

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

Lecture - 27
Principles of Pipelining Part II

Welcome back. In the previous lecture we talked about the different kinds of hazards. So, in this lecture we shall look at solutions both in software as well as in hardware.

(Refer Slide Time: 00:25)

Solutions in Software

- * Data hazards
- * Insert *nop* instructions, reorder code

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

↓

```
[1]: add r1, r2, r3 ← done
[2]: nop
[3]: nop
[4]: nop
[5]: sub r3, r1, r4
```

McGraw Hill Education

34

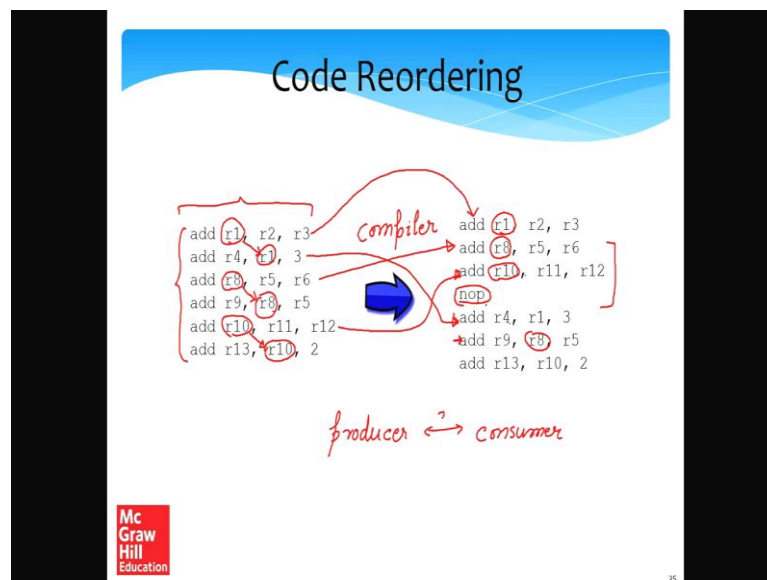
So, let us first consider an example. So, let us look at these 2 instructions, I am sorry over here. So, in this instruction as you see the first instruction writes to register r1 and the second instruction reads from register r1. So, there is a raw dependence, which is a problem in our pipeline, the reason it is a problem is because when instruction 2 needs a data for register r1 instruction one would not have written it. So, as we have seen in the past few slides it is necessary to have 3 instructions between instruction 1 and 2 such that the information flows.

So, how do we add 3 instructions well this is where the nop instruction comes handy. So, recall that we had introduced a nop or the no operation instruction way back in chapter 3. So, this was one instruction which does absolutely nothing, but now we see that it has some role to play in the sense that it ensures that a code in a pipeline executes correctly. So, let us see how. So, when the second instruction is in stage 2 which is the OF stage

the nop instruction that is right before it is in the EX stage the one that is before that is in the mth stage and the first instruction is done. So, which means that when the sub instruction tries to read the value of register r1 it will get the value of register r1 from the register 5, this is exactly what we wanted and one way of achieving this is by adding 3 instructions between the add instruction and the sub instruction.

In this case the 3 instructions are nop or no operation instructions. So, this ensures that our pipeline works correctly, but this is a bad solution the reason it is a bad solution is because we are essentially wasting the cycles and we are not doing anything.

(Refer Slide Time: 03:06)



So, better way would be that if we consider a larger piece of code we can reorder the code such that same constraint, such that the same constraint technique can be applied, but we can minimize the number of nop instructions that are added. So, let us consider this piece of code. So, we write to register r 1, and at the same time we read from register r 1. Similarly, over here we write to register r 8 and we read from register r 8. So, that is a raw dependence. And then we again write to register r 10 and we read from r 10.

So, these are the raw dependences that are there. And in a normal pipeline unless something is done this code will not be able to execute correctly. As a result, what we do is that we reorder the code we can do that. So, basically who reorders the code in principle the compiler can do that. So, the compiler can reorder the codes such that there are 3 instructions between every producer and every consumer instruction. So, this can

definitely be done or let us say the programmer can also do that. If let us say the code on the left hand side has been written by the programmer, then the programmer can take a look at the code and modify it appropriately, but it is not a good idea to put the owners off ensuring correctness on the programmer such kind of transformation should always be done at an automated level at the level of the compiler.

So, what we do is that we the first instruction is still the add instruction we write to r 1, but what we see is that the third instruction can be brought over here and we are not violating any dependence, because this instruction has separate has a different set of operands and this instruction has a different set of operands. So, we can happily bring this instruction over here. Similarly, we can bring this instruction which writes to r 10 over here no dependencies are violated. After that we can bring the consuming instruction which consumes the value of r1 over here, but we still need to add one nop instruction to ensure that there are 3 instructions between the producer of r1 and the consumer of r 1.

Now, let us consider the next instruction which is a producer of r 8, we can have this instruction over here which is the consumer of r 8. So, we see that there are also 3 instructions between them 1 2 and 3. So, essentially this instruction will get the right value of r 8 from the register point after this we have one more instruction add r 13 and r 2. So, here again it will get the right value of r 10 because there are 3 instructions between the producer and the consumer. So, what we see is that we can have an algorithm which can be implemented and executed by the compiler. So, the compiler can reorder the code without affecting the correctness of the program as we have done here, and it is still it still might be required to insert a few nops, where we want to minimize the number of nops because the nop mean that nothing is being done in that particular stage.

So, we should try to minimize it, and there are lots of algorithms in the literature that try to reorder code by minimizing the number of nops that are required to be inserted.

(Refer Slide Time: 07:21)

The slide is titled "Control Hazards" and contains the following text:

- * **Trivial Solution** : Add two nop instructions after every branch
- * **Better solution** :
 - * Assume that the two instructions fetched after a branch are **valid** instructions
 - * These instructions are said to be in the **delay slots**
 - * Such a branch is known as a **delayed branch**

Handwritten red annotations on the slide include "Q", "EX", and "nop" with arrows pointing to the "EX" stage and the "Better solution" text. A small "Mc Graw Hill Education" logo is in the bottom left, and a small "36" is in the bottom right.

So, we have some solutions to take care of data hazards, let us now try to take a look at control hazards. So, the trivial solution in this case would be to add 2 nop instructions after every branch. So, what we have seen is that the control hazard is resolved in the EX stage. So, there might be still 2 instructions in the previous 2 stages which were fetched incorrectly. So, one idea would be that after every branch we just sent 2 nop instructions and then send the right instruction which is the instruction at the branch target or the instruction just after the branch instruction.

But here is a better solution. So, let us assume that the 2 instructions fetched after a branch are valid instructions. So, these instructions are said to be in the delay slots of a branch. So, the delay slots will essentially in our case in our pipeline be the 2 cycles after the branch. So, let us try to do something, let us assume that the branch instruction is there in the program at this point let us try to bring 2 instructions before it, and insert them after the branch in the delay slots. So, in this case we are not actually paying a performance penalty, the reason being that we still are executing 2 valid instructions which are independent of the branch they would have otherwise executed before the branch. So, we are just bringing them after the branch.

If we do that what happens it that we do not have to cancel any instructions or insert any nops. So, we will not waste any time. So, this kind of a branch is called delayed branch. So, let us discuss the idea of delayed branches in some more details.

(Refer Slide Time: 09:21)

Example with 2 Delay Slots

```
Original Code:
{
  add r1, r2, r3
  add r4, r5, r6
}
beq, foo
add r8, r9, r10

Reordered Code:
beq, foo
add r1, r2, r3
add r4, r5, r6
add r8, r9, r10
```

* The compiler transfers instructions before the branch to the delay slots.

* If it cannot find 2 valid instructions, it inserts nops.

Mc Graw Hill Education

37

So, let us consider a code snippet of this kind. So, here we have a branch to the dot foo level and we have 2 instructions before it add r1 r 2 r 3, and add r 4 r 5 r 6. So, as we see these 2 instructions the 2 add instructions are per say independent of the branch right. So, they do not determine the target of the branch neither the direction of the branch. So, what we can do is that we can bring these 2 instructions to after the branch, and in vision a processor which treats a branch like a normal instruction and treats the instructions after the branch as normal instructions as well and a smart compiler essentially moves instructions which are before the branch to instructions that are after the branch.

So, they will be in the delay slots because when these instructions in their IF FO stages we would not know the direction outcome of this branch. So, they will still execute and they will execute correctly. So, the compilers job is to transfer instructions from before the branch to these delay slots, if it cannot find 2 such valid instructions than it need to insert nops. So, hopefully the idea is clear. So, the idea here is that we are not wasting any time after the branch, and instead of d this can beq bgt does not matter, let us assume it is beq it does not matter because anyway if the branch sees if the branch is not taken that is no problem, if the branch is taken these 2 instruction would have executed anyway, and they still are executing because they are still in a delay slots.

So, we realize the direction of the branch only in the E X stage by that time these 2 instructions there in the pipeline. So, they can be allowed to continue. So, such kind of a

processor which supports delayed branches are you know they were there in the earlier days and these processors suppose are you know termed as processor that support delayed branches, and the slots after a branch in which you can fetch such kind of instructions are called delay slots.

(Refer Slide Time: 11:54)

The slide is titled "Outline" and contains a list of topics. The topics are: Overview of Pipelining, A Pipelined Data Path, Pipeline Hazards, Pipeline with Interlocks, Forwarding, Performance Metrics, and Interrupts/ Exceptions. There are handwritten notes in red ink: "(bubble + stall)" next to "Pipeline with Interlocks", "(forwarding -> avoids bubble)" next to "Forwarding", and a blue arrow pointing left towards "Forwarding". The Mc Graw Hill Education logo is in the bottom left corner, and the number 55 is in the bottom right corner.

