Computer Architecture

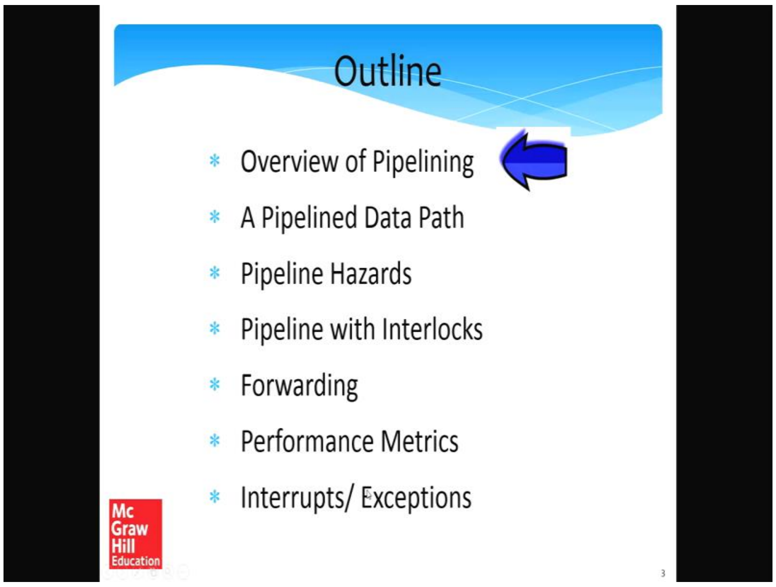
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 26
Principles of Pipelining Part-I

Welcome to Chapter 9. In this chapter, we will discuss the Principles of Pipelining. So, pipelining is by far the most important technique for improving the performance of the processor, so it is speeding them up. So, there is no other technique, that is, you know much more, that is more important than pipelining.

So, let us discuss this technique in a fair amount of detail in this chapter. So, this chapter is fairly long we have 125 slides. So, this discussion will be spread across several lectures again. This is chapter of the book “Computer Organisation and Architecture” written by myself, Dr. Sarangi from IIT Delhi. It has been published by McGraw Hill in 2015, and should be available in almost all locations worldwide.

(Refer Slide Time: 01:53)

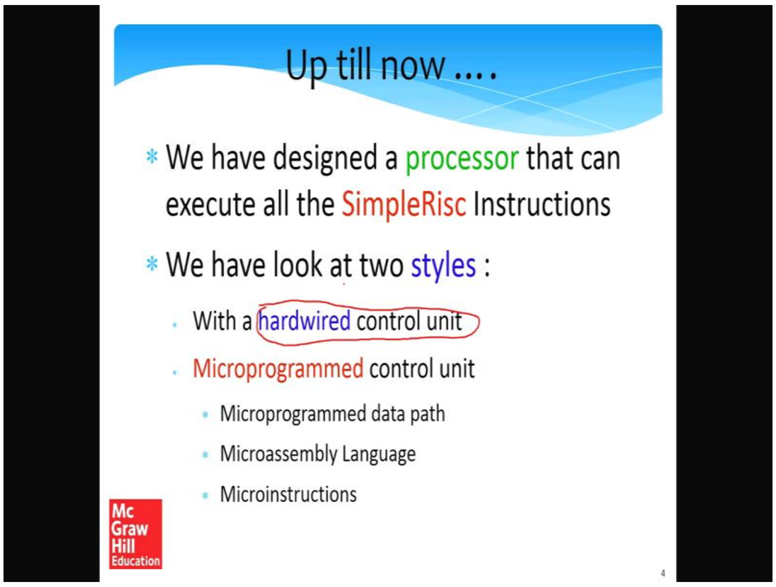


Outline

- Overview of Pipelining
- A Pipelined Data Path
- Pipeline Hazards
- Pipeline with Interlocks
- Forwarding
- Performance Metrics
- Interrupts/ Exceptions

So, the way we divide this chapter is as follows. We will discuss, we will have an overview of pipelining first. Then, we shall discuss the pipeline data path, where we change the data path of the processor to incorporate pipelining. Then, we will discuss several problems with pipelining called pipeline hazards and methods that we apply to fix those problems; pipelining with interlocks and forwarding. We then look at performance metrics for processors. And finally, end with a quick discussion on interrupts and exceptions.

(Refer Slide Time: 02:36)



Up till now ….

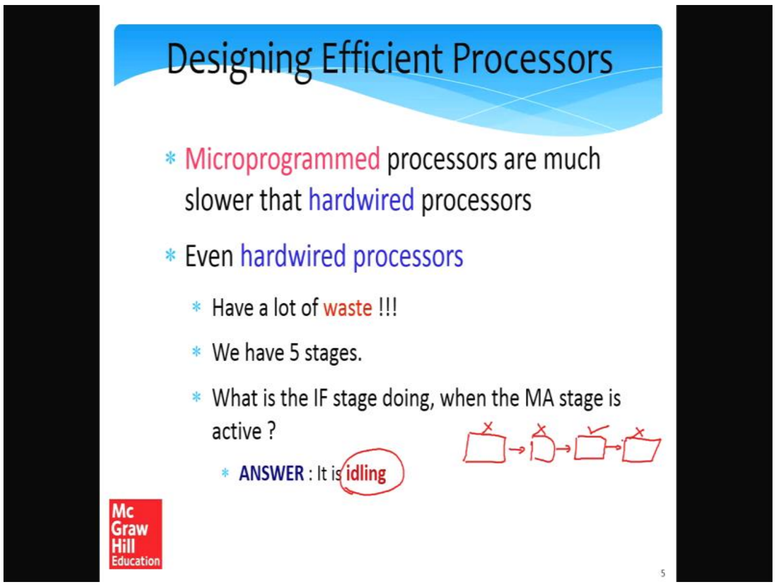
- We have designed a processor that can execute all the SimpleRisc Instructions
- We have look at two styles :
  - With a hardwired control unit
  - Microprogrammed control unit
    - Microprogrammed data path
    - Microassembly Language
    - Microinstructions

4

So up till now, we have discussed the design of a simple processor to execute the SimpleRisc instruction set. We have looked at two kinds of styles. We have looked at a processor with the hardwired control unit, which is faster. And, we have also looked at a very flexible processor that in which you know, even the implementation of instructions can be changed. So, it is called a processor with a microprogrammed control unit. So, we have taken a look at a microprogrammed data path, microassembly language and microinstructions.

So, we will, in this chapter only be working with processors with the hardwired control unit, which is processors of the first type, which is by far are the most common. So, we will only work with this and try to increase the performance of such processors.

(Refer Slide Time: 03:32)



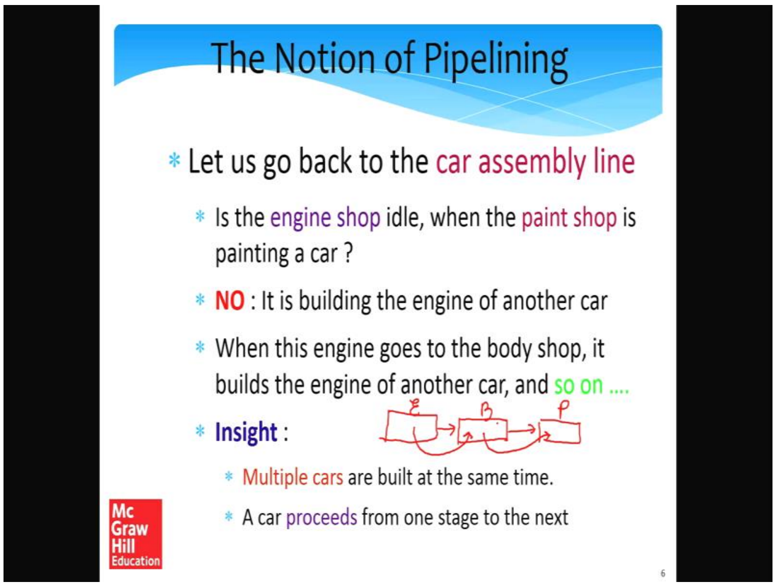
Designing Efficient Processors

- Microprogrammed processors are much slower that hardwired processors
- Even hardwired processors
  - Have a lot of waste !!!
  - We have 5 stages.
  - What is the IF stage doing, when the MA stage is active ?
    - **ANSWER** : It is idling

So, let us look at how to design an efficient processor. So, we will only look at hardwired processors here. So, what we claim is that there is a lot of waste, in the sense there is a lot of idling inside the processor. Most of the time, a lot of stages inside the processor are doing nothing. For example, what is the I F stage doing, when the MA stage is active? Well, the answer is it is doing nothing. It is just idling. So, it is absolutely idling and it is doing nothing at all. And, similarly what is maybe the EX stage is doing, when the MA stage is active? So, then also answer is there. It is idling and it is doing nothing at all.

So, this tells us that if there are 5 stages, only one of the stages is active at any instant of time you know these are the 5 stages; and, only one of the stages active at any point of time. And, let us say if this is active, the rest of the stages are not doing anything. And, this to us represents the source of inefficiency, which if removed can possibly give us more performance.

(Refer Slide Time: 04:47)



The Notion of Pipelining

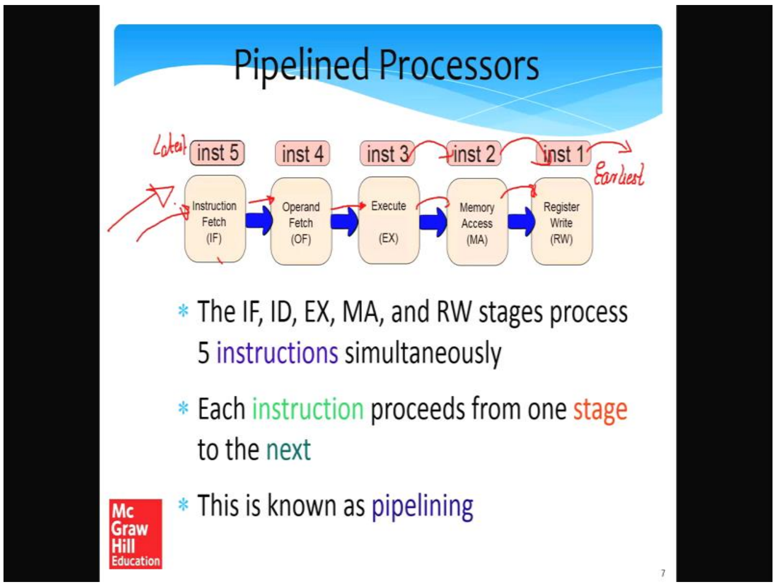
- Let us go back to the car assembly line
  - Is the engine shop idle, when the paint shop is painting a car ?
  - NO : It is building the engine of another car
  - When this engine goes to the body shop, it builds the engine of another car, and so on ….
  - Insight :
    - Multiple cars are built at the same time.
    - A car proceeds from one stage to the next

$$\begin{tikzcd} \boxed{E} \arrow[r] & \boxed{B} \arrow[r] & \boxed{P} \end{tikzcd}$$

So, let us look at the notion of pipelining. So, let us go back to our basic example which is the car assembly line. So, the question is the engine shop idle, when the paint shop is painting a car? Well, the answer is no. So, when we are painting one car, the people at the engine shop are essentially building the engine of another car. So, when this engine goes to the body shop, what does the engine shop do? It builds the engine of yet another car and so on. So, the insight is that multiple cars are built at the same time. And, pretty much half built car proceeds from one stage to the next.

