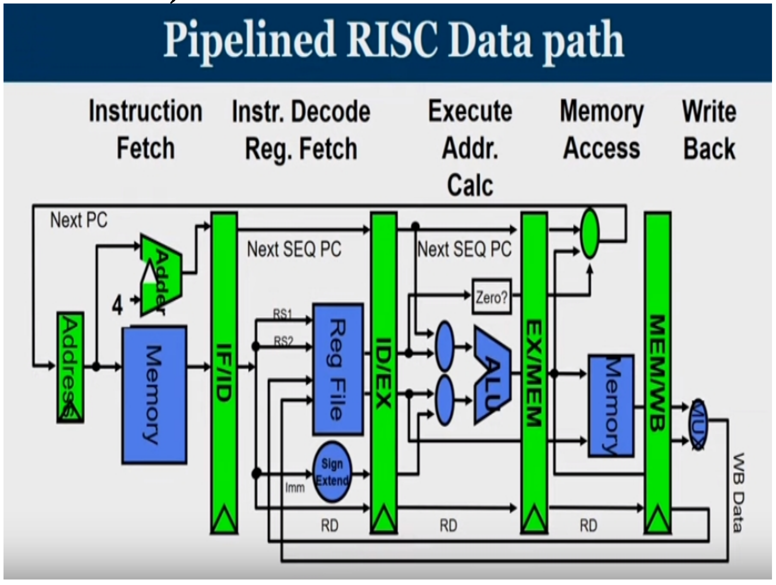
Advanced Computer Architecture

Dr. John Jose
Department of Computer Science & Engineering
Indian Institute of Technology-Guwahati

Lecture - 04
Pipeline Hazards

Welcome to the fourth lecture of the course. Today's lecture will be dedicated on discussion related to pipeline hazards. From the last lecture, we have seen what is the MIPS 5-stage pipeline and we have seen what are the functionalities that are being done on each of these 5 stages. We will have a quick recap on these 5 stages and see what are the scenarios in which the ideal pipeline will not work and these are called hazards and then we will try to address how these hazards are being resolved.

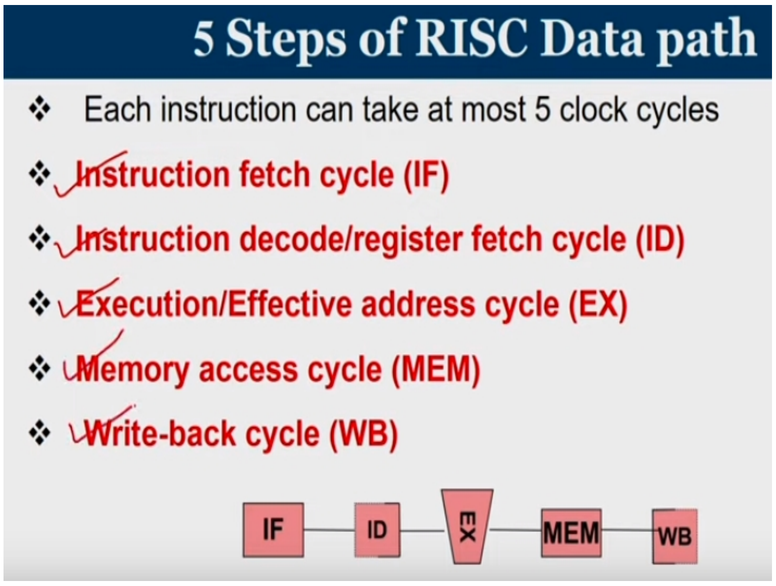
(Refer Slide Time: 01:08)



Pipelined RISC Data path

So this is the 5-stage risk data path that we have seen consisting of instruction fetch, instruction decode, execute, memory access, and write back.

(Refer Slide Time: 01:14)



5 Steps of RISC Data path

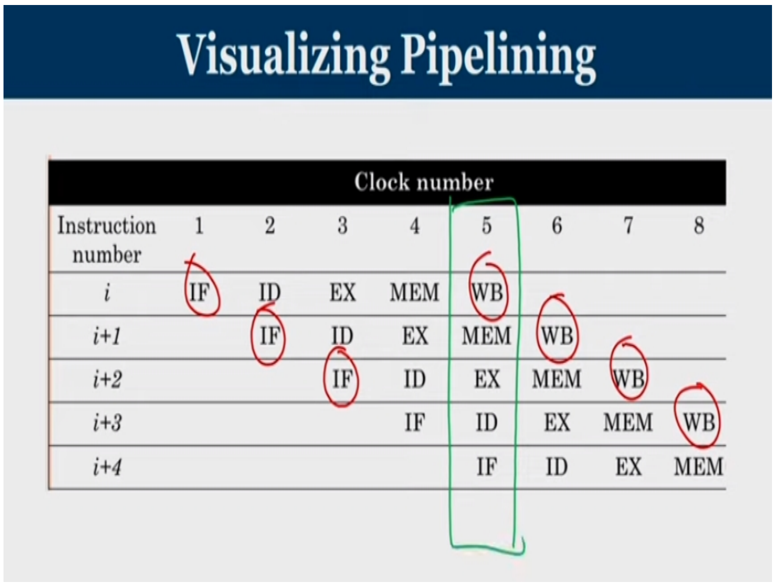
- Each instruction can take at most 5 clock cycles
- Instruction fetch cycle (IF)
- Instruction decode/register fetch cycle (ID)
- Execution/Effective address cycle (EX)
- Memory access cycle (MEM)
- Write-back cycle (WB)

IF
ID
EX
MEM
WB

And we will see what are the functionalities that are associated with this as far as instruction fetch is concerned. We will be working on the program counter value and based on PC, we are going to memory and then fetching the addressed word. In the decode we try to understand the operands and opcodes and if there are any register source operands then the contents of these registers are also been brought and kept in the pipeline register.

And when it comes to execution or effective address cycle, if it is an arithmetic operation the actual execution takes place. If it is a load or a store operation, effective address calculation takes place. And the fourth stage of MEM is used only for loads and stores wherein based upon the effective address that is calculated in the EX stage the memory is accessed. And the last stage is for writing the results into the registers.

(Refer Slide Time: 02:09)



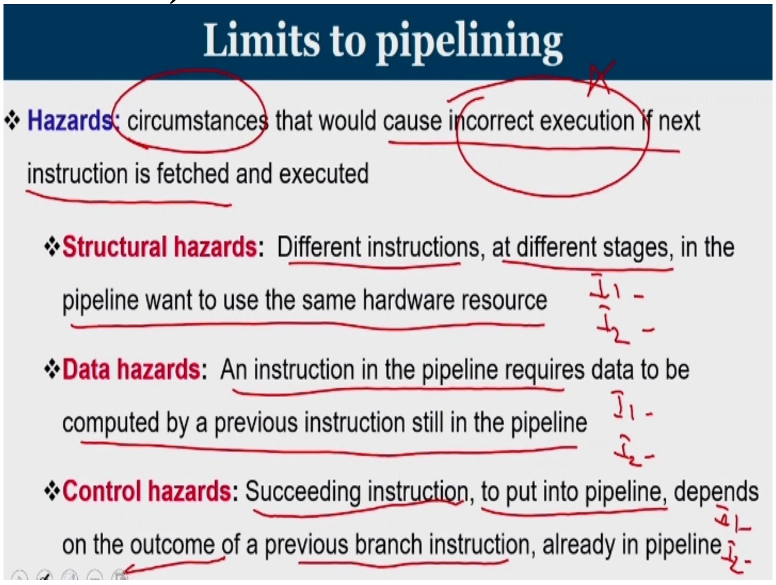
Visualizing Pipelining

	Clock number							
Instruction number	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				IF	ID	EX	MEM	WB
i+4					IF	ID	EX	MEM

And this is what we have seen every cycle we are going to start a new instruction even though the previous instruction is not fully over and after the initial latency for every cycle we can see that one-one instruction is getting completed. And if you look at any point, you can see that all the 5 stages of the pipeline are busy with 5 different instruction and this is the ideal case of visualizing a pipeline.

