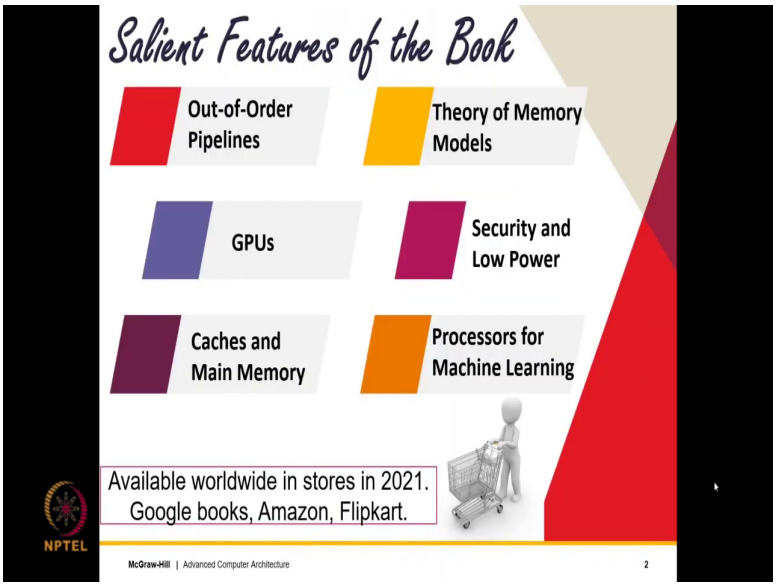
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

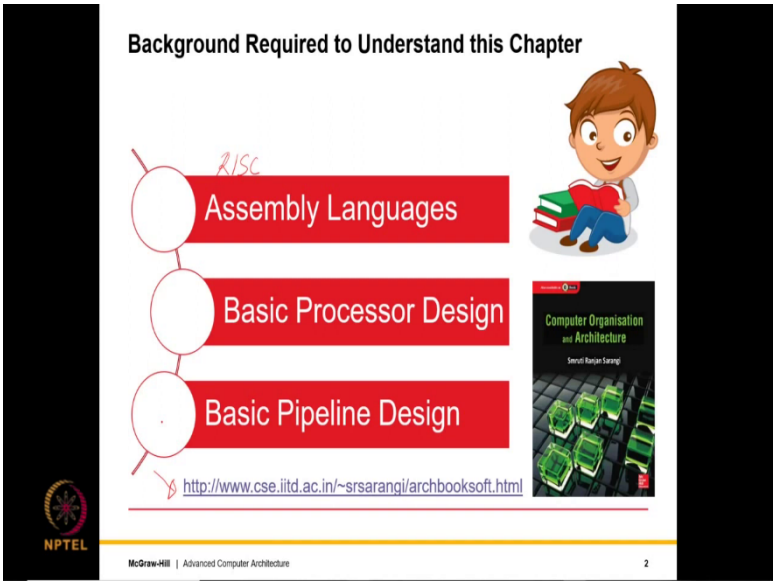
Lecture - 02
Out-of-Order Pipelines Part- I

(Refer Slide Time: 00:23)



Welcome to the chapter on the introduction to Out-of-Order Pipelines.

(Refer Slide Time: 00:50)



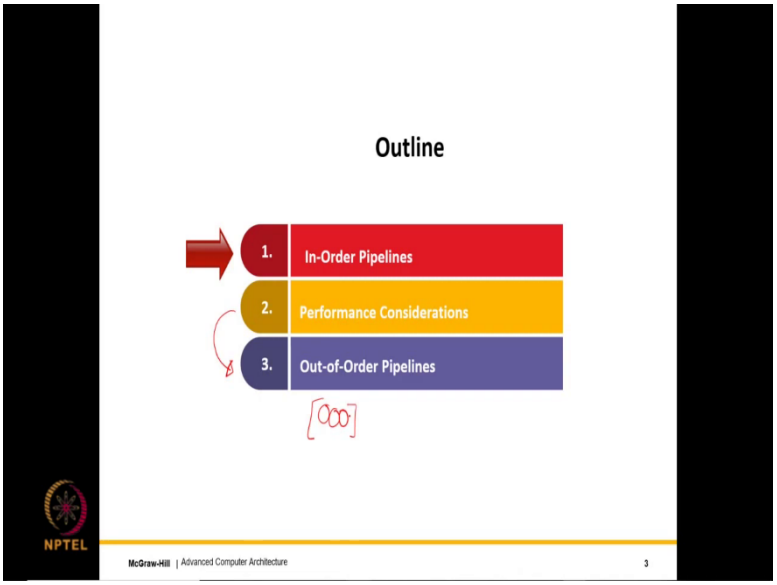
So, in this chapter, we will study everything about out of order pipelines which is a very different kind of pipeline as compared to traditional in order pipelines. So, since this is an advanced book on computer architecture, let us think of it as a second level course, so, something that you are supposed to read in the final year of your undergraduate studies or in your post graduate studies.

So, in this case, we would require some background and the background that is required to understand this chapter is the understanding of assembly languages specifically RISC assembly languages, so, this is definitely required. Furthermore, student also should know about basic processor design and finally, the student should also know about basic pipeline design of how to take a simple processor and pipeline it? So, these three basic concepts we require before somebody can actually proceed further.

So, I am giving a pointer to my earlier book on Computer Architecture which was published by McGraw Hill in 2015. So, this book is widely available and also online versions are available on Amazon, Kindle and Google Books. So, readers are advised to at least go through these three chapters, but again reading my book is not necessary, any other text in computer architecture is also sufficient as long as these three basic concepts are understood.

So, all the videos for these three basic concepts for my earlier book are also available on YouTube. so, students can also go through the PowerPoint slides and videos. So, the links to these PowerPoint slides and videos is given below right over here. So, I would advice anybody listening to this video to kindly look at the YouTube videos at least for these three topics before proceeding forward. So, this is a think of this as a mandatory requirement.

(Refer Slide Time: 03:23)



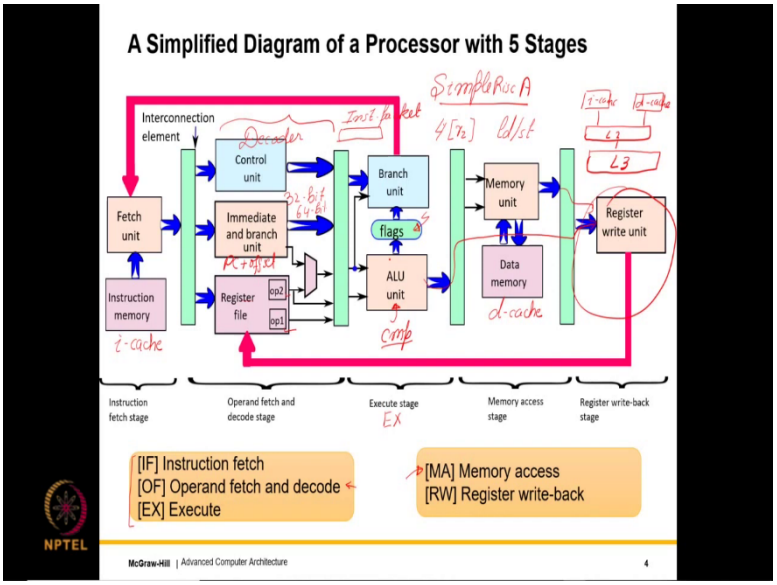
So, what we will do? is that we will divide this short lecture into three sub lectures, 1st we will do a quick recap of in-order pipelines. So, kindly note that the description will be at a very high-level and is by no means meant to be complete. So, this is a part of the prior background that I was talking of.

Then we will look at some performance considerations of how exactly do we model the performance of an in order pipeline? and what exactly are the matrix? what should be considered? and what should not be considered? on the basis of that, we will develop our motivation and also the key essentials features of what an out of order pipeline?

So, out of order pipeline is something that we do not know, but almost all processors in laptops, desktop, servers, even mobile phones these days use out of order pipelines. So, out of order is also called “OOO”, triple O. So, this is so, when we use this in the text, it will refer to an out of order pipeline. So, without further ado, let us move to discussing basic in-order

pipelines. Again as, I said another is the last disclaimer is you, and this is just a very high-level overview.

