

Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Module - 1
Lecture - 18
Basic Pipelining, Branch Prediction

(Refer Slide Time: 00:13)

Data hazards

- Pipelining disturbs the sequential thought-process
 - Data dependencies among instructions start to show up

```
add r1, r2, r3
sub r4, r1, r5
and r6, r1, r7
or  r8, r1, r9
xor r10, r1, r11
```

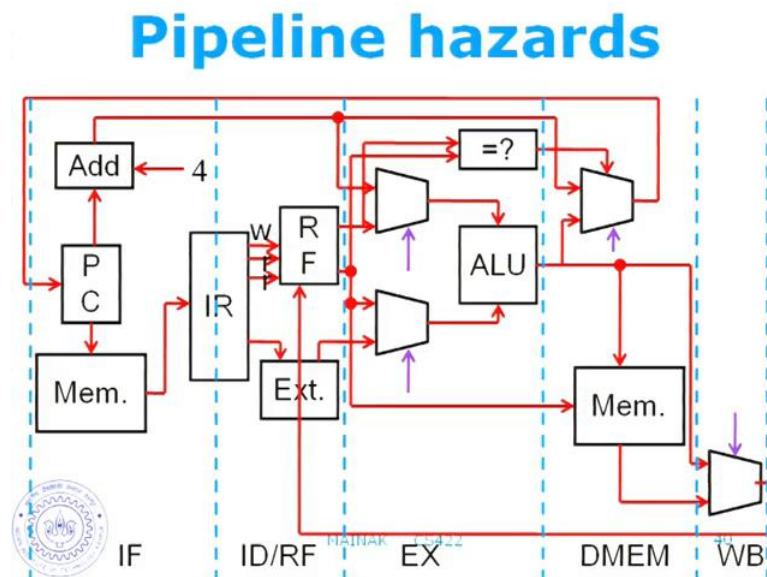
– Result of add is needed by all instructions (RAW hazard)

add	IF	ID	EX	MEM	WB				
sub		IF	ID	EX	MEM	WB			
and			IF	ID	EX	MEM	WB		
or				IF	ID	EX	MEM	WB	
xor					IF	ID	EX	MEM	WB

MAINAK CS422 78

And if you compare with respect to this particular point in time, you will find that all these instructions are actually using the value afterward. For examples how needs the value here, to the, and instruction needs a value here, and odd instruction needs a value here. So, it is quite possible to forward the value produced here to these instructions. You just need some extra hard work, and that is what is called a bypass (()); for example. Now put up the A L U here, we will get bypass to the input of the A L U in the next, so that the sub instruction can control the value. So, this is what it looks like, logically speaking that the add instruction produces the value here, and forwards it to the sub instruction in the next cycle, and it also forwards the value. So, you remember that the value is getting carried forward to the pipeline registers, until it receives this particular stage.

(Refer Slide Time: 01:53)



So, here in this register the value is still available, which is can forward to this instruction. So, it can consume here. So, the idea here is that you read a wrong value from the register file, the decode R F stage; for example here, but the bypassed value will override, and we will also discuss how to implement it. You will require much of comparators, which will dry the selection of the multiplexes at the input of the A L U, so that you can reside what value to contribute. So, the value in this particular latch, we essentially get bypassed to the input of these 2 multiplexers, or if you want to imagine separately you can also have 2 separate multiplexers, which will actually select between the register file value, and the forwarded value, based on the comparison of a register.

Similarly, you can have an extra multiplexer here, which you select between these value, and the value that is been forwarded from the output of the A L U. And then you have this multiplexer as usual, which will select based on opcode whether to look at the immediate, or to get the next p c of a branch. So, that is the basic idea for bypassing, and then of course, when you want to bypass from here, you do the same thing, only thing is that you will probably require longer value, so that will also come here. So, that now select between this value, and the value will get forward from here. So, now, as you can see your multiplexer is getting. Previously we had to select between this value, and the value can forward from this latch. Now we have one more value to select.


(Refer Slide Time: 03:17)

Data hazards

- How to avoid increasing CPI?
 - Stalling is clearly not acceptable
 - Phased register file solves three-cycle apart RAW
 - Can we forward the correct value just in time?

add	IF	ID	EX	MEM	WB		
sub		IF	ID	EX	MEM	WB	
and			IF	ID	EX	MEM	WB

- Read wrong value in ID/RF, but bypassed value overrides it (how to implement it?)
- Always feed bypassed value to the ALU input
- How many sources in bypass network?
- Do we need to bypass to MEM stage also?



