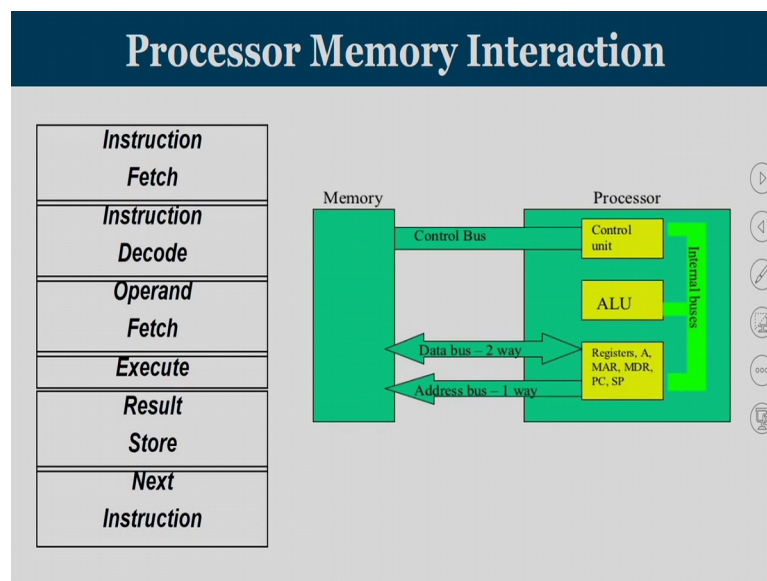


**Multi-core Computer Architecture - Storage and Interconnects**  
**Dr. John Jose**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Guwahati,**

**Lecture – 02**  
**Introduction to Instruction Pipeline**

Welcome to the 2nd lecture of the course. Today, we will be focusing our attention to Instruction Pipeline. Even though our course's main focus is on the Storage Aspect and the Communication Aspect on Multi-core devices. We have to have a sufficient background, how instruction execution is taken care of inside a processor. Typically, instant execution is been driven by the instruction pipeline circuit we will try to understand how a basic instruction pipeline works in the case of a RISC processor.

(Refer Slide Time: 01:22)



We have seen in our last lecture that the main job of a microprocessor is to execute the task, and we represent the task as a sequence of instructions, and the instructions are stored in the memory. Now, these are the various stages that an instruction will go through in order to get it executed. Now, these are all done by separate combinational blocks in the hardware. We have a separate unit, that take care of the instruction fetching, we have a separate unit that takes care of the instruction decode, operand fetch, then execution, finding out next instruction and all. Today, we will try to understand how these units are

connected each other, and how can you improve the performance of the system with the help of instruction pipeline mechanisms.

(Refer Slide Time: 02:14)

### Pipelining – Laundry example

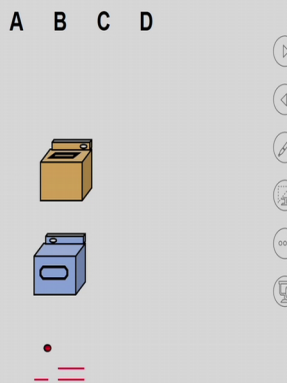
❖ A, B, C, & D each have one load of clothes to wash, dry, and fold

A   B   C   D

❖ Washer takes 30 minutes

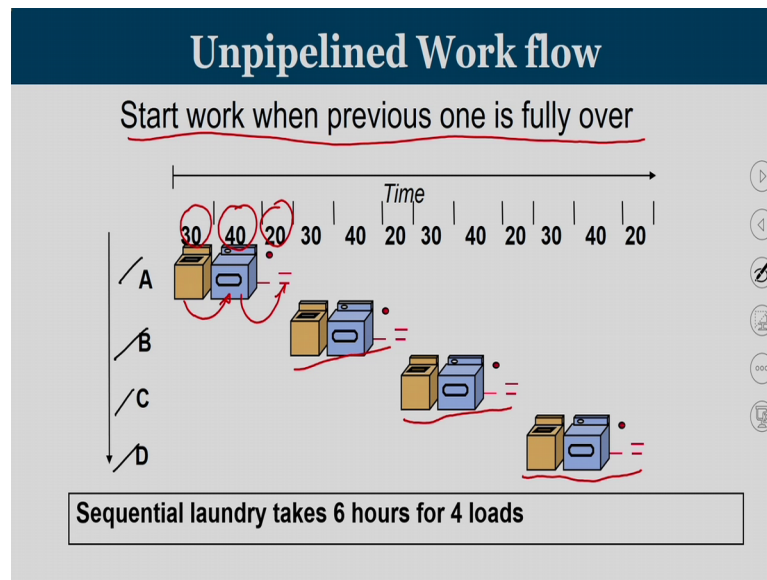
❖ Dryer takes 40 minutes

❖ Folder takes 20 minutes



Before drawing into the details and architectural features of instruction pipeline, let me draw your attention to a common example that is being used while teaching the instruction pipeline. Consider the case that we have four loads of clothes to wash, and every load or every unit of cloth has to go through basically three different types of operations, namely the washing, then the drying, and then the folding. Your washing will typically takes 30 minutes, drying will take 40 minutes, and the folding will take 20 minutes. We have four such operation that need to be carried out.

(Refer Slide Time: 03:06)

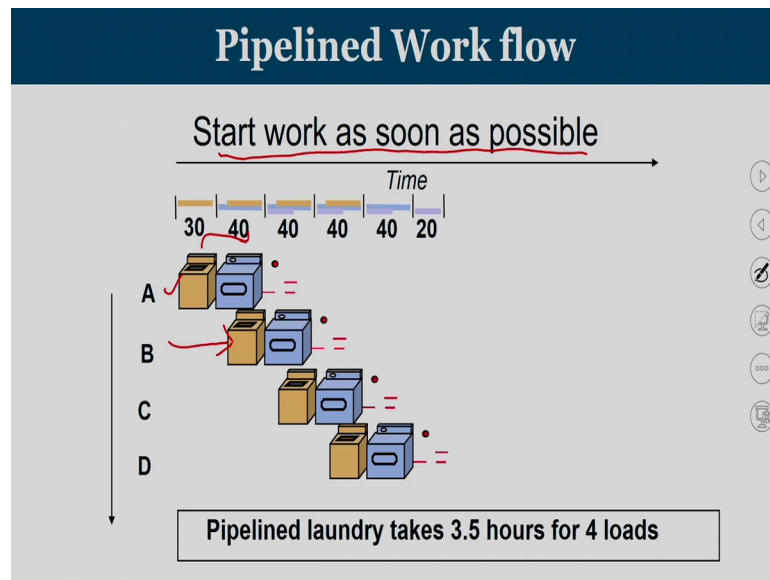


So, we will try to understand how this particular task can be done in the case of an unpipelined process. So, we have four loads of work that is to be done that is A, B, C, and D. Now, each of the work has three sub operations; washing, which will take 30 minutes of time; and then we have drying, which will take 40 minutes of time; forward by folding, which will take another 20 minutes of time.

In the case of an unpipelined work flow, the basic idea is to start the work only when the previous work is fully over. So, let us start with A, we spend 30 minutes in completing the washing, followed by 40 minutes in the drying, followed by 20 minutes in the folding. The peculiarities these three sub operations are carried out by three different units.

So, once the washing is over, we have to take your clothes into the next unit, and ones that is over, we will take it into the next unit. Once the work of A is over, which will take 30 plus 40 plus 20 that is 90 minutes. We take the work of B, and that also will take 30 plus 40 plus 20, another 90 minutes is gone. Once B is over, then we start with C; and followed by D. This is the way, how we typically carry out a work, if this is an unpipelined work flow. So, we call it as a sequential laundry that will takes roughly 6 hours for completion of 4 loads of work. Now, we will try to see how pipelining is applied on this.

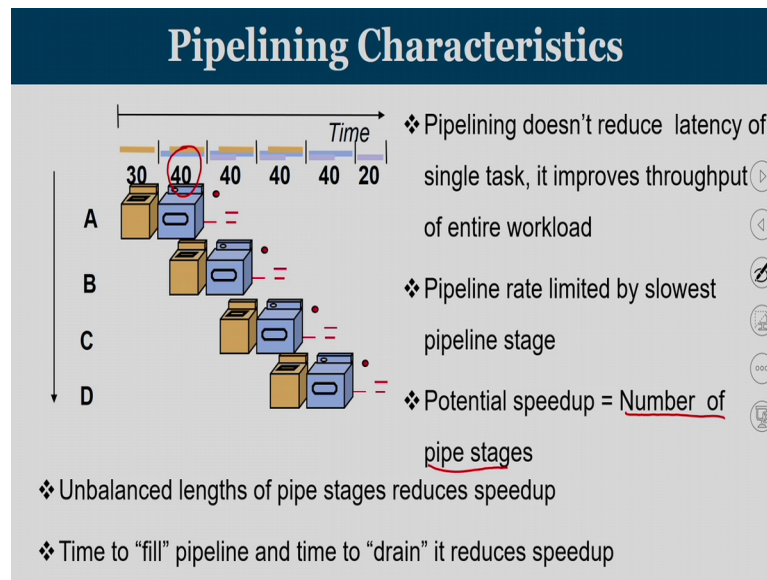
(Refer Slide Time: 05:05)



The concept of pipeline is start work as soon as possible. So, since we have three independent units, the washer, the dryer, and the folding; we can parallelized these three activities. When you are drying the clothes of unit A, your washer is free. So, we can start the work of B that is what is happening here. We can see that the first 30 minutes, it is only the washer that is working; once the washing is over, the clothes from A is moved to the dry; at the same time, the washer is free, I can start with the work of B.

So, even though A is not completed, I am trying to start B thereby making sure that the dryer is taking care of As clothes; at the same time, the washer is busy by washing Bs clothes. This is called the pipelined structure. So, we are going to do a work as soon as the unit in which the work is carried out is idle. So, pipelined laundry will take only 3.5 hours of time in order to complete these 4 loads.

(Refer Slide Time: 06:28)



So, what are the characteristics, let us sum up the characteristics of this pipelined work flow structure. The first feature is pipelining does not reduce the latency of a single task; it tries to improve the throughput of the entire workload. So, when you consider each of the unit, then the total time taken by them is still the same, rather slightly higher that we will see. Effectively what are we gaining here, it is a throughput the number of task that is completed in unit time.

The second feature of pipeline is pipeline rate is limited by the slowest pipeline stage. So, here 1 unit is taking 30 unit of time, 2nd one is 40 and the 3rd one is 20, the slowest that which takes more amount of time is called; is defined as slowest. So, the 2nd unit that is the dryer will take more amount of time. So, the advancing of the pipeline from one unit to other is restricted to dryers time. So, pipeline rate is limited by the slowest pipeline stage. Now, how much speed up you are going to get from this pipeline. The speed up is subjected to maximum of the number of pipeline stages.

Now, what are the other issues with respect to pipeline? Since, these stages are having unbalanced length, amount of time the clothes spend in each of the stages is not saying, we were spending 30 minutes on the washer, 40 minutes on the dryer, and 20 minutes on the folder. Since, the time that the clothes are being kept inside these stages are different. We have an unbalanced pipelined stage, so the this will reduce the pipeline speed up. And the time to fill the pipeline and time to drain the pipeline will also reduce the

performance. So, what do you mean by filling? I have to put your clothes from one unit, once it is over, I have to take away the clothes from that unit into the next one that is called filling and draining of each of this unit.

(Refer Slide Time: 08:56).

## Pipelining in Circuits

- ❖ Pipelining partitions the system into multiple independent stages with added buffers between the stages.
- ❖ Performance gained in a pipeline  $\rightarrow$  depth of a pipeline
- ❖ Pipelining can increase the throughput of a system.