(Refer Slide Time: 04:55)



Let us not go through the structure of a typical processor with 5 processing stages. So, I will describe this somewhat complicated slide. If you are having any difficulty in understanding, you need to heed to my advice given in the previous slide and go back and see the earlier videos.

So, I start this diagram is extremely easy to understand nevertheless let me just provide a brief overview for viewers of this video who have seen similar processors I will be designed slightly differently. So, what we will do? we will divide the entire processing of an instruction into 5 stages. So, the processing starts here where we read an instruction from instruction memory. So, in an actual processor, this is actually an instruction cache so, it is called an i-cache and what is a cache? Well, we will see that later.

But a cache is essentially a small memory that holds a subset of memory locations, this is called a cache and an i-cache has instructions and similarly, we have a data memory called d-cache that has data words. So, the i-cache instructions over here are the instructions are read and they are read by the fetch unit. So, the fetch unit of course, reads the entire contents of the instruction.

After that, it is passed, so, here the green box just indicates for this diagram a bunch of wires we call it an interconnection element and so, it passes it to this unit which is the next unit. So, next unit does several things, the first is we have a control unit, this we can also call the decoder. So, what the decoder does is that it decodes the instruction which means that it understands the instruction.

So, normally an instruction is like a compressed bundle of a lot of things. a compressed bundle of the opcodes, the register files, memory addresses all of that it kind of you compress it and keep it. So, the decoder essentially expands it and essentially, generates what are called control signals which control the rest of the elements in the data paths which essentially determine the execution of the instruction, the way it will be executed.

We then have an immediate and branch unit say any constant within the processor is called an immediate and we have an immediate and branch unit which extracts whatever constants are embedded inside the instruction and expands it to a full 32-bit or 64-bit value depending upon whether it is a 64-bit or a 32-bit processor depending upon the width of the data path, this value is expanded.

Then, for all the register operands, we read the values from the register file, recall that the register file is a set of named storage locations, a typical processor has 16 to 32 registers and they are used for different purposes as we shall see later. So, we read the registers from the register file. So, we call them op1 called the values that are read as op1 and op2, operand 1, operand 2.

And of course, we have a choice, so, we will be using the simple risk instruction set, this is provided as an appendix in the book the 1st appendix, appendix A. In addition, it is there in the previous book, and this is similar to any other RISC instruction set broadly and it is extremely simple. So, in this case, in the simple RISC instruction set, the 2nd operand can either be a register or an immediate, so, we have one multiplexer over here to choose whether it is a register or an immediate.

So, what do we get from the second stage which is operand fetch and decode? Well, what we get is that we get a set of control signals which determine the behaviour of the instruction in the rest of the processor, we extract the constants or the branch target. So, the branch target is

encoded as a fixed offset from the current PC, from the current program counter. So, essentially, we take the current program counter, here we add the offset to get the branch target.

So, whatever we compute we just take everything along with us and whatever we take along with us is called the instruction packet. this c got hidden behind the red line, but nevertheless what I wanted to write is it is the instruction packet, so, all of this information flows along with us. In the 3rd stage, which is the execute stage, the EX stage, we do the following.

So, we can either process the branch which means see if a branch is taken or not taken or we can perform an arithmetic or logical operation. So, let's look at the later first. In the ALU unit the arithmetic and logical unit, what we do is that we take in the instruction packet. If let's say it is an add instruction we add, if it is a multiply instruction we multiply, divide we divide, logical OR, and so on.

So, one important internal register is the flags register. So, whenever we see a compare instruction which is the same for almost all instructions sets, they work in a similar manner. Whenever we see a compare instruction, we set the flags register, we set certain bits within the flags register such that a later branch can look at the bits of the flags instruction and find out if the latest comparison resulted in an equality or a greater than or a less than and this can be used to decide the direction of the branch.

After, we have processed the arithmetic logical instructions and the branch instructions, we move to the 4th stage which is called the MA stage, the memory access stage. So, look at the mnemonics over here IF, OF, EX and now, we are at MA well, MA stage basically only processes the load and store instructions.

So, for both of them so, load and store are actually multipart instructions, so, we only support the base offset addressing mode in simple RISC which means that an address is of the form let's say you know some constant and a base register. So, we read in the contents of r2 here from the register file, we add 4 to it so, it becomes r2 + 4, this is the address. So, this addition of r2 + 4 is done in the ALU unit and subsequently, the address is sent to the memory unit.

So, the memory unit does either perform either a load or a store depending upon what the instruction actually is and finally, so, store of course, does not produce an output, it just writes to data memory in this case is the data cache. So, in the modern memory system we typically have an instruction cache at the top, a data cache a d-cache, both of them are connected to a much larger piece of memory called the L 2 cache, this is again connected to an even larger piece of memory called the L 3 cache.

So, well, why it is the case? We will look at that when we discuss the hierarchy of the memory unit, but we will not focus on that issue right now, but as far as we are concerned, there are two separate memories the instruction memory and a data memory. So, once the load is done, we have read something, so, we need to either write that to the register file or write the value computed by the ALU to the register file which of course, is passed directly via this stage.

And so, whatever be the case, we have a register write unit or register write back unit, so, this unit again writes the results back to the register file. So, this is a simple processor, it does not have pipelining of any kind and this is it we have logically divided into 5 stages because if you take a look at it, we have 5 distinct parts of the processor that do rather different things. so, that is the reason we divided it into 5 parts.

And then, as we see that is some amount of information flow between stages, so, of course, the 2nd stage produces something when the 3rd stage reads, the 3rd stage produces something when the 4th stage reads, but however, there is a back connection. So, both the back connections are shown with this magenta colour.

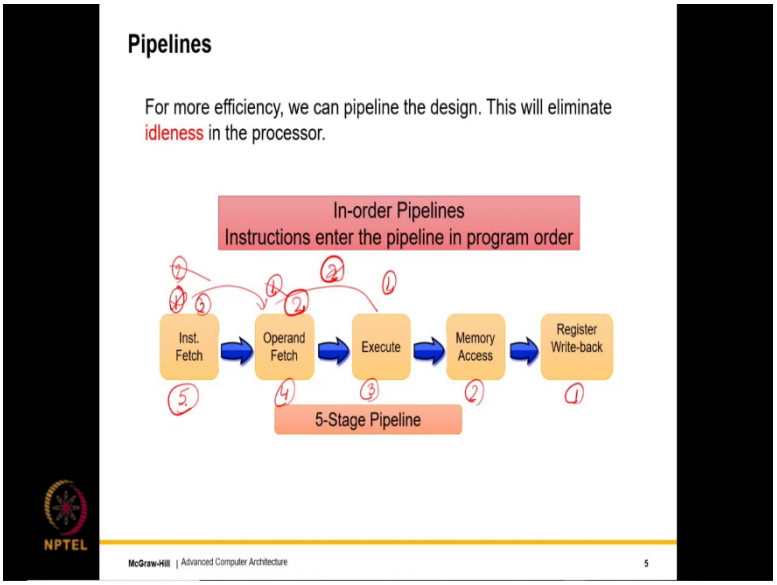
So, the colour that I see over here is somewhere between red and pink that's the best that my eyes can tell me, but you know if any of you have an exact name for this colour, kindly write it in the comment section. And so, whatever this colour is flavour of red, so, from the branch unit to the fetch unit, what we do is we have a back connection. So, this back connection essentially indicates the direction of the branch.

Branch is like an if statement, so, whether it is entering the body of the if statement or exiting it. So, this information is being conveyed. And similarly, in the register write unit also we are

writing from the 5th to the 2nd stage with the results that have been computed by either the ALU unit or the memory unit or the load, so, that has been used to write back.

So, we have discussed this complex diagram so, this is actually simple, but again as I said any difficulty, go back to the videos of the previous book, this is explained in full detail. In fact, there is one full chapter that discusses this diagram something that we finished in 10 minutes several hours are devoted to describing just this. So, I would advise that before we move to the next slide at least this part is completely totally fully understood.