MAINAK CS422

79

So, essentially means that as your source points of bypass increases; number of source points, you will have bigger multiplexers. For example, if you look at these instructions, what you can see between 2 multiplexer, you can have a value forwarded from here, and also a value forwarded from here. You cannot disable that bypass that is always on. Only thing is that I am bringing of course, another value coming from the register file. Out of these three values, you may select the correct one based on register identifier. What identifiers will compare here. So, this particular instruction, this instruction, will essentially take R 1 and R 7. These are the sources. It will compare against, what will compare against. You can compare against the destination of these instruction, which does not match. So, in which case it will discard these bypass; that is discarded, but you will find that this matches with this one. So, we will actually consume this particular bypass value. So, as you can see what it means is that, as your span of bypass network increases, we will also require more and more comparisons to get.

(Refer Slide Time: 00:13)

Data hazards

- Pipelining disturbs the sequential thought-process
 - Data dependencies among instructions start to show up

```
add r1, r2, r3
sub r4, r1, r5
and r6, r1, r7
or  r8, r1, r9
xor r10, r1, r11
```

– Result of add is needed by all instructions (RAW hazard)

add	IF	ID	EX	MEM	WB				
sub		IF	ID	EX	MEM	WB			
and			IF	ID	EX	MEM	WB		
or				IF	ID	EX	MEM	WB	
xor					IF	ID	EX	MEM	WB

HAINAK CS422 78

For example here in this case both of these will be compared against both of these, and of course, only one of them will pass. So, any question on this, is the basic concept here. So, we will also make sure one thing that is about of phased register file, and that helps you with the resolving this or instructions bypass. So, here the point is that, the value is written here in this particular cycle, and the or instruction reads the value in exactly the same cycle to the register file. if you do not take any extra care of course, the value that is rate by this particular instruction, will be known, you guarantee it to be known, because the value that will that is written here in this particular cycle will appear in the next cycle at the output of the register; that is how the registers to be done. So, what we are saying is that. Well if we can squeeze the register write, it only half the cycle, that is a positive; the first half of the cycle. and we deal from the register file only in the negative; the second half of the cycle, does this problem goes away, because now R 1 will be written in the first half of the cycle, and I will read R 1 in the second half of the cycle and then I will get the back.

So, that makes sure that this or instruction does not require any bypass, it will take get the right value from register file itself. So, of course, the mission here is that, even your processor frequency, you can squeeze register now in half cycle and register read in half cycle, otherwise it is not possible. Otherwise what will you do? Well, we have to enable another by pass. Well this instruction will actually forward it to the in execution stage, from the write back stage. Just one small, that is whenever you say that. So, this is the

end of the pipeline. So, there is probably known latch here, or know write pipeline register here. So, question is when I say that well I really do not have this phase register file write and read. So, register write will actually take the two side, then where do you actually bypass it from, how do you how do you accomplish this bypass, what is the meaning of this bypass, where it is coming from. There is no pipeline register here to bypass. The register file... The register file will have the value that is. So, we did not get the value from (()).

(Refer Slide Time: 00:13)

Data hazards

- Pipelining disturbs the sequential thought-process
 - Data dependencies among instructions start to show up

```

add r1, r2, r3
sub r4, r1, r5
and r6, r1, r7
or  r8, r1, r9
xor r10, r1, r11
  
```

– Result of add is needed by all instructions (RAW hazard)

add	IF	ID	EX	MEM	WB				
sub		IF	ID	EX	MEM	WB			
and			IF	ID	EX	MEM	WB		
or				IF	ID	EX	MEM	WB	
xor					IF	ID	EX	MEM	WB

MAINAK CS422 78

So, you will need a bypass from the register file itself. So, that is the important point here; that is indeed a bypass path in the register itself, which you forward the value to this particular write; that is the important.

Student: But we would have to identify which register are we going to.

All registers will have...

Student: but there should be some selector register (()).

No your question is very relevant. So, we have 32 registers here, which value to source from; that is what you are asking essentially yes. So, essentially we have to remember what value was just written in the previous angle. So, essentially it will affect you have to introduce the pipeline register, which will hold the value from one more extra cycle, which was written in the last cycle. So, in this case r 1 will be held in a special register,

which was written it in the last cycle, and what we used to bypass to this particular stage. This has one register which we always hold, the register that is written to in the last cycle.