Potential  $k$ -fold increase of throughput in a  $k$ -stage pipelined system

Now, coming to pipelining of circuits. Ideally, from this example, how can we correlate the concept into the pipelining that is being implemented in hardware circuits. Pipelining partition the system into multiple independent stages with added buffers between the stage. And performance gained in the pipeline is proportional to the depth of the pipeline. And pipeline can increase the throughput of system, we have already seen that.

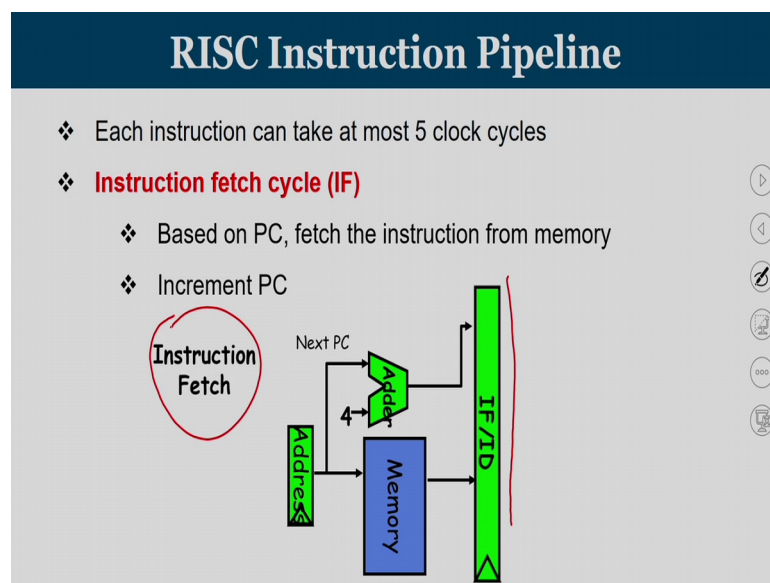
So, consider the case, we are going to do a task of execution of an instruction that is fetching, decoding, operand axis, executing the task, and then producing the result. This entire activity, which is carried out by a set of combinational circuits, we can assume that let there be  $n$  the combinational circuit is having an  $n$  gate delay. When if you could divide this  $n$  units into two independent sub task of  $n$  by 2 gates, then that is called a first level of partitioning the circuit. And these independent unit, we call this as a pipelined stage. We have two pipeline stages, and they are separated or interfaced by enlarge. If you can further divide them into  $n$  by 3 stages rather than  $n$  by 2, then we call it as a 3 stage pipeline. So, here we are going to have a 3 stage pipeline.

So, ideally the concept is whatever is the operation that, we are going to carry out in the execution of an instruction. If you could split the operations into independent sub task,

carry out an operation in one combinational unit, once that operation is done in that particular unit, pass it on to an interface register, which we call it as pipelined register, and then you go and take the next instruction.

In this way, all the stages inside the pipeline are busy with different instructions; all instructions are making forward progress, even though we are not able to increase the performance of a single instruction or we are not able to reduce the execution time of a single instruction. Overall, when we consider the throughput, we are going to gain much.

(Refer Slide Time: 11:11)



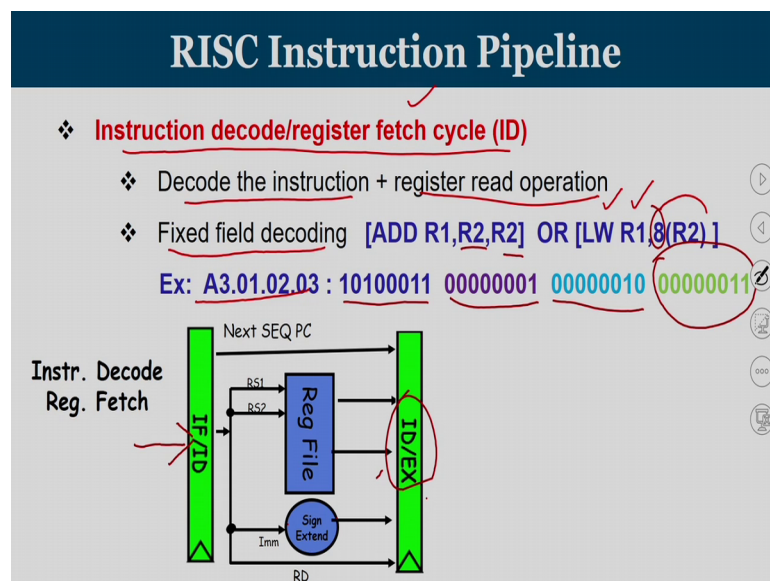
Now, we will see a specific case of a pipeline. For our case study, today we are going to consider the RISC five stage instruction pipeline. RISC stands for Reduced Instruction Set Computer. Now, in the case of RISC architectures, the instructions are designed in such a way that every instruction will take at most five stages. Today we are going to learn about these five stages; what are the operations that are carried out in these five stages that will give us a better understanding about how pipelining works.

The first stage is known as instruction fetch. So, what happens is, the instructions are stored in the memory, we have to fetch an instruction and then only we can carry out the remaining operations on it. So, it is a program counter a register that contains the address of the next instruction to be fetched. So, the contents of program counter from the contents of program counter, go and fetch the instruction from the memory.

And once the instruction is fetched, the instruction fetch unit has a special adder, which will increment the program counter to the next linear address. In the case of this RISC stage architecture, we assume that we are using a word length of 4 bytes that means, every instruction is stored as 4 bytes 32 bits. So, the next instruction is stored in program counter plus 4. So, PC is equal to PC plus 4, it is been done.

This will be the logical diagram of the unit. So, you are having an address. So, based upon the program counter, you go into the memory, access the contents. So, the fetch the instruction is going to be typically kept in the IF ID register, whereas this unit is called instruction fetching unit. And you have an another adder that will increment the PC. To summarize, the instruction fetch unit is capable of bringing the instruction from the memory, and it will update the value of program counter, so that it will be ready for fetching the next instruction from the memory.

(Refer Slide Time: 13:30)



Moving out to the second stage of the pipeline, this stage is known as instruction decode or register fetch operation. Here, two operations are being carried out; the first operation is you have to understand, what the fetched instruction is, whether it is an ALU instruction, whether it is a memory instruction, whether it is a branch instruction or not. Along with that, once you understand what the instruction is, once you decode what the instruction is. The next task is if there are some operands that are kept in registers, then we have to access these registers; that is why, it is known as register fetch operation.



So, the first task is decode the instruction, and then read the registers. And the RISC instruction pipeline basically uses a fixed field decoding. Out of the 32 bits that we are having, there are certain prefix to positions. So, for example, first  $n$  bits represent opcode, the next  $m$  bits will represent my first operand, next  $p$  bits will represent my second operand like that. So, once you get the instruction, since it is fixed length decoding, it is very easy that you could work, one unit will be trying to understand what the opcode is, parallelly another unit is trying to work what the operands are.

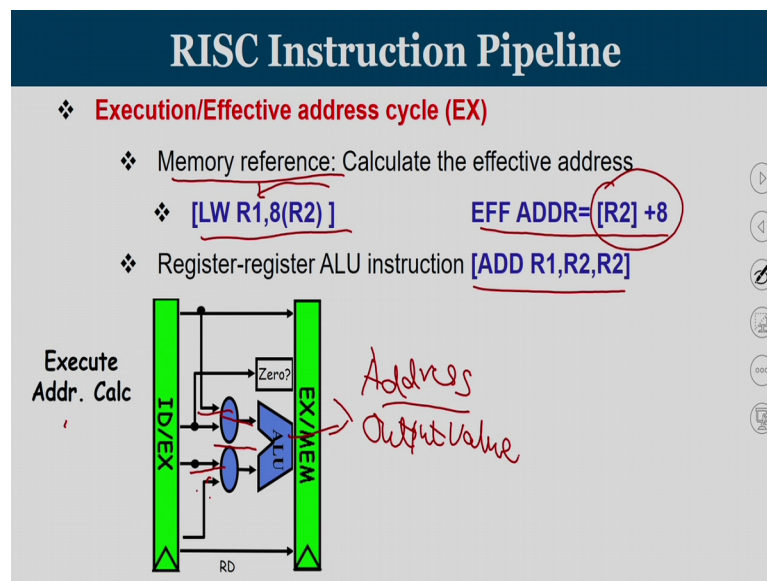
Considered an example, let us say we have an instruction add R 1, R 2, R 3, which means R 1 is equal to R 2 plus R 2. In this case, we try to understand, what how this is encoded. Let us say assume this is the encoding in hexadecimal, let us say this is a 32-bit value that is going to be used. So, since we use fixed length encoding, one example is, first 8 bits can represent, it will tell an add operand; this is the first operand; then second operand; and the third operand like that.

Let us say if it is going to be a load operation, this is a typical syntax of a load word instruction in the case of a MIPS RISC processor. Load word is the opcode; and this is one of the operand that is called destination operand R 1; 8 and R 2 are used to find out the address from which the load has to happen. So, we will see about what is the working of load later, but even if it is a different instruction other than an ALU instruction, still we can use the fixed length decoding that will tell the first 8 bits can always find out what is going to be the first 8 bits can tell what it is an opcode is load. Then this can tell, it is R 1; this can tell, it is R 2; and this portion can tell the value 8. So, this is what is known as fixed length encoding and decoding.

So, this is the register, where the output of the fetch instruction is being kept. Now, from the IF ID register, once you come into the second stage that is called instruction decode and register fetch unit, you can either read the register file, such that the contents of R 2 is been taken, and it is past to the next pipeline register that is called ID and DX. Sometimes, for these kind of instruction, all the operands may not be registers, sometimes we may have an immediate value like 8, then I could use this possibility of finding out the sign extended version that is a sign extension version of it, and that also can be stored in the ID EX registers.

(Refer Time: 17:01) To sum up in this unit what we are trying to do is once you have completed the fetching of an instruction, go and take this entire 32 bit, split into smaller pieces, first n bits will tell what is a opcode, next n bits will tell what is the first operand, what is second operand, whether it is an immediate value or not. So, this can be done in parallel with the decoding. And once the decoding is over, one of this combinations can be properly taken and work with coming into the third stage.

(Refer Slide Time: 17:30)



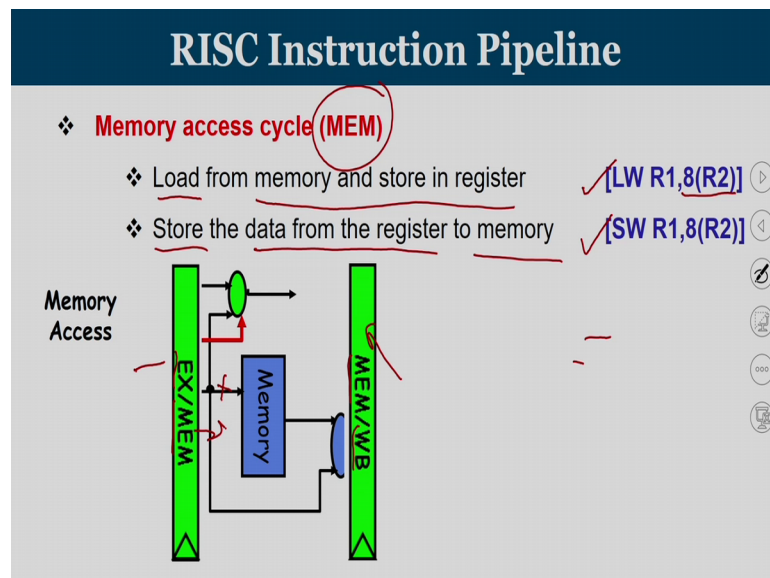
The third stage is known as execution stage or it is also known as an effective address computing stage. If the instruction is going to be an ALU instruction like add, sub, or increment kind of thing, then this unit will perform the corresponding operation. But, if the instruction is a memory operation like a load instruction or a store instruction, then the third stage that is a EX stage is performing computation of effective address.