(Refer Slide Time: 16:50)



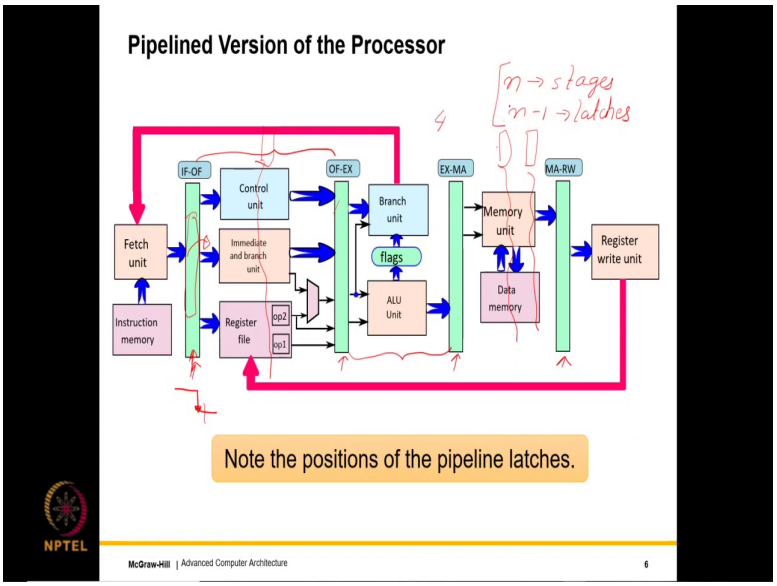
Now, let's come to pipeline, well, if we take a look at this diagram, there is a source of inefficiency. So, let me show this source of inefficiency over here that let's say you know this unit is active, the rest of the 4 units are not active. So, at any point of time only 1 unit is active, and 80 % of the chip is not active which does not convey a very efficient solution which is not a very efficient solution, it does not convey a good message.

So, ideally, we want the entire processor to be busy. So, in pipelining what we do is that we send 1 instruction down the pipeline. So, in instruction 1 is over here, so, we given the fetch unit is idle, we bring an instruction 2.

In a next cycle, instruction 1 moves here, instruction 2 moves here to this stage and instruction 3 is fetched so on and so forth. So, there will be a time when instruction 1 will be

here, 2 will be here, 3 is here, 4 is here and 5 is here. So, if you take a look at it, a pipeline manages to keep the entire processor busy. So, of course, this will increase the throughput and the performance, but we will see, but this is not that straightforward, so, we will see a when it will increase the performance and when it will not.

(Refer Slide Time: 18:42)



So, if I were to create a pipeline version of the processor well, I just make a simple change. So, look at this diagram and just look at the previous one. So, in the previous one we just had a bundle of wires that we call the interconnection elements, now what we do is we remove this and we replace this by a pipeline register as you can see right here.

So, the pipeline registers are regular edge triggered, negative edge triggered flip flops. So, if they are negative edge triggered at the end of a cycle when there is a negative edge from the left side whatever is being written that is written and in the beginning of the next cycle which is after than negative edge, the data over here gets transferred to the next stage, the next stage does its processing.

So, it is expected to complete its processing within one clock cycle which means from within one you know between two of these registers IF-OF and OF-EX, so, the signal is supposed to take a maximum of one clock cycle to propagate via these units which means that the latencies of these units is limited to one clock cycle.

Similarly, the latency between these two units is limited to one clock cycle and so, what does this allow us to do? Well, what it allows us to do? is that we can have five separate instructions in five of these stages and we can process them in parallel. So, of course, this does introduce complexities to the design.

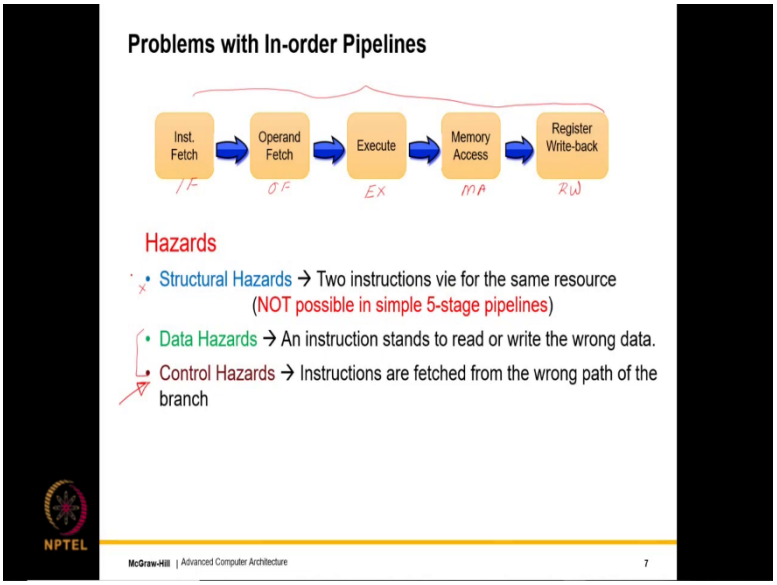
So, pretty much an instruction has to move along with all of its data, it is in entire instruction packet and hop across the stages that is one and we will see there are some other problems as well-known as hazards in the next few slides. So, just to repeat, the only change that we did is that we added a register called a pipeline register which of course, does not store one bit or one byte, but it stores the entire instruction packet which can be tens of bytes.

The entire instruction packet is stored and every cycle, so, each stage reads the input computes the output by the end of the cycle and writes it to the pipeline register at its end. So, of course, the register write unit does not have a pipeline register at the end because it is the last stage, but by the end of the clock cycle, it is expected to write the values to the register file and finish off the work.

So, see that if we have 5 stages, we need 4 pipeline latches, so, there is 5 is no hard and fast rules if we want, we can further split this into one more stage. So, create 2 stages and so, we will have one more pipeline latch over here.

And if we want, we can further slip this down or maybe we can take it into 3 stages, so, we will have one more pipeline latch here, one pipeline latch here. So, if there are n stages in the pipeline, we will need (n-1) the n stages, we will need (n-1) latches. so, that is easy.

(Refer Slide Time: 22:44)

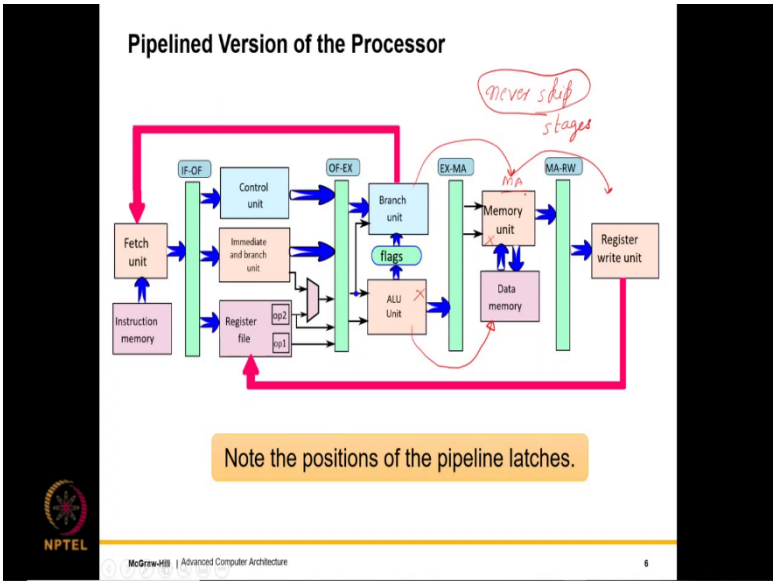


So, given that we have seen this, let us go forward and discuss what is it that can happen? or what is it that can go wrong? if we were to pipeline our instruction processing in this manner. So, we need to understand that in a pipeline which of course, I have shown it over here, it has been made far simpler of course, so, the 5 stages are shown like IF, OF.

So now, given that we will have 5 instructions residing here at the same time, there can be numerous issues, numerous collisions between them. So, we can have structural hazards where two instructions can vie for the same resource. So, this will not happen in the in-order pipeline that was just shown, but it is possible if you take a look at this pipeline, we always can have an instruction that tries to access the data memory.

So, we always can add an instruction that a new instruction of course that can access the data memory in the ALU unit, so, the ALU unit will also access it.

(Refer Slide Time: 24:09)



So, in this case, if the data memory can support only one access per cycle, there will be a conflict between an instruction over here and an instruction over here. If the instruction in the MA stage is accessing the data memory because it need not, because all instructions have to pass through the same route.