Now, let us try to understand when this pipeline is not going to work. So far what we have seen is the only ideal case, but we would not get such ideal cases always. So let us now practically move into the kind of hazards that we face, the kind of bottlenecks that we face in implementing the pipeline.

(Refer Slide Time: 02:58)



Limits to pipelining

- **Hazards**: circumstances that would cause incorrect execution if next instruction is fetched and executed
  - **Structural hazards**: Different instructions, at different stages, in the pipeline want to use the same hardware resource
  - **Data hazards**: An instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
  - **Control hazards**: Succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline

So hazards are circumstances that could cause an incorrect execution if the next instruction is fetched and executed. So hazards are actually circumstances. In certain scenarios, if you bring the next instruction in the very next clock cycle and trying to execute it, it can lead to an incorrect execution. So this we how to address. And basically there are three different types of hazards.

The first one is called structure hazard. The second one is called data hazard. And the third one is called control hazard. Structural hazard means when different instructions at different stages in the pipeline want to use the same hardware resource. So if you have two instruction I1 and I2, both are going to start the execution at different stage. But at one particular point, both these instruction want the same resource.That is shown as a structural hazard.
And in data hazard an instruction in the pipeline which is already in the pipeline requires data to be computed by a previous instruction still in the pipeline. So when you have two instruction I1 and I2, let us say the second instruction I2 wants a data which can be produced by I1, but I1 is also not yet complete, I1 is still in the pipeline. This basically happens due to data dependency.

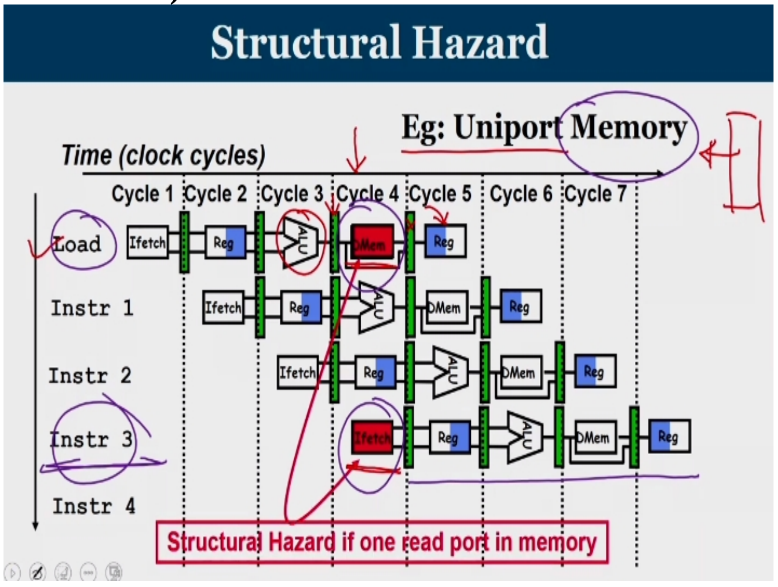
And the third one is control hazard. Succeeding instruction to put into the pipeline depends on the outcome of previous branch instruction already in the pipeline. We know that in every cycle, new instruction needs to be brought into the pipeline by the instruction fetch stage. So the new instruction to be brought into the pipeline actually depends on the outcome of a previous branch instruction that is already in the pipeline.

Here also we can have two scenarios, let us say I1 and I2. I will bring I2 if the branch in I1 comes out with one particular outcome. I am not supposed to bring I2 if the outcome of the branch in I1 is a different scenario. So whatever instruction that I am going to put into the pipeline, it depends on the outcome of a previous branch which is not yet completed in execution. So this scenario is known as control hazard.

So these scenarios, whenever we have a structural hazard or whenever we have a data hazard or if there is a control hazard, if you are trying to bring the next instruction in the pipeline and if you are going to execute that, then that will lead to an incorrect execution. So that is not what we want. Ideally, a pipeline should have additional support facilities to detect whether there is a hazard or not.

There should be a hazard detection circuit. The duty of the circuit is to find out whether bringing up the next instruction will cause an incorrect execution. If so what is the kind of hazard and what are the various steps that the control unit has to do in order to make sure that an incorrect execution like a hazard is not occurring?

(Refer Slide Time: 06:16)



Structural Hazard

Eg: Uniport Memory

Let us know try to understand more deeper about each of these hazard. The first one is known as a structural hazard. Let us take the consider of a uniport memory. So what do you mean by uniport memory? You have a memory which has only one read port. At any given point of time only one word can be read from the memory. So consider the case that we have a load instruction.

When we know that, in the case of an load instruction after the fetching and decoding is over, the address is computed in the ALU. Now you have the address here. Based on the address you are going to access the data memory and the word is accessed from data memory at the end of cycle 4 and you are going to write the result into the register.

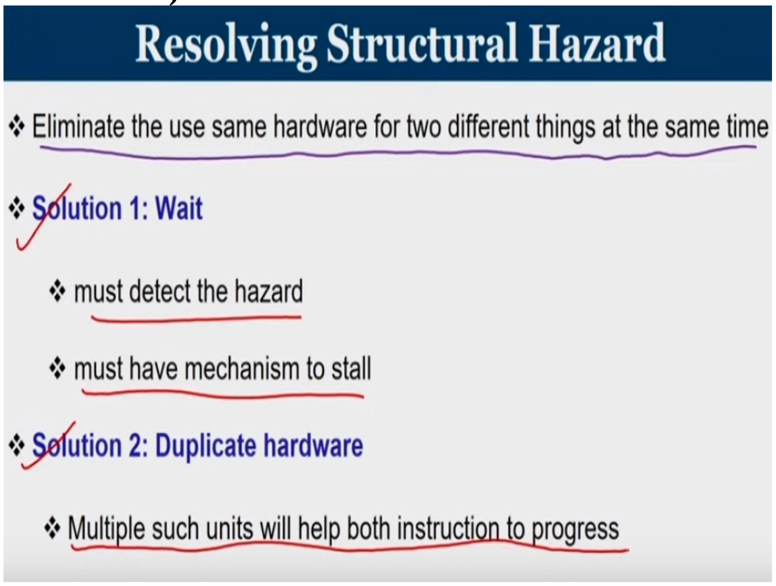
Consider the case that the second instruction is going to start when the load is in the decode stage. So this is how second instruction proceeds. Let us say there is a third instruction that happens. There is no problem with respect to the first and second instruction as far as this load is concerned, but the problem happens in the fourth instruction.

The fourth instruction is also trying to access memory in the instruction fetch state. So you give an address from the program counter and based upon that we how to fetch a word an instruction word from memory. So if you look at cycle number 4 the first load instruction is also trying to access the memory and the instruction I3 which happens fourth in the sequence, they are also trying to perform a reading operation from memory.

So you have a load operation from memory and you have an instruction read operation from memory. Both exactly happens at cycle number 4. Now what is hazard here? Attempting to use the same hardware. Attempting to use the same memory by two instruction in different stages of the pipeline exactly at the same time. Since I have only one ported memory, it is not possible to supply two words.

One for the load instruction and second one for instruction 3, which can solve this issue. This scenario is known as a structural hazard. So it is impossible. What are the solution to structural hazard. So the scenario is called structural hazard if one read port in the memory.