So, let us, may be, draw a diagram. So, let us assume that part of the factory, part of a car factory that is working on the engine. And then, this part of the car factory, let us say works on the body and this part of a car factory works on the paint. So, once an engine moves from the engine shop to the body shop, well, the engine shop does not idle. The workers are building the engine of another car. Similarly, the moment the body of a car has been made, the body shop is not idle, instead we build the body of another car. So, can we use the same logic in a processor? We shall see it.

(Refer Slide Time: 06:16)



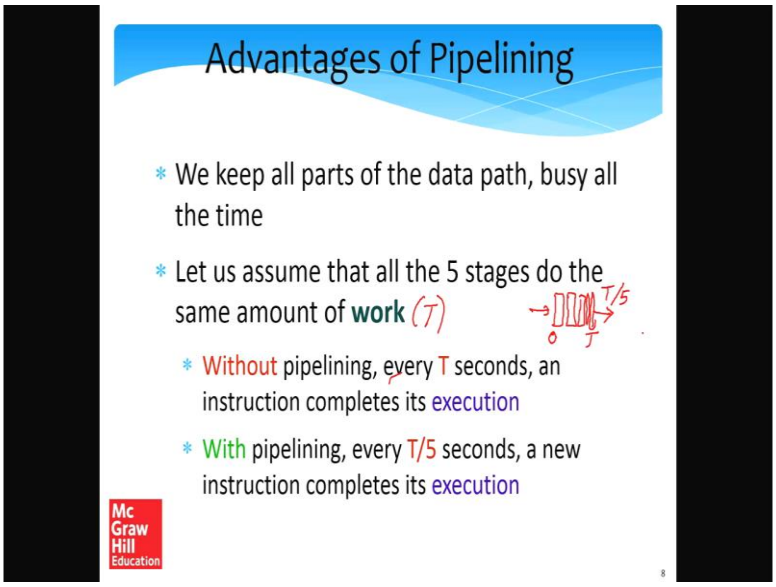
Pipelined Processors

- The IF, ID, EX, MA, and RW stages process 5 instructions simultaneously
- Each instruction proceeds from one stage to the next
- This is known as pipelining

So, let us consider a example of a pipeline processor. So, let us looks at the 5 stages instruction fetch, operand fetch, IF, OF, execute stage EX, memory access MA and register write RW. So, these are the 5 stages; IF, ID, EX, MA and RW. And, the stages process five instructions simultaneously. Let us assume that this is the case. So, this will be similar to a car factory, where a car factory also has multiple shops and a half built car move from one shop to the next. But, no shop is ever idle. So, one shop is building the body for one car; another shop is building the engine for yet another car.

So, if we straight forward bring the analogy over here, this is what we get to see. That if we divide our processor into the 5 stages, we can have one instruction in the RW stage, one instruction in the memory access MA stage, one in the EX stage, one more in the OF and IF stages, respectively; one more in the OF, one more in the IF. So, we can have five instructions. So, the earliest instruction would be in the later stages. So, this would be the earliest and this instruction would be the latest. And, as an instruction’s fetch is over, it will move to the; as IF is over, it will move to the OF stage and as operand fetch is over, it will move to the EX or execute stage so on and so forth. So, this process is known as pipelining. It might sound simple, but it is incredibly complicated.

(Refer Slide Time: 08:10)



Advantages of Pipelining

- We keep all parts of the data path, busy all the time
- Let us assume that all the 5 stages do the same amount of **work** (T)
  - Without pipelining, every T seconds, an instruction completes its execution
  - With pipelining, every T/5 seconds, a new instruction completes its execution

So, let us first take a look at what is the advantage of pipelining. So, the advantage of pipelining is that we are keeping all the parts, all the parts of the data path busy all the time. So, in this case if I can go back, what we get to see is that if in the data path we have 5 stages, the 5 stages are busy all the time. Basically, similar to a car factory, no part of a car factory is actually idle. And, same is the case here. That all the stages are busy all the time. And, one instruction process finishes its execution in one stage, moves to the next stage; finishes execution here, moves here. If we have 5 stages, we will have five instructions being processed simultaneously. And then, at the end of the processing, one of the instructions will move out, the second instruction will come here, the third will come here and a new instruction will enter the fetch stage.

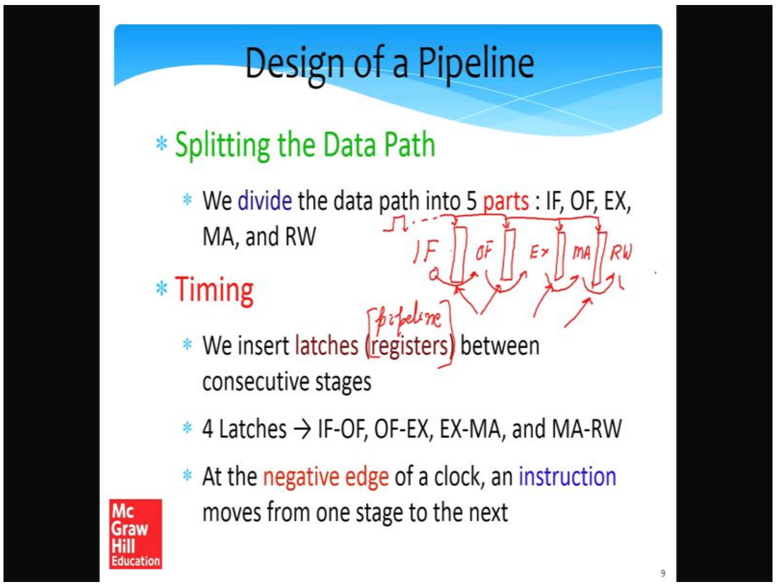
So, the idea sounds simple but the question is; where is the advantage. So, let us see. We will have a much deeper discussion on advantages of pipelining towards the end of the chapter. But, let us at least get an initial preliminary idea. So, let us assume. So, this is simplistic assumption. It is regularly made that all the 5 stages do the same amount of work. If we make this assumption without pipelining, so let us assume that the same amount of work means T seconds of work.

So without pipelining, every T seconds an instruction will complete its execution. So, which means if there an instruction enters the processor here at t equal to 0 after capital T seconds, the instruction will get out; which get out means it will have finishes its execution. So, means that it will, if it is a memory instruction that will finish the memory access and if a register needs to be written to it, it will write to the register. So, it will essentially complete the work of all the 5 stages. So, with pipelining what is happening that instead what we have done is that we have broken down this circuit into five parts; part one, part two, part three, part four and part five. So, in one, two, three, four, five, what we see is that in every T by 5 seconds; in every T by 5 seconds because now remember that since we have broken down the data path into five parts. So basically every part will take; instead of T seconds, it will take T by 5 seconds; which basically means that you know for an instruction to move from here to here, it will take T by 5; to move from here to here, it will take T by 5, so on and so forth.

So, basically which means that if for somebody who standing over here, so consider an observer who is standing over here and looking at what is happening. The observer will see that every T by 5 seconds, a new instruction is coming out. But, if you know instruction, the processor was not pipelined and the observer is standing over here and observer is looking, what the observer will see that one instruction enters at T equal to 0. At T seconds, it finishes. In a sense, it comes out.

Then, again one more instruction comes. In every T seconds, one instruction comes out. And, in this case every T by 5 seconds, one instruction comes out; which means that let us say after N seconds, the number of instructions that would have successfully executed in a pipeline processor is N divided by T by 5, which is 5 N by T. But, here after N seconds, the number of instructions that would have completed is N by T.

(Refer Slide Time: 12:45)



Design of a Pipeline

* Splitting the Data Path
  * We divide the data path into 5 parts : IF, OF, EX, MA, and RW
* Timing
  * We insert latches (registers) between consecutive stages
  * 4 Latches → IF-OF, OF-EX, EX-MA, and MA-RW
  * At the negative edge of a clock, an instruction moves from one stage to the next

IF
OF
Ex
MA
RW
pipeline

So, a pipeline processor in this case in its most simplest form can process five times more instructions than a non-pipelined processor. So, so that is the basic idea that you know that that is where the expected speed up will come from. That it can process more instructions at the same time. And also, if we are looking for, an observer who is looking at a pipeline processor from outside, every T by 5 seconds, a new instruction is coming out. So, I will just go back to the diagram. Every T by 5 seconds, a new instruction is coming out. So the instruction, the rate of instructions coming out is five times more as compared to a non-pipelined processor. As a result, if a program is N instructions, then you know it will execute five times faster.

Now, let us take a look at the design of a pipeline. So, the first thing that we need to do is that we need to split the data path among the 5 stages in a nice and clean way. So, we need it to divide it into five parts such that each part can contain a separate instruction.

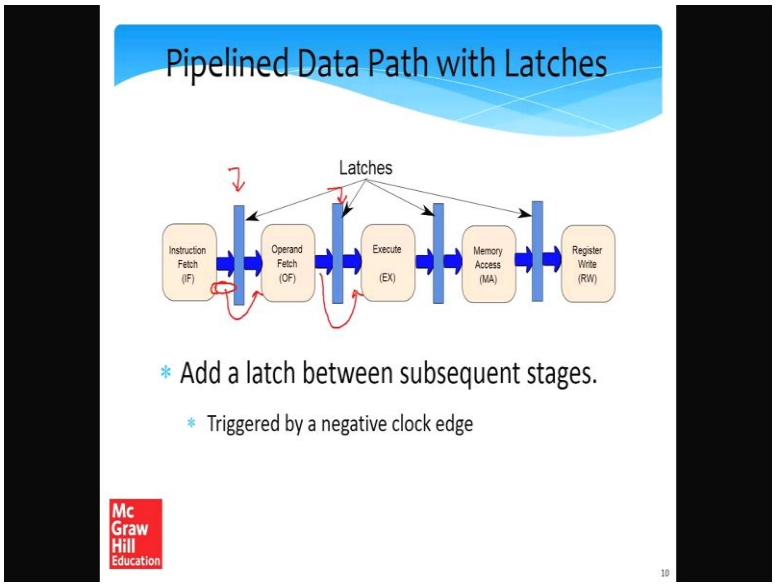
So, regarding the timing what we need to do is that we need to insert latches. They are not latches. They are actually flip flops because they are contained by; because they are controlled by a clock, so we will also call them pipelined, pipeline register or just registers. Well, just registers will confuse us with register in register files. So, let us call them pipeline registers. So, they will also call by latches. But by latches, here we mean that they are also connected to a clock. So, this is a little bit of an abuse of the symbols here. The little bit of naming here.

But, you know it should be clear from the context. So, these pipeline register are the pipeline latches. Let us add them between consecutive stages. So, we will add four latches. What we will do is we will have IF, then the IF stage will write its result to set of, you know, flip flops that are controlled by a clock set of pipeline, set of pipeline registers.