So, even an ALU instruction that does not access memory still has to go through this route and so, that is the reason there may be a conflict, there may not be a conflict, if there is a conflict, it would be a structural hazard which means that both the instructions cannot access the data memory at the same time, one will have to wait.

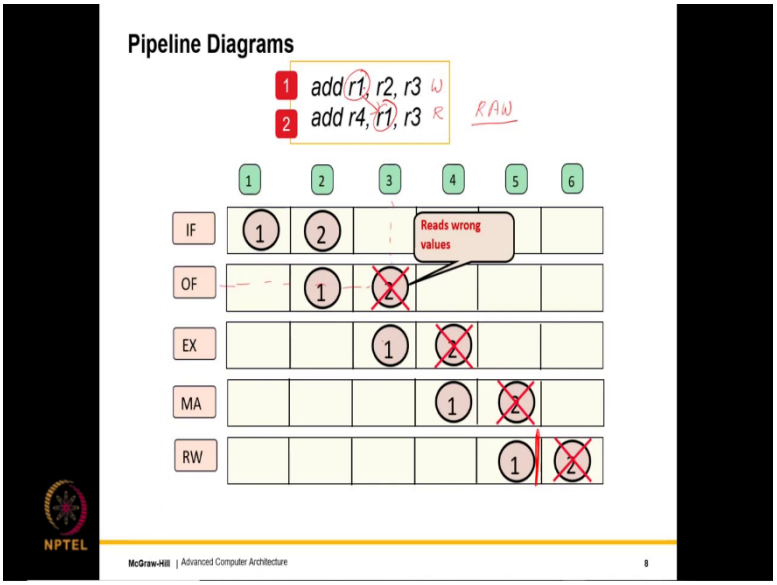
This will not happen in what in the simple version that we are using, but if I extend the instruction set with one more instruction that accesses data memory of this type, we can very well have a conflict and the other important point is that instructions never skip stages. So, I should this is an important concept many people often make a mistake, So, they never skip stages.

Even in the ALU instruction that does not access memory has to go via this step. So, even instructions that do not have an ALU component or a memory component, still go via this step. So, they always go from one stage to the next and they never jump at least in our basic in-order pipeline, they never jump. so, they never skip stages.

Now, coming back, we also can have a data hazards something that we will see in the next few slides where of course, there is a read and write dependency, so, there is a read after a write. So, it is possible as we shall show in the next few slides that the read can miss a value that is being written can have and we can also have what are called control hazards? which means that if a branch is taken, then there will be some problems.

So, we will discuss both of these structural hazards since they will not happen unless we extend our instruction set architecture, unless, we do that since they will not happen we will not discuss it henceforth.

(Refer Slide Time: 26:48)



So, let us create a mathematical tool called a pipeline diagram. So, a pipeline diagram the rows are the pipeline stages as we can see since we have 5 stages, the rows are IF, OF, EX, MA, RW these are the rows, and the columns are the cycles. Now, consider these two instructions add r1, r2, r3 and add r4, r1, r3.

So, here r1 is the destination it is being written to, in this case, r1 is a source, given the fact that this is a write, this is a read, we have a read after write hazard or a read after a write dependency over here a dependence and let's see what will be the problem? So, instruction 1 will be fetched in cycle 1 and as we have said earlier instructions never skip stages. So, from

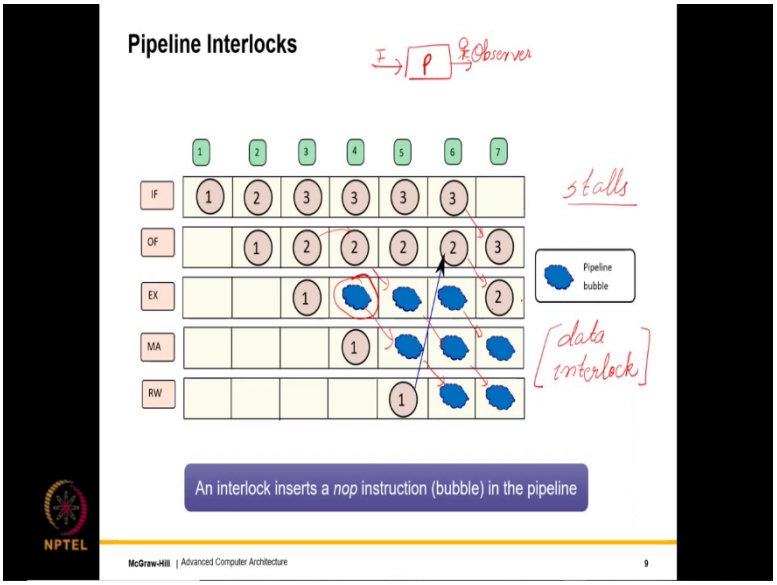
IF stage it will go to OF in the next cycle EX in the next cycle MA and RW say it goes on fine.

So, let's consider instruction 2, this is very important. So, instruction 2 will get fetched no problem. In cycle number 3, instruction 2 will enter the RF stage. In this case, it will try to read the value of r1, but where is the value of r1? The value of r1 will be computed by instruction 1 and that too at the end of cycle 5 which is pretty much at this point and so, this is when it will be computed and written to the register file. Since computing is not enough, you have to write it to the register file.

And in cycle 3, somewhere in the middle of cycle 3, instruction 2 wants the value of r1 from the register file which it is not going to get, it is going to get an old value, a stale value, a wrong value. So, it is going to read a wrong value and of course, it will continue, so, the execution of the program will be incorrect.

So, if you see, this is a very unique situation that has arisen primarily because we have parallel processing going on in a pipeline system where we have different instructions with dependencies between them. They cannot proceed at the same time because we write in this case at the end of the 5th cycle. whereas, in the 3rd cycle, we need the data from the register file, the data is not there. Hence, if we read the register file will stand to read a wrong value and a program will be incorrect.

(Refer Slide Time: 29:43)



So, what do we do? Well, what we do is? we allow instruction 1 to keep going, then instruction 2 when we enter the OF stage, we realize look there is trouble in there. In the sense that the value that instruction 1 was going to produce that value has not been produced, hence, what I do is that I keep instruction 2 in the OF stage. In the 4th cycle in the EX stage, I introduce what is called a bubble, so, the logic that inserts the bubble is called an interlock logic.

So, in this case, we are talking of a data interlock because of course, we have a data dependency. So, what do we do? What is a bubble? Well, the bubble is a nop instruction. What is a nop instruction? It is a no operation instruction; it is an instruction that precisely does nothing. So, nop instruction simply flows through the pipeline without doing anything whatsoever. So, this is the bubble, and this instruction simply goes through the pipeline does nothing and gets out.

In the 4th cycle also, we introduce one more bubble that does nothing and gets out. Finally, in the 5th cycle we do this. In a 6th cycle also, we introduce a bubble that does nothing; however, there is something interesting going on. So, the end of cycle 5, instruction 1 has produced the value of r1 as you can see over here, this value has been produced and this

value can then be written to the register file and in the 6th cycle instruction 2 can read the value of r1 from the register file.

So, as you can see the arrow indicates the dependency and it is going ahead in time, so, it is fine. So, the value of r1 is read by instruction 2 and after the value of r1 is read, instruction 2 can proceed and in the meanwhile, instruction 3 has also been fetched. We have not shown the instruction 3 in the previous slide but assume there is an instruction 3.

So, instruction 3 at that time is fetched, so, basically since we have in order processing, instructions will not overtake each other.