So, if it is a memory reference, then we are doing effective address computation. Consider the case that you are using a load word instruction, the meaning of this one is, I have to take a data from a memory, whose address is given by R 2 plus 8. So, contents of R 2 plus 8, let us say this value correspond to 2008, go to location 2008, fetch the data that is present, and that is being loaded into R 1. So, the contents of R 2 will be red in the ith stage that is a second stage of the pipeline, the value 8, and the contents of R 2 will be forwarded. So, you will get the value 8 here, and the contents of R 2 will be coming

through this, and then you effectively add. So, at the end of ALU stage what you get is, the address. So, effective address reaches here.

But, if it is going to be an add operation like `ADD R1, R2, R2` like that, then the contents of the registers R2 and R3 whatever be the case, they will be taken to ALU. In that case, this section may not be used, you will be directly taking two values, these values are added. In this case, what the result you get from ALU is not an address, it is an output value. So, your EX stage is capable of performing two task; if it is a memory instruction like load or a store instruction, it will compute the effective address; if it is going to be an ALU operation or a branch operation, it will perform the corresponding operation and get the result.

(Refer Slide Time: 20:08)



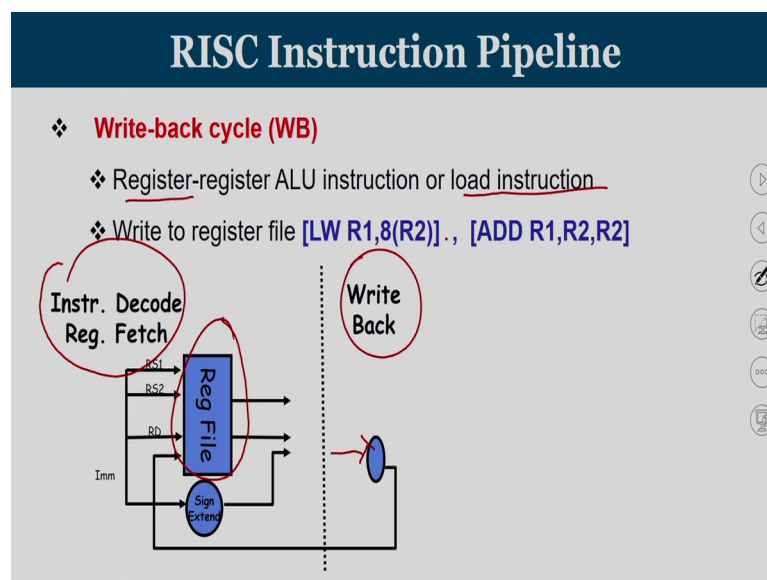
Going onto the fourth stage, this is called MEM stage. This stage is used only by two instruction, in the case of a RISC architecture; 1st one is a load instruction, 2nd one is a store instruction. For all other instruction, this will simply bypass the value that is available in the EX MEM register into the MEM right back register. So, the MEM stage is used only for those instructions, which are going to access your memory that is the load and store instruction.

So, load is an instruction, it will load the contents from memory and store into a register. And store is instruction that will store the data from a register into the memory. These are the syntaxes of load and store instructions. So, the effective address that is 8 plus content

of R 2 will be computed in the EX stage. So, by the time you reach the MEM stage, you are already having the effective address with you with the effective address you go to memory, so this will pass on the effective address. Once you have the address, you go to memory, get the data and that is being kept in the next pipeline register that is a MEM right back register.

There is a special naming convention for this pipeline registers. So, MEM write-back pipeline register means, one end of this pipeline register is the MEM unit, the other end is the write-back unit. Here in the case of this pipeline register, one end is the execution unit, and this end is the memory unit.

(Refer Slide Time: 21:27)



Going into the next stage that is called a write-back stage this is used by those instructions, where the results are written into the registers. When you have an ALU operation like add or sub, the result is written in their register. When you have a load operation, there also the result is written inside a register whereas, in the case of a store, we are not writing the result into a register, because the effect of the operation is reflected on the memory, not on the register.

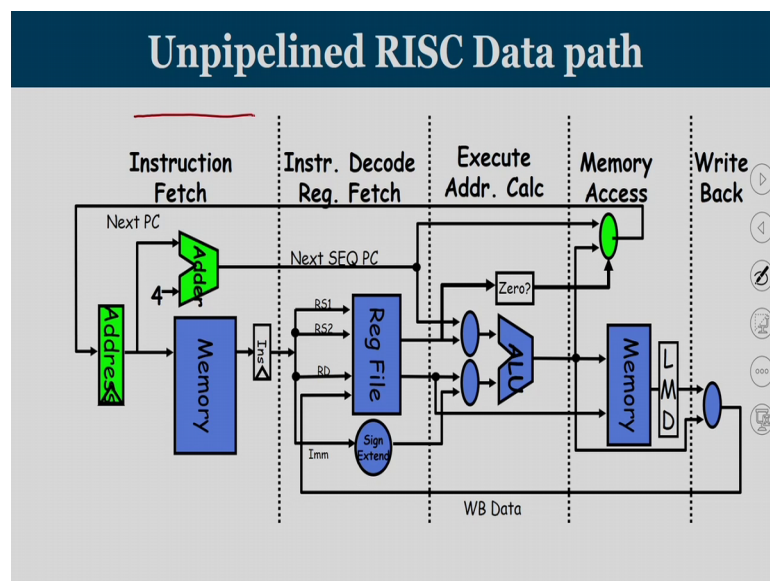
So, in this case, whenever you have a register-register ALU instruction or a load instruction, then the write-back stage is been used. The logical diagram, this is the second stage, which access reading of the registers. So, all reading of registers happened in the second stage; all writing of registers happen only write-back stage. So, whatever is

the output of MEM stage, you get the value, and that value is written into the appropriate so registers. These are two different stages, but they act upon the same register file that is why, I have shown the diagram like this.

So, these are the five stages of the RISC pipeline. So, the first stage is instruction fetch, which is responsible for fetching the instruction and updating the program counter. The second stage is called instruction decode and register read, where you decode the operation and apply fixed length the decoding to find out, what are the opcodes and operands are. The third stage is the execution stage, which can carry out two task if it is a load or a store instruction, the effective address is computed from the base register and the immediate value; if it is an ALU instruction, the corresponding task is being carried out in the arithmetic logic unit.

The forth stage is called MEM stage, which is used only in the case of a load and a store instruction. Whatever is the effective address that is computed by the ALU in the previous stage from the effective address (Refer Time: 23:29) to the effective address, the load or a store operation is been happening. The last stage is called a write-back stage, where whatever is the result of ALU operation or the memory operation, it is being reflected.

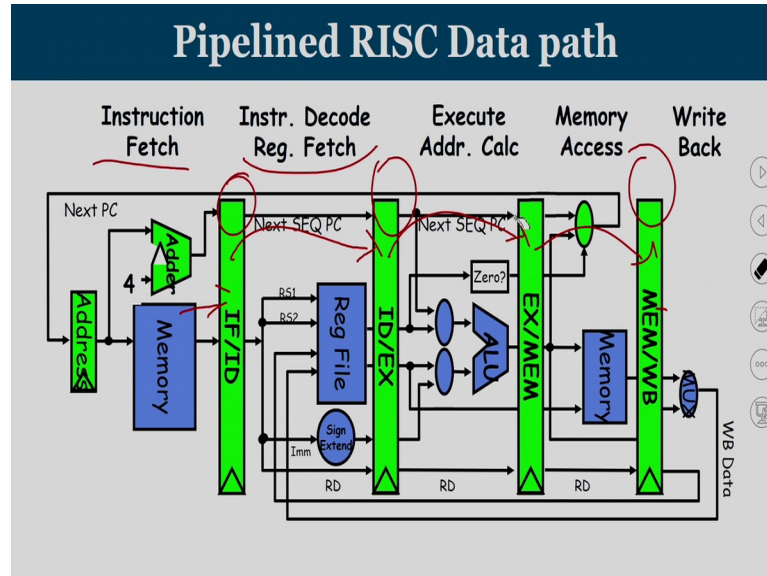
(Refer Slide Time: 23:52)



Now, this is the unpipelined RISC data path, where you can see the instruction fetching operation, it as an instruction decode the execute, the memory, and the write back stages.

If it is an unpipelined, this whole structure is going to be a single combinational block, where one task is over; it is given to the other combinational block.

(Refer Slide Time: 24:17)



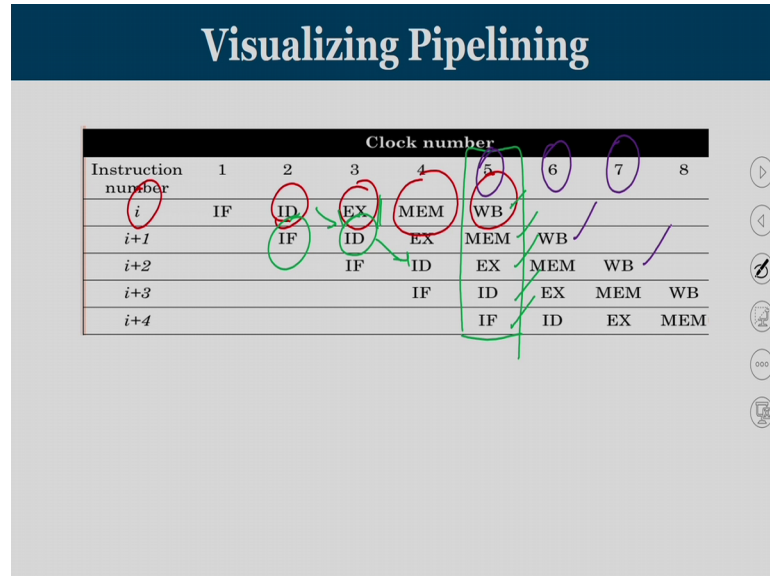
When you are going to pipelined structure, what you see is in between each of these independent combinational blocks, we are going to add pipelined registers. So, at the end, and this is been given to a common clock. So, the beginning of a clock, the instruction fetch unit will work, at the same time, instruction decode unit is also working, maybe on a different instruction.

At the end of this clock cycle, the operation instruction fetch will fetch the instruction and store it in the IF ID register. Whatever was been taken by the IF ID register in the previous cycle, the instruction decoder will produce the decoded output in the ID EX register. And whatever was available in the ID EX register, the ALU unit will perform this task, and it will produce in the EX MEM register. Similarly, it is from the EX MEM register into the MEM and write back register also.

So, for a pipeline to effectively work, we actually need a pipeline register. So, every pipeline stage, at the beginning of a clock cycle will take the input from one pipeline register, perform the activity, and keep the result in the next pipeline register. So, when the clock ticks again, this pipeline register goes into the first pipeline register, which might be already (Refer Time: 25:32) with the new set of task by its previous unit. So,

that every unit will take from one pipeline register, perform the task, and put the result in the second pipeline register.

(Refer Slide Time: 25:52)



Now, coming to visualizing the pipeline, this is how basically a pipeline works. Let us say and clock cycle 1; I am going to fetch the *i*th instruction. The fetching operation is over in the first cycle, second cycle I perform the decoding, third cycle I will perform the execution, fourth cycle memory operation, and fifth cycle, it is a write back operation. In the unpipelined case, your *i* plus 1th instruction can only start in the 6th clock cycle.