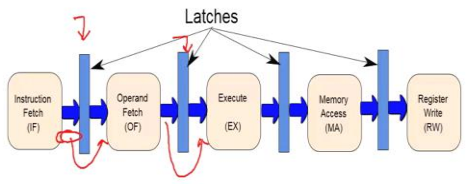
Then, we will have the OF stage. Again, a set of pipeline register or pipeline latches. Then, again the EX stage the same thing MA stage. And, the register write stage RW. So, what we shall see over here is that we are adding four pipeline latches. So, let us call this the IF-OF latch because it is between IF and OF stages. Similarly, we will add an OF-EX latch and EX-MA latch and a MA-RW latch. So, these latches are actually controlled by the clock. Say if this is the clock you know, so the clock goes to all the four latches; here, here, here and here. So, it goes to all the four latches. So, what we, the way we design it is rather at the negative edge of a clock, one instruction which is in one stage will move to the next stage.

At every negative edge of the clock, instructions move from one stage to the other, one stage to the other, one to the other. And, at the end they go out of the pipeline means that they finish their execution.

(Refer Slide Time: 16:19)



Pipelined Data Path with Latches

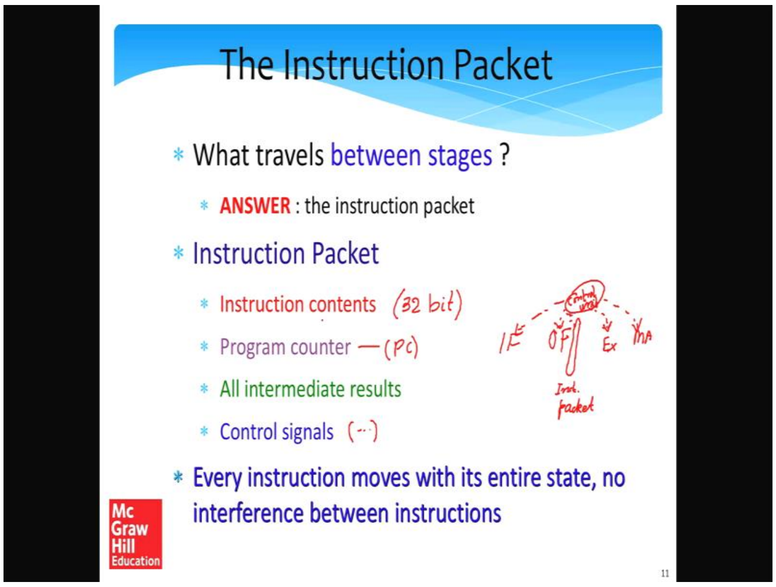


- Add a latch between subsequent stages.
  - Triggered by a negative clock edge

10

So, this is what a new diagram looks like. That we have these pipeline latches. So, mind you again a latch here is actually a flip flop. It is controlled by the negative edge of clock. So, we add four latches between the stages. And every negative edge, the instruction that is over, being processed in this stage, moves to this stage. Every negative edge, instruction here moves here. So, we add latches between subsequent stages. And, the latch is triggered by the negative clock edge.

(Refer Slide Time: 16:52)



The Instruction Packet

- What travels between stages ?
  - **ANSWER** : the instruction packet
- Instruction Packet
  - Instruction contents (32 bit)
  - Program counter —(PC)
  - All intermediate results
  - Control signals (--)
- **Every instruction moves with its entire state, no interference between instructions**

Control unit
IF
OF
EX
MA
Inst. packet

11

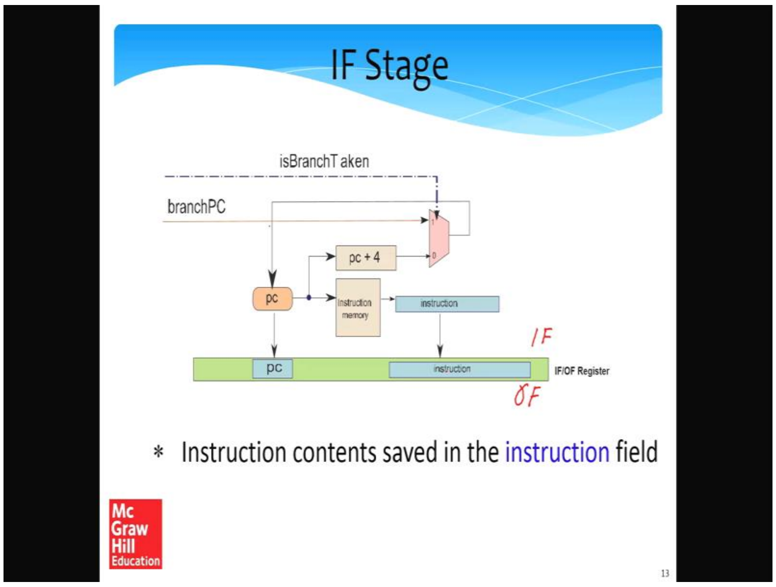
So, what is it that travels between the stages? We have already answered this in the previous slide that the instruction packet travels. So, what exactly is the instruction packet? Well, the instruction travels is an instruction packet. But, what exactly is in there? So, we have the contents of the instruction; which means instruction itself. It is 32 bit contents. Then, the program counter that is associated with the particular instruction. All the intermediate results that we compute, and they, all the intermediate results we compute and the result that are required in the subsequent stages, they also move along with the packet. And, all the control signals that we required to control the multiplexes in the subsequent stages.

So, basically every instruction will move with its entire state, such that there is no interference between instructions. Previously, what was happening is that the control unit? So, you know previously what was happening is that the control unit, which was there in the OF stage was sending the control signals to, well, it was actually dotted lines. Let us see. Let me remove this line. The control unit for sending control signals to all the stages, EX and so on. But, in this case this is not possible. The reason it is not possible is basically because we have five instructions. So, the control unit generates the control signals for which instruction?

So, either we need to have five control units, which is not reasonable idea. So, what we do is that when an instruction process to an IF to the OF stage, it collects all the control signals that are required in a subsequent stages that is the part of the overall instruction packet. Let us call it inst packets. And, inst packets moves from stage to stage. So, in some stages some information is removed; in some stages some information is added. But, broadly the instruction packet moves. And, what does it contain? It contains the contents of the instruction.

The 32-bit representation of the instruction, it contains the program counter, the PC, it contains all the intermediate results that would be required in the subsequence stages and it contains all the control signals that are required to control the multiplexes in the subsequent stages. So, this is the entire state of the instruction that moves across stages.

(Refer Slide Time: 19:55)



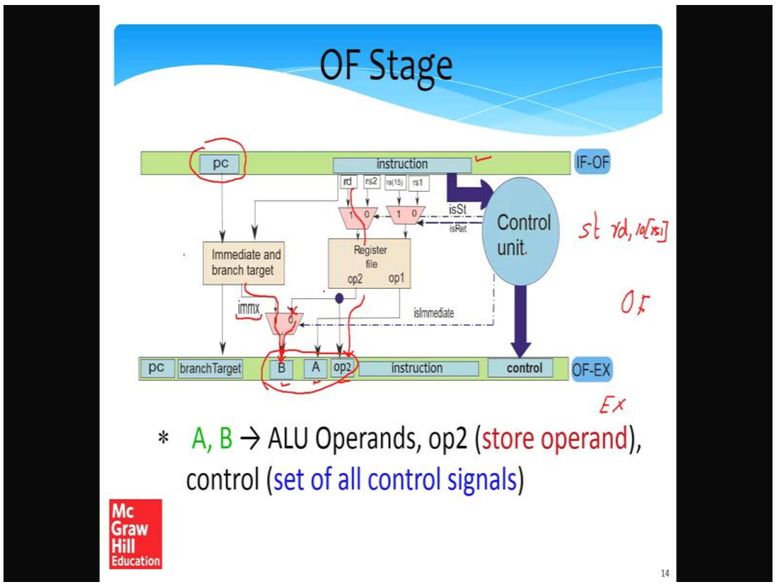
IF Stage

* Instruction contents saved in the instruction field

So, let us look at the design of the pipelined data path. So, almost everything remains the same. I would suggest the readers to take a look at the IF circuit that we had designed in the chapter 8. So, they will find that the circuit is the same. So, there is absolutely no change. It is just that we have this extra green box; the IF-OF register between the IF stage and the OF stage. Is this the pipeline latch or the pipeline register? We will use the term interchangeably.

So, what we store here? We store the program counter because we want this to. So, this is the part of the instruction packet. We store the program counter and the contents of the instruction. So, this is pretty much what we store over here. And, this is what moves to the next stage.

(Refer Slide Time: 20:54)



OF Stage

* A, B → ALU Operands, op2 (store operand), control (set of all control signals)

Let us now consider the OF stage. So, let us; so, OF stage begins from the IF-OF register. So, what did we get from the previous stage? We got the program counter, the PC, from the previous stage. So, this is what we got. And, we got the instruction packet from the previous stage. So, this is one more thing that we got from the IF stage. So after this, the rest of the circuit between the two green boxes, between the two pipeline registers is exactly the same as what was there in the non-pipeline processor. So, it is just that it was drawn vertically and this is being drawn horizontally. That is the only difference, but otherwise that the entire circuit is the same.

Let me nevertheless re explain. So, first is that the instruction goes to the control unit and the control unit generates all the control signals. The control signals are a part of the instruction packet. So, they flow along with instruction. So, this is stored directly in the next register, which is the OF-EX register because of the OF stage and the EX stage. The OF-EX pipeline register, pipeline or pipeline latch saves the control signals. Simultaneously, the control unit quickly generates the two control signals. It is stored is Ret, if you would recall what had been done in chapter 8.

So, my advice is that before taking a look at this chapter, please take a look at chapter 8; the processor design part. So, you can either read the book or take a look at the video and then come back. So, what we had designed in chapter 8 is that for the, register file had two read ports, two interfaces. The first read port, read the first source register or the written address depending upon the is Ret signal and a second read port based on isStore signal, either took in the contents of the rd register or the second source register, which is rs 2.

Then the register file, we have two outputs. We have op1 is one output and op2 is the other output. So op1, we directly save in the pipeline latch and we call it A. Simultaneously we save op2. So op2, we also save it. In the pipeline latch, we call it op2 only. Also, we choose, so depending upon the value of its immediate control signal, whether it has an immediate or not, for the second operand we choose between immx, which is the extended immediate and the second source.

So, basically depending upon the isImmediate flag, we choose one of them. And, we refer to this as B. So, what is the, why are we doing this because if the, isImmediate flag is 1, it means that second operand is an immediate. So, we should take the input from here. And, if isImmediate is 0, it means that the second operand is a register. So, we should take input from here. So, in any case this is saved as, saved in the field B and the first source is saved in the field A. We additionally save op 2 as well. The reason we save op 2 is that we will require this.

And, we will require this because this is a store operand. As there is a special case being made for store. So, let us see why is the special case being made. In the case of a store, the idea is that we will have; so, let us take a look at a store instruction once again. So, let us see. We will have in this case two registers, two source registers and also we will have one offset. So, basically we will store in to register rd. So, mind you for a store, the path is like this that we store rd. We read the result of rd, and the result of rd is basically the value that needs to be stored which comes in over here. But also, mind you that immediate flag is 1 for a store because the immediate stores the offset and which we add to the base register, which is rs1. So, basically the immediate also needs to be stored.