- * Overview of Pipelining
- * A Pipelined Data Path
- * Pipeline Hazards
- * Pipeline with Interlocks (bubble + stall)
- * Forwarding (forwarding -> avoids bubble)
- * Performance Metrics
- * Interrupts/ Exceptions

So, where are we right now we have taken a brief look at pipelining? We have discussed a pipeline data path; we have looked at pipeline hazards. Let me go back one more cycle one more slide we have also discussed 2 software methods of fixing problems at pipeline, one of these was reordering code and adding nop instructions such that what is the constrain between a producer instruction, and a consumer instruction there are at least 3 instructions. So, we need to insure that that has been done.

In the other case we have assumed the existence of delayed branching schemes. So, in this case what happens is that when the branch is there in the EX stage right let us let it be any branch instruction we have already fetched other the subsequent 2 instructions into the OF and IF stages see now the branch is taken these instructions would have been deemed to have fetched incorrectly. So, they would have been in wrong path essentially. So, instead of touching them what we can do is we can get to independent instructions before the branch, and put them in these delay slots which is essentially this slot and this slot right. So, we can put them in these delay slots.

So, since they would have executed any way they will still execute and they will execute correctly, and after the branch branches direction have been resolved we can fetch the right instruction, either from the branch target or the instruction which is just after the branch which is in this instruction. So, this instruction in this case will be fetched. So, there is no problem correctness is not been hampered.

So, these are 2 very common mechanisms, the code reordering and adding nops as well as delayed branches are very common mechanisms of solving issues with pipelines. So, we shall now look at 2 solutions in hardware. So, one is a pipeline with interlocks and other is a pipeline with forwarding. So, we will look at these 2 techniques because this was a software technique, and software is good, but a problem is that the compiler needs to be aware of all the details of the hardware. This is not always possible because it means that for every new kind of pipeline that we have we need to have a different and separate compiler. And Secondly, a one kind of code which runs on one processor will not run another processor.

So, to alleviate some of the some of these problems let us assume that the software does not do anything at all and everything is taken care off in the hardware.

(Refer Slide Time: 15:05)

The slide features a blue header with the title "Why interlocks?". Below the title, there is a list of four bullet points, each starting with an asterisk. The text in the bullet points uses color-coding: "trust" and "interlock" are in red, "compiler" is in red, "nop" is in blue, "Compilers" is in red, "pipeline" is in blue, and "compiler" is in red. The McGraw Hill Education logo is located in the bottom left corner of the slide content area. A small number "39" is visible in the bottom right corner of the slide.

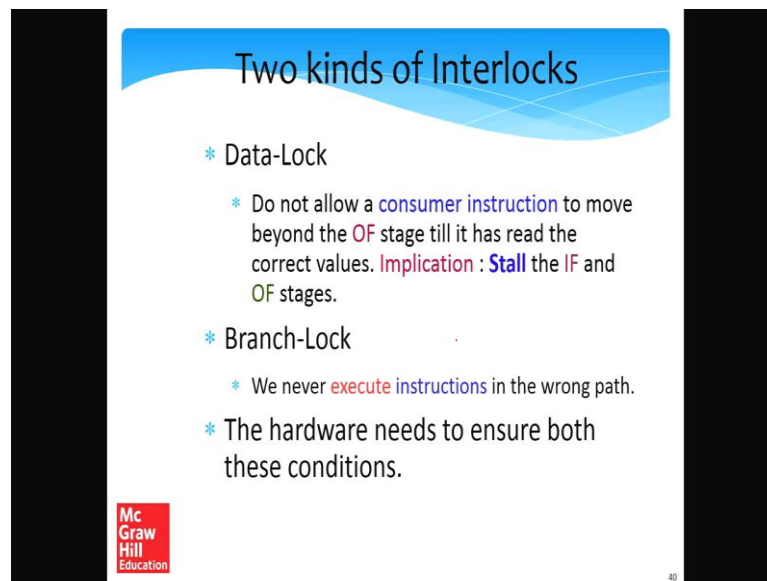
- * We cannot always **trust** the **compiler** to do a good job, or even introduce **nop** instructions correctly.
- * **Compilers** now need to be tailored to specific hardware.
- * We should ideally not expose the details of the **pipeline** to the **compiler** (might be confidential also)
- * Hardware mechanism to enforce correctness → **interlock**

So, let us look see the interlock is one hardware mechanism for this purpose. So, let me just reiterate what I said. So, we cannot always trust the compiler to do a very good job

and insert nop instructions correctly. Also from in this if you are going for software solutions you are always tying the compiler with hardware which is not the best idea.

We are also exposing the details of the hardware which the hardware manufacturers might not be very keen to do, that is the reason we need a mechanism entirely implemented in hardware it is called an interlock.

(Refer Slide Time: 15:42)



The slide is titled "Two kinds of Interlocks" and is presented on a white background with a blue header. It contains two main bullet points, each with a sub-bullet. The first main bullet is "Data-Lock", which includes a sub-bullet stating that a consumer instruction should not move beyond the OF stage until it has read the correct values, with the implication being that the IF and OF stages are stalled. The second main bullet is "Branch-Lock", which includes two sub-bullets: one stating that instructions are never executed in the wrong path, and another stating that hardware must ensure both conditions. The slide also features the McGraw Hill Education logo in the bottom left corner and a small number "40" in the bottom right corner.

Two kinds of Interlocks

- * Data-Lock
 - * Do not allow a consumer instruction to move beyond the OF stage till it has read the correct values. Implication : Stall the IF and OF stages.
- * Branch-Lock
 - * We never execute instructions in the wrong path.
 - * The hardware needs to ensure both these conditions.

McGraw Hill Education

40

So, there are 2 kinds of interlocks one is the data lock and other is a branch lock. Data lock basically says that do not allow a consumer instruction to move fast the OF stage till it has read the correct valves. So, basically you stop an instruction in the operant fetch stage till it gets all the correct values. And the implication of this is that essentially is stall the IF and OF stages. In a branch lock conditions, we will never execute instructions which are on the wrong path.

Why would instructions be on wrong path? They will be there in a wrong path because branch is resolved in the EX stage and in the IF and OF stage we will be fetching the 2 instructions immediately after the branch and these instructions might have been fetched incorrectly because of branch might be been taken. So, we will ensure that such kinds of instructions are never completely executed. So, the hardware needs ensure that both these conditions hold and these conditions the data lock and the branch lock conditions are together called interlocks.

(Refer Slide Time: 17:03)

Comparison between Software and Hardware

Attribute	Software	Hardware(withinterlocks)
Portability	Limited to a specific processor ✗	Programs can be run on any processor irrespective of the nature of the pipeline ✓
Branches	Possible to have no performance penalty, by using delay slots ✓	Need to stall the pipeline for 2 cycles in our design ✗
RAW hazards	Possible to eliminate them through code scheduling ✓	Need to stall the pipeline ✗
Performance	Highly dependent on the nature of the program ✓	The basic version of a pipeline with interlocks is expected to be slower than the version that relies on software ✗

41

So, let us compare the software and hardware approaches. So, in terms of portability software is limited to a specific processor. In hardware approaches any program can run on any processor irrespective of the nature of the pipeline which is defiantly a plus. In the case of branches, it is possible to have no performance penalty by you know optimally using delay slots, in the case of hardware in the interlocks we have to somehow stall the pipeline for the 2 cycles in our design a basic simple design. So, this is at least. So, in this case you know this is one positive point and it is a negative point, through code scheduling it is possible to eliminate raw hazards in our case we need to stall the pipeline. So, it is a bad idea.

In terms of performance well software approaches are highly dependent on nature as a program and in terms of a hardware with interlocks as we have said you know a very basic version is expected to be slower than software, but we will improve it and make it try to make it faster than software, but let us at least look at the basic idea behind a hardware interlocks.

(Refer Slide Time: 18:20)

Conceptual Look at Pipeline with Interlocks

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

- * We have a **RAW hazard**
- * We need to **stall**, instruction [2] at the OF stage for 3 cycles.
- * We need to keep sending **nop** instructions to the **EX stage** during these 3 cycles

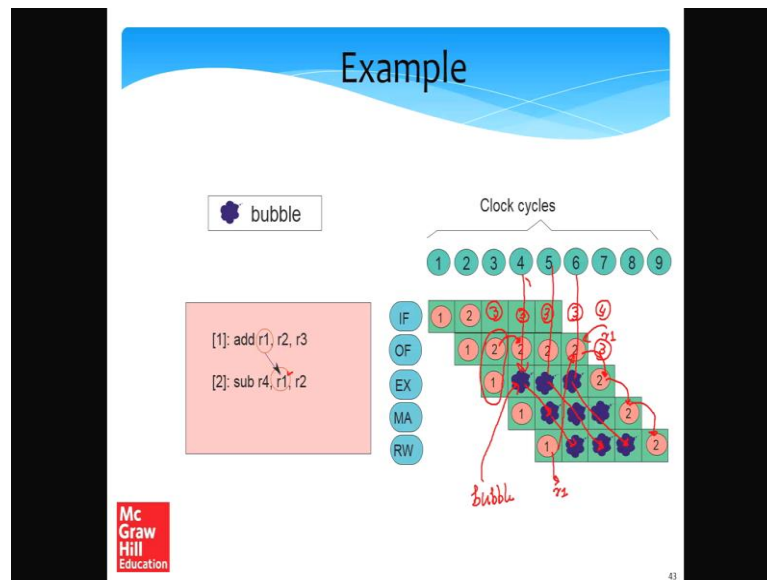
McGraw Hill Education

42

So, let us take a look at conceptual look at pipeline with interlocks. So, let us consider an add instructions. So, let us just consider 2 instructions with a read after write dependence between them. So, the first instruction writes to register r1 and the second instruction reads from register r 1. So, there is a read after write or raw dependence raw hazard. So, what we are saying that we need to stall instruction number 2 at the OF stage for 3 cycles till it gets the right value of r1 which instruction one will write.

So, how do we achieve that? Well, we need to in a sense simulate software in hardware. So, what we do is that we need to keep sending nop instructions to the EX stage during these for the next 3 cycles right, till r1 gets returned and instruction number 2 reads the value of r 1.

