point when the exception handler is running should the subtract instruction have completed what about the subtract instruction? In an out of order processor it is very well possible that these instructions could have been reordered. So, it is possible that the subtract would have completed its execution written the value to r 4 and then since the execution is out of order the divide instructions would have executed had a fault and gone to the exception handler.

So, before telling you why this is incorrect behavior, let us first tell you that an in order processor will never do this, why not? Well the reason it will not do it is because it executes instructions in program order, one after the other first this, then this, then this and then this. So, if there is a problem in the division instruction, it will immediately stop the execution not process the sub instruction subtract instruction at all most to the exception handler do whatever that the exception handler says.

And the instructions and exception handler execute them come back and then execute the subtract instruction. So, from the point of view of an outsider, the outsider will essentially see execution in full program order in the in order processor.

However, in an out of order processor if we jump to the exception handler, we are setting ourselves for extremely difficult non intuitive behavior where the outsider might see that the subtract instruction has completed or god knows what you can see a full mix, it is further more possible because of out of order execution.

Mainly you know some other instructions from here they have not completed, which are way before it a program order they have not completed and maybe instructions after days I have completed. So, in that case, it is not possible to actually stop the program at one point and restart it and of course, there are other things. So, normally what happens in the modern operating system, we run multiple programs in parallel.

So, of course, we never run multiple programs in parallel what actually happens is, we run one program for some time then we have a timer chip. So, the timer chip is outside the processor, it senses and interrupt. So, the interrupt basically says that look your time is up then the operating system loads another program on the processor after sometimes the timer

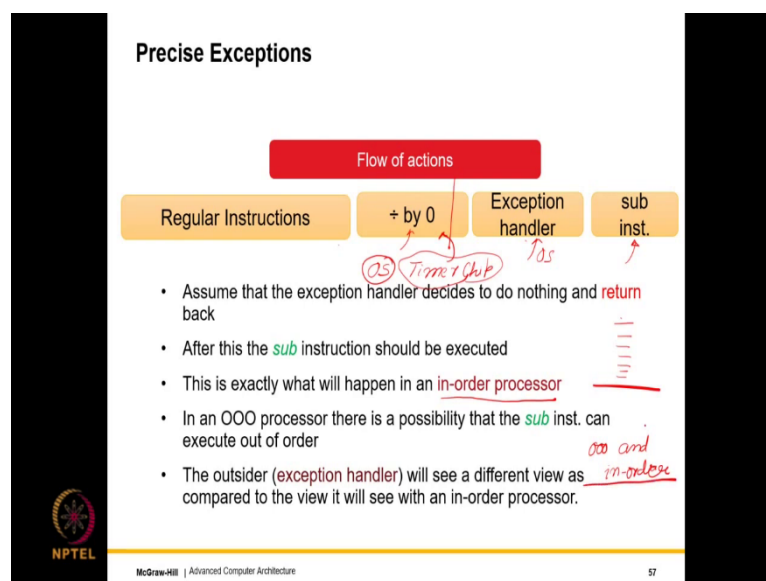
chip again tells look your time is up. So, that is why there is a need to effectively stop a program at one point and restart the execution of the program at the same point.

So, not only is this very helpful from the point of view of an exception handler, it is also very helpful from the point of view of an operating system which essentially sends an interrupting and interrupt to the program makes it stop at a certain point and then resumes the program at the same point.

If let us say things were executing out of order something before this has finished something after it. So, is something before this point has not finished something after it has finished the stop and resume will become very hard.

So, what we want is we need the power to arbitrarily stop the program at points and then resume the execution. So, what is important for us is this stop and resume behaviour and this is not possible appears to be very hard in an out of order execution.

(Refer Slide Time: 35:56)



So, what we define is the notion of a precise exception? where the flow of action should be as follows we have a regular instruction before the div which means all of these. Then we see the divide by 0, then we load the exception handler and then we execute the subtract instruction. Alternatively, it is possible that maybe there was no divide by 0, but the operating