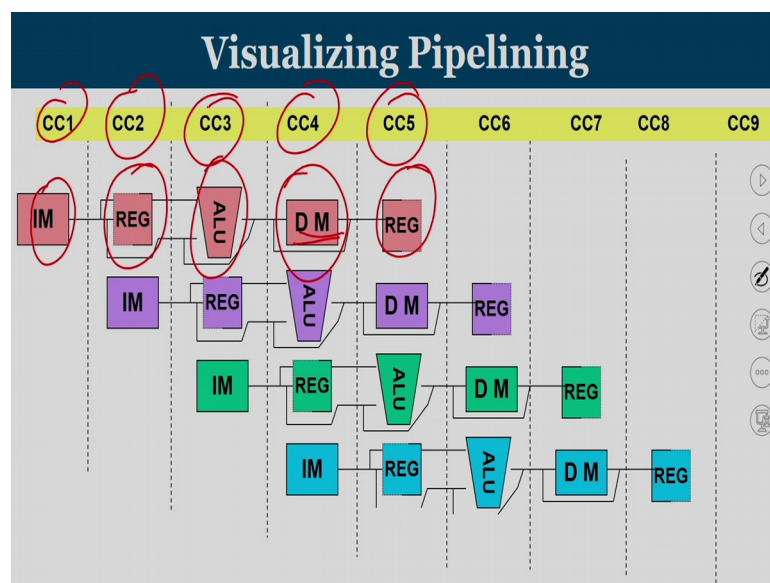
Now, we see what happens in pipeline. The *i* plus 1th instruction will be fetched in the second clock cycle that is the main advantage; the fetching happens in the second clock cycle. At the same time, the decoding is in progress for the first instruction. When the first instruction moves to the execution stage, then the decode unit is free. So, already fetched *i* plus 1th instruction will move to decode unit. So, the decode unit, this is the way, how the decode unit works, this is the way how a pipelined is been visualized.

Now, if you look at clock cycle 5, we can see that the write back stage is busy with the *i*th instruction, the memory stage is taking care of the *i* plus 1th instruction, the EX stage is taking care of *i* plus 2th instruction, the ID stage is taking care of *i* plus 3th instruction, and the IF stage is taking care of *i* plus 4 instruction. After clock cycle 5 write after clock cycle 5, you see the first instruction is completing in clock cycle 5. 6th clock cycle, my

second instruction is completing. In the 7th clock cycle, my third instruction is completing.

Thereafter for every clock cycle, one instruction is getting over. So, the idea of pipeline is one instruction is getting over in one clock cycle, on an average. Even though the 1st four cycles, we are not able to complete any instruction, from the 5th clock cycle onwards, one one instruction is getting over. So, pipeline increase this the throughput of the system, it would not would not reduces the latency of a single instruction.

(Refer Slide Time: 28:04)



Now, this is yet another way of visualizing the pipeline. In clock cycle 1, we are going to work on the instruction memory, where I am fetching the instruction. In clock cycle 2, I read the value from registers. Clock cycle 3, ALU is been use in order to carry out the task. Clock cycle 4, if there is any data to be access from memory, so I call it as instruction memory in clock cycle 1; I call it as data memory in clock cycle 4. And then, in the 5th clock cycle I am going to write the values into the registers.

So, similarly we can see that in pipeline, we are going to work with either memory. So, memory is been accessed in the 1st clock cycle, and the 4th clock cycle. Register is been accessed in the 2nd clock cycle, and the 5th clock cycle. And the functional units like ALU is been accessed in the 3rd clock cycle. To understand the storages, how cache memory works. This is the main area in which cache memory is been accessed. (Refer Time: 29:03) fetching an instruction and while accessing the instructions from memory,



these are the two places, where performance of cache memory is also crucial. Now, these are the pipelined registers that are been added in between them.

(Refer Slide Time: 29:21)

### Example- Design Overhead

Assume that a non-pipelined processor has a 1ns clock cycle and it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40% resp. Suppose that due to clock skew and setup, pipelining the processor adds 0.2ns of overhead to the clock. Ignoring any latency impact, how much speedup in instruction execution rate will we gain from a pipeline?

**Solution:**

$$\text{Avg.Exec.Time}_{NP} = \text{Clock cycle} * \text{Avg. CPI}$$

$$= 1\text{ns} * [(40\% + 20\%) * 4 + 40\% * 5] \rightarrow = 4.4\text{ns}$$

$$\text{Avg.Exec.Time}_P = 1.2\text{ns} (= \text{one clock cycle})$$

$$\text{Speedup from pipelining} = \text{Avg.Exec.Time}_{NP} / \text{Avg.Exec.Time}_P$$

$$= 4.4\text{ns} / 1.2\text{ns} \rightarrow = 3.7 \text{ times}$$

Now, consider this question, which is mentioning about what is overhead associated in pipeline. In the case of an unpipelined unit, we have a set of combinational blocks, which are one after another. Once you pipeline it, you have to independently divide them in between a pair of pipeline stage; we have to include a pipeline register. And reading into this register or writing into this register, all are going to create slightly extra overhead; we call it as pipelined skew and timing.

So, this particular question, assume that a non-pipelined processor has a 1 nanosecond clock cycle and it uses 4 cycles for ALU operations and branches, 5 cycles for memory operation. So, we have totally 3 category of instructions; they are ALU, branch, and memory operations. Assume that the relative frequency of these operations are 40 percent, 20 percent, and 40 percent, so respectively. Suppose, that due to clock skew and setup, pipelining the processor adds 0.2 nanosecond overhead to the clock. Ignoring any latency impact, how much speed up in instruction execution rate, we will gain from the pipeline.

So, average execution time, in the case of a non-pipelined processor is clock cycle into average CPI. So, we have 40 percent of instruction, which will take 4 clock cycle; another 20 percent of instruction, which will also take 4 clock cycle. So, 60 percent of

instructions will take only 4 clock cycles, the remaining 40 percent of instruction will take 5 clock cycle. This will give you how many clock cycles this particular program will take, and one clock cycle is 1 nanosecond. So, the total execution time for one instruction on an average is 4.4 nanosecond.

Once you come to pipelined design, you know that every instruction is going to get over in one clock cycle, if you eliminate the first few clock cycles for the very first instruction. So, the idea of pipeline is CPI is equal to 1; cycles per instruction is equal to 1 that is a target, we are trying to achieve using pipelining. So, when CPI is equal to 1 for a pipelined design, our average execution time for pipelined is one clock cycle, but in the question it is mentioned that due to pipelining our clock cycle is stretched by 0.2 nanosecond extra. So, already it is 1 nanosecond, now on top of this 1 nanosecond, we are going to add another 0.2 nanosecond; that is why, it is 1.2 nanosecond.

Now, what is the speed up that you get, average execute; this is what we have learnt from Amdahl's law. Average execution time of non-pipelined divided by average execution time of pipelined; so you will be getting 4.4 nanosecond divided by 1.2 nanosecond, and that is 4.24 nanosecond by 1.2 nanosecond, so it will be 3.7 times is the speed up that you get.

(Refer Slide Time: 32:32)

**Pipelining Issues**

- ❖ **Ideal Case: Uniform sub-computations**
  - ❖ The computation to be performed can be evenly partitioned into uniform-latency sub-computations
- ❖ **Reality: Internal fragmentation**
  - ❖ Not all pipeline stages may have the uniform latencies
- ❖ **Impact of ISA**
  - ❖ Memory access is a critical sub-computation
  - ❖ Memory addressing modes should be minimized
  - ❖ Fast cache memories should be employed

Now, we will see having said all these five stages, we feel that pipelining will work perfectly, but there are certain cases or certain design issues with respect to the pipeline;

we will see, what are them. The 1t one ideally we feel that there are uniforms of computations, that is all the stages in pipeline will be of uniform stage. The computation to be performed can be evenly partitioned into uniform latency sub-computation that is a ideal case.

But, in real case, we feel that not all pipeline stages may have uniform latency that means, the time required to carry out your IF instruction fetch may not be equal to the time required to carry out instruction decode. So, initially, when we discussed you told that all five units are there, and all five units will take equal amount of time, but once you write the combinational logic for them, we will understand that there will be taking variable delay.

So, how can we deal with this? That is the impact of instruction set architecture. Memory access is a critical sub-computation. So, wherever possible try to reduce memory access, because while you go to memory, it is going to take more amount of time, because it is not inside the processor, we have to go outside the processor to fetch from memory. So, wherever possible try to reduce memory access and use fast memory, such that the speed up access from on memory is proportional to or at par with the pipelined clock. Memory addressing module should be minimized, that is the second point, and wherever possible try to use fast cache memories.

(Refer Slide Time: 34:17)

## Pipelining Issues

- ❖ **Ideal Case : Identical computations**
  - ❖ The same computation is to be performed repeatedly on a large number of input data sets
- ❖ **Reality: External fragmentation**
  - ❖ Some pipeline stages may not be used
- ❖ **Impact of ISA**
  - ❖ Reduce the complexity and diversity of the different instruction types
  - ❖ RISC architectures use uniform stage simple instructions

Now, the second pipeline issue is we feel that identical computations are there on all instruction combination. The same computation is to be performed repeatedly on a large number of input data set, but the reality is some pipeline stages may not be used like we have discussed in the case of the RISC pipeline, we are not using the MEM stage for all the instruction. For ALU instructions, the MEM stage is just a bypass unit.

So, what can the instruction set architecture do, reduce the complexity and diversity of instruction, and try to see that instructions can be designed in such a way that they will go through the basic five stages. We remove to CISC architectures, such a kind of an optimization is possible, certain instructions will take five clock cycle, certain instructions will take ten clock cycle, some will take even more than ten, and some may take even less than five. But, when you go to a RISC architecture, we try to see that every instruction is taking uniform number of clock cycles.

(Refer Slide Time: 35:18)

**Pipelining Issues**

- ❖ **Ideal Case : Independent computations**
  - ❖ All the instructions are mutually independent
- ❖ **Reality: Pipeline stalls – cannot proceed.**
  - ❖ A later computation may require the result of an earlier computation
- ❖ **Impact of ISA**
  - ❖ Reduce Memory addressing modes - dependency detection difficult
  - ❖ Use register addressing mode - easy dependencies check

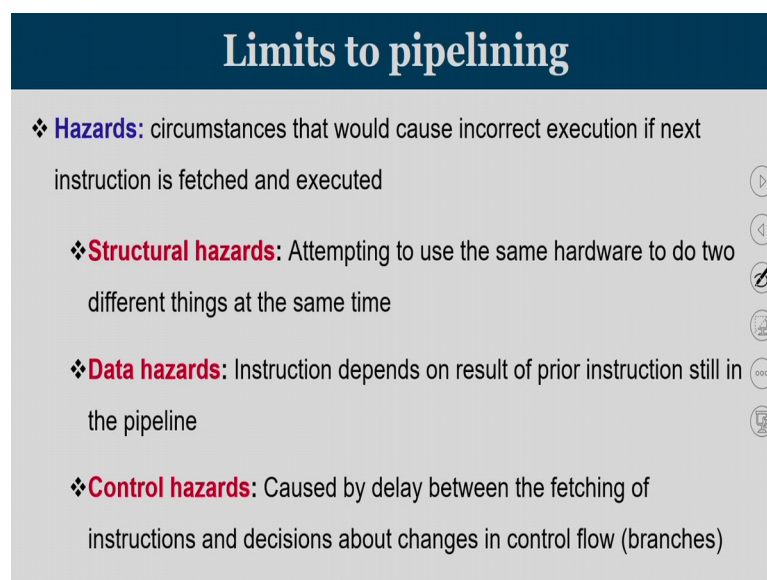
And the third issue is independent computations. Our idealistic assumption is all instructions are mutually independent. We cannot write a meaningful program, if all instructions are mutually independent. It is the dependency between the instructions that give the program a logical shape and the meaningful outlook. But, when you have instruction that are dependent, how will pipeline works. The reality is pipeline will stall; we cannot proceed that means, a computation may require the result of an earlier computation. So, sometimes the nth instruction can be executed, only if the n plus 1th

instruction is complete. So, in this case, parallelly doing things, what we have discussed in the pipeline, may not work.