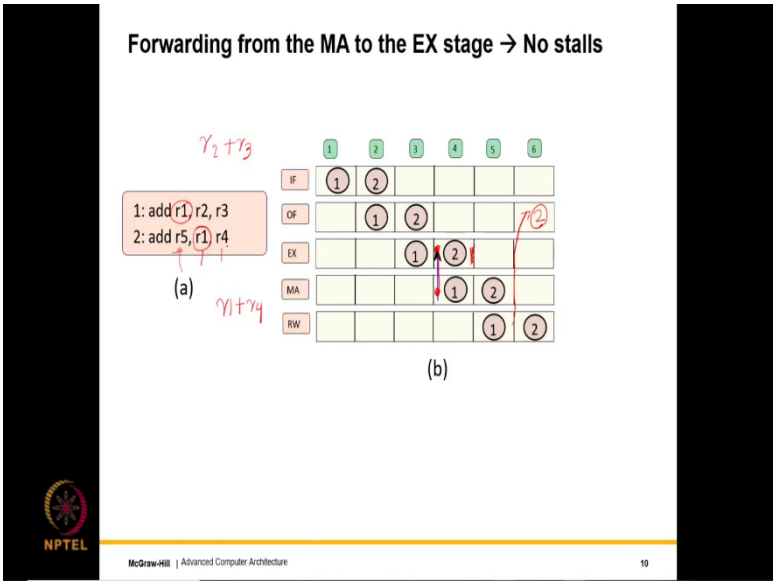
So, once instruction 2 gets the value moves from OF to EX, 3 moves from IF to OF. So, the data interlock logic what does it do? What it does is its job is to have a full global view of all the instructions in the pipeline, we looking at all of them and it figures out dependencies if and it sees whether it should allow an instruction to move to the next stage or insert a bubble in its place.

So, in this case, three bubbles are inserted in the place of instruction 2 and finally, when the data was available, it allowed instruction 2 to progress. So, it kind of put a lock on the pipeline until the data was available. Once the data was available, it releases a lock that is where the word interlock comes from.

So, this is an important concept. So, introducing bubbles essentially, what it does is that it stalls the pipeline. So, for an external observers assume that this is the instructions are going in, this is a processor over here that has a pipeline, and we have an external observer sitting here.

So, the external observer sees that for three cycles nothing is coming out of the pipeline.so, no instruction is completing its execution, this process is known as a stall. So, this process is known as a stall and a stall means a cycle in which no useful work is being done from the point of view of an external observer sitting at the end.

(Refer Slide Time: 34:05)



One way to fix this problem of adding so many interlocks is via forwarding. So, what is the problem? While the problem is that read after write dependency is not all that rare. So, read after write dependency can happen and what we further saw? in this slide that if there is a read after write dependency, we need to add three bubbles.

In fact, if there is a read after write dependency within a window of these four instructions, so, let's say after one instruction within the window of these three instructions rather if there is a read after a write dependency, we need to add a stall cycle at least one and if it is just after it, we need to add three otherwise two or one.

So, this severely constraints the way in which compilers can generate code and it also makes it rather hard for the programmer to write efficient code that can run on such pipelines because after all stall cycles are wasted cycles and they waste computational throughput. So, let us look at the same example once again the similar example where we write to r1 and the 2nd instruction reads from r1.

So, in this case, the 1st instruction which is instruction 1, this seamlessly goes through the pipeline no problem and as we had discussed before for instruction number 2, it tries to read the value of r1 in cycle 3, we happily allow it to read the wrong value it does not matter

because after all it will not use it in cycle 3, it will use it in its EX stage which it will enter in cycles 4.

So, at that point of time, if we take a look at where instruction 1 is? instruction 1 has already computed the result which means it has already added r2 + r3, it has done the addition and the result is there in its instruction packet. However, the result has not been written back to the register, it does not matter, the result is still present in the instruction packet of instruction 1 which is what we need.

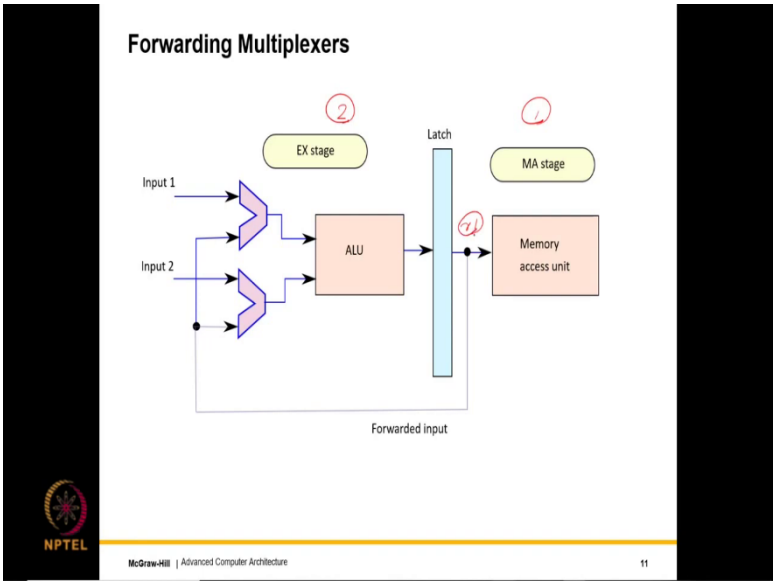
So, what we can do is given when the result is present here in the MA stage, we can simply what is called forward it to the EX stage. So, this is being done right at the beginning of the 4th cycle such that the data will reach and once the computer data, the data for r1 reaches the EX stage, the addition can happen which is r1 + r4, this addition can happen and the result will be available at this point which is the end of cycle 4.

So, what we can see over here is that by doing a forwarding of this nature, we are avoiding stall cycles altogether, there is no reason to add stalls, there is no reason to add any kind of bubbles or pipeline delays and the reason for that is essentially because we have the result in the pipeline.

It is just being forwarded to another stage internally we are just forwarding it not via the register file because if you were to forward it via the register file, we would have to wait for the end of the 5th cycle and then, we would forward it and then, instruction 2 will have to rewrite over here and that of course, would be a delay that of course, would be a big and major delay something that we do not want.

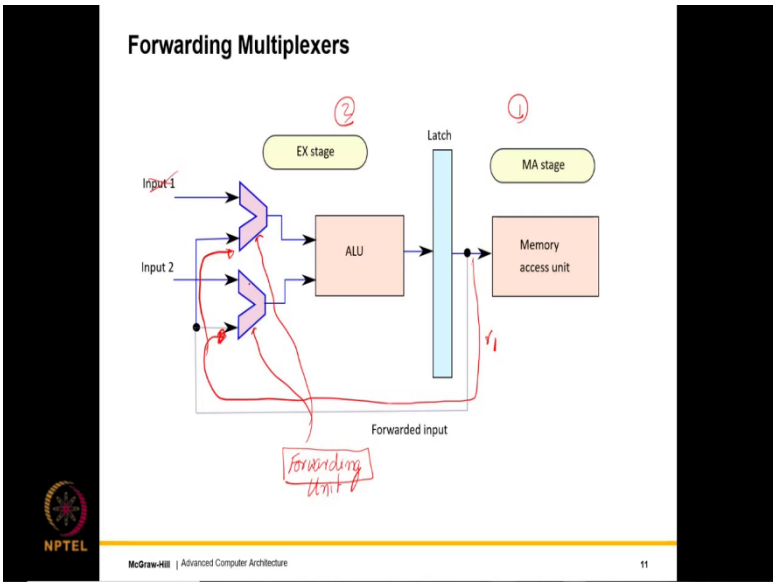
Hence to stop this delay from happening, what we do is we internally forward the data from the MA stage to the EX stage and as you can see, the instruction can execute correctly both the instructions, instruction 1 as well as instruction 2 and instruction 2 can get the value and it can then compute its result which is r5 correctly. So, r5 can be computed correctly, we can add r1 and r4 and we can compute r5.

(Refer Slide Time: 39:08)



So, how would we exactly implement this in practice? Well, this is very easy. So, look at the two inputs of the ALU, so, we have input 1 and input 2. Inputs 1 and 2 come from the previous stage, which is the OF stage, but what we do is instead of directly giving them to the ALU, we add a multiplexer, and the other input of the multiplexer is the forwarded input, the forwarded input comes from the next stage, comes from the MA stage.