(Refer Slide Time: 08:45)



Resolving Structural Hazard

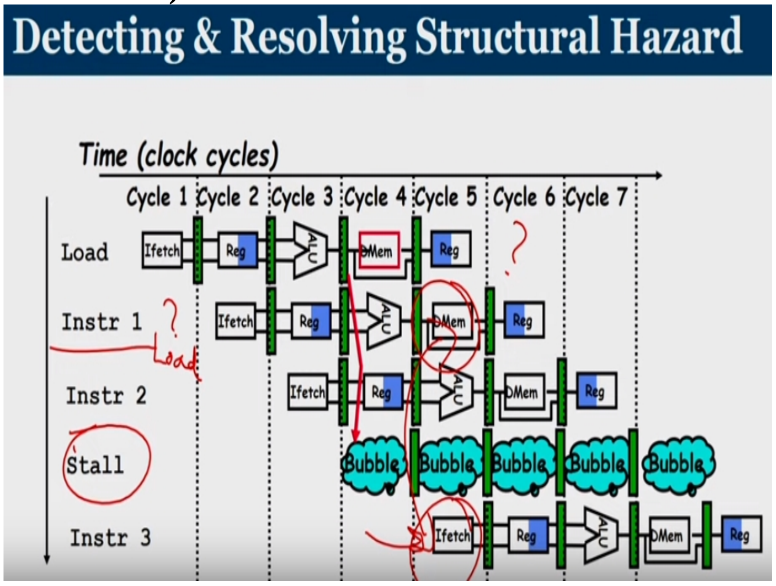
- Eliminate the use same hardware for two different things at the same time
- Solution 1: Wait
  - must detect the hazard
  - must have mechanism to stall
- Solution 2: Duplicate hardware
  - Multiple such units will help both instruction to progress

To resolve a structural hazard we how to eliminate the use of the same hardware for two different things exactly at the same time. So what are the solutions? The first solution is simple approach, wait. We must detect a scenario wherein two instructions are trying to access the same hardware. So detecting a hazard is occurring or not and then you should have a mechanism to stall one of the instruction.

So when you have two instructions that are going to access the same memory location or that are going to access the memory at the same time, then there should be a provision to detect such a kind of a scenario. Once we detect that it is going to be a structural hazard, we should try to delay one of the instruction in performing the operation. So detecting a hazard followed by a mechanism to stall.

The second approach is we should have a solution to duplicate the hardware. So have more number of units wherein this operation can be performed. Duplicate the hardware means multiple such unit will help both the instruction to make a progress.

(Refer Slide Time: 09:58)



Detecting & Resolving Structural Hazard

Let us try to see how the structural hazard we have discussed before in the context of a uniport memory should be resolved. So the same load instruction. We know that if the third instruction is going to run that is going to create a problem. So the mechanism is we detected there is going to be hazard and then a bubble, a no operation is been added.

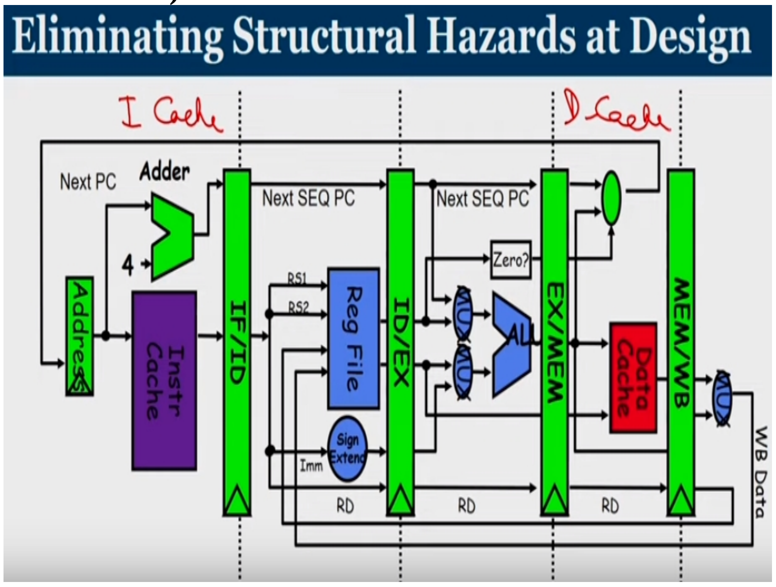
Bubbles are nothing but in the cycle number 4, no new instruction is been fixed. So naturally in cycle number 5, no instruction is decoded and subsequently no instruction is executed, memory access and write back. So there is one bubble that goes into it and then only your instruction 3 comes. So essentially what happens is your instruction 3 got shifted by one position.

This is the process of detecting a structural hazard and adding a new bubble or a stall in destruction sequence such that two instruction looking for the same hardware are not trying to access them at the same time. Now it will be interesting to note that you may get a doubt. What if this memory access that is happening for instruction 1 is going to have a conflict with the already shifted instruction fetch of instruction 3?

You may wonder whether this will create a hazard or not. This also will create a hazard if your instruction 1 is load or not. If instruction 1 is also a load operation, then its MEM stage is used for bringing an instruction word in memory. And we know that every instruction other than load and store the MEM stage is not used at all. If it is a normal ALU operation, then the MEM stage is not used.

And we have seen that even though the MEM stage is not used, it is not going to create any problem as far as the ADD instruction is concerned. Still the ADD instruction will take the normal 5 cycles. Since the MEM stage is not used for an arithmetic operation, this will not create an issue as far as a structural hazard is concerned.

(Refer Slide Time: 12:14)



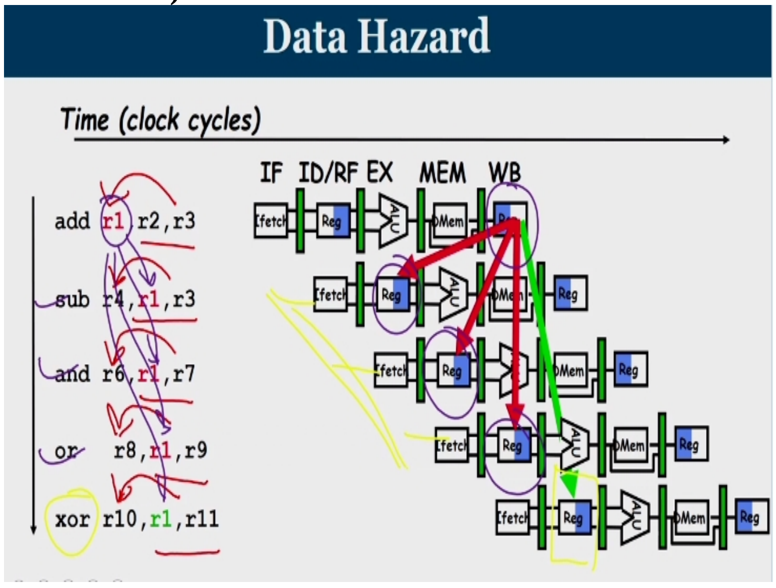
Eliminating Structural Hazards at Design

Another way, so the first approach that we have seen, the first approach that we have seen here is you try to detect the hazard and include a no operation or a bubble or a stall in the subsequent instruction, wherein simultaneous access of the same uniported memory is eliminated. But that can create delay in instructions. Another approach is like what we have already mentioned, we should duplicate the hardware.

So what was the memory contention problem? We are having one memory with one read port, either increase the number of read ports in memory such that one read port can be used always by the instruction fetch unit and the other read port can be always used by the data access unit in the pipeline. Another way of trying to looking at it have separate memory, one for exclusively for storing instruction and other one exclusively for storing the data.

So the instruction cache only will be accessed during the instruction fetch and data cache will be accessed only during the data access time. So eliminating the structural hazard, we can make use of two separate caches, this is what is known as I-cache and here we are having the D-cache. So I-cache and D-cache can be used parallelly. So all modern microprocessors have separate I and D cache, the basic purpose is to eliminate structural hazard that is happening.

(Refer Slide Time: 13:43)



Now let us try to see data hazards. Consider the instruction sequence that is given in the slide and all the instructions are of the format opcode, destination, operand 1, and operand 2. All of them are arithmetic or logic unit operations. So consider your ADD operations, so r2 and r3 has to be added into r1. And then the very next one is r1 and r3 has to be added into r4; r1 and r7 has to be and operated or you are going to perform a logical and operation to r 6.