Since the immediate flag is 1, so this will not get stored. The immediate will come in. So, that is the reason the store we will require both op 2, which is the contents of rd. So, let me use a laser pointer. So, we will need op 2, which is the contents of register rd. So, that needs to be stored. So, that will flow via this route because if isStore is 1, then we will access the register file with the contents of rd, which is the destination register. So, recall the special case that we are making for the store instruction. Then, we read that and we store op 2 in the op 2 field and then we store the first source register, which is coming via this route. So, we will store that in the field A in the instruction packet.

And, also in the case of the store, the isImmediate flag is actually true. Since the isImmediate flag is true, we will choose the immediate as the output of this multiplexer, the extended immediate, and that will be stored in field B.

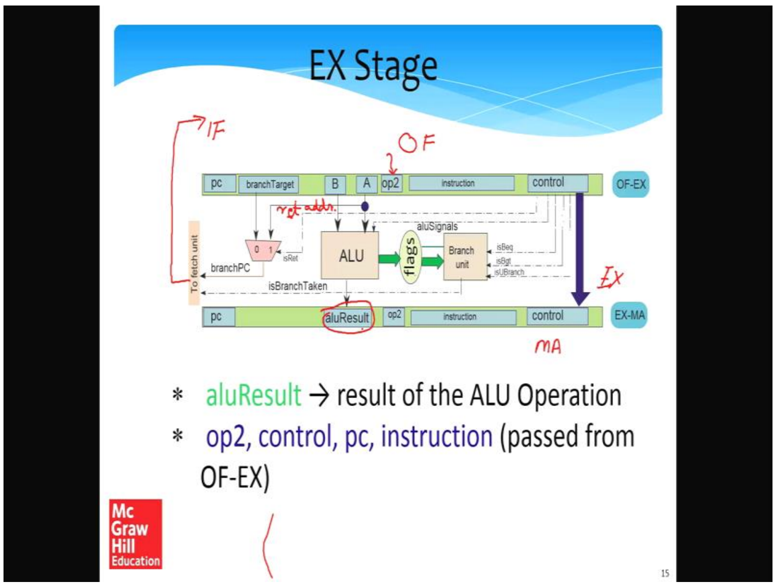
So, now the question that can arise is why do we store op 2 in B separately? Well, the answer is because for a store op 2 will not be chosen by this multiplexer. But, we still need the value of op 2. So, since we need the value of op 2, you better store it separately for the store instructions. For others, this problem does not arise. So, recall that we made a special exception for a store instruction because we did not want to add an extra instruction format. So, that is the reason we store the immediate in B. We store the source register in A (Refer Time: 27:53). And, we store the register rd, which actually contains the data that needs to be stored in op 2. So, the content that needs to be stored are stored in op 2, and the rest contain the immediate and the base register respectively.

For other instruction, this problem is not there. So, basically A contains the first source and B contains the second source. The second source can either be the contents of a register or can be the immediate. So, we again pass the instruction itself from; one from this latch to this latch. We compute the branch target in the branch unit over here in the OF stage. The branch target is also part of the instruction packet and the PC is also part of the instruction packet. So, this is pretty much all the information that is going to the next stage, which is the EX stage. So, we do not need anything else from the OF stage at this particular point of time.

So, those who have, you know those would recall in chapter 8, we had designed the process exactly in the same way. You know no difference at all. But, we did not. The only difference is we did not have these green boxes. We did not have the pipeline latches. So, we were not storing intermediate values. In this case, what are the intermediate values that we are trying to store in the pipeline registers? Well, intermediate values that we are trying to store at B, A and op 2, these are values which are either come from the register file or from the immediate unit. So, let me again erase everything and show you once again. So, the intermediate values that we are storing which are B, A and op 2 have essentially come from the register file or from the immediate unit. So, that is the reason they are being stored in this register. Along with that we are storing control signals, the contents of the instruction and also the branch target in the other intermediate values.

So, that is also what we are storing. And, the PC also passes along with everything else, it passes to the next stage, which is the EX stage.

(Refer Slide Time: 30:16)



EX Stage

* aluResult → result of the ALU Operation
* op2, control, pc, instruction (passed from OF-EX)

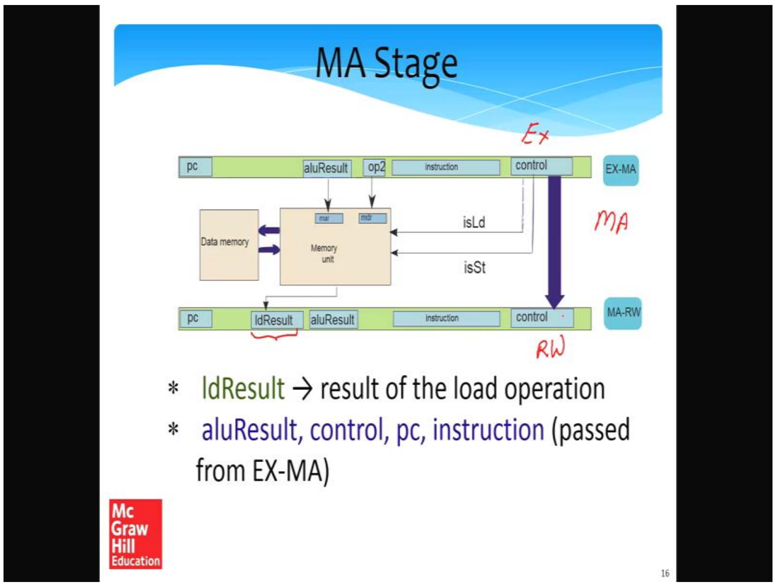
So, in a EX stage these are all the fields that we get from the previous stage, which is OF. So, we get the first ALU operand, which is A. we have the second ALU operand, which is B. In the case of the store, the data that needs to be stored is in op 2. And, the rest is all normal stuff; the instruction, the control for the instruction, the branch target and the pc. So, some of the control signals were out to the ALU. The ALU does its job. And, ALU stores the result in ALU result field of the EX-MA pipeline register. So, EX stands over here; MA stands over here.

So, the results of the ALU are stored in a field, ALU result. And the op 2, we just moved to the next register. And then, the ALU also update its flags and this is processed by the branch unit. The branch unit gets control signals from the control unit is Beq, is Bjt, is U branch. So, I just go back to the relevant part of chapter 8. So, what the branch unit produces is that it produces the control signal isBranchTaken, which needs to go to the fetch unit. And, also we have a multiplexer that chooses between the branch target that comes from the previous stage and the contents of register A, which is which will contain the written address in the case of a written instruction.

So in the case of a written instruction, the contents of, you know, the field A will contain the written address. Otherwise, it will be the branch target. So, using the isRet control signal, we choose between them to compute the branchPC. So, these are then sent to the IF stage, to the fetch unit, to the IF stage. So, this is the one case where something is actually flowing between stages without going via register.

So, we will see. This is an, you know, this is an exception. So, in this case isBranchTaken signal in the branchPC, both are going to the IF stage, but all the rest of the intermediate results are being written to the pipeline register at the end. So, that includes, I am sorry, yes, that includes the control signal, contents of the instruction, the data that needs to be saved, which is in op 2 in the ALU result.

(Refer Slide Time: 32:55)

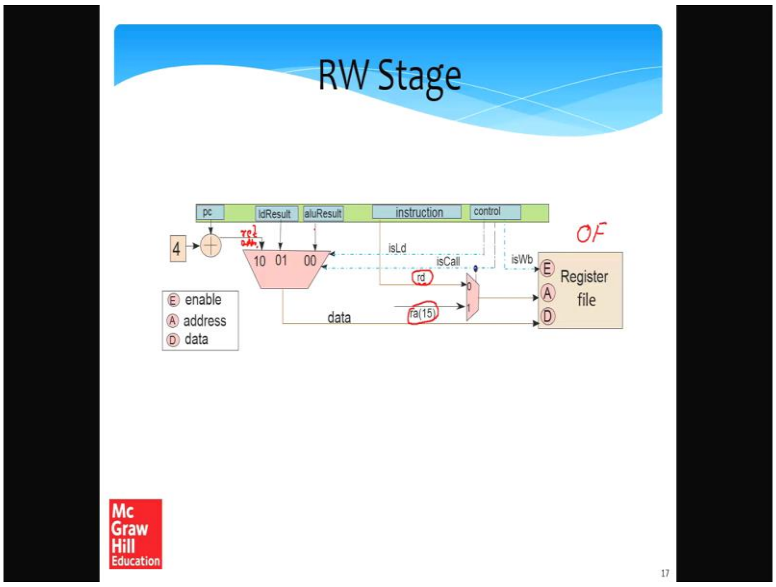


MA Stage

* ldResult → result of the load operation
* aluResult, control, pc, instruction (passed from EX-MA)

Now, we go to the next stage, which is the MA stage; the memory access stage. So, the memory access stage is simple; nothing, no rocket science here. So, it stands between the EX stage and the RW stage. So, the ALU result contains the address. So, this goes into the memory address register. Op 2 contains the data that needs to be stored, so it goes into mdr, memory data register. Then, the memory unit works. It interacts with data. Memory is controlled by two control signals isLoad and isStore. And, the result of the memory unit is stored; in the case of a load, it is stored in the field called load result in the MA-RW register. So, a load result contains whatever we loaded from memory. And, the rest of the fields just continue, the program counter continues. The ALU result that we had computed, the same thing is written in this register; the instruction and relevant control signals.

(Refer Slide Time: 34:06)



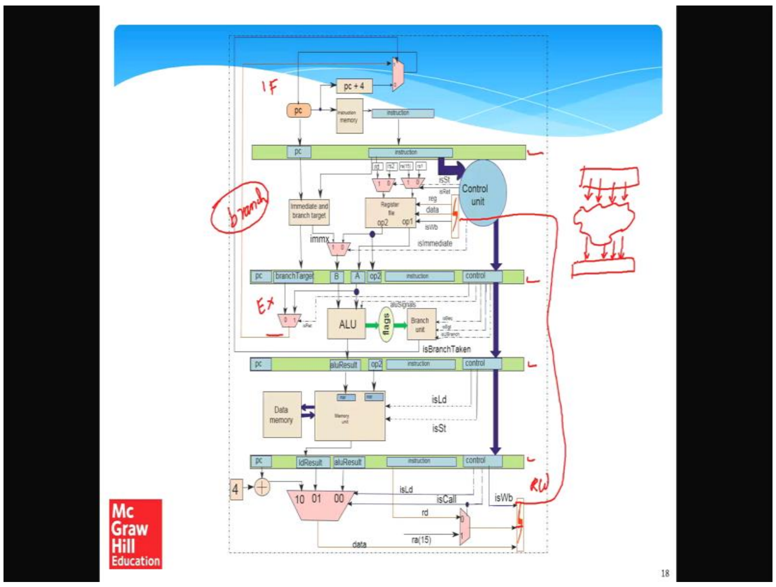
RW Stage

Now, let us move to the last stage called the RW stage. So, the last stage is exactly the same as what we had in chapter 8. In fact, all the stages are exactly the same as we had in chapter 8. No difference at all. And, so here, it is just that we have drawn it differently. So, it looks slightly different. It is the same. So, if you recall we had a three input multiplexer that first read pc plus 4. Pc plus 4 is essentially the written address which the call instruction will use. Then, we have the load result in ALU result. And, it was being controlled by two control signals coming from the control unit; is Load and is Call. So then, that one of the three inputs becomes the data that needs to be written to the register file. So, this directly goes to the register file.