Student: also read the destination of the r, need the instruction for the r instruction to correct the comparator.

The opcode.

Student: Opcode and destination register.

Destination is needed, but why you need the opcode.

Student: we do not need the opcode, we need the destination.

To guide the comparator.

(Refer Slide Time: 00:13)


Data hazards

- Pipelining disturbs the sequential thought-process
 - Data dependencies among instructions start to show up

```
add r1, r2, r3
sub r4, r1, r5
and r6, r1, r7
or  r8, r1, r9
xor r10, r1, r11
```

– Result of add is needed by all instructions (RAW hazard)

add	IF	ID	EX	MEM	WB				
sub		IF	ID	EX	MEM	WB			
and			IF	ID	EX	MEM	WB		
or				IF	ID	EX	MEM	WB	
xor					IF	ID	EX	MEM	WB

 MAINAK CS422 78

So, that is what I have said. If you look at this particular instruction, then compare r1 r7 in against r4 r1.

Student: No, but r1 it is actually.

Why do you get this form you are saying; yes. So, that is also in the latch. Up to this point it seem a pipeline register, using each that to write to the register file, but beyond

this point this is not there; that is what you see, so that is what you have to reMember in a register i d. So, here you require a special register, which you hold a value in the register identifier, which will be used here to activate the bypass. And what have we done, in effect you have increased one more bypass path to this multiplexer. Now we have a bypass from here to here, here to here, and here to here, and of course, one path coming from the register file itself that is here.

So, out of these 4 we will select 1, based on what. So, this instruction will compare r 1 r 9 against r 6 r 4 r 1. And this one will match with this 1, so this one will actually getting overriding. This value will come directly from the register file is not it. What we have a longer pipeline here. We require more bypass processor, we require a much bigger multiplexer. So, remember this implication that if you make your pipeline deeper, there are two things that is going to happen; one, your branch field is very increased that we what we discussed; that is if you increase your pipeline depth on this side of the pipeline, before the branch executes. And after the execution stage if we increase the pipeline depth, you will make your bypass network more and more complicated.


(Refer Slide Time: 03:17)

Data hazards

- How to avoid increasing CPI?
 - Stalling is clearly not acceptable
 - Phased register file solves three-cycle apart RAW
 - Can we forward the correct value just in time?

add	IF	ID	EX	MEM	WB
sub	IF	ID	EX	MEM	WB
and	IF	ID	EX	MEM	WB

- Read wrong value in ID/RF, but bypassed value overrides it (how to implement it?)
- Always feed bypassed value to the ALU input
- How many sources in bypass network?
- Do we need to bypass to MEM stage also?


MAINAK CS422
79

It will require more and more space, it will require bigger multiplexers, it will probably slow down your processor. So, we used this particular acronym, to identify these hazards. These are called after write hazards, why, because you are reading after this register is 2. So, you look at other times of hazard that issue. So, phase register file

solves 3 cycle apart raw, otherwise you require one more bypass path (()) here, and we always feed bypassed values to the A L U input. How many sources in the bypass network. So, in this case, if you imagine that your register file is phased, you have just how many 2 sources ; one coming from execution, one coming from here , so there are 2 sources of bypass. And till now we have only seen bypasses, going to the execution stage only. The destination that we have seen so far, it is only differing a bypass after the execution stage. So, the question that is asked here, is doing it a bypass to the Mem stage also. Can you think of an example, where there would be a bypass path, terminating in the main stage; that is the next stage needs some value coming from somewhere else. So, what are the values that a main stage requires. What are the inputs to the next stage. There are 2 things that it requires. What are they. A Memory needs 2 things what are they.

Student: Address and size.

Size; yes your size is also one thing, and value, what kind of instructions require value; source. So, the Memory needs an address for sure. Somebody has mentioned that it needs a size also, but; however, the size is not important here, because it comes from the instruction opcode. So, this is the dependency that size is not generally brought by somebody else, but address made by another instruction, and of course, there is a value for 4 instructions, we make it generated by the other instruction.

(Refer Slide Time: 13:17)

Data hazards

- MEM to MEM bypass

```
add r1, r2, r3
lw  r4, 0(r1)
sw  r4, 20(r1)
```

```
add    IF ID EX MEM WB
lw     IF ID EX MEM WB
sw     IF ID EX MEM WB
```

- How many destinations in bypass network?
- Overall complexity (i.e. number of MUXes and wires)?
- Yet another drawback of deep pipelining!