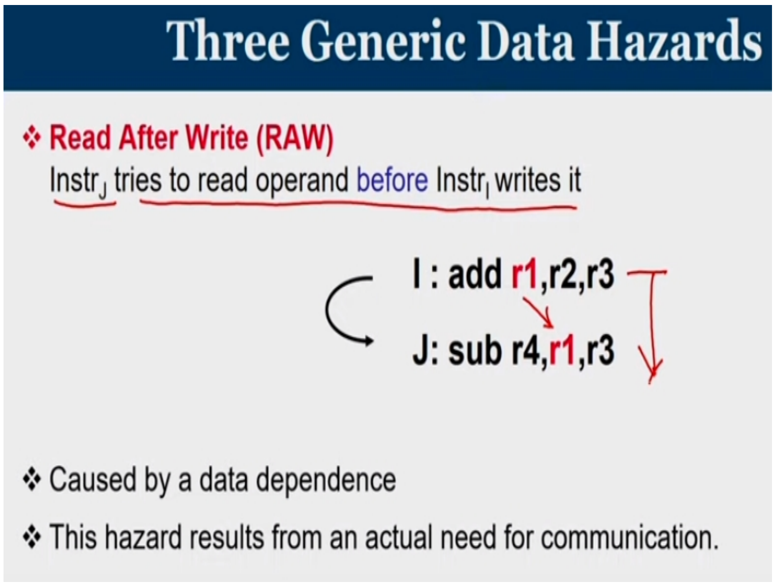
Again an r1 and r9 logical and result has to go to r8 and then you have an xor operation on r1 and r11 to store the result in r10. So the peculiarities across all these instructions, you can see that the result of the first instruction that is add r1, r2, r3 the result which is stored already in r1 is used by all others in these subsequent instructions. Then what is the problem in this?

We know that the result of r1 will be available only at the write-back stage. We are going to write the result only into write-back. Whereas, the subtraction instruction and the or instruction needs the value of r1 in their respective ID stages. So it is at this point, you want the value of r1 for subtraction. So the value that will be available only at the write-back stage is actually read prior to writing it.

Similarly, for the and operation also the value of r1 is needed here. For the or operation value of r1 is needed here. But when you take the xor operation, by this time already the value of r1 is written. So that the value of r 1 that is read by the xor instruction the fifth instruction will be correct. Whereas, all other previous instructions are going to read an older value of r1.

So if you read the older value of r1 and continue with the execution, it can lead to Incorrect result. So any circumstance that will lead to an incorrect execution of an instruction if these instructions are fetched, decoded and executed in its assigned time order, these three instructions, it will create a hazard. So this is what is known as a data hazard.

(Refer Slide Time: 16:20)


Three Generic Data Hazards

❖ Read After Write (RAW)
Instr$_J$ tries to read operand before Instr$_I$ 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.

So there are three generic data hazards that we are going to study today. The first and most common data hazard is known as read after write hazard. It is also known as RAW hazard. So instruction J tries to read an operand before instruction I writes on it. Consider these two instruction given in the slide. Instruction I perform

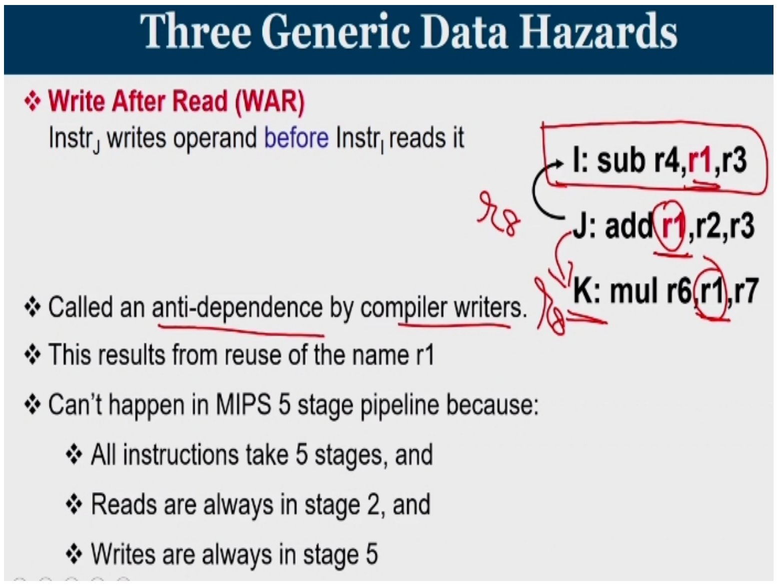
add r1, r2, and r3

Instruction J perform

sub r4, r1, and r3

And we know that there is a dependency that exist. The result of I instruction, Ith is needed for the Jth instruction as one of its source operand. This is known as data dependence. The data that instruction J want is dependent on the data that is produced by instruction I. This hazard results happens because of the fact that we need an actual communication from instruction I to instruction J because the result of that is produced by instruction I used to be used by instruction J.

(Refer Slide Time: 17:30)



Three Generic Data Hazards

- Write After Read (WAR)
  Instr$_J$ writes operand before Instr$_I$ reads it
- 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

I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
r8
r8

Now the second category of hazard is known as write after read hazard also known as WAR hazard. Look into the scenario instruction I is going to read from r1 and instruction J is going to write into r1 and instruction K will actually read that value of r1 which is written by J. So essentially, J is supposed to write a value into r1, which the K is going to use.

Having said this scenario, if instruction J writes its value into r1 before instruction I reads it. In the normal scenario, we feel that as per the program order I happens first followed by J and then K. So under ideal case, it will not create a problem. But think of a scenario if instruction I cannot read the value of r1. And the processor is permitting instruction J to go ahead with its work and complete it.

Assume that I reads the value of r1 after J writes on it. Then it lead to an incorrect execution. Actually, instruction I is supposed to read that value of r1 prior to the updation of r1 by J. So this scenario is known as anti-dependence by the compiler writers. This result from the reuse of the name r1. So during the time of translation of your high level language instruction to machine level language, then the compiler has to choose the appropriate register names for carrying out the execution.

Now how can compiler get impacted? In this scenario the compiler is using the register name r1. When the compiler is using the register name r1 that same register r1 will be used by the subsequent instruction which is dependent on it. Assume the case that if compiler has used the register name r8 for J and the same r8 is used by K, then whatever time your instruction I complete, it is never going to affect J or K or whatever time J is going to complete, it is never going to affect I.

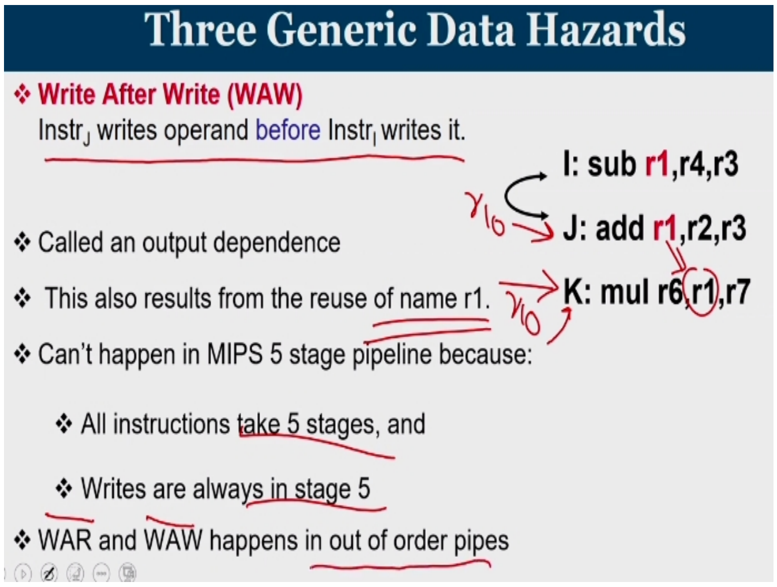
It is because of the reuse of the variable r1 or the register r1. Had compiler used a different name, this problem itself would not have exist. But in the previous case we have seen in the case of a RAW hazard, it is not the name of the register that is a problem. There exist a dependency between the second instruction with the first instruction. So whatever name that you use, the first instructions result is needed.

In this case there is not, this is not a case of transferring of result. It is a case of a name of a register that is a problem. So this is known as anti-dependence scenario that we have seen. Generally this kind of a scenario will not happen in a MIPS 5-stage pipeline, because all instruction take 5 stages. All the read happen in stage number 2 and all the write happen in stage number 5 and all instructions are executed in the order.

But down the course, after few couple of lectures, we see that all instructions are not taking 5, some instructions will be taking longer number of cycles and due to certain dependency issues, we are not having strictly in-order execution. Some of the instructions may get executed out of order.