The register file can be located either in the RW stage or in the OF stage. But, OF stage is a better place. So, essentially it goes to the register file in the OF stage. Now, we had one more multiplexer that shows between the rd field and the array field; the array field for the call instruction. So, so basically the isCall control signal chooses between these two values. Whether we write it to a register destination or whether we write it to the written address register, depends upon whether the instruction is a call or not; because the call instruction writes the address of the next instruction to the RA register. So, we make this choice and pass it to the address field of the register file, which tells it which register to write to.

And, we use the isWriteback control signal to enable a write. So, we have seen now in all of this; all of this we have seen in chapter 8. The only addition is a green box. So, all the control signals come from the green box. Previously, they used to come from the OF stage. And, here also all are values come from this pipeline latch over here, and not from other stages. That is the only difference.

(Refer Slide Time: 36:34)



See if I just summarize, this is a diagram of the entire processor. So, there is no. Yes, so, there is, there should be no ambiguity because this is exactly what we have seen earlier. The only differences just are these green boxes, where we store the temporary values and these flow from latch to latch to latch. And, at every stage we do some processing. So, the idea is that every stage we read the inputs from this register and then what we do is that we do some processing and write the outputs to the next register.

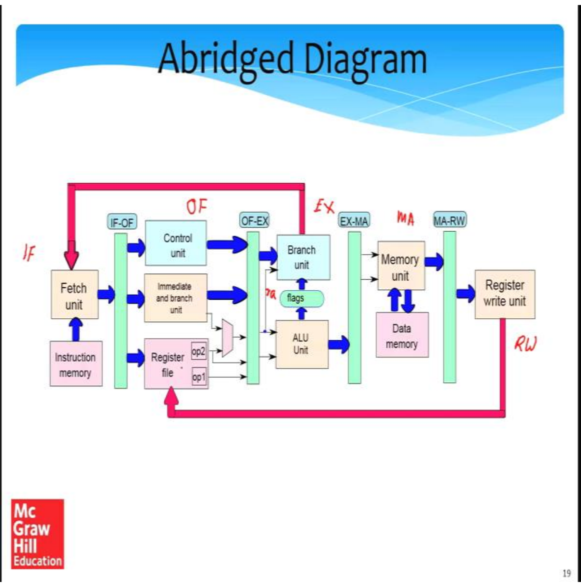
Similarly, in here also we read the inputs and write the outputs to the next register, so on and so forth. There are only two instances. So that is; this is the normal flow of action.

So, the flow of action is that we have this one green box, which is the pipeline latch. At the beginning of a clock cycle which means at a negative edge; we get data from the pipeline latch. We have a cloud of logic which essentially processes all the data. And, we can access memory's elements as well, then all the results of this activity are written to another latch and we then finish.

There are only two instances where data is directly sent between stages. So, one is the case of a branch, in a taken branch, where the fact that the branch is taken and the new target are sent from the EX stage to the IF stage, right, using these wires. And, the other instance is when we write to something in the RW, in the register write stage, so that is when all the values if you see essentially this is like (Refer time: 38:33). So, I did not want to draw a long wire. So, pretty much is the value is entered here and they come out here, which is same as you know taking the wire and going all the way here. And, it is just that I did not want to draw that many long wires. So, I used this convenient trick. But, essentially when you are doing a register write, the id of the register, the data and the fact that we are writing, these pieces of information need to go from the RW stage to the OF stage.

So, these are the only two instances in our data path where something flows; some kind of data flows between stages without going via pipeline registers or pipeline latches. So, these are the only two examples and they need to be kept in mind. The rest of the circuit is the same.

(Refer Slide Time: 39:28)



Abridged Diagram

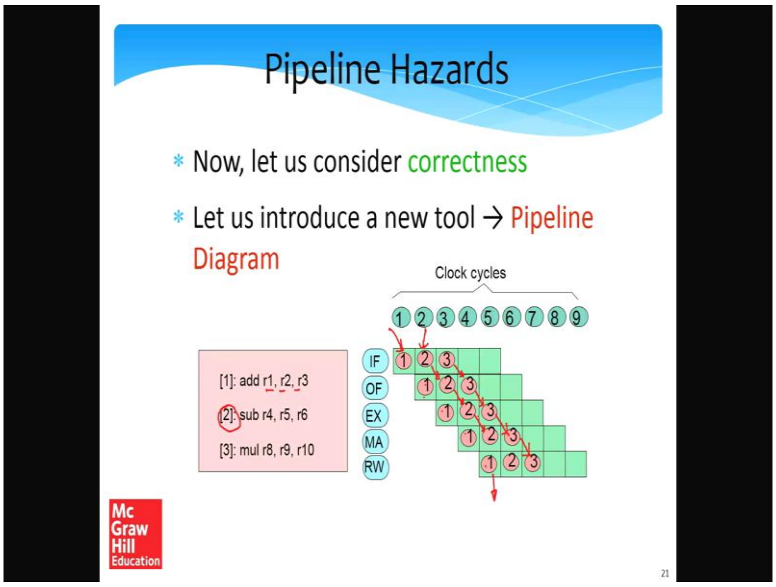
The previous diagram was pretty complicated. So, we shall not be using that in most of our discussion. We shall rather be using this diagram, which we call an abridged diagram. So, the abridged diagram is fairly easy. We have the fetch unit over here and the instruction memory. So, basically we just use one arrow to show all the contents that are transferred to the IF-OF register. At the OF stage, we have three units; the control unit, the immediate branch unit and the register file. So, we have a single multiplexer over here to choose between the register operand and the immediate operand. And also, for the special case of the store we store op2 separately in instruction packet.

In the EX stage, we have some information that flows back to the IF stage. So, as we had mentioned in the last few slides, there are only two instances when information flows between stages without, you know, without going through pipeline latches. So, one. So, both these examples are shown in red. So, the first case is when branch information, taken branch information flows from the EX stage to the IF stage. So, the X, sorry, the EX stage additionally has the ALU units, the flags. And, the ALU result is written to the EX-MA register. There is one connection between the inputs to the ALU unit and the branch unit. This is for the written address. Then, we have the EX-MA pipeline registered in the MA, memory access stage. The memory access stage simply communicates with data memory and writes the result of load instructions to the MA-RW pipeline register.

And finally, we have the register write unit. So, this is the second instance when information flows between stages. So, basically from the RW stage we have a connection to the OF stage, where register write values flow to the register file. So, we shall mainly be using this diagram because this diagram is very simpler than the previous diagram which is far more complicated. So, we shall mainly be using this diagram and we shall keep on making small modifications to this.

So, now that we have seen the basic organisation of a pipeline, so let us take a look at some of the problems that can happen. So, these are called pipeline hazards. So, let us take a deeper look at pipeline hazards. And then, we shall look at different methods of fixing them or taking care of them.

(Refer Slide Time: 42:33)



Pipeline Hazards

- Now, let us consider correctness
- Let us introduce a new tool → Pipeline Diagram

[1]: add r1, r2, r3
[2]: sub r4, r5, r6
[3]: mul r8, r9, r10

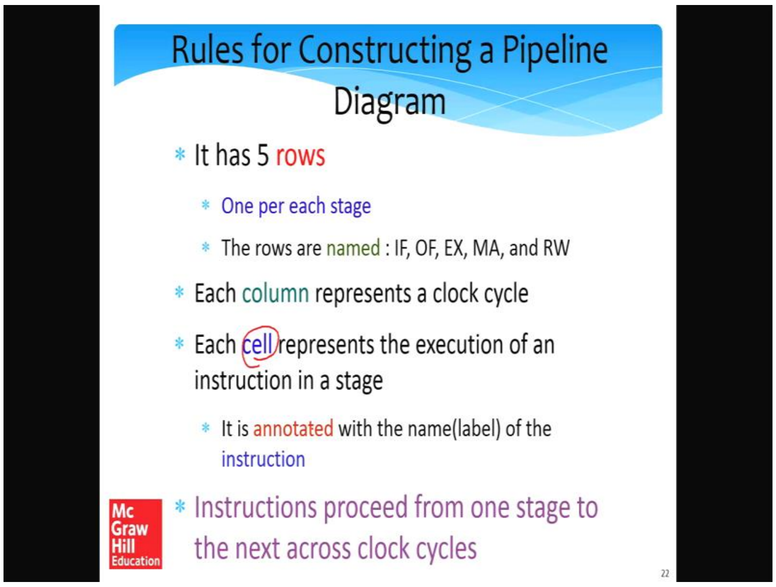
So, let us consider the correctness of executing program. So, let us introduce a new tool called a pipeline diagram. So, a pipeline diagram is an analytical tool for us. So, it will be shown in this fashion. So, essentially the columns in the diagram represent the clock cycles; the clock cycle 1, 2, 3, 4 to 9. And, the rows represent the pipeline stages; so IF, OF, EX, MA and RW.

So, what this indicates? So, basically if you see there are these cells. There are these square shaped cells. So, what we are essentially saying is that instruction 1 which can be of the form add r1, r2, r3, enters the IF stage in cycle 1, subsequently it enters OF stage in cycle number 2, it enters the EX stage in cycle 3, the MA stage in cycle 4. And finally, the RW, the registers write stage in cycle number 5. And then, it exits the pipeline. So, it finishes its execution, it completes.

Similarly, we have instruction 2. So, this instruction enters the pipeline in the next cycle. So, it enters the pipeline in cycle number 2. Subsequently, it proceeds. So, it proceeds to the third cycle which is the OF stage, to the fourth cycle which is the EX stage, the fifth cycle which is the MA stage and then in the sixth cycle, it reaches the RW stage. And, the same is true for the third instruction as well. So, we can see that in this pipeline diagram, all the instructions proceed diagonally towards the bottom right. So, this is the main tool that we shall be using in, for our discussion of correctness of programs inside pipelines. And, a pipeline diagram is pretty much a matrix, but we do not show all the cells of the matrix because they are not required.

So, what we will show is some of the cells of the matrix. And, pretty much an instruction enters the pipeline diagram at the IF stage. And, at every subsequent cycle it moves to the next row. We shall see. There are some conditions on when an instruction can move and when an instruction cannot move. But, this is the basic idea.