(Refer Slide Time: 40:05)



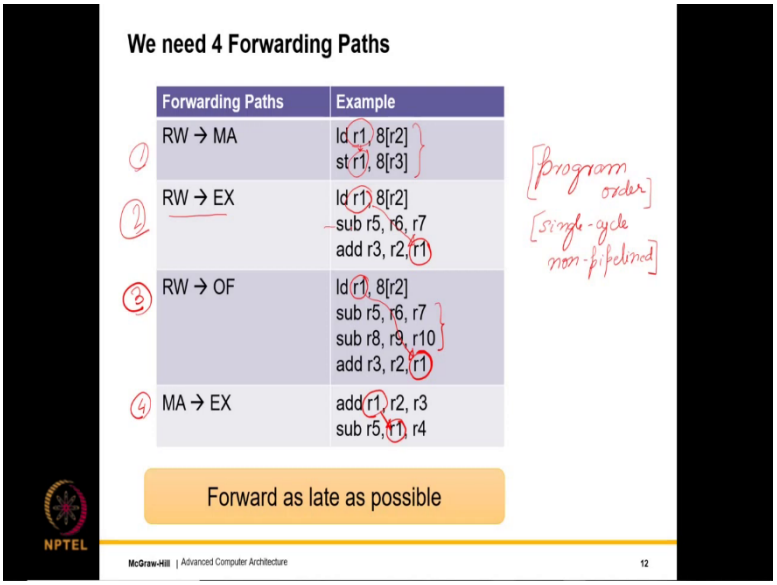
So, if these were instructions 1 and 2, so, instruction 1 would be over here, instruction 2 will be over here and you can see the destination is r1. So, r1, the value of r1 will be there in the instruction packet of instruction 1, this will flow this way and it will come here. So, this is the value of r1, the value of r1 will flow this way and it will come here and we can use it.

So, in this case, r1 was the first input, so, r1 instead of coming here, will actually come here and r1 can then be used. So, what you can then see is what this multiplexer will do is that it will discard input 1 because input 1 which is coming from the OS stage is not correct instead it will use the forwarded input which is essentially the temporary result computed by instruction 1 which is resident over here in the MA stage.

It will use that as the first input of the ALU and as we can see it is correct because the r1 is the first input r4 is the second input and r1 in this case is being forwarded from the next stage which is the MA stage or the memory access stage. So, this is where r1 comes from and similarly, if we can give the forwarded input to the second multiplexer for a different instruction combination.

So, we have a dedicated forwarding unit it is called a forwarding unit. So, the forwarding unit computes the controlled signals of these multiplexers and figures out which input should be selected, the forwarded input or the default input that is coming from the previous stage. So, if we r2 effect a forwarding, then the forwarding multiplexer chooses the forwarded input.

(Refer Slide Time: 42:06)



So, this is explained in a lot of detail in the previous book, so, I will not go into the exact derivation of this, but I will just state the result somewhat without proof that for the simple pipeline that we have described, we will actually need 4 forwarding paths not more than that.

So, we will follow two axioms. One is obvious I am not mentioning that so, that is that we always forward from a later stage to an earlier stage which if you see this diagram. So, we can always forward from here to here or forward from here to here from always from a later stage to an earlier stage depends on how you define? So, let us say later I am defining something to the right and earlier is something to the left.

The other is that we try to forward as late as possible. So, this is the design choice that we make such that we can reduce the number of forwarding paths and also, we forward data only when it is just about immediately required, so, that is the other principle that we use. And this does not tamper with correctness because in any case before a value is being used by a functional unit, it is there so, we need 4 forwarding paths, so, one is RW to MA.

One example of RW to MA would be a consecutive load store instruction pair where we load a value into the register r1 and in the next instruction, the instruction just after that in program order. What again this program order? It is the order of instructions, the dynamic order of

instructions in the program and so, basically the same order that will be seen by a non-pipeline processor that is program order, so, we will use this.

So, it is the same order of instructions as they appear to execute by a single cycle non-pipeline processor. So, a processor that just picks one instruction and executes it picks one, executes it. So, it will see a dynamic string of instructions which are set to be in program order. So, assume that it is the case that after a load we have a store, we load to register r1 and then, we store r1 to another memory location.

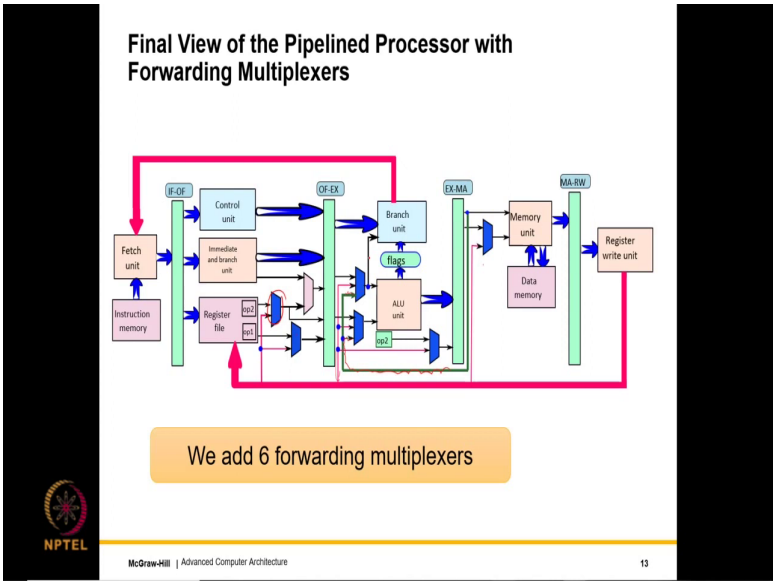
So, in this case, what we have is that we need RW to MA forwarding to execute this piece of code correctly. Similarly, we need to add the 2nd forwarding path which is RW to EX. So, in this case, we load into register r1, then we have another instruction we do not care and finally, we have an instruction that reads from r1. As you see, this is another read after write dependency, this will use the RW to EX forwarding path.

Then, the 3rd one is RW to OF. Well, again here we load to r1, we have two the subsequent two instructions are totally independent, can be totally independent does not matter, but we need another forwarding path here from RW to OF, if the 3rd instruction reads from r1 again a read after write instruction. So, as we can see from the last stage, the RW stage, the register write backstage, we need 3 forwarding paths to the 3 earlier stages.

And then comes the forwarding path and example of which we just saw the MA to EX forwarding path. So, we have just seen this where we have consecutive ALU instructions where the earlier instruction writes to one register the later instruction reads from it. So, this is a read after write dependence pattern where we write to r1 and then, we read from r1.

So, essentially, in the previous book, it has been proven that these are the only 4 forwarding paths that are required. So, only these 4 paths are required, no other path is required. So, we will not get into the proof that I mentioned, but again of course, a given the longer pipeline, we will have many many more such forwarding paths and that is why we often avoid having very long in order pipeline because they would necessitate extremely complicated forwarding logic, lot of multiplexers, lot of paths.

(Refer Slide Time: 47:16)



So, let me again show you a bird's eye view of the entire pipeline. So, recall that this processor we have seen with the pipeline version, what is extra? What is extra are these blue multiplexers, there are total of 6 of them 1, 2, 3, 4, 5 and 6, these are the forwarding multiplexers which are controlled by a dedicated forwarding unit.

What is the role of the forwarding unit? Well, the forwarding unit has a complete global view of the pipeline, it as a view of all the instructions in the pipeline. So, it decides which instruction moves, which stops, which stalls and it also decides who forwards where. So, from the register write unit whatever is written, we can clearly see this is the RW to MA forwarding path, this is the RW to MA forwarding multiplexer.

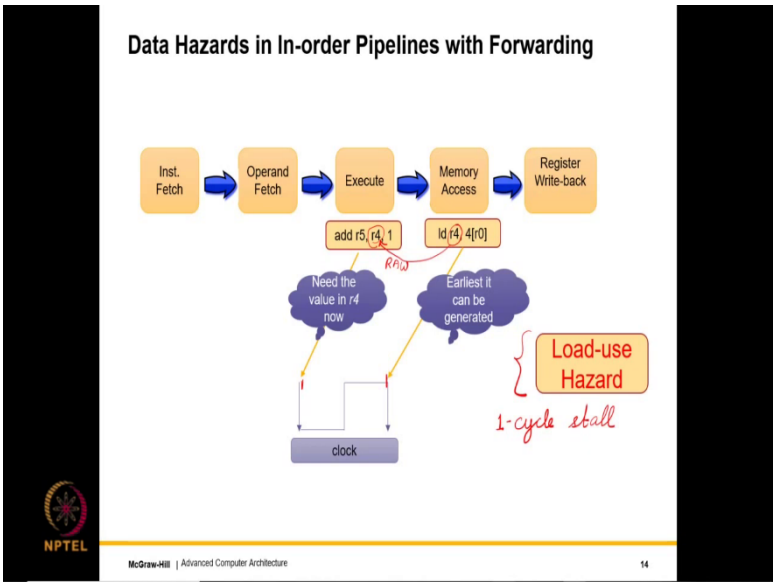
This is the RW to EX forwarding paths as you can see the input goes to both the multiplexers and this is the RW to OF forwarding path. Similarly, we have an MA to EX forwarding path which is this green line over here the one that I am taking you through, so, this goes to both the multiplexers. So, the multiplexers here have a choice of three inputs; two of these are forwarding paths and one is a default input that comes from in the previous stage.