So when we have such kind of scenario when you have longer pipelines, which takes more than 5 cycles and when we have out of order execution, then this condition whenever there are instructions which have WAR pattern, they are going to create issue.

(Refer Slide Time: 21:57)



Three Generic Data Hazards

- Write After Write (WAW)
  Instr$_J$ writes operand before Instr$_I$ 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
r10
r10

The third type of hazard is known as write after write hazard where instruction J writes to an operand before instruction I writes to it. So consider the case you have three instruction I, J, and K. I is going to write to r1. J is going to write to r1. And then K is going to read that value of r1 which was written by J. That is a logical order in the program.

Now think of a scenario where instruction I is going to write into r1 after instruction J is going to write into r1. So both I and J are writing into r1 and we expect that I completes followed by J. If there is a violation in the order in which the instructions are written, then whatever value K is reading that is wrong. So K is going to read the wrong value of r1. K is supposed to get the value of r1 from J, not from I.

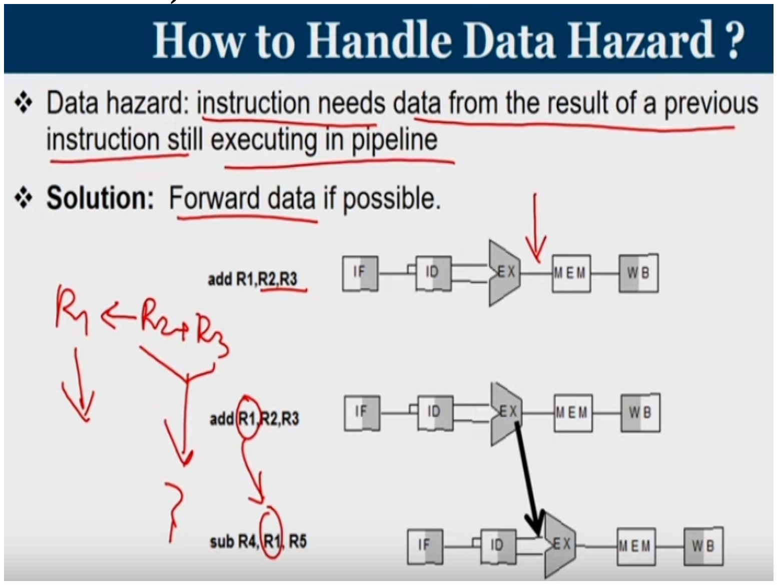
This is known as output dependence. This is also happening because of the reuse of the name r1. Had J used r 10 and the same r10 has been used by K also then whatever time I completes, it is no longer going to affect your J and K. This also happens because compiler is using the name r1. This also can happen in MIPS 5- stage pipeline because all instruction take 5, all the write happen at 5 and all instructions are executed in order.

But, the last two hazards that is WAR hazard and WAW hazard will happen in out of order pipelines. So far we are considering in order pipelines and all instructions we assume that they will take only 5 stages. As far as this assumption remains true, the

WAR hazard and WAW hazard will not happen. Only the RAW hazard we need to take care of.

For the time being while dealing with hazards we will try to address the RAW hazards, but later, once we learn longer pipelines and out of order execution, we will come back and revisit the concept of WAR and WAW hazards.

(Refer Slide Time: 24:00)



How to Handle Data Hazard ?

- Data hazard: instruction needs data from the result of a previous instruction still executing in pipeline
- **Solution:** Forward data if possible.

Now how to handle a data hazard? We know that data hazard is a scenario where an instruction needs data from the result of a previous instruction still executing in the pipeline. So the solution that is proposed is data forwarding. So consider the case we have

ADD R1, R2, R3

three instructions and we know that the execution of instruction

R2 + R3

will be completed after the EX stage.

Now consider you have one more instruction

sub R4, R1, R5

So clearly we know there is a data dependency. The destination register of ADD instruction is same as the source register of the subtraction instruction. Now it is not the content of R1 that is important to us. We know that the second instruction sub is going to start fetching when the first instruction is in the decode stage.

Now do you really want R1? If you carefully look into, it is not R1 that is of interest to the subtraction instruction, it is

$$R2 + R3$$

which is going to be written into R1. So if you carefully look into the ADD instruction it performs

$$R2 + R3$$

and the result is to be written to R1. It is that result which the subtraction instruction wants. So even though we are not getting the value of R1, but if you could get the result of

$$R2 + R3$$

and if that result can be forwarded to the subtraction instruction, then we can actually address this data hazard scenario.

And how is it possible or is it possible? It is possible if from the point the result is obtained to the point where the result is needed, if we can forward it from the output of the execution unit which will produce

$$R2 + R3$$

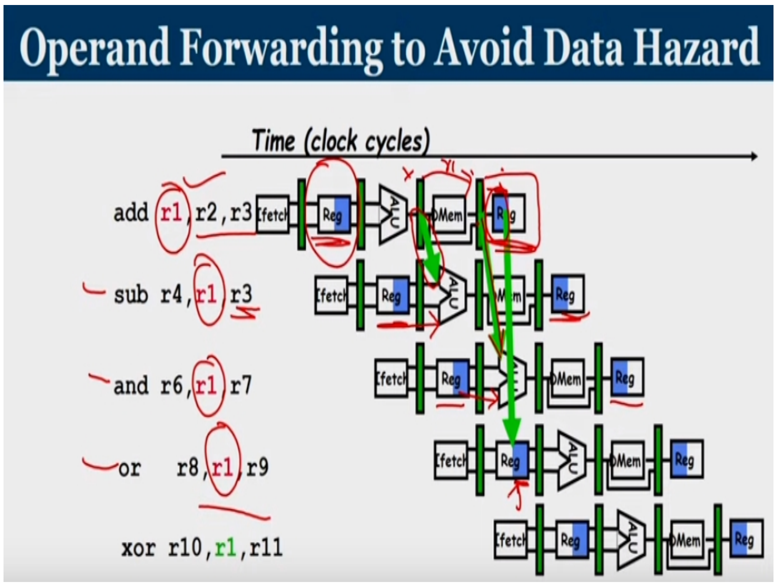
to the input of execution unit which wants this

$$R2 + R3$$

for subsequent computations, then that can actually solve the problem.

So the problem will be solved by having a forwarding mechanism from the execution unit. This technique of forwarding the operand as and when it is available into those execution units which further want them later in the execution is known as operand forwarding or data forwarding.

(Refer Slide Time: 26:31)



Operand Forwarding to Avoid Data Hazard

We will try to see how this operand forwarding works. So consider the case in which we are going to have these 5 instruction that we have seen earlier and or and xor instruction we can see. Wherever there is this red color that has been marked that indicates data dependency. Your result is going to be written in r1 and that result is been used by the subsequent 3 instructions.

In the time axis they all happens before the writing of r1. In the normal case of a pipeline, subtraction instruction read r1 before the value of r1 is written by the ADD instruction. Similarly and and or instructions also read the value of r1 before writing happens in the first instruction. Now you see, the value of

$$r2 + r3$$

will be available at the output of ALU and the value of r1 is needed for subtraction instruction prior to ALU start.

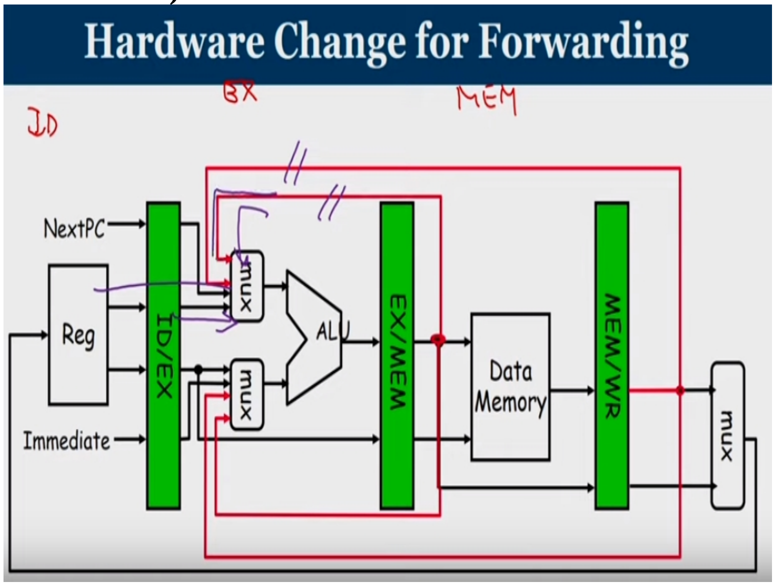
So the value of r3 will typically come from the register and the value of r1 will come from the output of ALU. That is called operand forwarding. Now in the second case, the value of r1 is needed. So you get the value of r7 here. The value of r1 will be forwarded from the MEM stage of the first instruction. So the value to be returned to the r1 it is now traveling from this EX/MEM pipeline to MEM writeback pipeline.