So, the impact of instruction set architecture is you try to reduce memory addressing mode, such that your dependency detection is easy, because in memory addressing modes, finding out whether the  $i$ th instruction is having a dependency between  $i$  minus 1th instruction is very difficult. But, if you use a register addressing mode, it is easy to know, whether the source register of one instruction is the destination register of the next instruction. So, using register addressing mode is slightly easy in this case.

So, we have seen there are basically three issues that the pipeline has to take care off. Our 1st idealistic assumption is all the units will be of uniform latency, which is not true. So, the instruction set architecture has to be designed in such a way that every instruction will have an uniform kind of or more or less uniform latency. 2nd one is that every instruction need not go through all this sub-computation. And 3rd one is there maybe dependency between different stages in the pipeline, so we have to take care of that aspect as well.

(Refer Slide Time: 37:22)



**Limits to pipelining**

- ❖ **Hazards:** circumstances that would cause incorrect execution if next instruction is fetched and executed
- ❖ **Structural hazards:** Attempting to use the same hardware to do two different things at the same time
- ❖ **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
- ❖ **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches)

Now, we will see certain other problems that the pipelines are having. So, they are the limits to pipeline, we call it as hazards. Hazards are circumstances that would cause an incorrect execution if an instruction is fetched and executed in its pipelined slot. So,

hazards are certain scenarios in which our normal pipelining sequence would not work well.

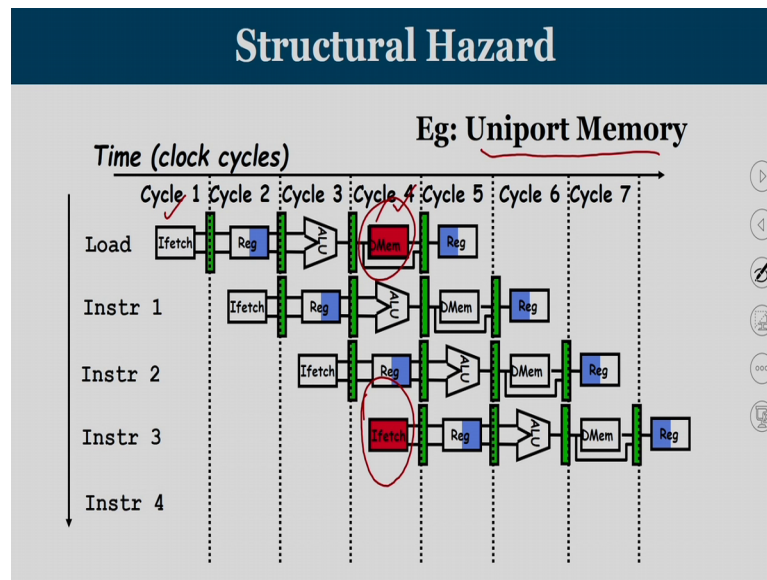
There are three different types of hazard. The 1st one is called structural hazard; attempting to use the same hardware to do more than one operation at a time. Let us say  $i$ th instruction and  $j$ th instruction, both wanted to use one particular unit at the same time. Such a scenario is called structural hazard, where your resource cannot be shared across two instructions.

So, if you do that, that leads to an incorrect execution. The 2nd one is called data hazard; the instructions depend on the result of the previous instruction. So, only if the previous instruction is complete, then only the current instruction can do. So, the current instruction fetch runs in its normal pipeline slot, then it is going to give us incorrect answers, that is called data hazard.

And the 3rd one is going to be the control hazard. Control hazard is caused by the delay between the fetching of an instruction and decision about changing the control flow. When you fetch an instruction, the very next cycle the next instruction is fetched. Let us say the first instruction is going to be a branch instruction, only if you know the condition of the branch, then only you will be knowing, whether the adjacent instruction is fetched or a target instruction is being fetched.

But in the normal case, the very subsequent cycle itself the next instruction is fetched. So, if you bring a wrong instruction, then that also will create issues so, these three problems, structural hazard, data hazard, and control hazards, will create issues while working with pipeline.

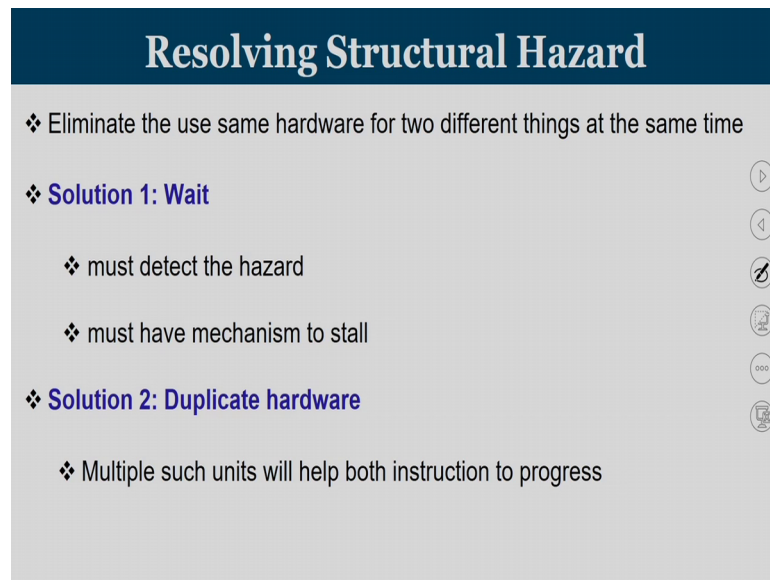
(Refer Slide Time: 39:12)



Now, we will see what a structural hazard is. One of the classical example of structural hazard is uniport memory. Consider the case; you have a first instruction that is load. The load instruction is going to fetch the instruction from memory in the 1st cycle, and going to take the data from the memory in the 4th cycle.

Consider the remaining instructions. When it comes to the 4th instruction, the 4th instruction will perform fetching in the 4th clock cycle, the instruction will be fetched. At the same time, memory has to be used for fetching data of the first load instruction. So, this is actually same resource that is your memory is going to be used by two instruction, exactly at the same time. Memory has to supply the instruction that is the 4th the 3rd instruction has to be given, as well as the data of the first instruction also has to be supplied with memory. One resource shared by two instruction, this is called a structural hazard.

(Refer Slide Time: 40:32)



**Resolving Structural Hazard**

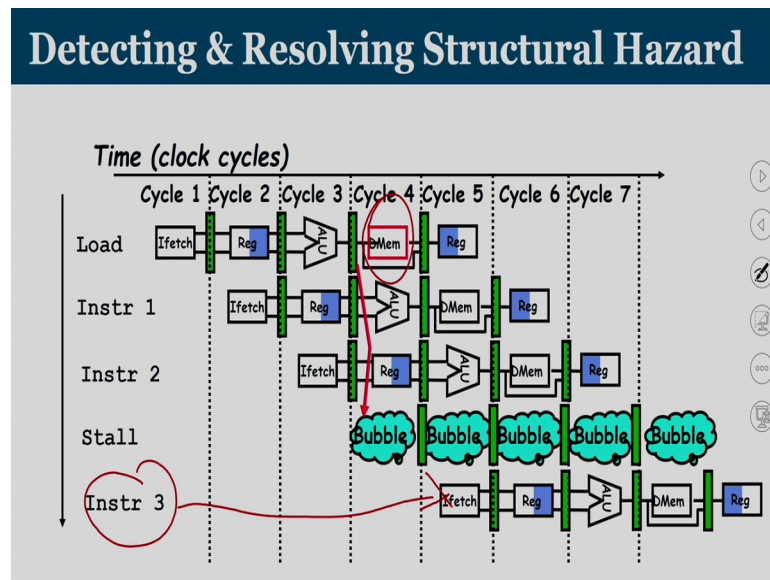
- ❖ Eliminate the use same hardware for two different things at the same time
- ❖ **Solution 1: Wait**
  - ❖ must detect the hazard
  - ❖ must have mechanism to stall
- ❖ **Solution 2: Duplicate hardware**
  - ❖ Multiple such units will help both instruction to progress

Now, how will you resolved a structural hazard. Eliminate the use of same hardware for two different things at the same time, there is a more simplest principle, but what are the two ways in which you can do. 1st one, whenever you detect a hazard, you wait until one instruction has used it. So, that in the next cycle or then subsequent cycles, the functional unit which created structural hazard may be free for the current instruction, so that you can use.

So, detect whether there is a structural hazard, if so ask one of the instruction to wait, we call it a stalling, the instruction will stall and subsequently try accessing this particular functional unit at a later stage. Second solution is duplicate the hardware. That hardware, which was creating structural hazard, create multiple instances of those hardware, and that is way how we can do.



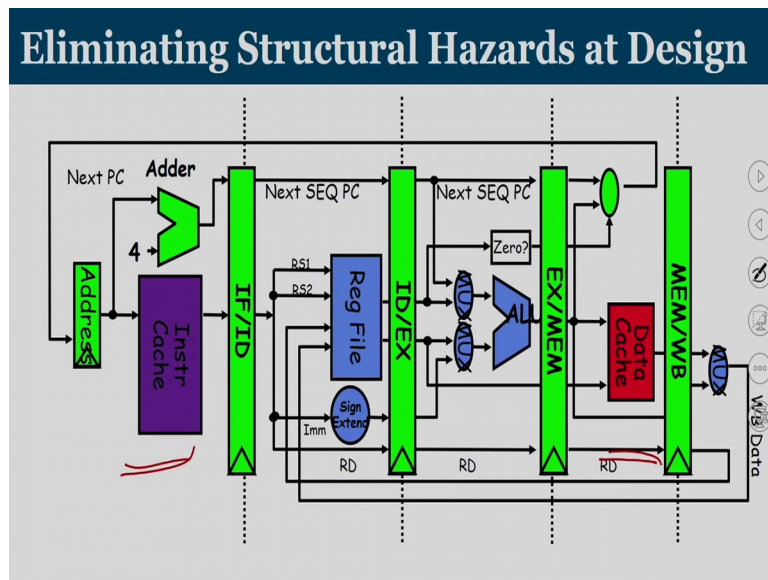
(Refer Slide Time: 41:10)



Now, detecting and resolving a structural hazard, the first mechanism we see in the same example we have seen. First instruction is load, and then it followed by another three instructions; so second instruction will work, third instruction will work. When it comes to the next instruction had it been run in the same slot, it is going to create a problem with this memory access. So, what we do is, we do not fetch the next instruction that is going to create a bubble or a stall.

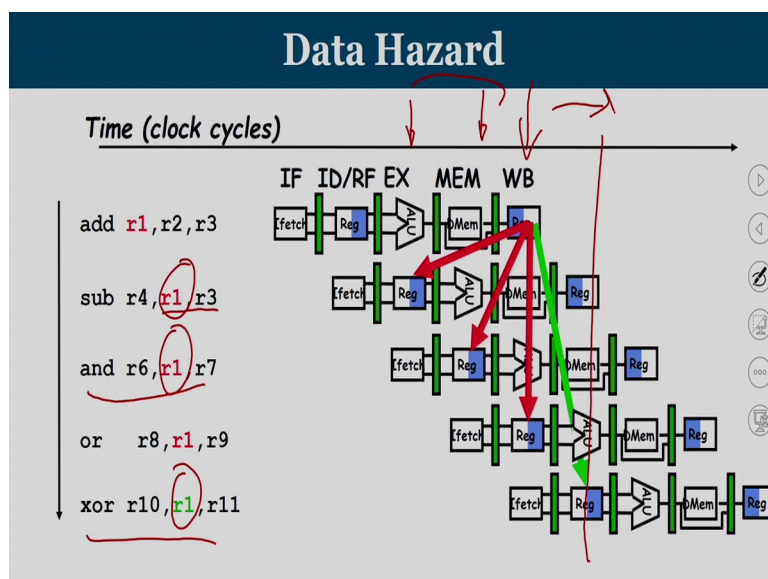
And so, we are trying to delay your instruction 3 to cycle number 5, rather than starting instruction 3 at cycle number 4, now we are delaying it by one cycle. This is called asking one of the instructions, detecting a hazard and asking one of the instructions to execute or to get its operation done after one cycle. So, if in this cycle also if that would have been a problem then we will shift one more this is the first approach detect and wait.