(Refer Slide Time: 19:26)



So, let us consider an example. So, let us consider again the 2 instructions add 2 r1 and read from r 1. So, in this case let us see. So, the first instruction enters the pipeline in clock cycle number 1 and then it just proceeds in the pipeline without stalling. So, cycle number 2 it enters the OF stage cycle number 3 it enters the EX stage cycle 4 enters the m s stage and cycle it finishes it is job and exists the pipeline.

So, in the pipeline diagram we see that instruction to enter the IF stage in cycle one enters the OF stage in cycle number 3. So, at this point an important decision is taken it is found out that instruction 2 needs register r 1. And register r 1s value it is not going to get from the register file because instruction one is not done yet. So, in this case what we do is for the next 3 cycles it waits right. So, it waits in the OF stage, but in cycle number 4. So, in cycle number 4 we do not allow instruction 2 to progress to the next stage we keep it at it is current stage right.

So, if we in the e f stage we essentially send a nop instruction which is called bubble, a bubble means a nop right. So, it is nothing it is a same as the bubble in water. So, it essentially means a nop. So, we send a bubble to the next stage and the bubble travels along the pipeline does absolutely nothing, but it travels along the pipeline and it finally, exits the pipeline in cycle number 6. Similarly, in cycle 5 we insert a bubble in the EX stage, and that also travels in the pipeline. In cycle 6 we also insert a bubble in the EX stage it travels in the pipeline, but there is something interesting that is happens in cycle

number 6. So, in cycle 5 we write the value of the register r1 to the register file. So, in cycle 6 the value of r1 can be read by instruction 2 from register 5.

So, now it has the value of r1. So, in cycle number 7 instruction 2 proceeds to the EX stage then it subsequently proceeds to the MA stage and finally, in cycle number 9 it finishes its execution in the RW stage. So, what we see is that we need to introduce 3 bubbles, 3 bubbles mean 3 nop instructions in the pipeline in the EX stage in cycles 4, 5 and 6. So, cycles 4, 5 cycle let me maybe draw an arrow cycles 4, 5 and 6, we need to introduce 3 bubbles.

And these bubbles just travel along the pipeline and since they are nop instructions they absolutely do nothing. We wait till the instruction to wait in the OF stage until it gets the value of r1 and that is when it proceeds. So, one point is noteworthy which has not been shown in this diagram and in cycle number 3 we fetch you know if there was a third instruction we would fetch it into the IF stage, but since instruction 2 is not moving instruction 3 will also not move. So, it will get stuck in the IF stage.

Finally, when there is you know this in this let us still stuck finally, when there is some movement and instruction 2 moves to the EX stage instruction 3 will move to the OF stage. And a new instruction will be fetched in the IF stage. So, if bubble basically stalls a part of the pipeline does not allow any new activity to happen in a part of the pipeline. So, in this case in the IF and OF stage we have not allowed any new activity to happen till the instruction in the OF stage reaches the right value and it is able to proceed. So, such kind of an approach this is a data lock interlock and we have essentially introduced nops in the pipeline dynamically and a nop in a pipeline is also called a bubble.

(Refer Slide Time: 24:17)

A Pipeline Bubble

- * A pipeline bubble is inserted into a stage, when the previous stage needs to be stalled
- * It is a nop instruction
- * To insert a bubble
 - * Create a nop instruction packet
 - * OR, Mark a designated bubble bit to 1

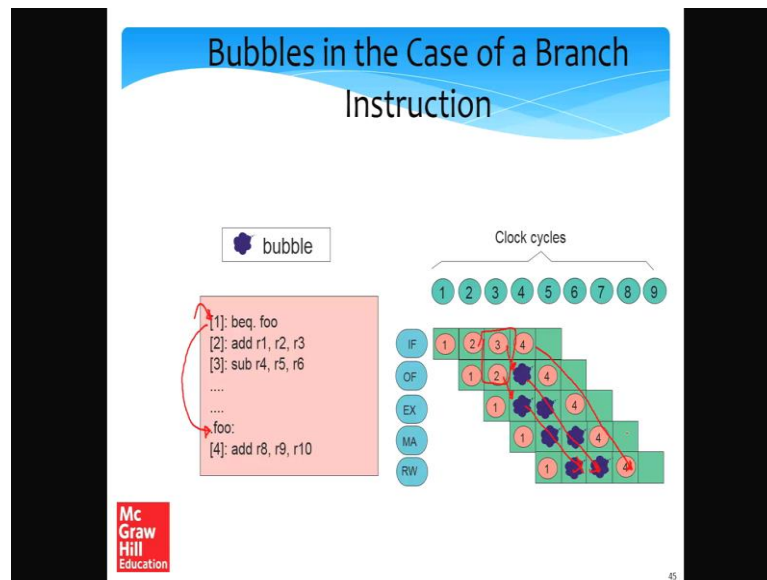
McGraw Hill Education

44

So, let us look at a pipeline bubble in some more detail pipeline bubble is pretty much a nop instruction. And to insert a bubble what needs to be done is that we need to create a new instruction packet as was discussed in the last chapter, chapter number 8. So, we create a new nop instruction packet which basically tells the pipeline register to do nothing.

And we can also keep a designated bubble bit to say that a bubble and mark a designated bubble bit in the instruction packet to 1. So, as t instruction packet moves across the pipeline stages the pipeline will be well aware of the fact that this is a bubble and nothing much needs to be done I mean nothing needs to be done.

(Refer Slide Time: 25:10)



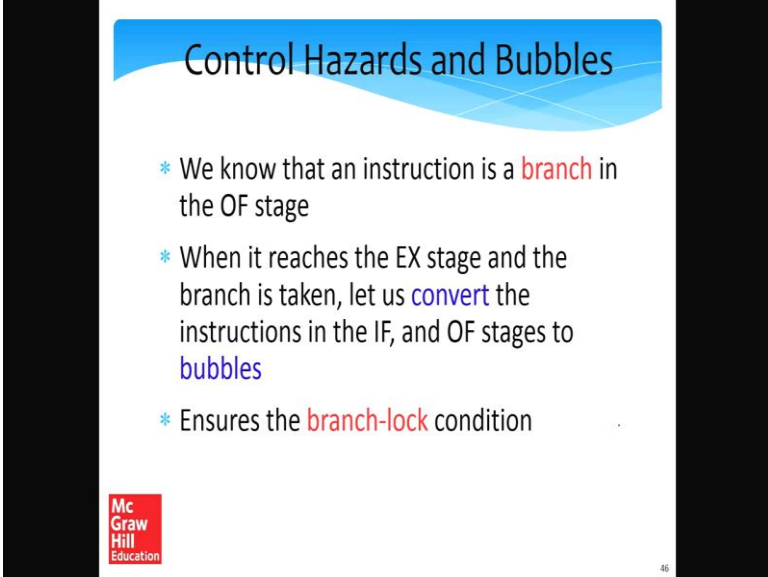
So, now let us look at bubbles in the case of a branch instruction. So, in the case of a branch instruction when delayed branches are not there right the way the things work is as follows. So, let us assume that we have a conditional branch instruction `beq dot foo`, where if the condition is correct then we directly branch to the instruction at `dot foo`. So, we have 2 instructions after the branch instruction. So, what happens is that once we fetch the first instruction which is the branch till it reaches the EX stage we are really not sure about the direction of a branch. Once it reaches the EX stage we have already fetched instructions 2 and 3 by that time.

So, let us assume that the branch is a taken branch and delayed branches are not there. So, in this case what we need to do is that we need to first realize that instructions 2 and 3 are on the wrong path they should not have been fetched. So, dynamically instruction 2 is converted to a bubble and instruction 3 is converted to a bubble which basically means that we either mark a bubble bit to be one or convert them to a `nop` and they are cancelled in the pipeline.

Since in the IF and the OF stages they would not have returned to the register file or to the memory it is possible to cancel them. So, there is no harm in that sense and cancelling them. So, the way we cancel them is that we convert them into bubbles and then the bubbles move through the pipeline and exit the pipeline. And when we make a determination that the instruction one is a taken branch we then fetch the branch

target which is instruction 4 in the fourth cycle and instruction 4 sends it on the correct path it continues along the pipeline, had instruction one not been a taken branch then we there was no necessity of doing it because instructions 2 and 3 would have been on the right path would have been on the correct path, and they would have continued their execution and gone down the pipeline.

(Refer Slide Time: 27:39)



The slide features a blue header with the title "Control Hazards and Bubbles". Below the header, there are three bullet points. The first bullet point states: "* We know that an instruction is a **branch** in the OF stage". The second bullet point states: "* When it reaches the EX stage and the branch is taken, let us **convert** the instructions in the IF, and OF stages to **bubbles**". The third bullet point states: "* Ensures the **branch-lock** condition". In the bottom left corner, there is a red logo for "McGraw Hill Education". In the bottom right corner, there is a small number "46".

- * We know that an instruction is a **branch** in the OF stage
- * When it reaches the EX stage and the branch is taken, let us **convert** the instructions in the IF, and OF stages to **bubbles**
- * Ensures the **branch-lock** condition

McGraw Hill Education

46

So, let us talk about control hazards and bubbles. So, we now know that an instruction is a branch in the OF stage. So, when it reaches the EX stage and the branch is taken we convert the instruction in IF and OF stages to bubbles we have to do that, this will ensure the branch lock condition which was basically that we do not allow any instruction in the wrong path to execute and move down the pipeline.

(Refer Slide Time: 28:10)

Ensuring the Data-Lock Condition

- * When an **instruction** reaches the **OF** stage, check if it has a **conflict** with any of the instructions in the **EX, MA, and RW** stages
- * If there is **no conflict**, **nothing** needs to be done
- * Otherwise, **stall the pipeline** (IF and OF stages only)

Mc
Graw
Hill
Education

47

And to ensure the data lock condition when an instruction reaches the OF state we check if the read after write can conflict with the any of the instructions in EX MA and RW stages if there is no conflict. So, conflict in this stage means the read after write dependence nothing needs to be done otherwise we stall the pipeline the IF and OF stages only, till the correct data is available in the OF stage and we can proceed.