From the MEM writeback pipeline to the appropriate cycle this value get forwarded to the ALU. So you know that in the case of a subtraction instruction output of ALU should be given to input of ALU. In the case of and instruction output of MEM stage has to be given to the input of ALU. Whereas when it comes to the or instruction, you can see that it is a direct forwarding from register to register.

This, generally the convention is writing into the register generally happens in the fastest half of the clock cycle whereas reading from a register happens in the second half of a clock cycle. Since the register access is a very fast operation an entire clock cycle is not needed to access the registers. So in order to take care of the data reading and writing from a register, generally writing happens on the first fraction or first half of a clock cycle and reading happens on the second half of a clock cycle.

That is why the register access stage we can see a white color and the blue color always. So the blue color tells where the activity happens. As far as the ID stage is happened, is concerned, then the blue color that is reading of the register happens in second half. And as far as the writing is concerned, it will happen in the first half. So if in first half if you perform writing into r1 then the **or** instruction can read it in the second half. This is the way how you are going to deal with operand forwarding.

(Refer Slide Time: 29:48)



Hardware Change for Forwarding

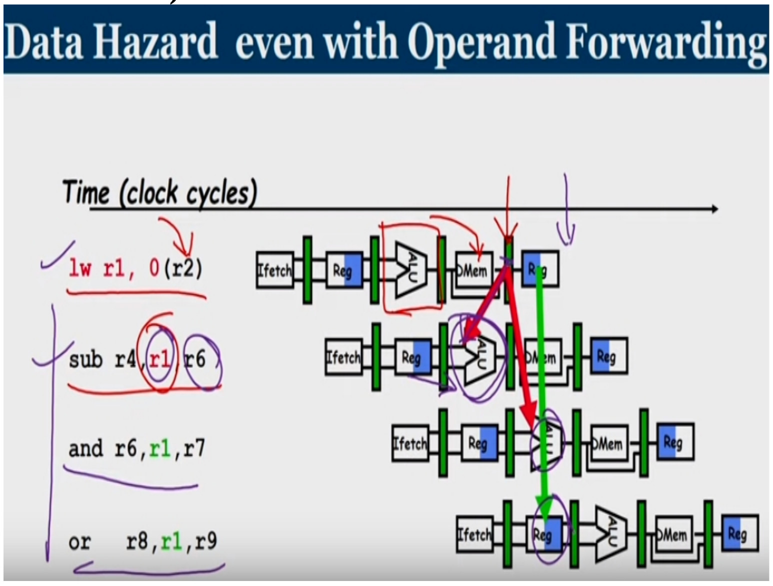
Now, let us look into how operand forwarding circuit really behaves. So this is the ID stage what we are talking about and we have the EX stage. This is the EX stage and this is the MEM stage. We have seen that the output value of ALU should be forwarded to the input of ALU. So there should be a path that connects from the output of ALU that is from the EX/MEM register which is going into this.

So this multiplexer will take care of whether the value should come from this pipeline register or the value should come from this arrow. So to make it more clear, let me draw with a different color. Either the value can come from the ID/EX register or the value can come from the EX/MEM register. And we have seen one more scenario in which the output of MEM writeback register has to be given to the input of ALU.

So in certain cases, the forwarded data will come through this channel and in certain cases it will come through this channel. So in the input of ALU we have a multiplexer that depending upon the instruction from the control unit will take the appropriate input to the ALU. Either the input is from the register file or the input is from output of ALU that is EX/MEM register that is what is this line.

Or it can be from the output of MEM writeback register, that is the third line. This is the operand forwarding circuit.

(Refer Slide Time: 31:22)



Data Hazard even with Operand Forwarding

So operand forwarding is a important concept that can actually reduce hazards, especially RAW hazards between a pair of ALU instruction. So as long as you have only ALU instructions operating on various operands and if there is a dependency between them by the operand forwarding technique we have just explored, it can be completely eliminated. So there is no hazards at all.

But we know that inside a program, it is not only ALU instructions that we are going to deal. We have heterogeneous instruction. It can be load store instruction, branch instruction and other control transfer instruction apart from ALU. So look into this example what you see in the slide where we have a node instruction followed by a sequence of arithmetic logic instructions.

We know that operand forwarding is allowed as and when the data is ready the data is forwarded from one pipeline register to other pipeline register and using appropriate multiplexers we can take the data. To try to understand what happens in this scenario where we have a

load r1, 0(r2)

We know that the address from which the loading happened is content of r2.

So you how to access r2, add zero to it, the content of r2 and that will give you the effective address. And this effective address calculation happens in the ALU stage. Once you get the effective address then we are going to access the data memory and take the data that is to be written. So at this point, the data is ready from memory. Now consider the second instruction actually wants the content of r1 or it wants the result that has been produced by the previous load instruction.

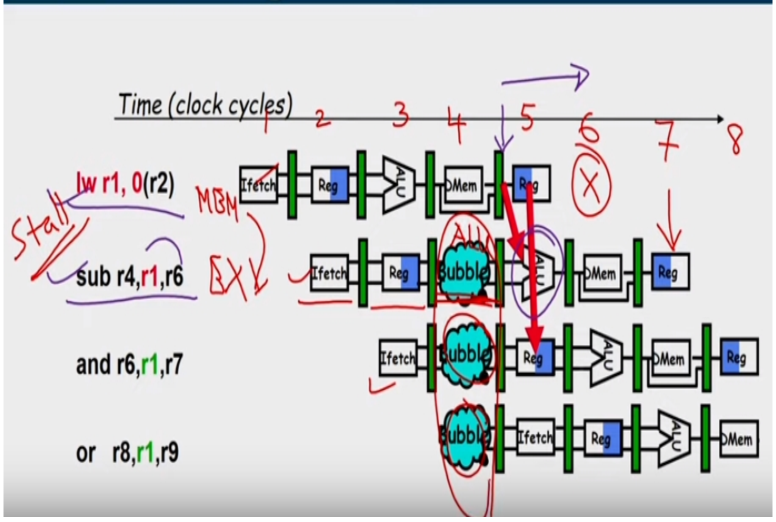
The second instruction will perform its ALU operation here but the data that it wants, one data is r6, other data is r1; r6 will come from the register file. The value to be written to r1, in the ideal case it is available only after this point. But if you use operand forwarding, from this pipeline register the data is available. But you actually wanted one cycle before. That is why it is shown in the red color.

So even with operand forwarding, the subtraction instruction will not be able to get the value in the ALU so that it can complete its operation in its assigned time slot. Whereas this and instruction and or instruction with proper operand forwarding, they are going to get the corresponding operands.

So what we have seen here is whenever we have an ALU instruction, which immediately follows a load instruction and if there is a dependency on that load instruction, then even with operand forwarding we cannot resolve the hazard. So what is the solution?

(Refer Slide Time: 34:16)

Resolving the Load-ALU Hazard



The only solution is for this load instruction, that operand is available at this point. For the subtraction instruction, the execution of r1 + r3 can happen only after this point. So the ALU operation has to happen only after the MEM stage of the load instruction is over. I will repeat once again, the ALU stage or the EX stage of subtraction instruction can happen only after the MEM stage of load is over.