(Refer Slide Time: 42:49).



Now, second approach is by using design. So, we were having a problem like instruction fetch of one instruction is happening at the same time of memory stage of another instruction. If instruction fetch and memory can be carried out from two different units, then the problem can be resolved. This is called duplicating hardware. Can we have a separate hardware that will store instruction, and can we have a separate hardware for storing your data. This is your this led to the concept of having split caches; one cache for instruction, and other cache for data. So, that whenever the IF is happening instruction fetch is happening, it is not conflicting with the data fetch.

(Refer Slide Time: 42:58)



Now, the second type of hazard is called data hazard, where there exist a dependency between the data. One instruction cannot go or cannot execute cannot start until a previous instruction is over. Consider the instruction sequence, add, r 1, r 2, r 3 that is your 1st instruction, where r 2 and r 3 are added together, to store the result in r 1. 2nd instruction is sub r 4, r 3 and r 2, where the value that is produced in the first instruction r 1 is going to be used as a source operand in the second instruction. In the 3rd one is also r 1 is a source operand, 4th one is also r 1 is a source operand, and in the 5th 1 is also r 1 is a source operand.

Ideally, this is an instruction, where first instruction will produce, the result be stored in r 1. All other instructions are going to use these instructions used to going to use the value that is stored in r 1. So, let us this is your first instruction, which will produce the result at the WB backstage; in the write back stage only, you will get the result. The second instruction is supposed to run in these time slots, but you can see that the second instruction wants the value of r 1, at the ID stage that is the IF stage of just after IF the ID stage.

Third instruction also has this problem, it also reads from r 1. But, if you look at the time axis, r 1 value is ready only at this point. Whereas, somebody is trying to read r 1 at this point that is subtraction instruction, the and instruction is trying to read the value of r 1. So, whatever is be the value of r 1, in this points will be absolute values or old values. So, sub instruction cannot read r 1, and instruction cannot read r 1, in its designated slots.

This is also another problem for r instruction also the same issue is there, you would not get the correct value of r 1, if you try to read r 1 at this point, because you will complete writing only at the stage. But, when it comes to the XOR instruction, you are going to read the value of r 1 after the w backstage of the add instruction is over. So, this value will be correct. So, this scenario, where which is marked with the red arrows, they if you read the value of r 1 in those instructions, then it will lead to incorrect execution coming a hazard. This is called a data hazard.

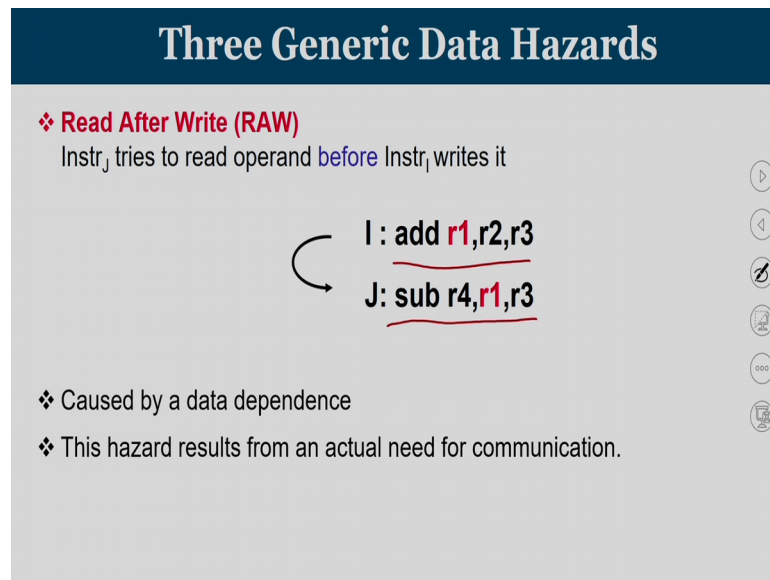
(Refer Slide Time: 46:01)

## Three Generic Data Hazards

❖ **Read After Write (RAW)**  
Instr<sub>j</sub> tries to read operand *before* Instr<sub>i</sub> writes it

I : add r1,r2,r3  
J : sub r4,r1,r3

❖ Caused by a data dependence  
❖ This hazard results from an actual need for communication.

A slide titled "Three Generic Data Hazards" with a dark blue header. The main content is on a light gray background. It defines Read After Write (RAW) as instruction J trying to read an operand before instruction I writes it. It shows two instructions: I: add r1,r2,r3 and J: sub r4,r1,r3. A curved arrow points from the 'r1' in instruction J back to the 'r1' in instruction I. The 'r1' in both instructions is underlined in red. To the right of the text are several small circular icons: a play button, a left arrow, a magnifying glass, a refresh symbol, a list icon, a search icon, and a refresh icon.

There are three different types of data hazards, we will see one after another. The 1st one is called read after write, where instruction J tries to read an operation before instruction I writes on it, we call it as a data hazard. First instruction is going to write a value into r 1, second instruction is going to read a value from r 1. If the second instruction read the value before the first instruction writes, then that is a hazard, so second instruction reading has to be delayed. This is this hazard result from an actual need for communication. So, whatever the result obtained in instruction I has to be forwarded to instruction J, in order to properly get the result.

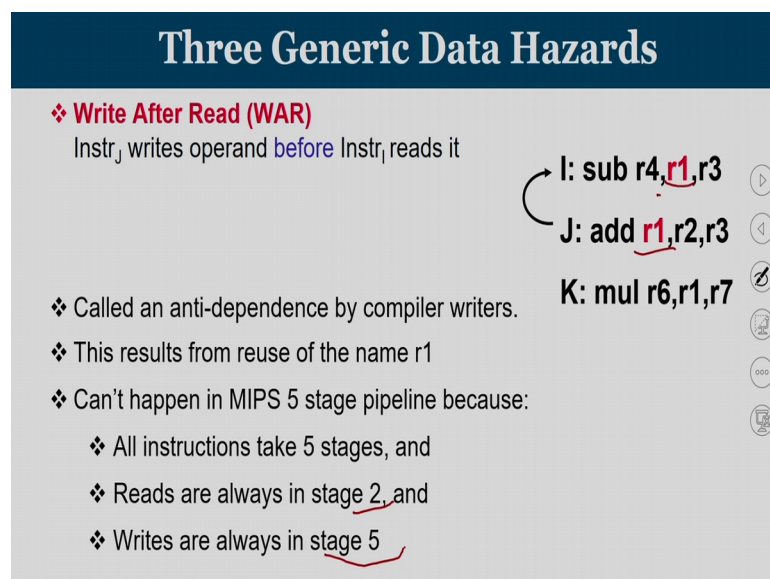
(Refer Slide Time: 46:31).

## Three Generic Data Hazards

❖ **Write After Read (WAR)**  
Instr<sub>j</sub> writes operand *before* Instr<sub>i</sub> reads it

I: sub r4,r1,r3  
J: add r1,r2,r3  
K: mul r6,r1,r7

❖ Called an anti-dependence by compiler writers.  
❖ This results from reuse of the name r1  
❖ Can't happen in MIPS 5 stage pipeline because:  
    ❖ All instructions take 5 stages, and  
    ❖ Reads are always in stage 2, and  
    ❖ Writes are always in stage 5

A slide titled "Three Generic Data Hazards" with a dark blue header. The main content is on a light gray background. It defines Write After Read (WAR) as instruction J writing an operand before instruction I reads it. It shows three instructions: I: sub r4,r1,r3, J: add r1,r2,r3, and K: mul r6,r1,r7. A curved arrow points from the 'r1' in instruction I back to the 'r1' in instruction J. The 'r1' in both instructions I and J is underlined in red. To the right of the text are several small circular icons: a play button, a left arrow, a magnifying glass, a refresh symbol, a list icon, a search icon, and a refresh icon.

The second category of hazard is called write after read hazard also known as WAR hazard. Instructions J writes operand before instruction I reads it. So, consider this scenario, you are going to read a data, the second I is going to write a data into r1. This kind of a hazard is called anti-dependence, because we are going to use the same r1; r1, which was the source operand, we are going to write on r1.

But, in the ideal 5 stage pipeline of RISC, this is not a problem can happen in a MIPS RISC 5 stage pipeline, because all instruction take 5 stages. Reads typically happen in the 2nd stage, and writes happen in the 5th stage. So, if Ith and Jth instruction are executed in the normal sequence in the pipeline, then they are not going to create any problem. But, when it goes to out of order processors, which we will see later, then this kind of hazards can happen.

(Refer Slide Time: 47:36)

### Three Generic Data Hazards

- ❖ **Write After Write (WAW)**  
Instr<sub>j</sub> writes operand **before** Instr<sub>i</sub> writes it.
- ❖ Called an output dependence
- ❖ This also results from the reuse of name r1.
- ❖ Can't happen in MIPS 5 stage pipeline because:
  - ❖ All instructions take 5 stages, and
  - ❖ Writes are always in stage 5
- ❖ WAR and WAW happens in out of order pipes

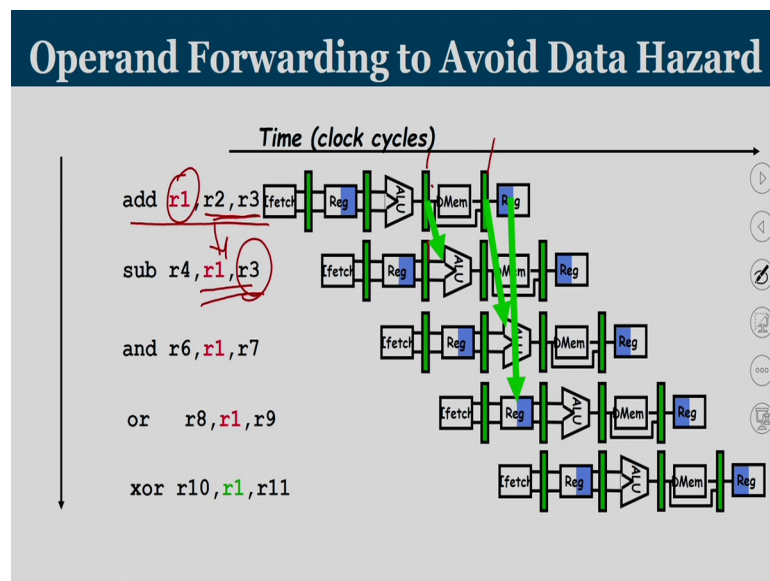
**I: sub r1,r4,r3**  
**J: add r1,r2,r3**  
**K: mul r6,r1,r7**

The next type of hazard is WAW hazard. When instruction J writes the operand before instruction I writes on it. Considered this instruction, where I and J both are going to write into r1, and your instruction K is going to read from r1. The program meaning is the value that instruction K reads the value of r1 should be the result of Jth instruction. Ideally, we expect Ith instruction will write first followed by J, but if by chance, I is delayed in out of order kind of execution we will see that.

If the order is violated, then what our the value K will read is a wrong value. And this is called output dependence, this also result from the reuse of the name r1. Add J use a

different name or (Refer Time: 48:18) I use a different name, then this problem would have been there. So, compiler should have done this optimization, such that we can avoid WAW hazards. This also typically (Refer Time: 48:27) happen in a MIPS 5 stage pipeline, because all instruction take 5 stages, and the write happens typically in the 5th stage. But, as mentioned already WAR and WAW hazards can happen in out of order processors.