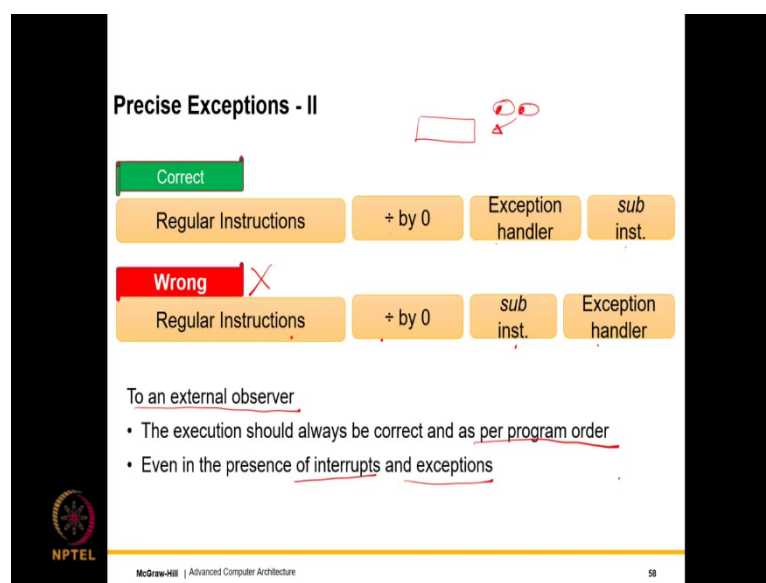
system decided to shell the program and run another program on the core. So, know that operating system is also a program.

So, it is nothing special, but there is an external timer chip which essentially sends the interrupts the operating system pre programs the timer chip. And the timer chip is in the external entity sends an interrupt and so, then the entire state of a program is stored over here and the exception handler in this case is the OS it will load another program. So, what we want is we need to stop and resume semantics.

So, the stop and resume semantics is very easy in an order processor because we are following program order we execute, execute, execute stop at one point no problem and we just store the register state and then go and execute some other program. With out of order it is hard.

So, what we want is that we want all external agents which is anybody is sitting outside the processor to actually look are the processor and not be able to say whether it is out of order or in order. Which means that, internally an out of order processor can do whatever it wants, but externally it should appear to be as if it is in order processor.

(Refer Slide Time: 37:57)



What do I actually mean? What I actually mean is that externally which means if this is the processor and I have a pair of eyes and if I have a pair of eyes looking at it externally, they

should see this order, the regular instructions divide by 0 exception handler and the subtract. They should not see this order which is regular instructions divide by 0 the subtract and then the exception handler this is indeed out of order.

But out of order should be visible inside not outside, which means that to an external observer the execution should always be correct, should always be as per program order even if there are interrupts and exceptions externally it should be in program order.

(Refer Slide Time: 38:51)

### Precise Exceptions - III

- We thus need **precise exceptions**
- Assume that the dynamic instructions in a program (ordered in **program order**) are:  $ins_1, ins_2, ins_3 \dots ins_n$
- Assume that the processor starts the exception/interrupt handler after it has just finished writing the **results** of instruction:  $ins_k$

$ins_1$

$ins_2$

$ins_3$

$\dots$

$ins_k$

Exception

$ins_{k+1}$

$\dots$

- Then instructions:  $ins_1 \dots ins_k$  should have executed completely and written their **results** to the memory/register file
- AND,  $ins_{k+1}$  and later instructions should **not** appear to have started their execution **at all**
- Such an exception or interrupt is **precise**

*Stop and Resume*

- External agent on proc. should appear to be an in-order processor.

*done stop Resume NOT started*

*Outsiders see an in-order execution*

*Memory*

McGraw-Hill | Advanced Computer Architecture

59

So, this is making our life very difficult. So, this is a rather complicated thing that we are just discussing we are saying that look if you want a seamless stop and resume of a program to an external agent. An external agent can be anybody an external agent can be the programmer, can be the compiler, can be the operating system or can simply be an oscilloscope that has probes outside the processor does not matter. An out of order processor should appear to be.

So, this will kind of keep our life very simple internally it can do whatever it wants. So, what are we saying? We are saying that look for outsiders you appear to be an order processor internally you do whatever you are doing, you want to execute out of order no problem do it. How do we ensure both at the same time? Well chapter 4, so in chapter 4 we will discuss how to do it not now, but at least for us this is a goal.

So, this chapter is all about goal setting, this chapter is all about telling us what is it that we expect from the out of order processor and a crucial aspect of this goal setting is this particular requirement. So, this automatically takes us to the notion of precise exceptions which means that the following.

So, let us first define a terminology. Let us say the dynamic instructions in a program are numbered instruction  $ins_1, ins_2, ins_3, ins_4, \dots, ins_n$  assume that the processor starts the exception interrupt handler after it has just finished writing the results for the  $k$ th instruction to an outsider it should appear.

And as I said the outsider is very generic can be the programmer, compiler, operating system or even an oscilloscope that has probes plugged in to the interface of a processor. It should appear as if all of these instructions have executed completely written in the state to the memory or register file and none of the instructions after this have even started. So, these are fully done and these have not even started.