(Refer Slide Time: 28:43)

B
A

```
Algorithm 5: Algorithm to detect conflicts between instructions
Data: instructions, [A], and [B]
Result: conflict exists (true), no conflict (false)
if [A].opcode ∈ {nop, b, beq, bgt, call} then
    /* Does not read from any register */
    return false
end
if [B].opcode ∈ {nop, cmp, st, b, beq, bgt, ret} then
    /* Does not write to any register */
    return false
end
/* Set the sources */
src1 ← [A].rs1
src2 ← [A].rs2
if [A].opcode = st then
    src2 ← [A].rd
end
if [A].opcode = ret then
    src1 ← ra
end
hasSrc1 ← true
if ([A] ∈ {not, mov}) hasSrc1 ← false
```

Mc
Graw
Hill
Education

48

Now, let us present an algorithm to detect a conflict between 2 instructions. So, a conflict is depending is defined as read after write kind of dependence. So, let us consider

instructions A and B where instruction B is the instruction which comes first and instruction A comes later. So, let us go here instruction B comes first and instruction A comes later. So, the result is if a conflict exists if there is a read after write kind of dependence then we return true otherwise we return false.

So, let us start. If a dot op code means if A is op code is. So, this is the first line I will just correct it in the slide should be indented here. So, if A is op code is nop a branch a beq a bgt and a call. So, this essentially means it does not read from any register right. So, it does not say does not read from any register there cannot be a read after write conflict. So, we can happily return false. So, falls means that there is no conflict similarly bs op code is element of is in this set and nop I compare a store branch beq bgt and ret this means that it does not write to any register writes a no register is being returned to. So, a result again a read after write conflict cannot happen. So, we return false.

If that is not the case in theory read after write conflict can happen. So, we need to talk a look at the source registers and the destination registers. So, let us term src 1 as the first source rs 1 a dot rs 1 and let us term src 2 source 2 as the second source register in the rs 2 field. So, let us make a small correction for the store instruction. If A is a store instruction, then recall that the second stores are not the rs 2 field rather is the rd field. So, we make this was a special case made for the store instruction. So, this is incorporated here in this algorithm and if as op code is equal to ret then. So, the returned instruction actually reads from the return address register ra and so, we set src 1 as ra.

So, recall that these are all the special cases we had made one special case for store one special case for return. So, that is the reason you know this is further complicated around the logic pipeline. So, now, hopefully the readers can realize how important it is to have a simple instruction set. Never the less you know sometimes special cases do happen. So, they need to be taken care of like in the case of a store where one of the sources comes from the rd register and in the case of the ret or the return instruction where one of the sources comes from the ra register.

So, it is possible that not all instructions will have actually both the sources. So, let us by default set the variable as source 1 equal to 2. So, if the instruction A is not or a move. So, recall that in that case we do not store the source in source 1. So, it will actually store in rs 2. So, that is the reason we set as source 1 to false. So, basically for this readers

need to go to chapter number 3 to find out why this is the case, but that was just a format. So, when we have designed the instructions we have designed them in such a way that the knot and move instructions the register source or the immediate stores was actually in rs 2 it was not in rs 1 hence we set the has source 1 flag to false because it does not have the first source in rs 1 there is nothing.

(Refer Slide Time: 33:32)

```

dest ← [B]rd
if [B].opcode = call then
  dest ← ra
end
/* Check the second operand to see if it is a register */
hasSrc2 ← true
if [A].opcode ≠ (st) then
  if [A].r then
    hasSrc2 ← false
  end
end
/* Detect conflicts */
if (hasSrc1 = true) and (src1 = dest) then
  return true
else if (hasSrc2 = true) and (src2 = dest) then
  return true
end
return false

```

Handwritten annotations on the slide include:

- Red text: `Src1`, `dest`, `Src2`
- Red text: `add r1, r2, r3` with arrows pointing to registers `r1`, `r2`, and `r3`.
- Red text: `Src1`, `Src2`, `dest` grouped by a bracket with `has Src1`, `has Src2`, and `Boolean`.

Now, let us look at the destination. So, in the destination field it is by default the rd register is the destination; however, here also once the special case needs to be considered. So, if the instruction is a call instruction then the call instruction writes to the ra the return address registers the ra register. So, the ra register is saved in dest that is one special case. And then we need to see after taking a look at the destination we need to see if the second operator is in the register or not for A. So, we set again has source 2 to true are the only special case that will come here. So, so what is the order we first looked at source 1 then we looked at destination then we looking at source 2, in source 2 the special case that we need to consider is the case of the store instruction.

So, in the case of a store instruction; well, I am sorry the special case that we need to consider is when the second source is an immediate. So, since the store instruction was special in the sense it has an immediate as well as a second register source we first eliminate this from our condition. Subsequently if the instruction has an immediate, then it does not have a second source for example, if we consider `add r1 r 2 and r 3` I am sorry

not r 3, but should be an immediate let us give it a second immediate operand r add r1 r 2, and let us say 3.

So, in this case we have a single register source which is r 2 and this is stored in field rs 1 we do not have anything stored in field rs 2, and we have a single register destination which is an rd. So, this exactly what this part of the logic is trying to capture what this part of the logic is trying to say that if the instruction is not a store which is special and if a dot i is equal to one means it has an immediate operand then it does not have a second register source. So, has src 2 is false.

So, what have we achieved till now? So, up till now we have set the values of src 1 which is the first register source src 2 and the value of dest which is the id of the destination register we also have set the values of the bullion flags as src 1. So, lot of instructions might not have the first source then has src 2. So, particularly instructions with an immediate operand will not have a valid second source register. So, all of these 3 fields are pretty much 16 bits I am sorry 4 bit fields because each bit requires 4 bits for encoding and these are bullion fields which essentially say if the source 1 exist if source 2 exists.

Now, detecting conflux is very easy, if certain instruction has it is first source it has a valid first source register and src 1 is equal to dest. So, this basically means that there is a read after you know dest is a write and the src 1 is the read. So, there is a read after write dependency then we return true right. So, then a dependence exists else if has src 2 equal to true which means that the second source exists and src 2 is equal to dest which means again there is a read after write dependence then also we return true which indicates that between instructions B and A there is a read after write dependence.

So, let me summarize our discussion in the last 2 slides you know this slide, slide number forty 8 and slide number 49. So, what we are essentially showing is an algorithm to find dependencies between 2 instructions between instructions B and A where A comes after B. And this algorithm needs to be implemented in hardware not in software. So, we need to do a lot of checks and a hardware circuit needs to be written to implement this algorithm, what complicates our life is the existence of many special cases, which we are realizing now that special cases you know sense or a bad thing, but sometimes are unavoidable. For example, we have made a special case for store where we store the

second source and rd. So, this requires this introduces an additional s statement over here we had made a return special case where it writes to a return address return address register. So, that introduces some complexities as well.

Similarly we had we had to take care of the call instruction also, because it writes to the return address register and we need needed to take care of the immediate format as well. If we would have had more formats this logic would have become far more complicated, but thankfully it is only limited to slides, then detecting conflux is easy if we have a valid first source we check if the destination of B is the same as the first source away.

(Refer Slide Time: 39:36)

The slide is titled "How to Stall a Pipeline?". It contains the following text:

- * Disable the write functionality of :
 - * The IF-OF register
 - * and the Program Counter (PC)
- * To insert a bubble
 - * Write a bubble (nop instruction) into the OF-EX register

At the bottom right of the slide, there is a hand-drawn diagram of a pipeline. It shows two stages: OF and EX. The OF stage is labeled "OF" and the EX stage is labeled "EX". A red arrow points from the OF stage to the EX stage. A red box labeled "nop" is drawn inside the OF stage, indicating a bubble. The diagram is numbered "50" at the bottom right.

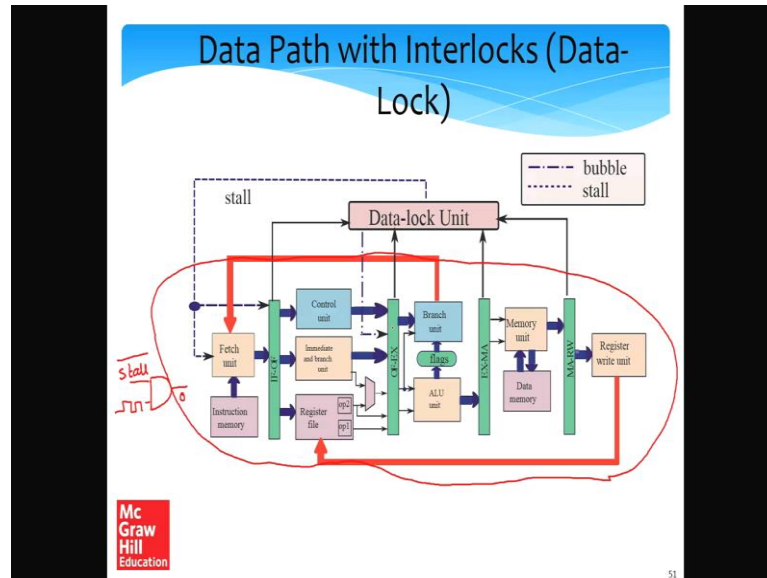
Mc Graw Hill Education

And if we have a valid second source we check if the destination of the B is the same as the first source of sorry the second source of A. So, after this logic once we have found out the conflict exists how do we stall a pipeline. So, what we do is that we disable the write functionality of the IF OF register and the program counter pc. So, this ensures that it is not overwritten with new data right. So, that helps us stalling the pipeline. And to insert a bubble we write a bubble or a nop instruction into the OF EX register which basically ensures. So, if this is the OF EX register OF on one side EX on the other side, this is the OF EX register. So, if we do this then it ensures; so this end.

So, if we write to the register it ensures that this value will propagate along the pipeline as well. So, what we need to do is that we need to write a nop instruction into this register, in the next cycle it will propagate along the pipeline and in the next register

which is the EX MA register it will reach that register in the next cycle. So, this is the way bubble propagates the same way normal instruction propagates it is just that a bubble does not have any effect.