(Refer Slide Time: 48:43)

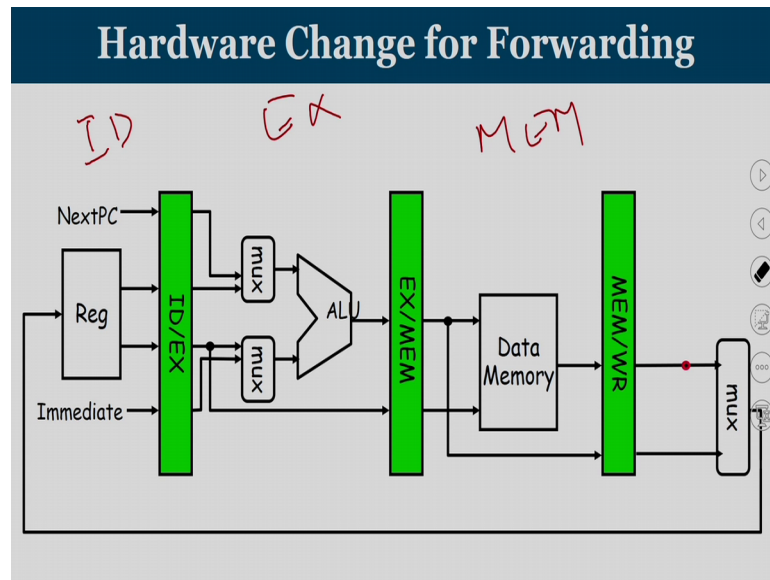


Now, how will you take care of these hazards data hazards? Whenever there is a RAW hazard, read after write hazard that can be taken care by a technique called operand forwarding. So, consider this case, you have the first instruction add, you have the first instruction add is there, where it is not the content of r 1 what we want, we want the sum of r 2 and r 3 that is to be forwarded to the next instruction. And it is a ALU will produce the sum of r 2 and r 3. So, from the output of ALU, I can give the value to the input of ALU, such that the next instruction when it reaches, it will get the value.

So, even though the instruction says that we how to read from r 1, we are not specifically looking for reading from r 1 rather than that, we are interested in the result of r 2 plus r 3; r 2 plus r 3 is computed in the ALU. Once ALU produce the result, then that result is forwarded to the input side of ALU, such that when r 3 is ready, it can be added with this r 2 plus r 3, such that it will replace the contents of r 1; it can be used instead of contents of r 1. So, wherever you see this green arrows, there actually forwarding the result and

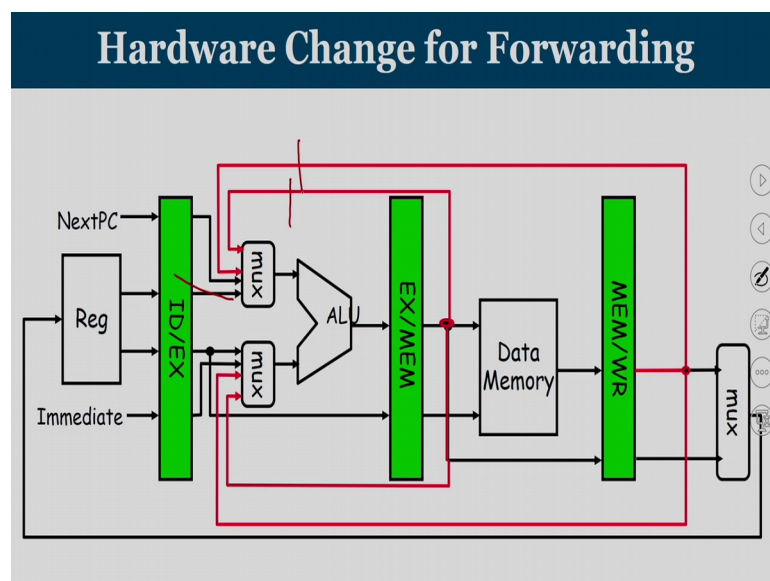
result forwarding or operand forwarding, can happen only from one pipeline register to another pipeline register. It can be from MEM stage to the input of ALU or it can be from output of ALU to the input of ALU.

(Refer Slide Time: 50:17).



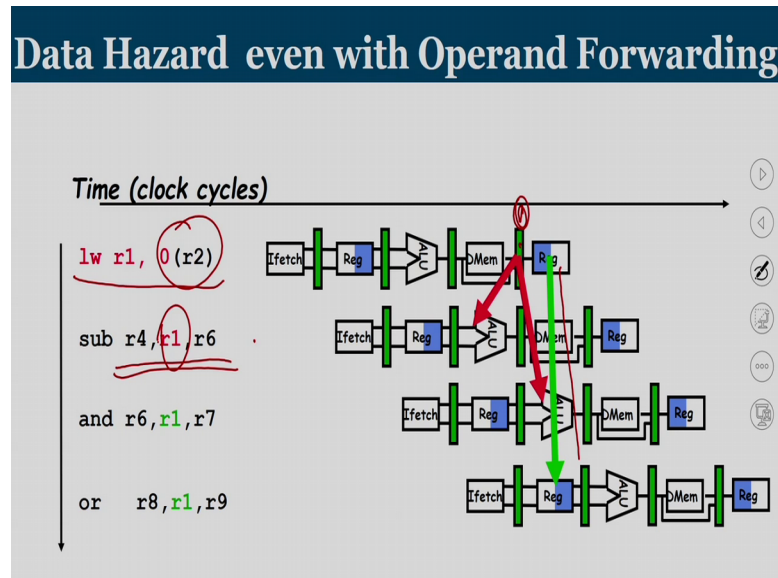
So, this is how it works. This is your the ID stage, this is your EX stage, and this is your MEM stage. So, these stages can forward the appropriate (Refer Time: 50:29). So, these are the pipeline registers, which connect the stages.

(Refer Slide Time: 50:37)



And now, what happens in forwarding is, the output of ALU is being connected to the input of ALU by a proper multiplexer. Similarly, output of the MEM stage is also connected to this multiplexer. And the control bits in the multiplexer will take a call, whether the data should be taken from this pipeline register or the data should be taken from one of the input of the multiplexer. This is the technique of operand forwarding.

(Refer Slide Time: 51:04).

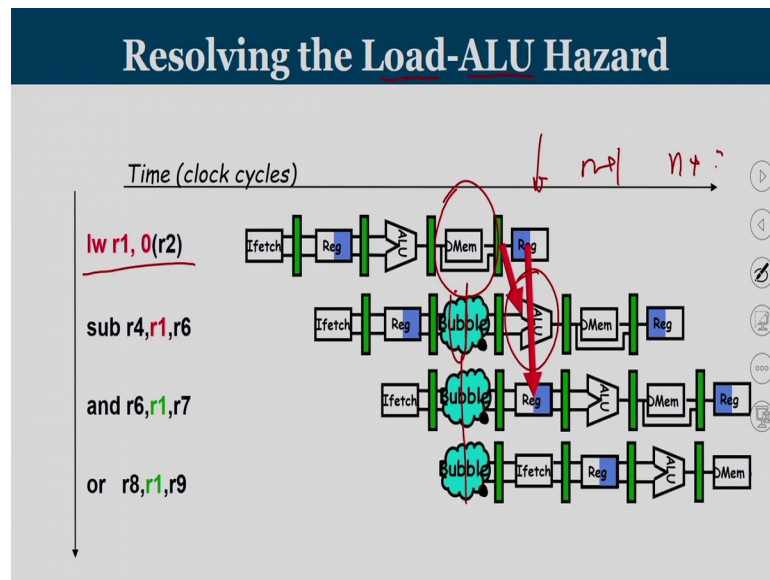


Now, even with operand forwarding, still data hazard exist. This is a special case, where is the first instruction is a load instruction, where the value that is to be loaded is obtained not at the end of ALU. At the end of ALU, you will get only the effective address, and then you go and access memory. So, the data to be written to `r1` is available only at the end of MEM stage. But, even if we use operand forwarding, we cannot forward, because the second data the `r1` is needed latest by the ALU stage for the subtraction instruction. So, when you forward a data from this pipeline register, it cannot forward to ALU, because it is in the negative time access.

So, in this case, whenever there is an ALU operation that makes use of a value that is been taken from a load instruction, then even with operand forwarding, techniques would not work. But, in this case is the green arrow, whatever it is shown by this green arrow, the operand forwarding will work, but this red one, it would not work. So, in this case, our subtraction instruction cannot execute in its assigned slot, it has to be delayed.



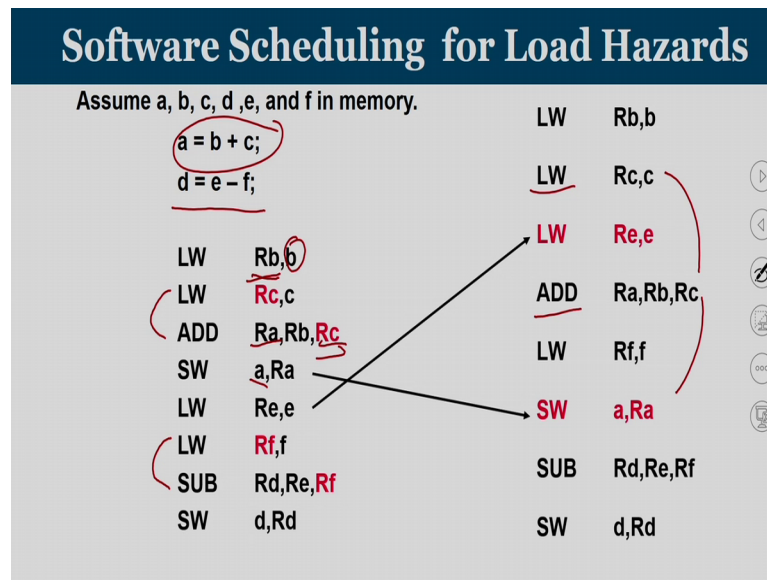
(Refer Slide Time: 52:14)



So, this is what happens, your load will run with operand forwarding, you how to make sure that the execution of the subtraction instruction happens after the memory stage of the load instruction. Execution of the subtraction instruction should happen after the memory stage of the load instruction. So, in order to facilitate that, we have to insert a stall or a bubble, such that IF and ID will happen, the ALU stage will be happening only one stage after that, subsequently the bubble is propagated for the remaining instructions as well.

So, whenever there is a load instruction, and immediately after that when you have an ALU instruction, then they cannot run in adjacent clock cycles, there has to be a delay for one clock cycle that is what we can (Refer Time: 53:02). If this happens in  $n$ , ideally we expect that  $n + 1$  the next instruction will get over this next instruction will get over only at  $n + 2$ .

(Refer Slide Time: 53:17).



Now, compiler also can play a very significant role. If it understands that, there is a data hazard. So, compiler if (Refer Time: 53:23) reorganizes these instructions, when it comes to hardware, a add immediately after a load kind of thing a dependent load can be removed. We will see an example, where how compiler helps. Consider the case of a simple fragment of code a equal to b plus c, and d equal to e plus f, where a, b, c, d, e, and f, are in memory locations. So, how will be the machine code looks like. First, you load the word into register Rb, the value of b is loaded into one of the registers. Value of the second register is loaded into another register Rc. So, Rb and Rc are registers, then you are going to add.