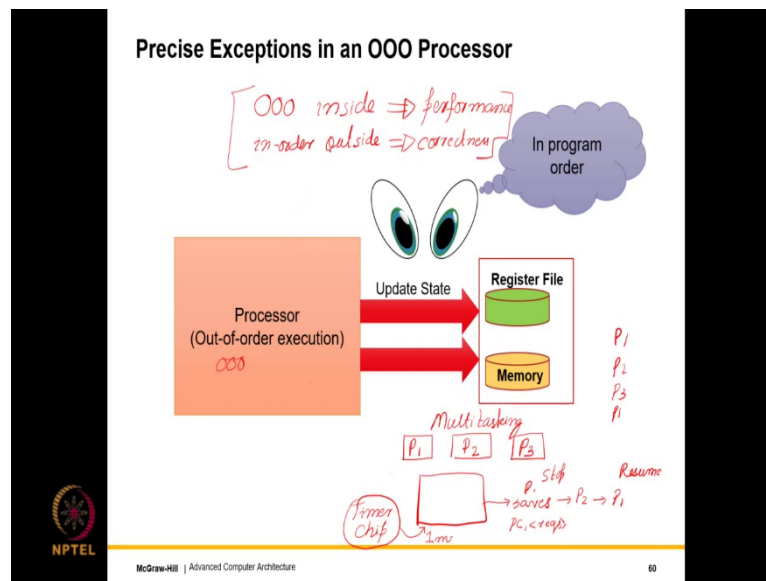
So I should have a not started in bold not started and these are done and this is of course, where we have the exception. So, we have an option at this point or stopping the program storing its state. So, which is easy we just need to store the value of the architectural registers at this point, process the exception and resume, stop resume. For a seamless stop and resume what you will agree with me that the first  $k$  instructions have to be fully done and all the instructions after it should not even have started.

So, this notion is known as a precise exception and almost all out of order processors as of today provide the notion of precise exceptions which essentially means same things which are all equivalent that outsiders will see in order execution. So, basically I have so, the processor, in this case, consists of the cores the core and pipelines are the same and of course, the memory.

So, let me just all the memory that we are using does not matter let me just draw a full circle around them that is a black box say any outsider outside this black box will see in order execution, which will allow us to essentially stop the program at some point. So, we are sure everything before this is totally done in program order execute the exception handler and again restart at the same point.

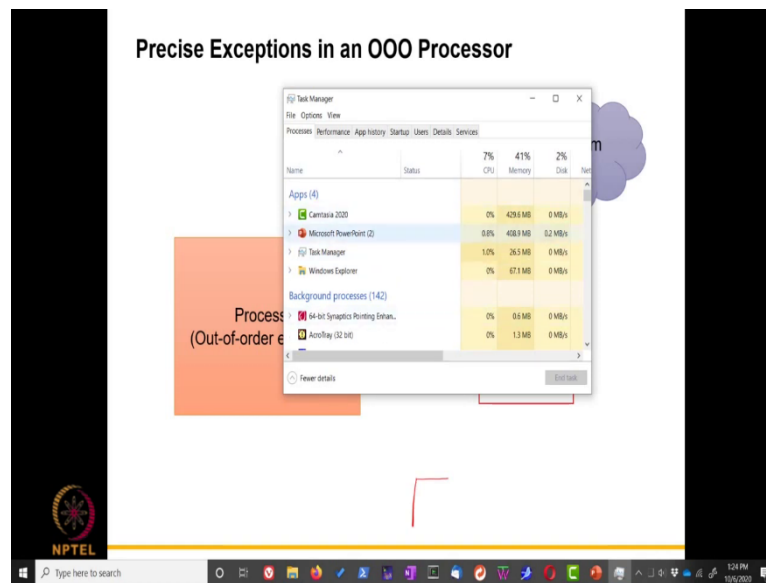
So, we are sure that nothing after the kth instruction has even started. So, this ability is very important not only is it required for intuitiveness and correctness, it is also required for processing exceptions as you will see it is also required for recovering from branch wise predictions, because even that is an exception. We will also see that it is a very vital requirement for operating systems.

(Refer Slide Time: 44:09)



Because what normally happens if I can use some of the additional space over here what normally happens just do a control all delete and you will see, so in fact, let me do that right now. So, what you can actually see over here in my task manager that look I have a lot of tasks and you can see your power point task over here.

(Refer Slide Time: 44:28)



See if I have a single core what I am actually doing is that I am running multiple programs right program 1, program 2, program 3 etcetera on a single core. So, on a single core what will actually happen is that program 1 will execute till sometime. As we have said, we will have a timer chip it is very important to have the timer chip it is not a part of the operating system it is a separate piece of hardware.

It will send an interrupt, it is typically programmed to send an interrupt once every one milli second. Once that happens, what the processor does is that it saves what is called the context of the program. The context is simple it is the program counter and the set of architectural registers because the physical registers are not visible to software. So, is essentially the program counter and the set of architecture of registers that is saved. And then well it starts executing some other program may be program 2.