So that is the tricky condition here. So when that can happen, that means, if I am going to start fetching here and decoding here, ideally the ALU should come here. But now the condition tells that the execution stage of subtraction instruction can happen only after the MEM stage of load word. That naturally means that my execution cannot happen at this point. I am going to insert a bubble.

That means the subtraction instruction will continue in its ID stage or the register reading stage for one more cycle. That means, I cannot perform the decoding for the and instruction so that the bubble will continue. So once you insert a bubble then the rest of the process will move smoothly. So how are we going to deal with this?

Whenever we have an ALU instruction that is immediately following a load instruction and if there exist a dependency between them then there is a stall that is compulsorily needed between the load word and the sub. So what is the impact of the stall, The first instruction will complete at clock cycle number 5. So if we look at 1, 2, 3, 4, and 5 ideally with operand forwarding we have seen the previous example that in clock cycle number 6, the second instruction will complete.

But here we do not have a case in clock cycle number 6, no instruction is reaching the write-back. The next instruction is getting complete only at clock cycle number 7 and thereafter in every clock cycle one-one instruction is getting over. So after the first instruction, there is a stall. That means, one instruction will not be getting over in the subsequent clock cycle.

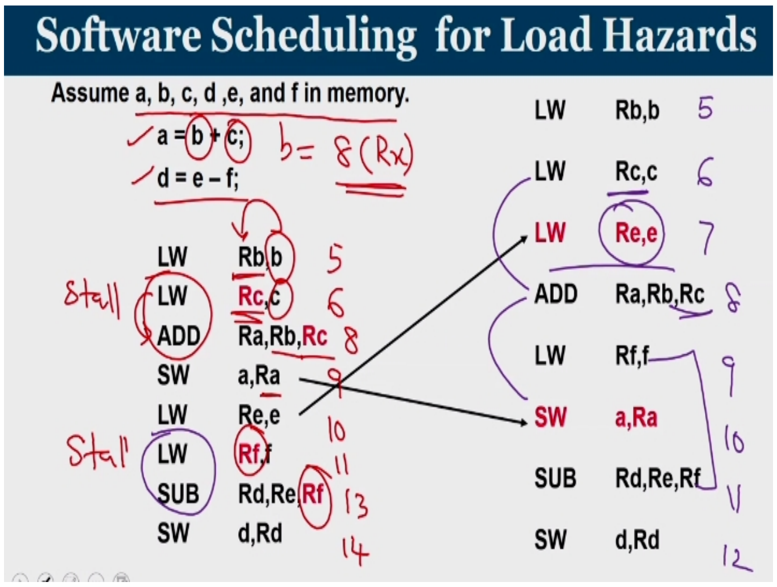
That sift in completion of the instruction is happening because of the bubble that is been inserted here. So the conclusion is an arithmetic instruction just after a load instruction will incur a stall between them in order to make sure that an incorrect execution is not happening. Now, we will look into the software scheduling. It is yet another approach to handle the hazards.

We all know that when you write a high level language program for a given task, this high level language is been converted to machine language by a compiler and then it is been stored in memory. So looking at the register names the compiler can easily understand whether there exist a RAW hazard between them. If a compiler is going to generate a code for a load instruction and immediately after if there is some arithmetic operation that is to be performed on the loaded value.

And if these instructions are adjacent to each other, then at the time of code generation itself, the compiler can understand once this instruction goes to hardware there has to be a stall. Now a compiler if it is intelligent enough can understand this kind of scenarios by finding out register dependencies. And what do you mean by the register dependency?

Trying to detect dependency between instruction by looking similarity of the registers that are being used in them. If compiler can slightly rearrange the instruction such that a load instruction and the corresponding ALU instruction that should have been followed this load if they are sufficiently separated, then we can actually solve the scenario.

(Refer Slide Time: 38:39)



Software Scheduling for Load Hazards

Assume a, b, c, d ,e, and f in memory.

a = b + c;

d = e – f;

b= 8(Rx)

Stall

Stall

```
LW    Rb,b        5
LW    Rc,c        6
ADD   Ra,Rb,Rc    8
SW    a,Ra        9
LW    Re,e        10
LW    Rf,f        11
SUB   Rd,Re,Rf    13
SW    d,Rd        14
```

```
LW    Rb,b        5
LW    Rc,c        6
LW    Re,e        7
ADD   Ra,Rb,Rc    8
LW    Rf,f        9
SW    a,Ra        10
SUB   Rd,Re,Rf    11
SW    d,Rd        12
```

So consider the case that we are going to perform a simple operation

a = b + c,

d = e – f

Let us say a, b, c, d, e, and f are memory locations. So I am trying to list out what should be a rough MIPS code for carrying out this program. So first since it is a load store architecture, prior to performing the operation, the operands have to be kept in registers. Let us call the register Rb wherein the value of the memory location b.

Let us say b correspond to some 8(Rx). So this 8(Rx) I am going to call it as b. So whatever is b, the displacement and the name of the register is not relevant, whatever it is, that has to be loaded into register Rb. And then I am going to load the register value Rc with the corresponding value of c. So now you can see that the value of b and c is now available in registers Rb and Rc. Now I have to add them.

So I am going to add Ra, Ra is my new register. Rb and Rc values are added and then put it in a. Now we know that it is a load instruction and immediately after that we have an add instruction. So naturally there is going to be a stall here. That is why stall for because of the data dependency in the name of the register Rc. So and then we are going to store the value whatever is the value that is available at the end of ALU operation we store it back into memory location a.

The same thing can be continued with Re and Rf. In order to carry out the remaining operation e – f, load the value in Re, load the value in Rf, subtract and then you store. Here also you see a dependency between the register Rf. So for this particular code, we are going to have stalls. So here also we will encounter a stall.

So if you look into by what time we are going to complete the first instruction, the first instruction ideally should complete by clock cycle number 5, second will complete at clock cycle number 6. Between second and third there is a stall. So nothing will complete in clock cycle number 7. So it is completing at clock cycle number 8, then 9, 10, 11.

In 12 nothing is going to complete because this load and sub is going to have a dependency. So it is 13 and the store will get over at 14. Now if you look into can I get rid of the stalls that is happening? If you wanted to get rid of the stall then this load and add that we have seen it here, they should not come together. The load and add should be separated. Similarly, this load and sub also should need to be separated.

So if compiler knows this is the code that will be generated normally can compiler rearrange the code, the same load word is there. Now what I do is I am going to put up this instruction

LW Re, e

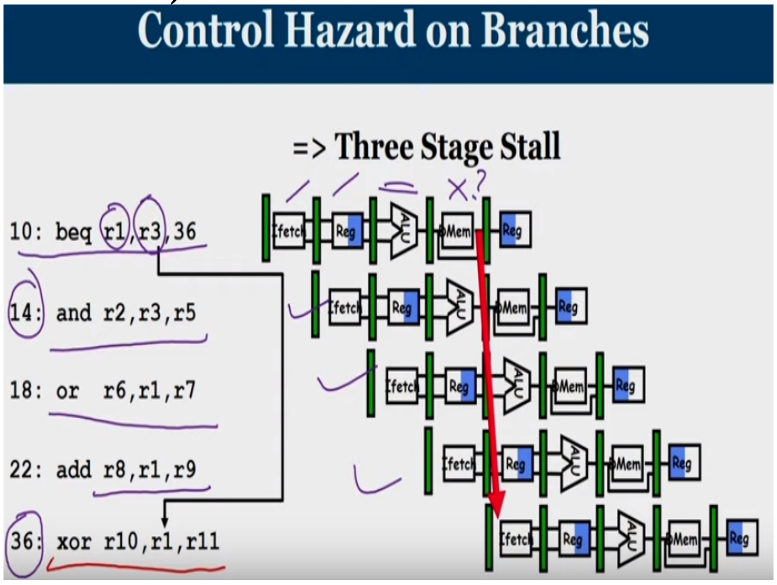
It is a new load which was initially down the instruction sequence coming up and then add. So these are the dependent instruction.

Now they are separated by inserting another instruction which will never change the meaning of the code. So it is only working on e and Re. By this time value in Rc is complete and add can perform. So the dependency that there was existing between load and add is now effectively addressed. Then we have this one. In the meantime I am completing the store operation also.