So, as I said if we have a longer pipeline with more units, we will add more multiplexers that will make our forwarding logic more complicated, but we have whether we will derive whether it is an advantageous thing or not we will see, but this to you is an entire functional

working and also commercially viable in-order pipeline that has all the forwarding multiplexers that are needed which are the multiplexers with a blue colour.

So, this is pretty much where a traditional course and computer architecture will stop in so far as pipeline designing is concerned. There are a few corner cases which we will discuss in the rest of the chapter, but this is pretty much the meat and bones of pipelining.

(Refer Slide Time: 49:50)



So, now let us look at one hazard which forwarding cannot take care of. So, this is a very celebrated example, it is called a load use hazard. So, here, we will see that a one cycle stall is required, and this will also motivate us to think further and this has implication important applications in the design of out of order pipelines as well.

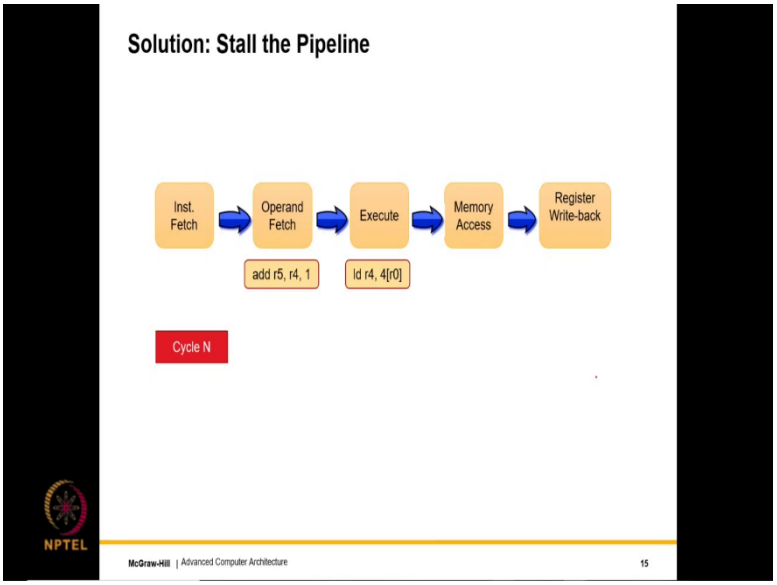
So, consider these two instructions add r5, r4, 1 which is a later instruction and consider this instruction load r4, 4 r0 which is the earlier instruction appears earlier in program order. So, there is a dependency, so, r4 is being written to r4 is being read from so, there is a read after write dependency that we can see.

So, consider a clock cycle. So, in this clock cycle, the earliest the value of r4 can be generated is essentially the end of the clock cycle when the memory axis is finished. Consider the earlier instruction when do we need the value of r4? Well, we need it at the beginning of the clock cycle because after that we need to add 1 to r4 and produce r5, so, r4 is needed at the

beginning so, mind you the these are the same clock cycles. So, at the beginning of a clock cycle we need r4 whereas, the earliest that the forwarding path can give us the value of r4 is at the end of the clock cycle.

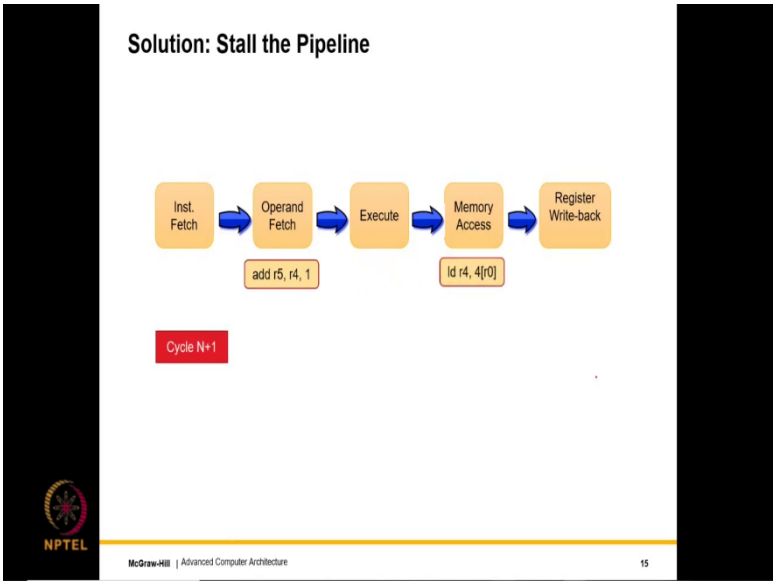
So, in this case, this is called a load and a use hazard because this is a load and this is its use, so, the load use hazard does not have a solution with forwarding, we have to have a 1-cycle stall, there is no choice otherwise, there is absolutely no choice, we need a stall needs to be done.

(Refer Slide Time: 52:02)



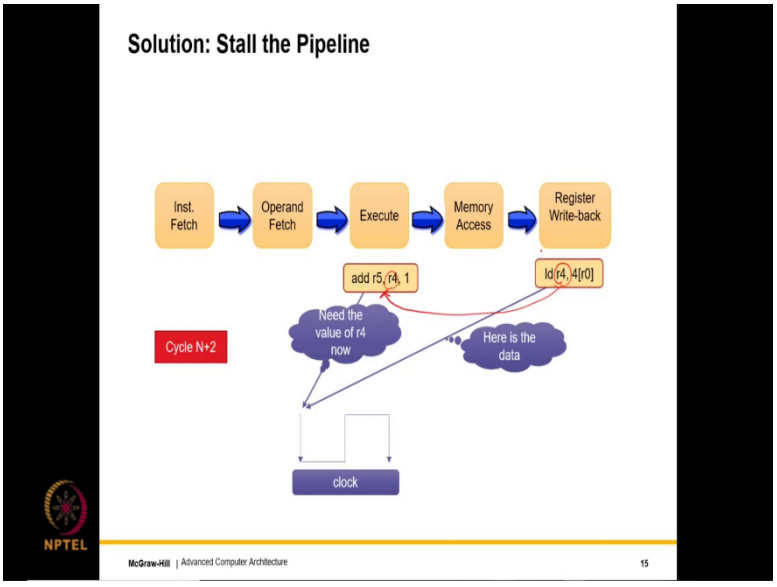
So, what do we do? We have a small animation over here that shows you the picture. So, this picture over here tells you that look at the operand fetch stage at cycle N, we allow, we detect that look there is a load use hazard nothing can be done, we need to stall so, we stall the add instruction, we make the load instruction move.

(Refer Slide Time: 52:13)



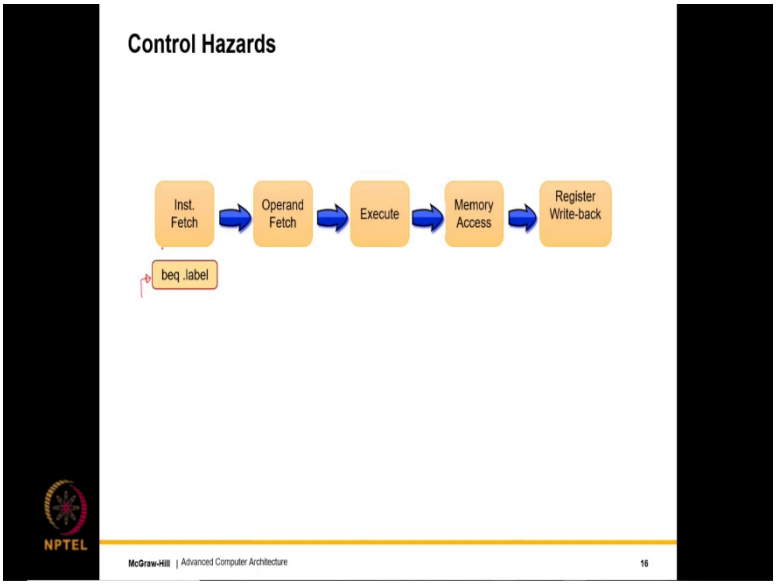
In cycle N plus 1, both the instructions move.