(Refer Slide Time: 45:26)



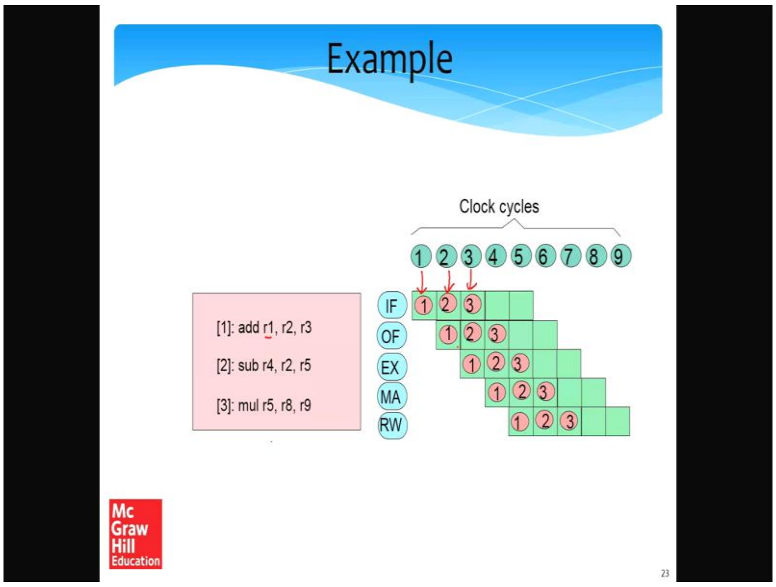
Rules for Constructing a Pipeline Diagram

- It has 5 rows
  - One per each stage
  - The rows are named : IF, OF, EX, MA, and RW
- Each column represents a clock cycle
- Each cell represents the execution of an instruction in a stage
  - It is annotated with the name(label) of the instruction
- Instructions proceed from one stage to the next across clock cycles

22

So, what are the rules for constructing a pipeline diagram? Again, explained textually. So, it has five rows one per each stage. The rows are named by the stage. Each column represents the clock cycle. And, each cell represents the execution of an instruction in a stage. So, it is annotated with the name or the label of the instruction. And, instructions proceed from one stage to the other across subsequent clock cycles, consecutive clock cycles.

(Refer Slide Time: 45:58)



Example

[1]: add r1, r2, r3

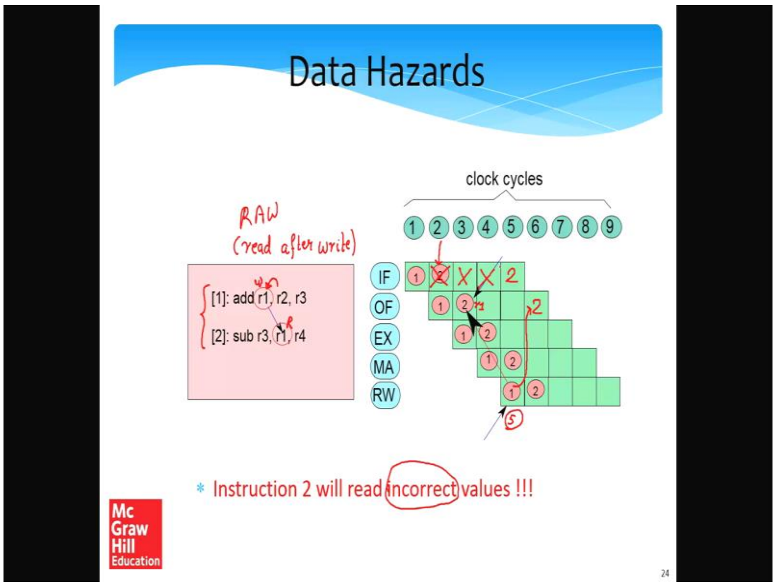
[2]: sub r4, r2, r5

[3]: mul r5, r8, r9

So, let us take a look at one more example. So, this is add r1, r2, r3, then subtract instruction and a multiply instruction. So, let us take a look at this example in a different way now. So, there are no dependencies across the instructions, in a sense that there is no dependency between the first, the second and third instructions. So, pretty much the instructions can operate independently of each other.

But, since our pipeline can read in one instruction every cycle, so in the first cycle one instruction will get into the pipeline, in a second cycle one more instruction, third cycle one more instruction. So, the crucial thing is that in each cell only one instruction can be there. You cannot have two instructions in one cell that will be tantamount to saying that in the same stage there are two instructions, which is not possible.

(Refer Slide Time: 47:00)



Data Hazards

[1]: add r1, r2, r3

[2]: sub r3, r1, r4

* Instruction 2 will read incorrect values !!!

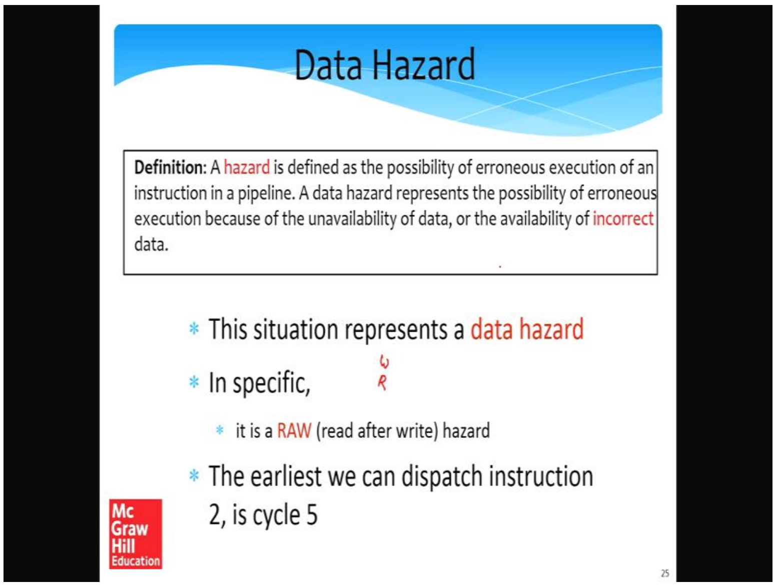
So, now let us take a look at some of the problems that can happen. The some of the problems that can happen will be shown explained like this. So, let us consider a sequence of two instructions; instruction 1 and instruction 2. So, let the instruction 1 be add r1, r2, r3, where we add the contents of r2 and r3. And, we transfer it to r 1. Then, let the next instruction subtract r 3, r 1, r 4, take the results of instruction 1 and use that as operands. So, here essentially instruction 1 is writing to r 1 and instruction 2 is reading from r 1. So, we call it a read after write dependency error or a raw dependency. So, r a dependency is a read after write dependency, which is also called a true dependence because we first write and then we are reading it.

So, let us consider the pipeline diagram. So, in the pipeline diagram instruction number 1 enters the pipeline in the first cycle and subsequently it keeps on moving to the next stages in every consecutive cycle. So, let us consider instruction 2. It enters the pipeline in the IF stage in cycle number 2. So, in cycle number 3 instructions 2 enters the operand fetch stage that is when it needs the value of r 1. So, the value of r 1 is needed in the cycle number 3. But, the problem is that when will instruction 1 produce the value of r 1 instruction 1? Well, it will produce the value of r 1.

In the sense, it will write r 1 to the register file in cycle number 5. So, base, we need the value of r 1 in cycle 3, but it will be written to the register file in cycle 5. So, basically we have an arrow, a big red arrow that is running backwards in time. So, so this is a problem over here. So, instruction 2 will read incorrect values because in cycle 3, the value of r 1 has not been produced yet. So, it will read a value which is not right, which is not correct and that will lead to wrong or incorrect execution. So, this is something that should be avoided. And, that should be fixed. And, what this translates to in a pipeline diagram is that the dependency between, when between 1 and 2, particularly between the cells in which 1 writes the data and when 1 and in which 2 needs to read the data. If you draw a line between these cells, then the line will flow backwards in time. It is flowing from cycle 5 to 3.

And, this is something which is not possible something which is not feasible. As a result, we will read incorrect values if you follow this scheme.

(Refer Slide Time: 50:22)



Data Hazard

Definition: A hazard is defined as the possibility of erroneous execution of an instruction in a pipeline. A data hazard represents the possibility of erroneous execution because of the unavailability of data, or the availability of incorrect data.

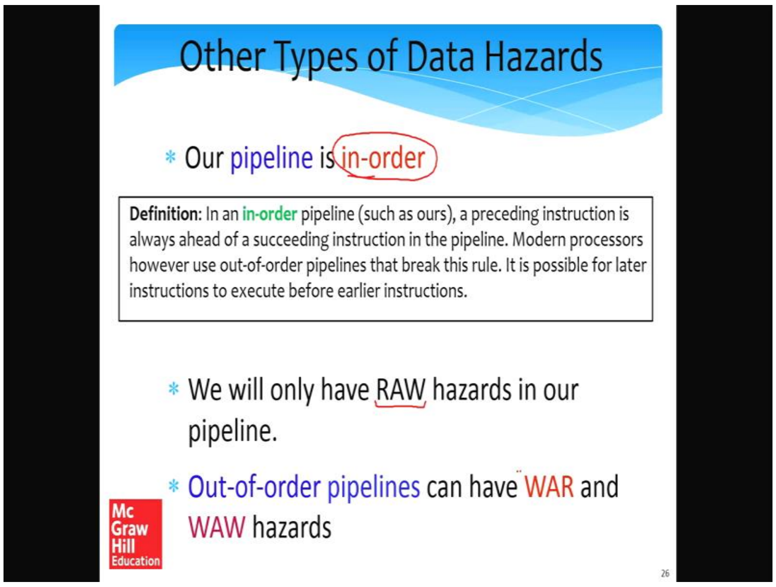
- This situation represents a data hazard
- In specific, W R
  - it is a RAW (read after write) hazard
- The earliest we can dispatch instruction 2, is cycle 5

25

So, let us define some terminology. Let us define the term a data hazard. So, hazard is defined as the possibility of erroneous execution of an instruction in a pipeline. So, in specific a data hazard represents the possibility of erroneous execution because of the unavailability of data or the availability of incorrect data. So, in this case it is what is happening is that in cycle number 3, if we go and try to read r 1 from the register file, we will simply not get the right data. So, we will get incorrect data because the right data has not been written to the register file yet. So, that will cause a problem, and that is called a data hazard. There is a possibility of an error. In specific, this is RAW read after write data hazard because we have two instructions. So, the first instruction is writing and a second instruction is reading. So, the earliest we can dispatch in instruction 2 is actually cycle 5. And, you know here there is a problem. So, basically if we let us say have instruction 2 over here. If let say instruction 2 enters the pipeline in cycle 5, in cycle 6 it will be over here. And, it can happily read the data from the register file.

So, the question is that for these three cycles we will not be able to push in any instruction into the pipeline, which will ensure that the pipeline is idle. This is a bad thing. So, precisely we did pipelining to fix the problem of idleness. But, as we see you know for this specific pattern, which is very common, we will have issues. So, basically for three cycles, we will not be able to put instruction 2 into the pipeline. And, that is something which is very problematic. So, we will need to fix it.

(Refer Slide Time: 52:37)



Other Types of Data Hazards

- Our pipeline is in-order

Definition: In an in-order pipeline (such as ours), a preceding instruction is always ahead of a succeeding instruction in the pipeline. Modern processors however use out-of-order pipelines that break this rule. It is possible for later instructions to execute before earlier instructions.