So, there are two things that we need for the Memory module. Now can you think about the dependence. So, there is a Memory instruction, which needs its address and value, and can you think of an example where one of these, both of these getting generated by another instruction. So, here in the example for that. We have an add instruction here which adds $r_2 + r_3$ and put the result in r_1 . Then there is a load instruction which uses r_1 to compute the address, and push the value in r_4 form. And immediately after that it stores r_4 back to some other address. So, actually I am copying a value from one part of Memory to another part of Memory. So, let us see how the pipeline line will work in this case. So, here if I add pipeline. So, remember that time goes in this direction. The load instruction requires r_1 getting generated by the add instruction. So, there is a bypass for that will be activated here, so r_1 will become new from this instruction in this bypass. And otherwise the load does not require anything else, it will come to 5. The store instruction have a double dependence. It needs r_1 form add to compute the address, and it needs the value from load to get at the store value.

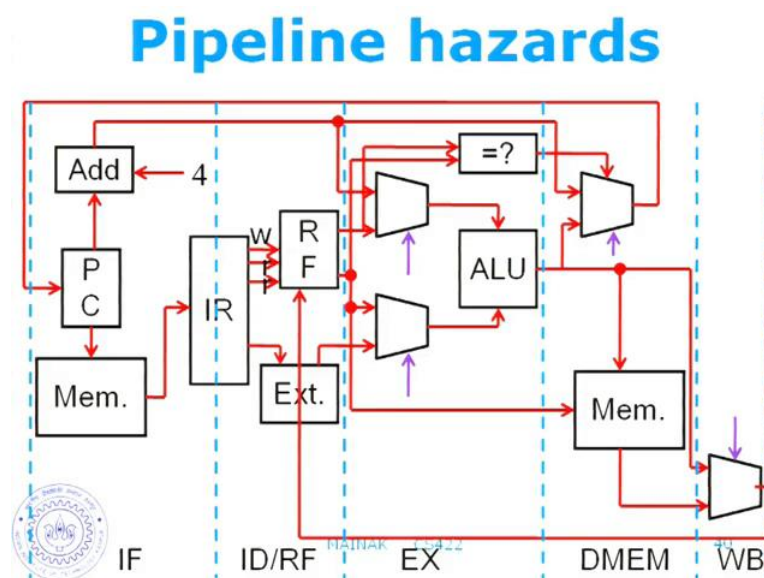
So, let us see how it works. So, the Mem to ex bypass for this instruction make it activated, to pass the value of r_1 . So, r_1 is needed here to compute the address. So, remember that, the Memory operations compute addressing the ex state. And the load instruction will bypass the value from here to here for the store. So, it is getting generated here, you can bypass it here. This is an example where you (()). So, is there any other situation, where a main stage makes to have the bypass value, any other case. So, here is a general question that you might think about; that will help you answer many of this question that is. Suppose I have a pipeline, and your focusing on a particular sting of pipeline, stages, and you are asking tell me what is the 5 stages that can bypass values to the stages.

What can you say about those things that the bypassing value to stages. Will they come before stages or after stages. For example here, if I look at the ex pipeline stage, will I require or let us suppose, let us look at Mem pipelines actually. Will I ever require a bypass, coming from the ex stage. I think you know, can you see why, because ex always appears before Mem. There is no way, there is no reason why I need to bypass actually, because if you layout the pipeline here, you will always find that ex appears before Mem. So, there is no way I need a bypass from ex to Mem, because whatever currently in ex, cannot. it has to after the way instruction; instruction that is currently Mem. Let us

suppose this load instruction, why it is in the MEM stage. Instruction that is currently in ex, is actually after it. So, bypass always transform previous instructions, cannot come from the next instructions, that is impossible.

So, whenever you are looking at a bypass, trying to resolve why are the possible sources of by bypass, you can only come from stage is after, it cannot come from stage before that is impossible actually, and of course, it will come from myself, that is possible here Mem to Mem bypass; that is possible. So, in this case since the only stage after Mem is lies back. There could be a possibility of having a bypass transform write back and Mem. We will come to that very soon that particular possibility. So, how many destinations in the bypass network, so in this case we have 2 ex and Mem; there are two destinations. So, what is the number of multiplexers that you require. So, whatever number of destinations you have, whatever number of sources you have, that (()), and of course, you have to look at how many inputs are there. So, in this case for example, you will require ex to ex, these bypass will require two multiplexers, because there are two inputs to the execution pipeline stage.