So there was initially a dependency between this load and the sub. They have to be separated by at least one instruction. So you meaningfully put the store instruction there. And then you are going to complete the store operation. Now if you look into the first instruction complete by 5, 6, 7, 8, 9, 10, 11, and 12. So previously there were 2 stalls. Now the whole of the stalls is been eliminated.

So in this way compiler can actually help in rearranging the instruction by seeing the register names that are being used in the instruction. So wherever there is a RAW dependency and if it is an arithmetic instruction immediately after a load instruction, and any dependency between them it can be easy addressed. So a little bit of hardware knowledge if it is been supplied to the compiler, compiler can actually reschedule the code and it is known as software scheduling for load hazards.

(Refer Slide Time: 43:11)



Control Hazard on Branches

=> Three Stage Stall

```
10: beq r1,r3,36
14: and r2,r3,r5
18: or  r6,r1,r7
22: add r8,r1,r9
36: xor r10,r1,r11
```

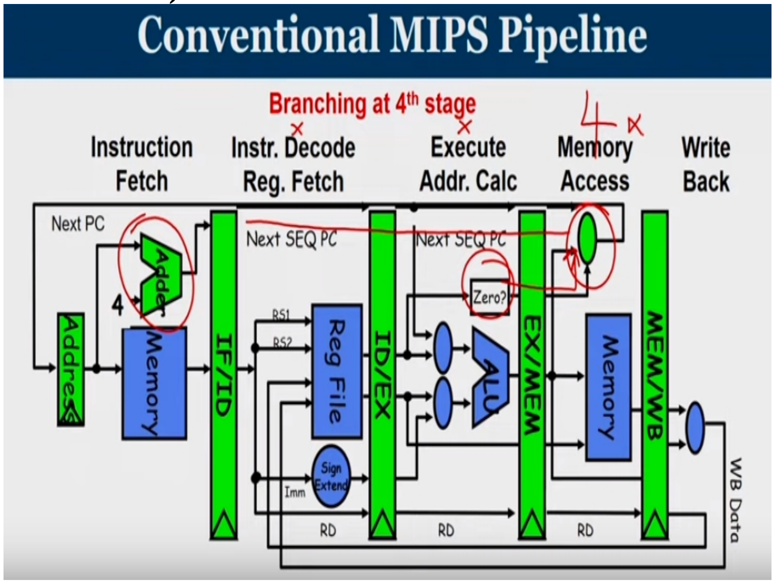
Let us now deal with the third category of hazard. It is known as controller hazard on branches. So consider the MIPS instruction branch is equal to, the meaning is if the value of r1 and r3 are equal, then I have to jump into line number 36 or address 36. If they are not equal, then I will go into 14. Now look at the instruction sequence, You are going to fetch the instruction, decode.

During decoding the value of r1 and r3 are red and they are now subtracted. So we will come to know whether this branch will take place only at the MEM stage because it is essentially comparing two registers and this comparison happens only down the line the pipeline.

By the time you come to know the outcome of the branch or by the time the comparison result of r1 and r3 is ready already these instructions have entered the pipeline; instruction at 14, instruction at 14, 18 and 22 have reached the pipeline. Let us assume that this branch is taking. That means content of r 2 and r 3 are matching. So the next instruction to be fetched and executed is in line number 36.

So if this is to be executed already three wrong instructions have entered the pipeline. So if you permit these instruction to continue execution that will lead to an incorrect state, that is a hazard. Such a kind of a hazard is known as a control hazard. So we have to finalize this.

(Refer Slide Time: 44:43)



Conventional MIPS Pipeline

So normally your branching happens in the fourth stage. Let me address this. The normal value of PC + 4 that is available here. And here we are seeing whether if you subtract two register value, the resultant is zero and that is going to tell you whether a branch will happen or not. So branches typically happens from flag registers or flag values, equality checking of two registers.

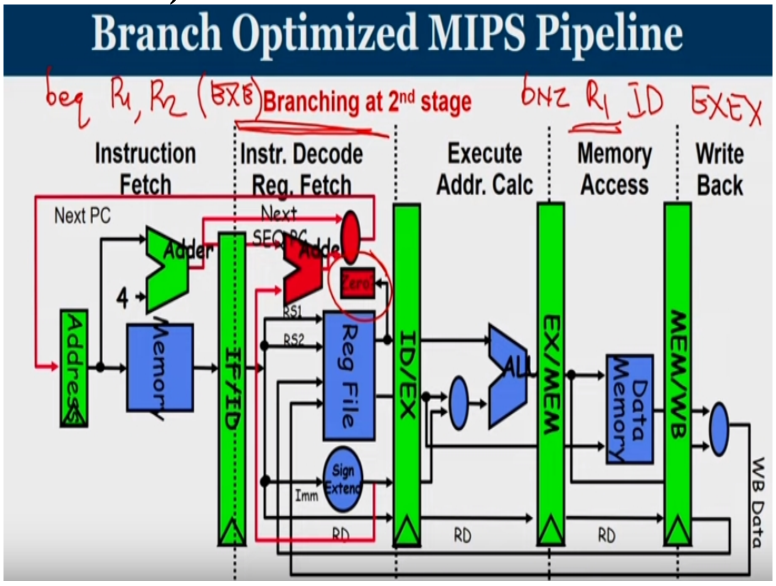
So the outcome of branch is known only at this point. Should I continue with the current program counter value or should I move into or jump into a target location. By the time, since this happens in the fourth stage, already three instructions will be entering into the pipeline. When it is in the decode stage, one instruction will enter. When it is in the execution stage, the second instruction will enter.

And once it is in the memory access stage, the third instruction will enter. So if at all whatever entered is wrong, that means the branch is taking, you are going to jump into target location pipeline should have facilities to undo whatever task that has been done. So you might have already bringing some instruction into the pipeline you have to flush it out. It requires some extra hardware unit and some mechanism to detect whether branches happened or not.

Let us assume that the branch never happened even though you come to know that 4 stage in the pipeline in the MEM stage you come to know the branch is not happening we are in a safe state. Even though without knowing that we have brought three instructions, but they are the instruction that are to be brought. So there is nothing wrong in what we have brought. Why it is happening? Because the decision of a branch is very late in the pipeline.

Can you do branch handling? Or can I know the outcome of a branch little early in the pipeline.

(Refer Slide Time: 46:34)



Branch Optimized MIPS Pipeline

So in modern architecture the branches are happening in the second stage and that is been done by test of zero register. So if you are going to use an instruction branch is equal to R1, R2, then it involves reading the value of R1, reading the value of R2, subtracting them and then look at the sign of the resultant. That can happen only at the EXE stage.

Let us say apart from that, if I am going to rephrase my branch instruction as branch is nonzero of R1. So my branch instruction is now not a comparison of two registers, I am just checking whether a register value is equal to 0 or not. That is called test on zero. This test on zero, I can do it in the ID stage itself. I do not want an EXE unit to carry that. That is called the zero unit that we are seeing in the pipeline.

So the PC + 4 that is our normal one will be available and test of zero can be done as early as second stage because it is reading a register value. Then and there itself I can check whether all the bits of the value that is been red is 0 or not. So if we properly manipulate branch instructions or if you define what are the operands of a branch instruction then those operations can be done early in the pipeline.

So equality check of two registers cannot be resolved at the ID stage, because to check whether two registers are equal or not we have to subtract these register contents and then look for the sign of the resultant. It can happen only at the third stage. But if you are dealing with only one operand, let us say R1 and checking whether

$$R1=0$$
or
$$R1!=0$$

and based upon that outcome if a branch is going to be taken or not taken, then the outcome of a branch can be known as early as the second stage.

We will deal with more of branches in the subsequent lectures. So today, we have learned what is a hazard. We have three categories of hazards, the structural hazard happening due to usage of the same resource at the same time. Data hazard happening due to dependency between instructions and control hazard happening due to bringing of wrong instructions before knowing the outcome of the previous branch.

And then we have seen what are the mechanisms that we use in order to take care of these hazards. So that completes today's lecture. Thank you.