- We will only have RAW hazards in our pipeline.
- Out-of-order pipelines can have WAR and WAW hazards

26

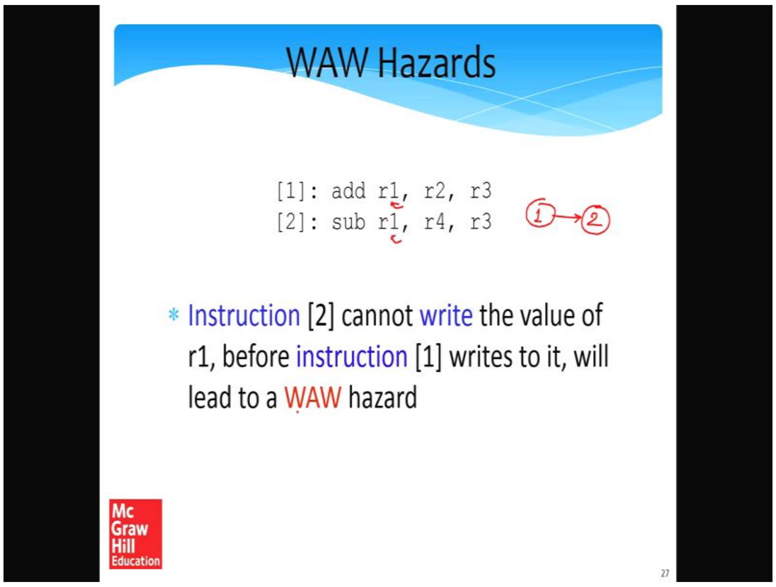
So, there are other types of data hazards. So, but these data hazards too; will actually not show up in our simple pipeline. So, in our pipeline is an in-order pipeline. So, I will discuss what that is. In an in-order pipeline, such as ours, a preceding instruction is always ahead of a succeeding instruction in the pipeline. So, let me explain what that means. So, what this means is say if I consider, maybe let me see a cleaner, yes, it is the cleaner version of the pipeline diagram.

So in this case, 1 is the earlier instruction. It is the preceding instruction. And, 3 is the succeeding or it is a later instruction. So, what we see in our pipeline diagram is that at all points of time. So, let us say, you know, let us consider cycle 5. At all points of time, the earlier instruction is at a later stage. It is ahead in the pipeline. And, a later instruction, such as instruction 3 will always be behind instruction 1 and 2 in the pipeline. It is never allowed to overtake. So, this property is called an in-order pipeline.

So, the modern processors however do not use in-order pipelines, they use out-of-order pipelines that break this rule. So, it is possible for later instructions to execute before earlier instructions. So, this causes many kinds of complexities. And, so you know discussing out-of-order pipelines is clearly not within the scope of this course. But, it will be a part of an advanced course. But, in this course we will only look at in-order pipelines. And, a lot of processors also do use in-order pipe lines; it is not that they are not used. But, nowadays they are mainly used for smaller processors, embedded processors and so on.

(Refer Time: 54:39) in a high performance is not really the key. We are more looking for power efficiency. So, in all such processors which have in-order pipelines and a preceding instruction, an earlier instruction is always ahead of a succeeding a later instruction in the pipeline. So, in such pipelines we only have read after write hazards. But, out-of-order pipelines can have other kinds of hazards like a write after read or write after write hazard.

(Refer Slide Time: 55:07)



WAW Hazards

```
[1]: add r1, r2, r3
[2]: sub r1, r4, r3
```

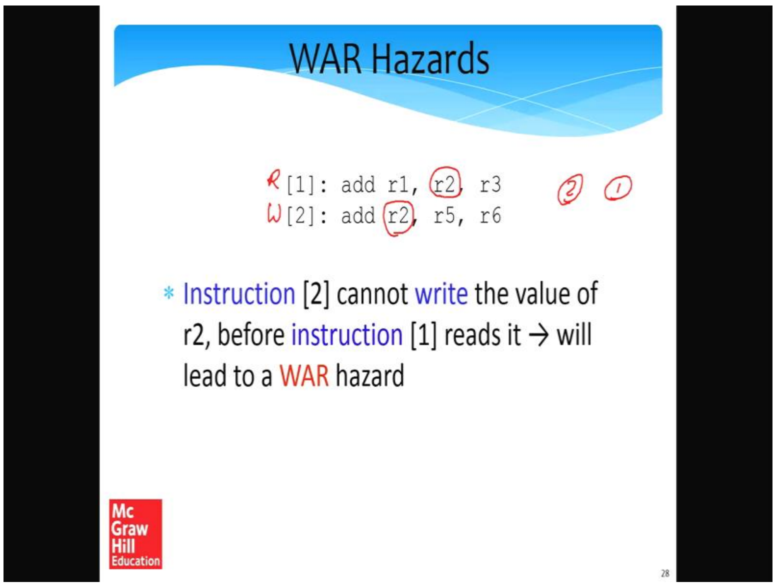
$1 \rightarrow 2$

- Instruction [2] cannot write the value of r1, before instruction [1] writes to it, will lead to a WAW hazard

So, let us discuss what these two types of hazards are; write after write and write after read. So, let us consider a sequence of instructions where 1 is the earlier one and 2 is the later one. So, both these instructions are writing to register r1. So, instruction number 2 cannot write the value of r1, before instruction 1 writes to it. Otherwise, what will happen is that the final value inside r 1 will become wrong. So, we want instruction 1 to write the value of r 1 first. And then, we want instruction 2 to write the value of r 1.

So, this is the sequence of actions. This is the flow of action that needs to happen. This will always happen in our case because we do not allow instructions to overtake each other. But, in a pipeline which allows instructions to overtake each other, this problem can happen. But, we need not be concerned about that. But, nevertheless what readers, listeners, should know is that we can have a hazard of this type, which is a write after write hazard, where if we allow instructions to overtake each other, then erroneous execution is possible in this case.

(Refer Slide Time: 56:35)



WAR Hazards

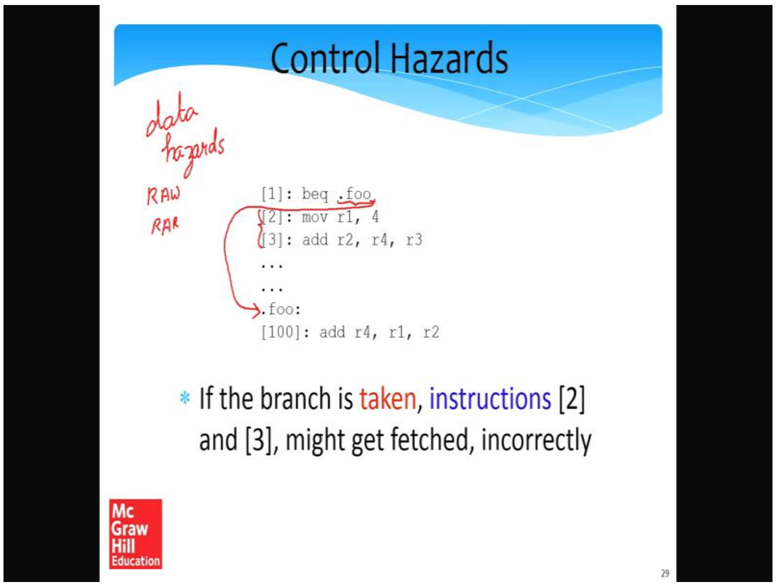
```
R [1]: add r1, r2, r3        ② ①
W [2]: add r2, r5, r6
```

* Instruction [2] cannot write the value of r2, before instruction [1] reads it → will lead to a WAR hazard

Similarly, we can have a write after read hazard. So, let us consider this example. So, instruction 1 is reading the value of r2. And then, the later instruction instruction 2 is writing to r 2. So, this is doing some computation; adding r5 and r6 and it is writing to r 2. So, in our case we will never have incorrect execution because we do not allow instructions to, you know, overtake each other. So, instruction 1 will always be ahead. Say if this is a pipeline, instruction 1 will always be ahead of instruction 2. So, that is the reason instruction 1 will always read the correct value of r 2 and instruction 2 will then write to r 2 at the end.

However, you know in our case instruction 2 cannot write the value of r2, before instruction 1 reads it. In an out-of-order pipeline, it is possible. And, this condition is a write after hazard. We have a read first and then the write. So, this is called a write after read hazard. Again, it is not a problem for us. But, if you allow instruction 2 to overtake instruction 1, then it becomes a problem.

(Refer Slide Time: 57:55)



Control Hazards

```
[1]: beq .foo
[2]: mov r1, 4
[3]: add r2, r4, r3
...
...
.foo:
[100]: add r4, r1, r2
```

- If the branch is taken, instructions [2] and [3], might get fetched, incorrectly

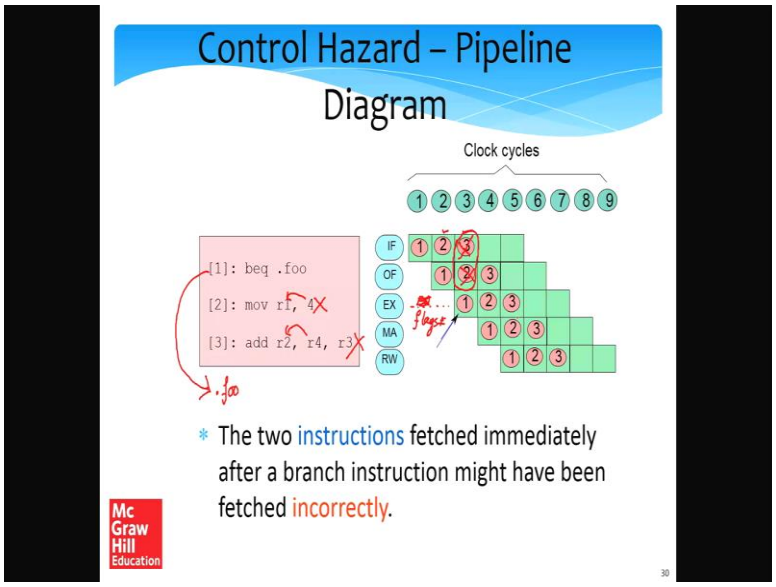
So, what have we seen up till now? We have seen data hazards. So, we have seen write after read hazard, which can definitely happen in our system. And so, we showed a simple example. And, we showed a problem that can happen that after instruction 1, if you immediately put instruction 2 in the pipeline, it will get a wrong result. So, we need to, you know, wait for three cycles and then put instruction 2 in the pipeline. Then, we looked at WAR and WAW hazards, which can happen in more advanced pipelines, not in our case.

And, RAR has situation is not a hazard because you know you have one read and one more read after that in the same register. In an advanced pipeline also they can overtake each other. There is no problem. As long as the write is not involved, there is no problem. So, these are the kinds of data hazards.

So, we can have another kind of hazard called a control hazard, which we shall see now. So, let us consider an instruction of the form branch is equal. So, if the equality condition is true, we shall branch to the label dot foo. After that, we have several instructions. And then long time later, we have the label dot foo and instructions after it.