(Refer Slide Time: 52:32)



And in cycle N+2, this is the situation, and this is where a forwarding can be done where the r4 can be supplied and since we need the value of r4 now, well, the value of r4 can be happily supply to the EX stage. So, this is an example of a load use hazard as we just saw where a combination of interlocks and forwarding both actually solve the problem.

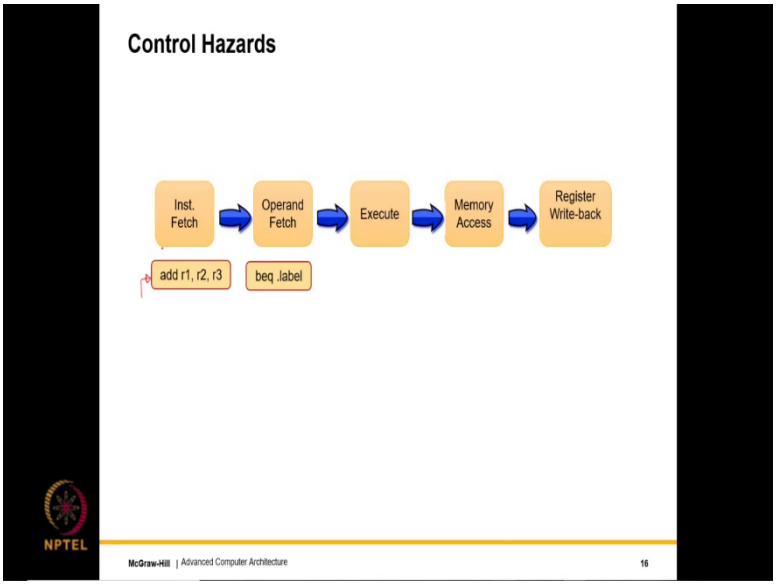
(Refer Slide Time: 53:03)



Now, we have looked at data issues. So, with data issues we have seen forwarding is mostly fine, the only point at which it does not work is when we have issues with the load use hazards that is the only exception. Now, let us look at control hazards. Control hazards are branches particularly, conditional branches which can be taken or not taken.

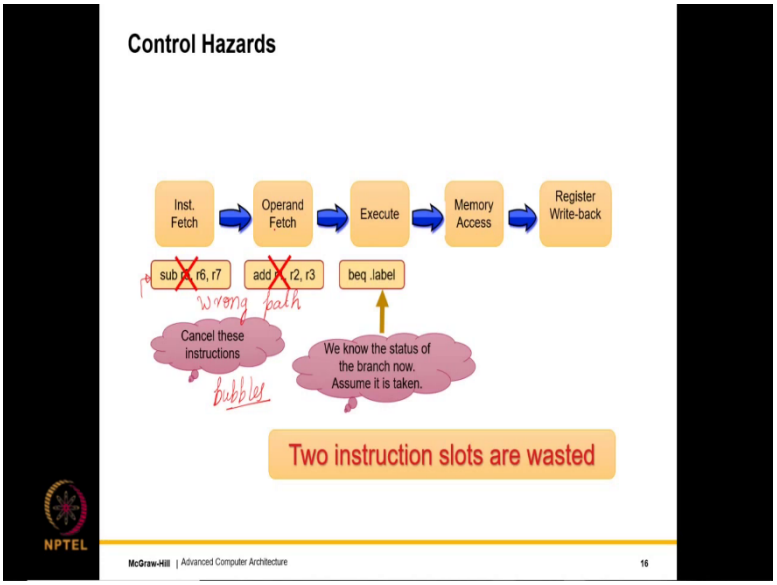
So, here I am showing one example of the beq instruction branch of equal which means that if the earlier comparison resulted in the equality, the branch is taken otherwise it is not.

(Refer Slide Time: 53:50)



So, the branch instruction keeps moving, so, as it moves, other instructions keep getting added to the pipeline.

(Refer Slide Time: 53:55)



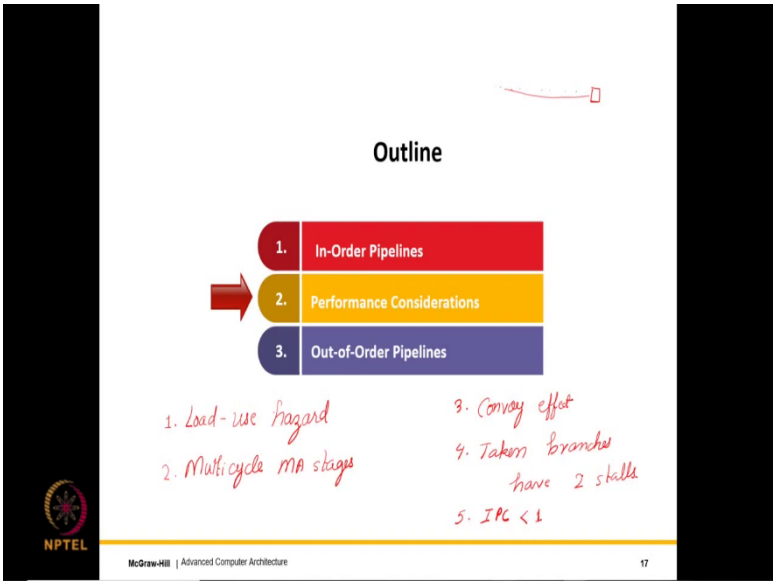
So, of course, these are in program orders say the branch is not taken, these instructions will be on the correct path. So, they will be on their correct path which means then they should be executed, they are logically speaking in program order, so, they will be in the correct path.

But we get to know the status of the branch in the execute stage assume it is taken. In this case, these instructions are on the wrong path because they should not have been fetched in the first place, we should have instead fetched instructions from the branches target.

And of course, we did not know that the branch will be taken, we did not know its target so, we could not have fetched, hence, we ended up with two instructions which are on the wrong path, so, they need to be cancelled. What this means? is that we convert them into bubbles or nops. essentially, you may convert them dynamically into dummy instructions.

So, these two instructions slots get wasted which is a bad thing, which means that every time there is a taken branch effectively, we stall for two cycles, it is a really bad thing is something that we will have to look at.

(Refer Slide Time: 55:11)



So, given that we have looked at in-order pipelines given that we have looked at their limitations, what are the limitations? Well, the limitations are several. The first is that well, assuming even if you are forwarding, one limitation is that the load use hazard well we cannot take care of it.

And also we have made a big assumption that the memory access stage takes one cycle that is not true. So, in general, we can have multi-cycle MA stages that is in fact, that is the norm

because modern caches and memories are rather slow, so, it can take tens of cycles for us to actually get data back.

The second is that we can have what is called head of the line blocking which means that let us say we have a stream of instructions. One instruction is blocked even though the rest of the instructions do not have any dependencies with it just because they made the crime of being after return program order, all of them are stopped, so, this is called the convoy effect.

Consider a convoy of cars, so, consider a set of cars moving in a very narrow road with a single lane. If there is a car break down, all the cars behind it just stall over there say in-order pipelines will have this convoy effect and of course, branches are not dealt with correctly, so, taken branches have two stalls.

Well, there are other problems as well, the other is that the number of instructions that we can process per cycle that is typically not one, it is less than 1. We will take a look at it which is why the next section is titled performance considerations.

So, we will take a look at the performance, but essentially what we will show is that in a pipeline processor IPC, the instructions per cycle is typically less than 1. So, we process for an outside observer the number of instructions that are being processed per cycle will appear to be less than 1, it is again a bad thing we want to process as many instructions in parallel as possible.

So, clearly inside the processor, there is no parallel processing. So, this is why in order pipelines are by and large not used for you know large processors in industry, they are not considered practically. So, now, we will kind of quantify the performance issues and move towards motivate gradually motivate ourselves towards out-of-order pipelines that will solve all of these issues pretty much to a very large extent.