(Refer Slide Time: 41:15)



So, let us now slightly update the averaged diagram of the pipeline that we have been using. So, recall that this diagram was introduced in the last lecture with only the core parts of the pipeline. So, we have now augmented it with data path with interlocks. So, what we do over here is that the first thing we do is that we send the contents of all the pipeline registers because they contain all the instructions. So, the IF OF register contains the contents of the instructions that are there in the OF stage. Similarly the MA RW register contains the contents of the instructions in RW stage.

So, all of their contents are sent to the data log unit which finds conflicts. If it detects a conflict, then it sends a stall signal to the fetch unit as well as to the IF OF register. So, there is a stall signal. So, the stall signal basically stops the register from accepting any new data from essentially overwriting its value and the easy way of implementing the stall signal is to basically have a NAND gate, where you know the NAND gate will have the clock entering on one side, and the output of the stall signal entering on the other side.

So, the stall signal is 0 the output is the clock if the stall signal is one, then the output is 0. So, this can be the clock input for the fetch unit, as well as the IF OF unit. So, for both of these units they will stall and no new instruction will be able to enter into them.

Simultaneously we have an additional set of lines that go from the data lock unit to the OF register. So, they introduce the bubble.

So, when we want to introduce a bubble the value in the OF EX register is written is overwritten, with an instruction packet that corresponds to a bubble or a nop instruction. So, this you know line ensures that and subsequently the bubble propagates to the pipeline. So, as we see most of the pipeline remains the same. So, at the risk of cluttering the diagram let me draw a big circle. So, this was the previous the older pipeline and to have a data lock you know we just take inputs from all the stages and we stall the first 2 stages and in the EX stage we can introduce the bubble.

(Refer Slide Time: 44:02)

The slide is titled "Ensuring the Branch-Lock Condition". It lists two options:

- * Option 1 :
 - * Use delay slots (interlocks not required)
- * Option 2 :
 - * Convert the instructions in the IF, and OF stages, to bubbles once a branch instruction reaches the EX stage.
 - * Start fetching from the next PC (not taken) or the branch target (taken)

Handwritten annotations include a red circle around "branch" in Option 1, and the letters "IF", "OF", and "EX" in red next to Option 2. A small red box with "Mc Graw Hill Education" is in the bottom left corner, and a small "52" is in the bottom right corner.

So, let us now look at the branch lock condition. So, let us consider the solution the software only solution which was to use delay slots. So, while using delay slots interlocks are not required. So, pretty much in every pipeline after a branch instruction we have a couple of slots in which there is a possibility of wrong path instructions being fetched. So, that is the reason what we do is that we pick instructions that are before the branch and move them to these slots such that we do not instead of executing possibly wrong path instructions we execute correct instructions.

So, delay slots are very simple they do not require hardware solutions. So, let us look at a second option. So, second option is that we convert the instruction in IF and OF stages to bubbles. Once a branch instruction reaches the EX stage why. So, the reason is that in the

EX stage we make a determination, that the branch whether it is taken or not taken. So, what we do is we just cancel the instructions by converting them to bubbles, the instructions in IF and OF stages right are cancelled and they are converted to bubbles.

Then we start fetching for the next pc in the case when the branch is not taken the next pc would be the branch instruction plus 4 bytes. And if the branch is taken then we start fetching from the branch target.

(Refer Slide Time: 45:43)

Ensuring the Branch-Lock Condition - II

* Option 3

$\begin{matrix} X & X & NT \\ IF & OF & EX \\ \hline p+3 & p+4 & \end{matrix}$

- * If the branch instruction in the EX stage is taken, then invalidate the instructions in the IF and OF stages. Start fetching from the branch target.
- * Otherwise, do not take any special action
- * This method is also called predict not-taken (we shall use this method because it is more efficient than option 2)

Mc Graw Hill Education

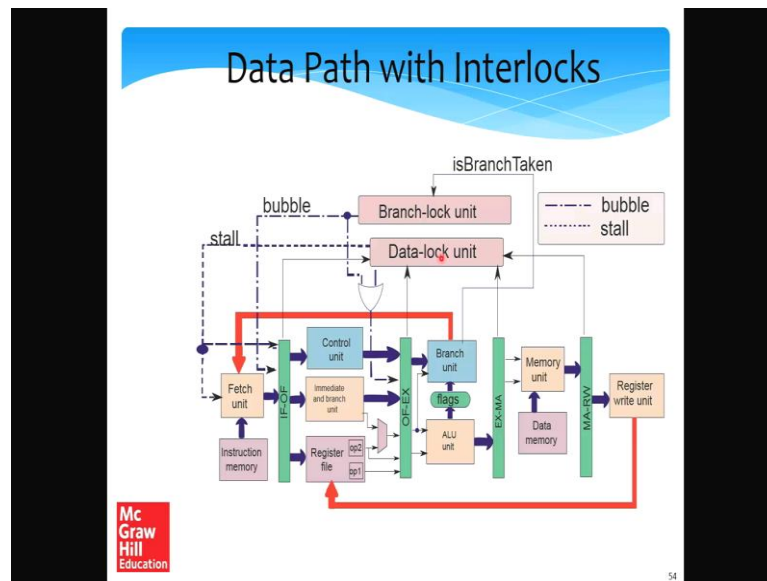
53

So, option 2 is not really the best option as we shall see. So, option 3 is the better option. So, the option 3 is as follows let us assume that if the branch instruction in the EX stage is taken. Then only we invalidate the instructions in the IF and OF stages and we start fetching from the branch target. So, this is basically saying that instead of invalidating and you know converting instructions into bubbles all the time that is a bad idea it is a waste of useful work.

So, instead in the EX stage if the branch is taken then only the instruction in the IF and OF stage are not valid. So, then they can be converted to bubbles and made to flow through the pipeline; however, if the instruction in the EX stage is not taken then the instructions in the IF and OF stages are correct right. They should have been correctly fetched and they are. So, no special action needs to be taken. So, we do not convert it into bubble.

So, in this method we are essentially predicting that the branch is not taken. And we are optimizing for this case. So, we are essentially if the pc of the branch is if p we are fetching the instruction at p plus 4 and we are fetching the instruction at p plus 8 if the branch is not taken there is no problem these are valid instructions; however, if the branch is taken these instructions have to be cancelled nullified converted into bubbles and then made to flow across the pipeline.

(Refer Slide Time: 47:25)



Let us now take a look at the data path with interlocks. So, we have our data lock unit in place. So, the addition in this diagram is pretty much the branch lock unit which is the new unit over here. So, the branch lock unit pretty much takes the inputs from the branch unit which says it just gives a one bit input whether the branch is taken or not and since we have chosen option 3 which means that when a branch is taken. Then only we stall I am sorry then only we insert bubbles right convert instructions in IF and OF stages to bubbles. What happens is if we decide that branch is taken then we send a bubble signal to the IF OF register.

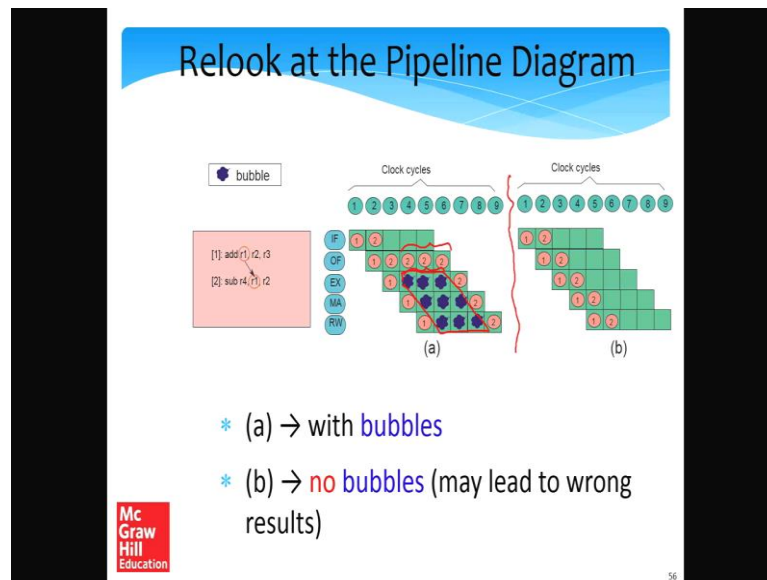
So, this converts the instruction which is going from the IF stage to the OF stage this converts the instruction to a bubble. And simultaneously we also send a bubble signal to the OF EX register, such that in the next cycle that the instruction that goes from the OF stage to the EX stage that also gets converted to a bubble. We have an or gate over here

because the bubble signal can come from the data lock unit as well and a bubble signal can come from the branch lock unit.

So, mind you the connotation for this is slightly different in the sense the data lock unit introduces an extra bubble into the pipeline. Whereas, for a branch lock unit all that it does is that it basically nullifies the instructions in the IF and OF stages. The way it does that is it does not actually change the instruction, but for the next cycle when the instruction in the let us say the OF stage will flow to the EX stage. Instead of sending the original instruction it is a nop similarly for instruction that supposed to flow from the IF stage to the OF stage instead of sending the original instruction we send a nop that is the only difference no other difference and the rest of the circuit that remains the same. So, all that we see is that we have this extra 2 interlock unit the branch and the data lock units whose job is to ensure that all data hazards and control hazards are taken care of correctly in our pipeline.

Now, that we have seen interlocks we looked at a far more efficient mechanism. So, it is not necessary to stall the pipeline. So, what was the main idea of interlocks? The main idea of interlocks was pretty much bubble install. So, it is not necessary to stall the pipeline all the time we can get a lot of performance the massive amount of performance by actually forwarding data between stages. So, we can actually avoid a lot of bubbles in this process right. So, forwarding is a good thing in a sense it avoids bubbles right. So, we can write here avoids bubbles. So, in terms of performance it is a fantastic idea.

(Refer Slide Time: 50:58)



So, let us look at this. So, let us consider this code snippet once again where we have 2 instructions the first instruction generates the value in a register and the second instruction reads the value.