Now, if the branch is taken, so let us assume that the branch in this case is taken, we will show that instruction 2 and 3 might get fetched incorrectly. So, so what will happen if the branch is taken? If the branch is taken, then the control should immediately jump to this point. And, these two instructions should not be fetched or executed.

(Refer Slide Time: 59:54)



Control Hazard – Pipeline Diagram

```
[1]: beq .foo
[2]: mov r1, 4
[3]: add r2, r4, r3
```

* The two instructions fetched immediately after a branch instruction might have been fetched incorrectly.

But what we see in this diagram? So, this is a pipe line diagram of a control hazard. Let us take a look at this in some detail. So, what we get to see over here is that we have three instructions. The first instruction is a conditional branch. And after that, we have two more instructions. Say, the first instruction’s condition is true, and then we will jump to the dot foo label. And, if the first instruction’s condition is not true, then we will execute instructions 2 and 3. So, now when I draw the pipeline diagram, what we see is that in the first cycle in cycle number 1, instruction 1 enters IF stage and it will continue.

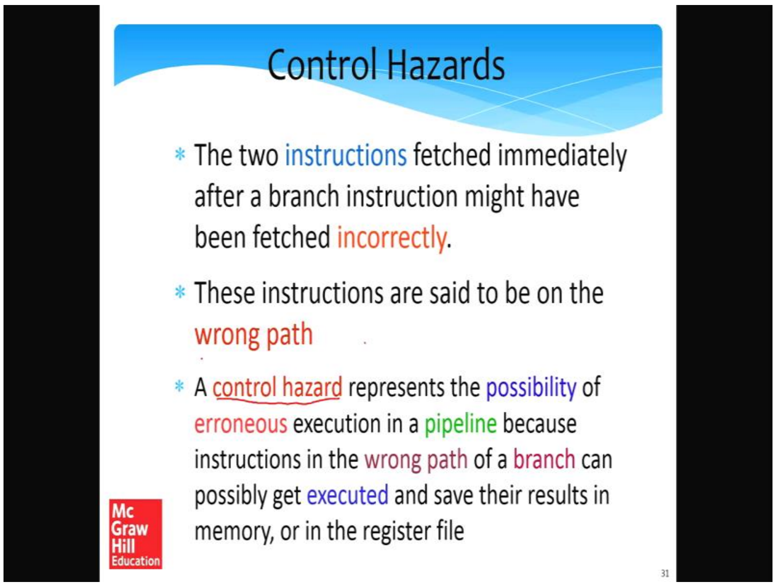
The instruction 1: so the instruction 1 essentially does not stop. So every subsequent cycle, it moves from one stage to the other. But, the important point is in cycle number 3. So, in cycle number 3 what happens is that a decision is made whether the branches taken or not because in cycle 3, instruction 1 is in the EX stage. So, in an EX stage the branch unit takes a look at the flags. In particular, it is the beq instructions. So, it takes a look at the eq, the equality flag of the flags dot E, actually. Flags; I should write it flags dot E. It takes a look at flags dot E. And, if flags dot E is 1, then the branch is deemed to be taken.

So, the problem that happens is that at this point is if the branch is taken, then instructions 2 and 3 were fetched incorrectly. This is what it means because in a second cycle when instruction 1 was in the OF stage, instruction 2 got fetched. So, it enters the IF stage. Then, at the third cycle, when we are making a decision about whether the branch is a taken branch or not, at that point the processor was busy in fetching instruction number 3 into the IF stage. So, the problem that arises now is maybe towards the middle of the cycle, towards the end of the third cycle, we get to know that this is actually a taken branch. But, by that time we have fetched two instructions; instructions 2 and 3. We have fetched them from memory and they are in our pipeline.

If you do not do anything, then instructions 2 and 3 will simply continue down the pipeline and they will execute. And, they will write 4 to r1. And then, they will add r 3, r 4 and write to r 2. And, clearly you know these two instructions should not be executing, if the execution will be incorrect. So, this problem is a control hazard. In the sense that there is a possibility of erroneous execution; and the possibility arises because the two instructions fetched immediately after a branch instruction, might have been fetched incorrectly. And, it definitely fetched incorrectly, if the branch is a taken branch. So, that is the reason if the branch is taken, there is, there has to be a method of cancelling these instructions, killing them, such that they do not, they do not proceed down the pipeline.

If they do and if they are allowed to modify memory or to modify a register in the register file, then essentially our state will get corrupted and the program will be deemed to have executed incorrectly. And, this is the situation that cannot be allowed. So, this situation is referred to as a control hazards.

(Refer Slide Time: 64:16)



Control Hazards

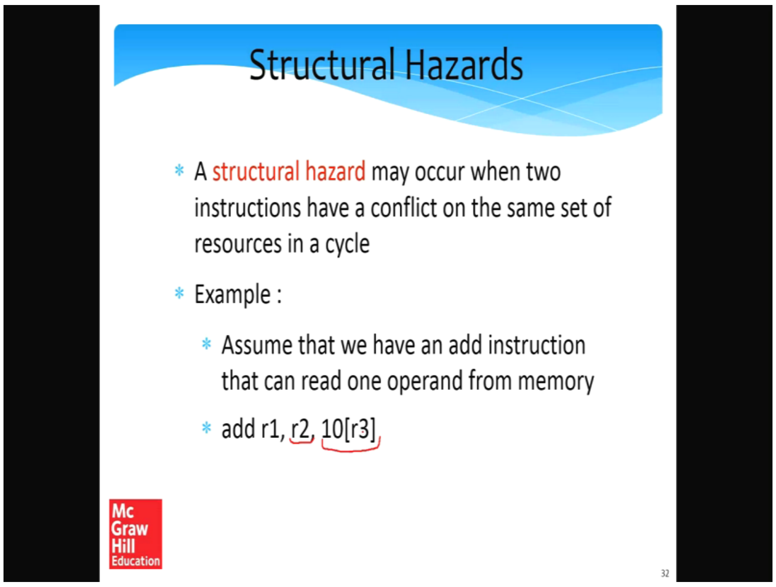
- The two instructions fetched immediately after a branch instruction might have been fetched incorrectly.
- These instructions are said to be on the wrong path
- A control hazard represents the possibility of erroneous execution in a pipeline because instructions in the wrong path of a branch can possibly get executed and save their results in memory, or in the register file

31

Let me now explain once more what a control hazard is textually. So, the two instructions fetched immediately after a branch instruction might have been fetched incorrectly, if the branch is a taken branch. So, these instructions are said to be on the wrong path. So, basically these instructions if they need to be cancelled later, then these instructions are known to be on the wrong path. So, controls hazard essentially, a control hazard represents the possibility of erroneous execution in a pipeline because instructions that are in the wrong path of a branch can possibly get executed and save the results in memory or in the register file.

So, it is our job to ensure that the instructions which are on the wrong path. For example, these two instructions; instructions 2 and 3 they do not get executed, if branch at instruction 1 is taken.

(Refer Slide Time: 65:17)



Structural Hazards

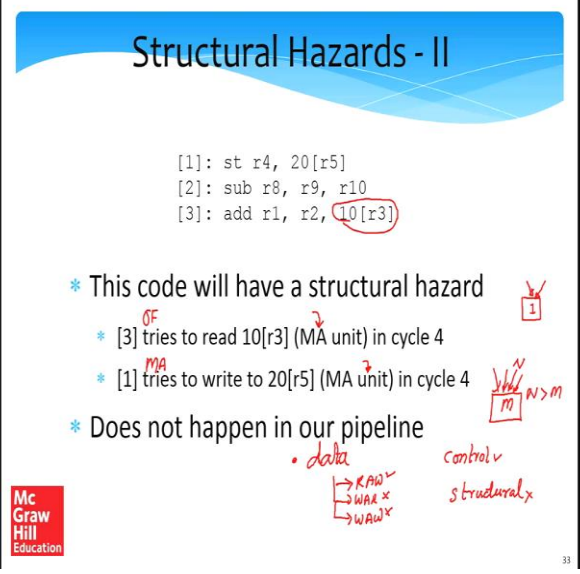
- A structural hazard may occur when two instructions have a conflict on the same set of resources in a cycle
- Example :
  - Assume that we have an add instruction that can read one operand from memory
  - add r1, $\underbrace{r2}$, $\underbrace{10[r3]}$

32

So, now there are, let us come to the third kind of hazard. So, we have discussed data hazards, we have discussed control hazards. So, let us come to a third kind of hazards called structural hazards. A structural hazard may occur when two instructions have a conflict on the same set of resources in a single cycle. So, this will not happen in our simple pipelines. It is not a concern for us. But, it can happen in other pipelines. It is good to know the concept in general.

So, assume that we have an add instruction that can read one operand from memory. If you have such kind of an instruction, then so in such kind of an instruction we will have one of the operands which is coming from memory.

(Refer Slide Time: 66:02)



Structural Hazards - II

```
[1]: st r4, 20[r5]
[2]: sub r8, r9, r10
[3]: add r1, r2, 10[r3]
```

- This code will have a structural hazard
  - [3] tries to read 10[r3] (MA unit) in cycle 4
  - [1] tries to write to 20[r5] (MA unit) in cycle 4
- Does not happen in our pipeline

OF
MA
1
N
m
N>m
• data
RAW ✓
WAR ×
WAW ×
control ✓
structural ×
Mc Graw Hill Education
33

Now, let us take a look at an example in which a structural hazard can happen. So, let us consider these three instructions; instructions 1, 2 and 3. In the first instruction, we do a store to the location 20 plus the address in r 5. Then, we have a subtract, and then we have the special kind of add instruction that we are talking about in the last slide.

So, here one of the operands is the memory operand. And, that is going to be the second source operand. So, reason that this code will have a structural hazard is as follows. That in cycle number 4, instruction 3 will be in it is OF stage. So, it will try to read the operand 10 r 3 from memory. So, it will access the memory access unit. And, similarly in cycle numbers 4, the store instruction will also be in its MA stage. It will also be in its MA stage. It will try to access the MA unit in cycle 4. And, the MA unit can process only one instruction per cycle. Then, clearly there is a conflict between instructions 1 and 3. And, this is called a structural hazard.

So, structural hazard is basically a situation in which two instructions try to access the same resource or defined in a more generic way. If a resource can satisfy m simultaneous requests, but there are N requests that are coming into it, where N is greater than m, then we will have a structural hazard; because we simply do not have enough resources to satisfy all the request. So, in this case the memory access unit can satisfy only one request at the time. But, if two requests come, then one of them has to wait. And, this is called a structural hazard. This will not happen in our pipeline. The reason being that we have designed it in such a way that we never have resource conflicts across stages, but it can happen in many other kinds of pipelines. It is a good concept to keep in mind.

To summarize: the three kinds of hazards and once again data hazards. Data hazards, there are three kinds; read after write, write after read and write after write. So, in all pipeline only the read after write hazard will happen. The rest two will not happen. Then, we have control hazards. And, control hazards can definitely happen in our pipeline. And, we have structural hazards, which will not happen in our pipeline. So, given the fact that we have, you know, idea of what the hazards are. Let us try to look for solutions to take care of RAW and control hazards. So, this will be part of the next lecture. So, the current lecture will end at this point.