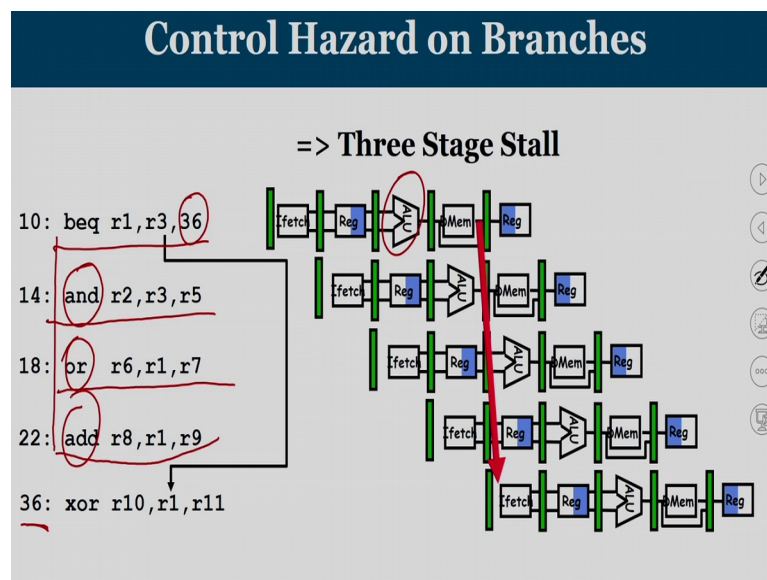
So, we have a load, and immediately after that we have a add, the red color indicates that this is the data dependency. When it runs on hardware, these load instruction and add instruction cannot run in adjacent clock cycles. Now, the second leg, then you store the value of Ra into a. So, with this a equal to b plus c portion is over. And you repeat the similar code, we have the value to be loaded into e and f, then we have to subtract, and then you have to store.

Here also you can see that between this load and ALU operation, there is a delay. Even with operand forwarding, this stall cannot be avoided. So, how can you reorganized this? Compiler can schedule, we call it a software scheduling, compiler can reorganize them, so we will keep the 1st two instructions as such. Now, what we do is we are going to take

the next load instruction into this point, and then so that means, these two are now separated by one clock cycle. We are finding out some other useful instruction and keeping it in between this load and add, such that there is no delay between them.

And then, we continue, then we are going to put the store value, whatever is available. So, so this add and store is also sufficiently separated, and then we have the storing of the remaining value into the d. So, compiler actually reorganizes this instructions. So, compiler should have the intelligence that is available, such that compiler should know if you run this instruction in the same sequence, it is going to create two delays; one at this point, and the other one is this point. So, compiler has to reorganize them, find out suitable instruction, such that they can be filled in this corresponding slots.

(Refer Slide Time: 56:01)



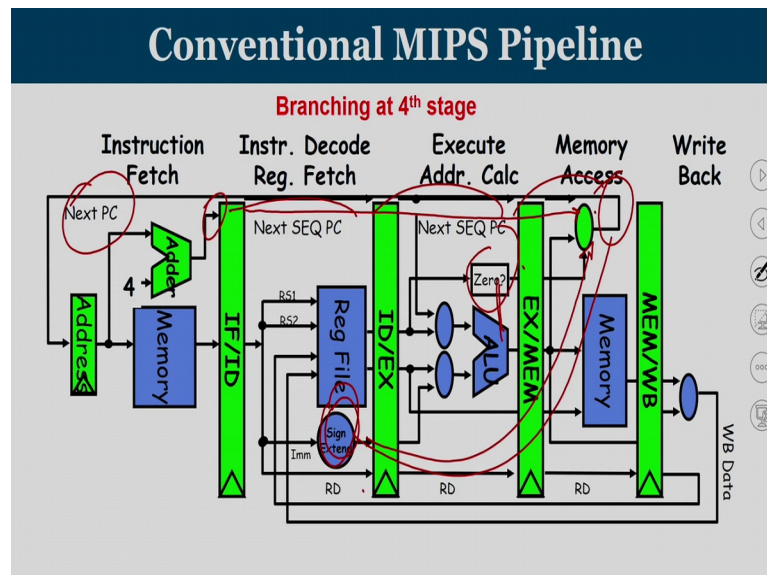
Now the third category is called control hazards. Consider the case that you have an instruction (Refer Time: 56:04) equal to r 1, r 3, 36. The meaning of this MIPS instruction is if the value of r 1 and r 3 are equal, then we have to jump into instruction 36. So, how will you get it. If the value of r 1 and r 3 is equal, then I can to jump into 36.

But, how will I know r 1 and r 3 is equal, that I can know only at the ALU stage. So, till then what instructions you will fetch, that is the tricky part of working with the pipeline. So, by the time, I am decoding this instruction, the next instruction will be fetched. And by the time, I am executing the OR instruction, the instruction stored at memory address

18 will be fetched. And by the time, I am doing the memory operation or comparison, the next is also fetched.

So, suppose let us say my instruction finds that r 1 and r 3 are same, then all these instruction that is there in 14, 18, and 22, they should not have executed now. So, whatever we have fetched and executed, they are wrong instruction. So, here is yet another scenario that if you execute this instruction and, or, and add, then they are going to create a wrong result, this is because they are not supposed to be fetched and executed, because there is a branch condition. Handling this is known as a control hazard.

(Refer Slide Time: 57:39)



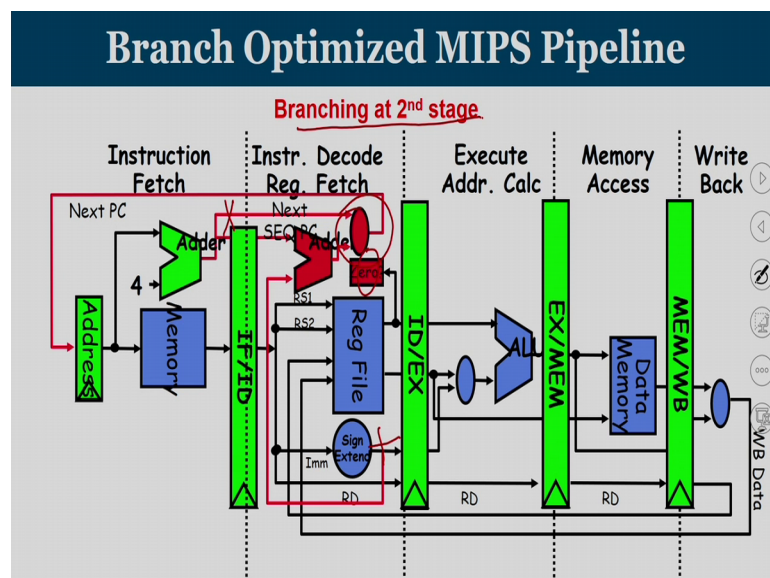
So, in conventional MIPS pipeline, what we have seen previously, we have a unit. So, what happens is this r 1 and r 2 is compared in the previous case, whatever we have shown r 1 and r 2 is compared. So, you subtract them, if it is equal to zero, then in this stage so your next value that means, PC plus 4 is available here, and whatever value that is available in the instruction that will reaches here that means, the target instruction as well as the next instruction is available here. And then, you check what is the condition, and that is going to give you like what is going to be your next program counter.

So, it is the fourth stage in the pipeline in the kinds of a conventional MIPS processor, we deal with branching conditions. So, in branching means either you can go to adjacent instruction that is PC plus 4 will be computed here, and then value has to be passed across various stages or you have to find out the value, which is part of the instruction,

the target instruction. So, both this branch values are available to you, now you check the branch condition.

Here, the previous case we have seen, you have to check the value, whether r 1 and r 3 are equal; how will you know they are equal, you have to subtract them and check, whether the zero flag is set, that is what equal zero flag. An optimized version, so this will make sure that for every branch will come to know what is the output of branch, only at the 4th clock cycle by then already three instructions are fetched, which sometimes you may have to flush out.

(Refer Slide Time: 59:22)

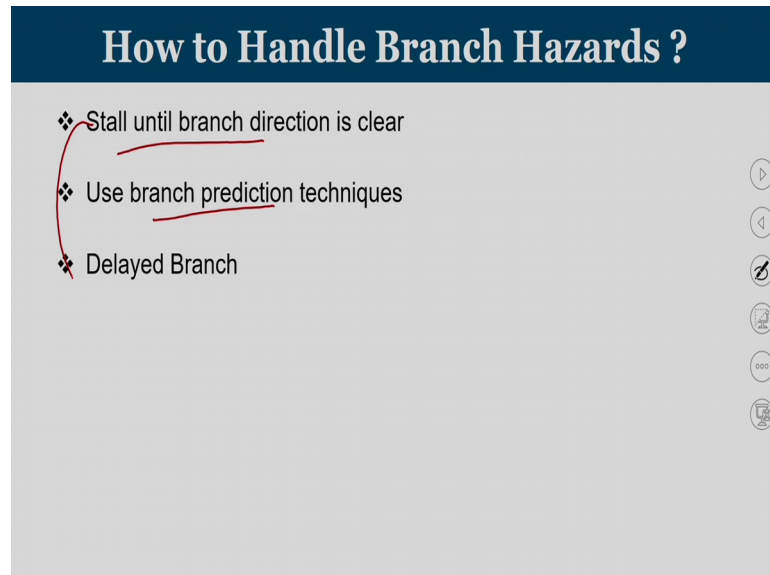


So, an optimized version of MIPS pipeline is we have to rearrange the instruction, such that we cannot compare two registers. If you want to compare two registers, then they can be done only in the ALU stage. But, if you slightly change your instruction set design in such a way that, conditional checking can be done only on test of zero. Your branch instructions can be done only like in following cases, I can check whether r 1 is equal to zero, whether r 2 equal to zero, whether r 3 equal to zero. Test of zero if that is the only branching condition, zero checking can be done very early in the pipeline.

So, in the second stage itself, I know what is PC plus 4, I know what is going to be the target instruction. So, both the target instruction as well as PC plus 4 will come to this unit, and the zero testing flag will tell whether should I take PC plus 4 that is an adjacent

or follow through instruction or the target instruction, such that I could resolve branching in the second stage.

(Refer Slide Time: 60:28)



Now, how to handle branch hazards? One simple way of handling branch hazard is whenever you find that there is a branch instruction, the moment you fetch an instruction, when you are going to decode it, you know this is a branch instruction. The moment you understand, it is a branch instruction, do not fetch any other instruction, you wait until the branch condition is resolved. Once the branch condition is resolved, then you have an idea, whether should the follow through instruction be fetched or the target instruction to be fetched, so then you can go and do. So, ideally you are stalling pipeline for few clock cycles, till the branch condition is resolved, that is called stall until the branch direction is clear.

2nd one we use sophisticated mechanisms called branch prediction techniques, these are mechanisms by which the moment you see on a branch, you know whether it will be taken branch will be taken that means, you have to go to target location or branch will not be taken that means, you have to go to the adjacent instruction. Based upon that, appropriate instructions can be fetched. In the next class, we will learn deeper about how branch prediction techniques are been applied.

And the 3rd concept is called delayed branch. So, this is a combination of the 1st one. Whenever you come to know it is a branch, then until the branch is resolved. You try to

fill up some kind of an instruction that is being done by compiler, compiler will find out some useful instruction and put it immediately after the branch instructions, and these instructions are those instructions, which how to be executed for sure, whether branch is taken or not taken. So, we will see deeper about branch prediction techniques and delayed branch techniques, in the next class.

So, with that we are going to conclude today's lecture. Just to summarize, we learned about the concept of instruction pipelining, what are the advantages of pipeline. And then, we have seen the five stages of RISC architecture, RISC pipeline architecture. And we see what are hazards; we have seen structural hazard, data hazard, and control hazard, and some of the techniques that are used to resolve these hazards. So, if you have any doubt in this session, please feel free to post your queries in this online discussion forum, we will so resolve your queries by getting back to you.

Meanwhile spend time on working on the tutorial sheet or the problem or assignment sheet that is being uploaded. And hope that you enjoy the session of instruction pipeline. Next day we will learn about some advanced pipelines and branch prediction mechanisms.

Thank you.