So, in this case the first instruction generates the value for r1 and the second instruction reads the value of r1. So, it is a read after write dependence. So, what we would do in our older pipeline diagram with bubbles is that we will essentially once instruction one enters sorry instruction 2 enters the OF stage we will make that wait there for 3 additional cycles and introduce 3 bubbles in the pipeline, that will flow through the pipeline. So, pretty much this entire amount of this entire region covered by bubbles rest at work. So, this is the work that we are losing right the efficiency that we are losing because we have bubbles in the pipeline.

(Refer Slide Time: 52:14)

Crucial Insight (Figure (b))

- * When does instruction 2 need the value of r1 ?
 - * **ANSWER** : Cycle 3, OF Stage (**wrong!!!**)
 - * **CORRECT ANSWER** : Cycle 4, EX Stage
- * When does instruction 1 produce the value of r1 ?
 - * **ANSWER** : Cycle 5, RW Stage (**wrong!!!**)
 - * **CORRECT ANSWER** : End of Cycle 3, EX Stage

McGraw Hill Education

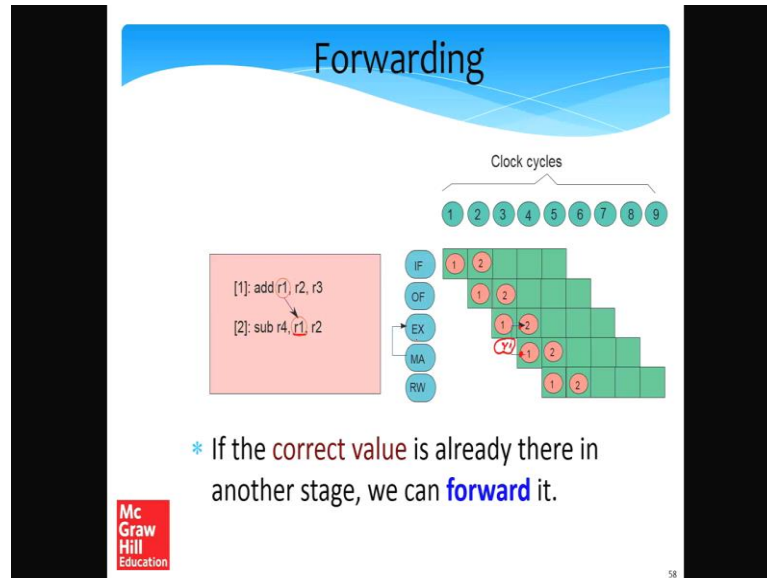
57

So, what might be a better idea is to have no bubbles at all, but in this case it will lead to wrong results. So, let us take a crucial insight from figure b in the previous slide I will go back to it. So, when does instruction 2 need the value of r1. So, instruction 2 needs the value of r1 we have been saying that it needs that in cycle 3 because it will read it off the register cycle. So, we are saying cycle 3 OF stage, but that is exactly speaking not correct we actually need it in cycle 4 which is the EX stage. So, let me go there once again. So, we need it in cycle 3 in the sense that we will be needing it off the register file. So, we need it in the register file cycle 3, but we will not be using it we will actually be using it if in this case instruction 2 enters the EX stage which is cycle 4 that is when we will actually be using the value right.

So, we will be needing it in cycle 3, but we will be using it in cycle 4. So, to be technically and factually correct we need the value of r1 instruction 2 needs the value of r1 in cycle 4. So, when does instruction r1 produce the value of r1 up till now we have been maintaining that instruction one produces a value of r1 in cycle 5 which is it is RW stage. So, cycle 5 is when we produce a value of r1 and write it to the register file, but that also is not correct. So, the correct is the answer is at the end of cycle 3 when it finishes its execution that is when at least the value of r1 is there even though the register r1 is not been returned to yet.

So, maybe we can use this particular pattern and do something. So, what we shall do is that we shall take a look at this diagram once again and realize that in cycle 4, so at the end of cycle 3.

(Refer Slide Time: 54:16)



So, first thing we need to realize for this whole snippet is that at the end of cycle 3 the value of r1 which is the result when we add r2 and r3 the result is ready. So, in cycle 4 at the beginning the result is ready and also in cycle 4 we need the value of r1. So, since the value of r1 is present with instruction 1 which is currently in MA stage and instruction 2 which is currently in EX stage needs the value of r1 we can always connect the producer with the consumer, because the value is already there in another stage we can take it from the other stage.

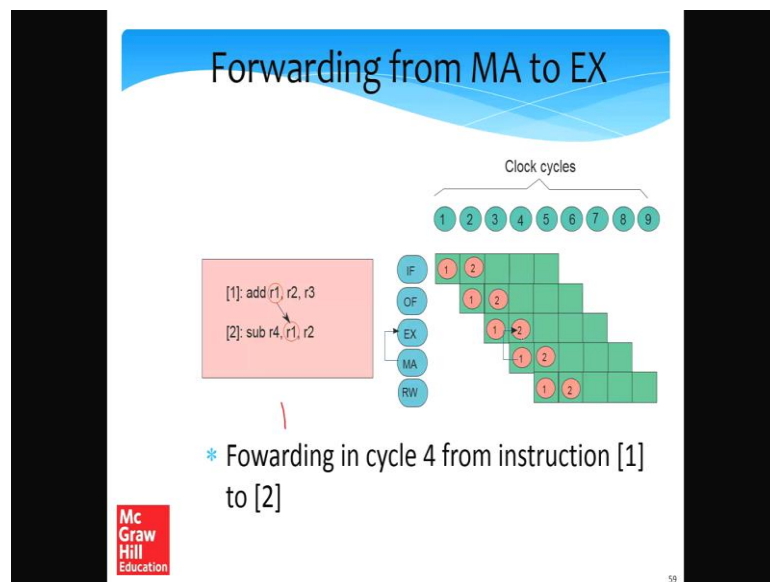
So, up till now we have not been opening we have not been discussing this because our only method of information flow across instructions was via the register file. So, one instruction wrote something to a register, and the other instruction read the contents of the register, but in this case what we see in cycle 4 specially that the value of r1 is present with instruction one in the image stage and at beginning of cycle 4 instruction 2 needs the value of r1 because it needs to add r2 with it.

So, since the need is there as well as the value is there all that we need to do is we need to connect the EX and MA stages with a wire and forward the value of r1 from the MA stage where instruction 1 is there to the EX stage where instruction 2 is there and it meets

the value of r1. So, this method is called forwarding this will work it will not cause any correctness problems in our pipeline, and it will also avoid 3 stalls it will also avoid 3 bubbles right.

So, the bubbles where you know pretty important issues in terms of the fact, that this is any producer consumer fare we had to either find 3 other instructions to put between them or we used to put nops or insert bubbles using hardware technique, but in this case none of that is required, because we can directly forward the value of r1 from the MA stage to the EX stage as shown in this example this is called forwarding.

(Refer Slide Time: 56:55)



Let us look at one more example. Well it is the same example it is just we have changed the title if you see this is forwarding in general, but there are different kinds of forwarding. So, let us look at forwarding from MA to the EX stage which is again this example. So, we are forwarding the value of r1 from instruction 1 to 2 and as I said doing it is very easy we just need to add a wire from the MA stage to the EX stage. So, the value will go from instruction 1 to 2 in cycle number 4.

(Refer Slide Time: 57:32)

Different Forwarding Paths

- * We need to add a multitude of forwarding paths
- * Rules for creating forwarding paths
 - * Add a path from a later stage to an earlier stage
 - * Try to add a forwarding path as late as possible. For example, we avoid the EX → OF forwarding path, since we have the MA → EX forwarding path
 - * The IF stage is not a part of any forwarding path.

McGraw Hill Education

60

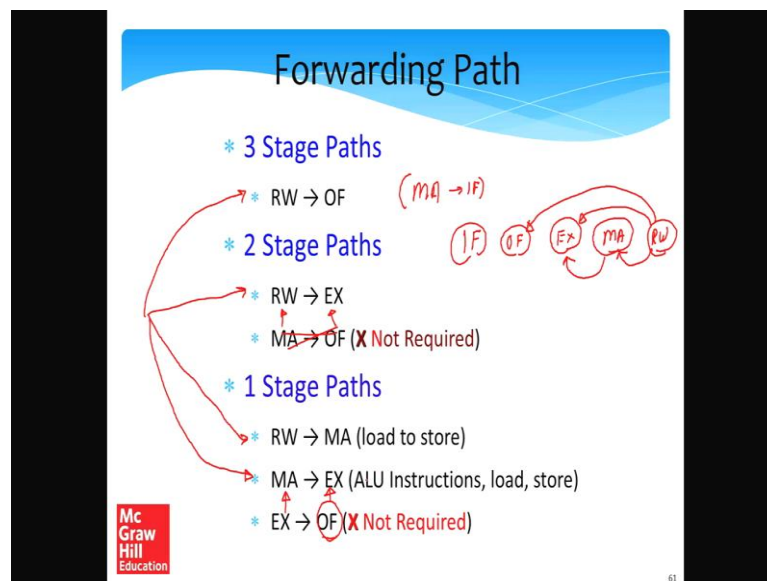
We can have many more forwarding paths right a multitude of forwarding paths can be there, but let us look at certain rules for creating forwarding paths. So, one path we already know that is from MA to EX, but let us look at some more forwarding paths. So, what we shall do is that we shall add a forwarding path from a latest stage to an earlier stage like from MA to EX right. So, we always you know flow forward in time. So, it is always an earlier instruction forward data to a later instruction.

We tried to add a forwarding path as late as possible which means we will not have a forwarding path as a form EX to OF right that is not required, instead we will have a forwarding path between MA to EX why is this the case, the reason is that no instruction actually needs the value in the OF stage in the OF stage we read the value, but we do not actually use it that is the reason instead of forwarding from EX to of a better idea is that we forward something from MA to EX because something. So, we want to forward as late as possible. So, we want to forward in other words as close as possible to the actual use.