Again, after one milli second another timer interrupt comes. The operating system then might decide to start executing program one back again from exactly the same point and you will not perceive a differences like magic. So, what is happening over here?

What is a happening over here is that unbeknownst to program 1 we stopped it at one point in the middle, we executed another program again we brought program 1 back, we were able to.

Because, program one supports precise exceptions we were able to nicely restore its state and execute it from that point.

So, operating systems what they do is that they take a small chunk of program 1 executed then execute program 2, then execute program 3, maybe then 1 then 2. Maybe the under execute program 2 for several time slices for one-time slice is defined as 1 millisecond. And if we have tens of programs as you just saw then maybe you can take all of these programs and just execute them in any order.

The reason we do not perceive this is because this is happening hundreds of times a second. So, it is happening, so quickly that we actually do not perceive the fact that the operating system is actually executing lots and lots of programs quickly one after the other on a same core. So, of course, if we have different cores where core is basically just an encapsulation of a pipeline then of course, these programs can be run on separate cores.

But even if we have a single core like in the good old days, the operating system can happily switch between these programs what is known as multitasking. And just keep to a switching between them hundreds of times of second, we will never be able to perceive what is happening and the reason that we are able to switch is because there is a way to stop a program at a point and resume it from exactly the same point.

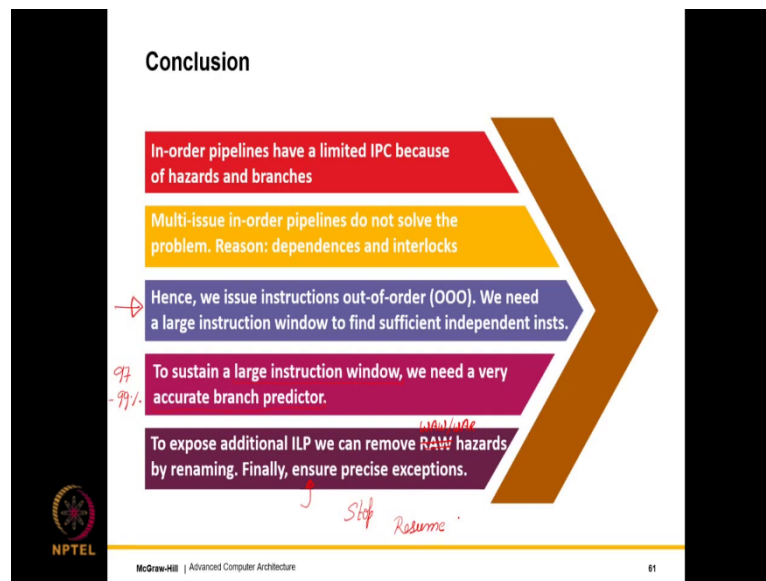
This stop and resume behaviour is enabled because we have precise exceptions and what are precise exceptions? Well, they just say that look for a pair of eyes outside the processor and memory black box, they essentially perceive in order execution which means that all software entities be it the programmer, the compiler, the operating system all of them perceive in order execution, even the inside the processor we are actually doing out of order.

So, our important challenge most important challenge would be or our motto out of order inside in order outside. So, out of order inside will give us what? It is going to give us performance in order outside will give us simplicity intuitiveness and correctness, correctness from the point of view of seamlessly stopping and resuming programs.

So, where have we stop? We have stopped at a nice point where we say out of order inside in order outside and precise exceptions are a natural corollary of the in order outside philosophy.



(Refer Slide Time: 49:51)



Now, we come to the last slide of this lecture set. So, this talks about the five most important points that we have learnt in this chapter. So, the first is that in order pipelines which we have studied with so much a pleasure in our undergraduate texts have a limited IPC because of branches and data hazards.

If I were to have a complex multi issue in order pipeline that is not really going to solve the problem. The reason being that there will be a very complex structure of dependencies and interlocks, hence the problem will not get solved.

What we instead need to do is we need to change our thinking and we need to go for out of order processors. So, in out of order processors, we need a large instruction window to find a sufficient number of independent instructions. To sustain the large instruction window we would need a very accurate branch predictor. So, we have looked at some accuracy numbers, but something like 97 to 99 % is the minimum that we need.

More not less moreover to expose additional I L P instruction level parallelism. we can remove I stand connected here write after write and write after read hazards by renaming. So, for additional ILP we can remove the false dependencies or the write after write and write after read hazards by renaming which essentially assign a separate physical register to each avatar.

Finally, for the sake of correctness we need to ensure precise exceptions such that we can transparently seamlessly and easily stop a program at an arbitrary point and resume the program at the same point after a long time. So, I will meet you in the next chapter which discusses the fetch logic and branch prediction in great detail.