In this case we see that forwarding any instruction or any value to the OF stage is not the right idea, the reason being that we were not actually use anything in the OF stage. So, we would rather allow the instruction to proceed through the pipeline and use and forward at a later point in time the IF stage is not a part of any forwarding path, that is the next thing because you know we do not have to sort of follows from the second point

that instruction will use any will use any data in IF and OF stage. So, it is not required and also the other thing is we do not have to do it for a very simple reason that the instruction in the IF stage right we can either use a later forwarding path and we will get the data and if the instruction is in the RW stage it would have returned to the register anyway we will be able to read the value when the instruction in the IF stage moves to the OF stage.

(Refer Slide Time: 60:08)



So, let us look at some forwarding paths. So, let us first look at the 3 stage forwarding path which you know is across 3 stages. So, that would be from RW to OF right. So, recall that we do not have to add a forwarding path from MA to IF it is not required we just allow both to progress by one stage and we add from RW to OF. So, this will this basically means that you have one instruction is writing to a register and simultaneously another instruction reading from the register, then instead of getting the data from the register file we will get it from the forwarding path. So, let me may be write down the stages it will become easier to visualize.

We are adding one path from RW to OF and MA to IF is not required because if you allow both instruction to proceed by one stage we can use this path. So, it is a 3 stage path because as you can see the RW stage is 3 stages ahead of OF. Now let us take a look at 2 stage path. So, one is RW to EX the other can be from MA to OF, but MA to OF is not required because in MA what would we get we would you know I really do not want

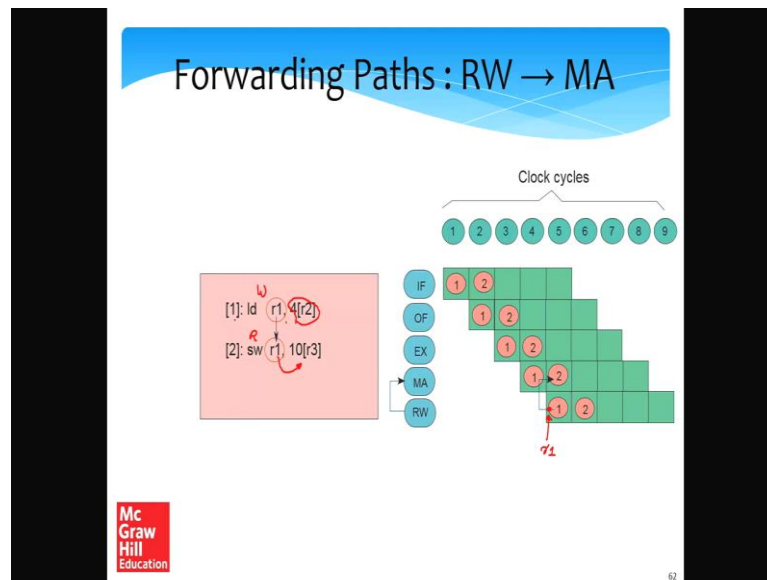
to you know it is sort of our second axiom over here we want to forward as late as possible. So, a OF is not required we can rather use, the RW to EX forwarding path because MA would proceed in one cycle to RW and OF would proceed to EX and we do not need this path that is the reason is not required.

So, an intelligent reader can always ask why did you need this path from RW to OF, well the answer is simple because we do not have any stage after RW. So, when we reach the EX stage the producer instruction is out of the pipeline. So, it will not be in a position to forward that is the reason we use the RW to OF forwarding path here, but we need not have a path between MA to OF because we can allow both to progress by one stage and use the RW to EX path.

Similarly, we can have a one cycle one stage path which is RW to MA we will see an example of this we can have MA to EX we have already seen an example of this. EX to OF is again not required because in the OF stage we do not need any value hence you can allow both the instructions to progress by one stage, and use the MA to EX forwarding path instead.

To summarize; we might not be in a position to appreciate these things completely as of now, but gradually over time we will appreciate forwarding much better. So, we will have only 4 forwarding paths in our design. We will have one path from RW to OF one from RW to EX one from RW to MA and one from MA to EX. So, let us take a look at examples understanding why examples is always much easier.

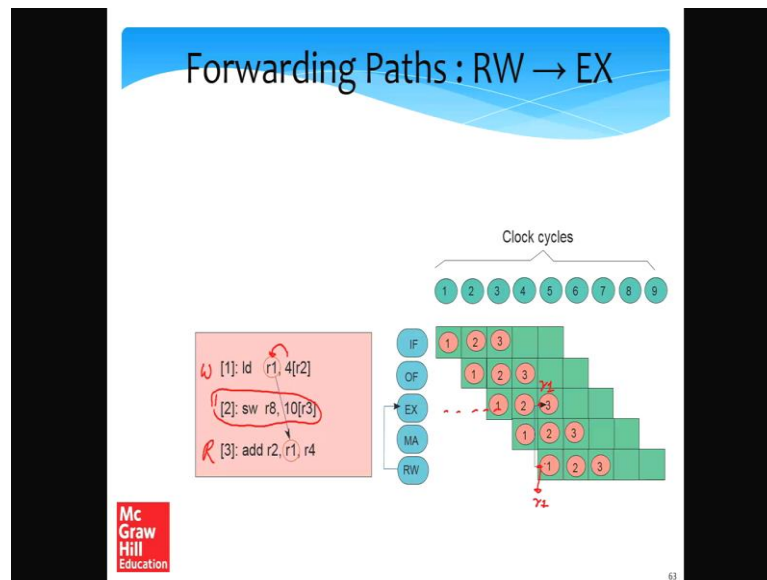
(Refer Slide Time: 63:50)



So, let us take a look at the RW to MA forwarding path right. So, let us you know in no particular order let us start taking the look at these forwarding paths. So, the RW to MA forwarding path let me give an example of 2 instructions. So, let us assume we have a load instruction. The load instruction reads from this address 4 or 2 and it writes to the register r 1, and then assumes that we have instruction number 2 which is a store instruction which reads the value of register r1 and writes to memory address. So, there is a clear read after write dependence over here where the load instruction in doing a write and the store instruction is doing the read. So, it is reading the rally of the register and writing it to memory.

So, in this case what we see is in cycle number 5 the load instruction would have already got10 the value from memory. So, the value of r1 would be there with it at the beginning of the cycle also in cycle number 5 store instructions would be there in the MA stage the memory axis stage and in the beginning of cycle number 5 it would actually need the value of register r 1. So, that is an ideal opportunity for us to forward. So, we forward the value from the RW stage to the MA stage basically from instruction 1 to instruction 2 and this helps us avoid stalls and it also helps us execute a load and a store back to back. Back to back means in back to back cycles or in consecutive cycles.

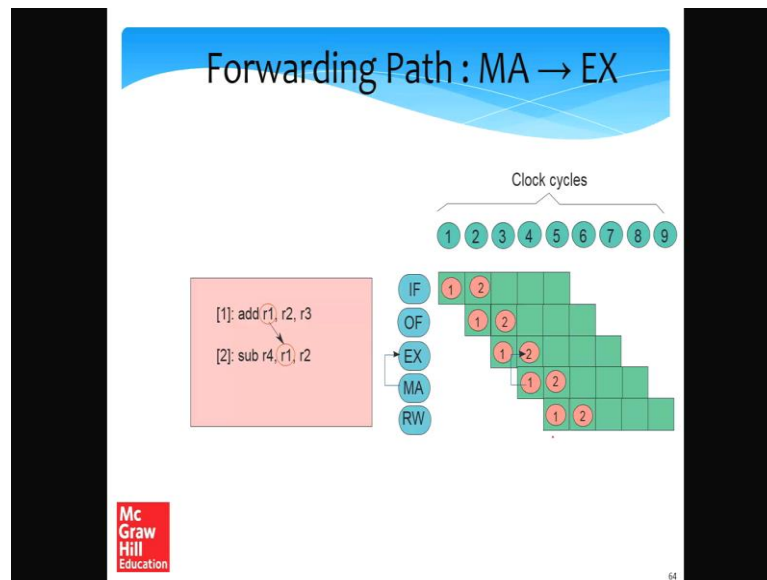
(Refer Slide Time: 65:48)



Let us take a look at the next forwarding path which is RW to EX. So, RW to MA we have seen let us see RW to EX. So, let us have a load instruction which again reads the value into register r 1. Then let us assume the store instruction which is absolutely independent. So, it accesses a different set of registers and then we have an air instruction that has a raw dependence with the load instruction, instruction 1. So, it reads register r1. Let us again consider cycle number 5 where the load instruction would have completed it is load. So, the value of r1 would be there with it and we do not care about intervening store instruction because it does not have any independencies, but the add instructions would be there in the EX stage and that is exactly when it would require the value of r1.

Again this is the ideal case of forwarding because the value of r1 is already there in the later stage it is there in stage number 5 which is RW stage. So, from RW to EX we can just connect a set of wires and forward the value between instructions one and 3 and thus avoids stocks.

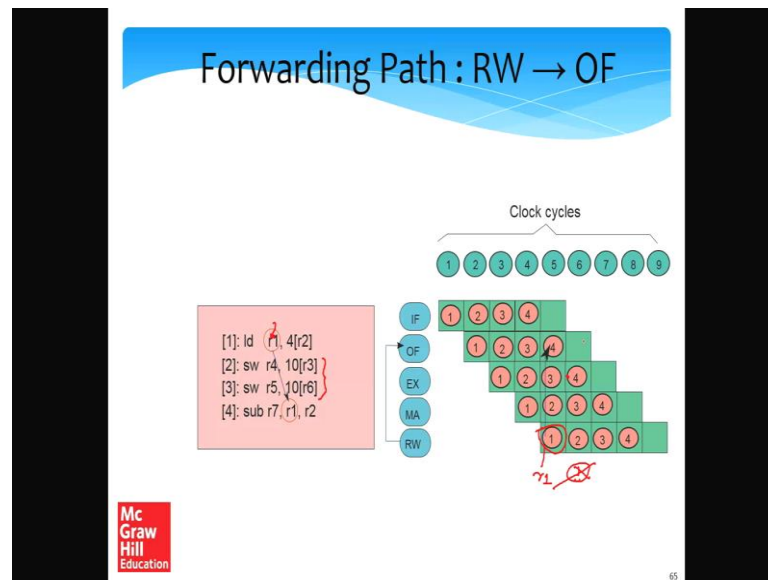
(Refer Slide Time: 67:24)



Now, let us look at the MA to EX forwarding we have already seen this example twice. And all that needs to be done is that we need to connect a set of wires between the MA stage and the EX stage and this is mainly useful for forwarding values between ALU instructions like add subtract multiply divide.

So, previous RW to MA and RW to EX forwarding paths are mostly useful when the producing instruction is a load because the loads value becomes available only in the RW stage right after the memory axis is done. So, that is the reason the load instruction uses those paths and the ALU instructions is mostly used MA to EX path.

(Refer Slide Time: 68:16)



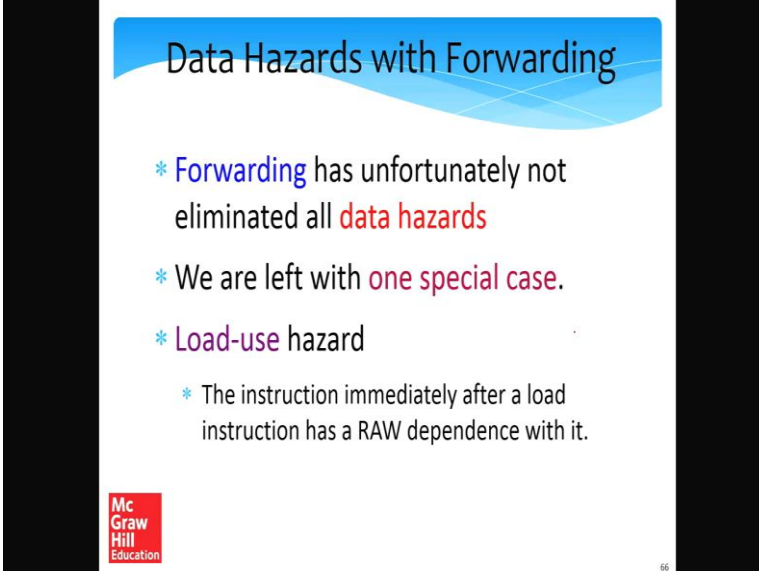
Now, let us take a look at long forwarding cycle from the RW not a cycle, but a chain 3 stage path from RW to OF. So, in this case we load the value into register r1 and again since RW is involved we have a load instruction. We have 2 more intervening instructions which are absolutely independent. So, we do not care about them and then we have a subtract instruction which uses r1 which is the destination of the load as one of its sources.

So, let us again consider cycle number 5. So, in cycle number 5 instruction one the load instruction is done, it has the value of r1 with it instructions 2 and 3 are in the middle of the pipeline. And instruction 4 subtract instruction is in the OF stage. So, my claim is that at this point we strictly instruction 4 does not strictly require the value of r1, but unless we forward it we never be able to forward it or use the right value the reason is that at this point of time if instruction 4 tries to read the register file. It will not be able to get the value of r1 because r1 is being simultaneously written. So, we never want to have that kind of a situation whether simultaneously read and write for the same register in the register file.

So, instead we can use a forwarding path from the RW to the OF stage to get the value of r1. If we do not forward, then in the 6th cycle instruction one is out of the pipeline. So, it is not in a position to forward. So, it is true that at this point at instruction 6 sorry in cycle 6 instruction 4 needs the value of r1 right urgently needs it, but there is nobody to

forward it to that is the reason we forward it early right in a sense we break one of our rules, but we have no choice. So, we forward it in cycle number 5 between the RW stage which has the value and the OF stage.

(Refer Slide Time: 70:50)



The slide features a blue header with the title "Data Hazards with Forwarding". Below the header, there are three main bullet points, each starting with an asterisk. The first bullet point states that forwarding has not eliminated all data hazards. The second bullet point notes that one special case remains. The third bullet point identifies the "Load-use hazard" and provides a sub-bullet point explaining that it occurs when an instruction immediately following a load instruction has a RAW dependence on it. The slide includes the McGraw Hill Education logo in the bottom left corner and a small number "66" in the bottom right corner.

Data Hazards with Forwarding

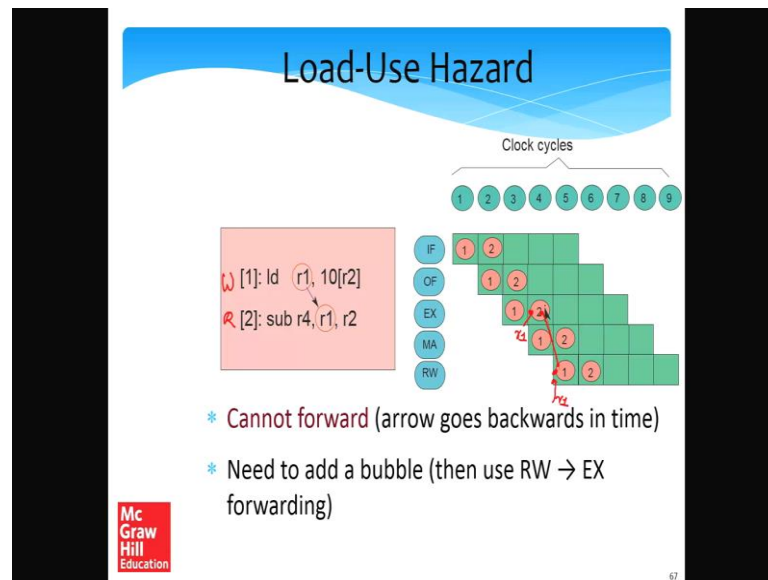
- * Forwarding has unfortunately not eliminated all **data hazards**
- * We are left with **one special case**.
- * **Load-use hazard**
 - * The instruction immediately after a load instruction has a RAW dependence with it.

McGraw Hill Education

66

So, in this case also we have successfully helped avoid stalls. So, let us look at data hazards with forwarding. So, forwarding has eliminated a lot of data hazards right a lot, but unfortunately not all. So, we are left with one special case. So, there is one case which forwarding cannot take care of. This is called the load use hazard and this is something where forwarding will not be useful. So, unfortunately we still have to stall and insert a bubble in the pipeline. So, what is this case this case is when an instruction immediately after a load instruction has a raw dependence raw dependence with it. So, let us see.

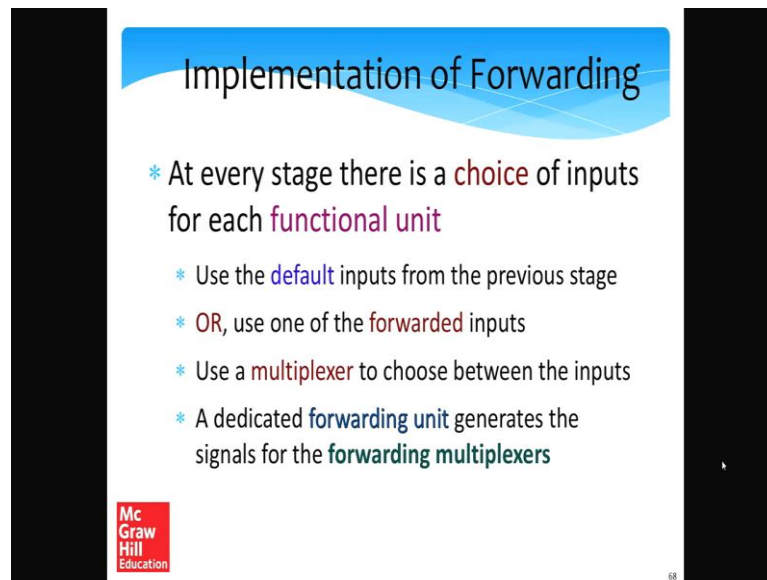
(Refer Slide Time: 71:28)



Let us consider a load instruction whose destination register is r1 as we have been doing. And let us consider an ALU instruction whose source operand is also r1. So, there is a clearly read after write dependence that we see forming over here. So, let us look at cycle number 5 when the value of the r1 will be available you know would just be available with instruction 1, but the sad part is if we consider instruction number 2 in cycle 4 it will be in the EX stage and at the beginning of cycle 4 is when we need the data right is when we need the value of r1 and unfortunately we cannot forward in a sense backward in time right all are forwarding arrows where pretty much in the same column in the same cycle.

So, the sad part here is that in the beginning of cycle 4 is when we need the data. And the beginning of cycle 5 is when we actually have the data. So, we cannot actually forward the value from cycle 5 to cycle 4 because that would be tantamount to going backwards in time which is not possible. So, that is why forwarding in the case of a load use hazard which means that there is a load instruction, which writes to a destination register and the immediately consecutive instruction reads from it. This load use problem cannot be taken care of in forwarding that is because when the data is available in the beginning of the MA stage that is too late, we should have had the data ready earlier which cannot be which is not possible. So, as a result the only solution is that we need to add a bubble in this case and then use a forwarding path otherwise it is not possible.

(Refer Slide Time: 73:40)



Implementation of Forwarding

- * At every stage there is a **choice** of inputs for each **functional unit**
 - * Use the **default** inputs from the previous stage
 - * **OR**, use one of the **forwarded** inputs
 - * Use a **multiplexer** to choose between the inputs
 - * A dedicated **forwarding unit** generates the signals for the **forwarding multiplexers**

McGraw Hill Education

68

So, we will see that at every stage there is a choice of inputs for each functional unit and we need to design our pipeline in such a way such that we can choose between the default inputs and the forwarded inputs. So, certain complexities will be introduced into a small and simple basic pipeline. So, we will discuss this in the next lecture.

In the next lecture we will talk about how to make a processor with forwarding, and then possibly move on to more advanced aspects for you know design processors that can deal with operating systems IO devices and look at performance aspects as well.