(Refer Slide Time: 01:53)



And there is a Mem to ex also. So, there will be two multiplexers, each accepting two inputs from bypass, and one from a register file .If I ignore the write back to ex bypass for now. And the Mem stage will have just one bypass part for now. We will come back to write back to Mem bypass very soon, and there will be one multiplexer. So, that is in

between this bypass value, and the value coming in from the execution stage. So, just to show you the diagram on the board. So, this is the store value, coming from the register file here. So, you will be selecting the store value between this, and the value that is getting bypass form the output of the Memory stage, where is my; this value, this value you can bypass here.

(Refer Slide Time: 13:17)

Data hazards


- MEM to MEM bypass

```

add r1, r2, r3
lw  r4, 0(r1)
sw  r4, 20(r1)
  
```

add	IF	ID	EX	MEM	WB	
lw		IF	ID	EX	MEM	WB
sw			IF	ID	EX	MEM WB

– How many destinations in bypass network?
 – Overall complexity (i.e. number of MUXes and wires)?
 – Yet another drawback of deep pipelining!


HAINAK CS422
30

And we are selecting to this and this in a register cycle, based on the comparison of the register identifiers. In this case I will be comparing r 4 by the destination of this 1, and though they match, so I will say I have to take the bypass now. So, as you make the pipeline deeper, you need more and more bypass pass.

(Refer Slide Time: 18:47)

Data hazards

- Can we always avoid stalling?

```
lw r1, 0(r2)
sub r4, r1, r5
and r6, r1, r7
or r8, r1, r9
```

lw	IF	ID	EX	MEM	WB			
sub		IF	ID	EX	MEM	WB		
and			IF	ID	EX	MEM	WB	
or				IF	ID	EX	MEM	WB

- Need some time travel (backwards)! Not yet feasible!!

- Hardware *pipeline interlock* to stall the *sub* by a cycle

- Early generations of MIPS (Microprocessor without Interlocked Pipe Stages) had the compiler to fill the *load delay slot* with something independent or a NOP 31



So, now the question is that, you know till now we have seen that in all the examples, even though we do not have the value getting the register file, will avoid all stalls by bypassing. There is no bubble in any of this example. We have bypassed the values exactly in time. No bubble here, no bubble here. There the question is, always possible. The answer is no, there will be stalls. So, here is an example for that. So, this is a typical scenario where you will require a bubble. And instruction here consumes the value that is loaded. So, whenever you find the value that is getting produced by the stage ex, and needed by the next instruction, before stage ex. Next instruction recalls the value in the ex stage; that is not possible bypass that is impossible actually, why is that. So, this is how it is pictorially. So, this is where the value is produced, and this instruction needs a value here; that is impossible, that is a negative in time; however, this and instruction is, because it will require the ex value the value here, that will get bypassed. So, in this case you really have no choice, but to stall the sub instruction from one side.

We have to delay its execution by one cycle, and that has to happen there, when the value can be bypassed. So, these are normally called, the technical term is pipeline interlock. So, these are actually stall cycles. So, you need a hardware pipeline interlock to stall the sub by a cycle in sub instruction. So, this brings us to the acronym MIPS. So, now, I can explain what it stands for. It stands for microprocessor without interlocked 5 stages. So, that is what wave stands for. And this is the only problem that we had. So, question now is, the name suggest that there is no interlock cycle. So, there has to be a

reason. Now there has to be a weight to avoid this stall. So, they rely on a compiler. So, they call this particular slot a load delay slot, which they rely on the compiler to be filled by an independent instruction. So, currently compiled MIPS code; the early generations of MIPS, never had a dependent instruction on load in the load delay slot.

So, that reads you of this particular bound, and which is why is called microprocessor without interlock pipe stages. It had the compiler to fill the load delay slot in something independent or a (()). So, what it essentially means that the code that I am showing here, is actually illegal this code. So, of course, the load delay slot to remove later 1, in the later generations of MIPS, but in the earlier generations this was the case. So, this code will never be generated by MIPS compiler. So, for your second assignment, the compiler that is handled out. It actually I mean more advanced, it can generate this piece of code. So, we have to actually implement this particular interlock, keep that. Any question. Sir phase execution not possible in this case. That is independent. Phase execution of the register file, that is independent. No I mean different. It could be possible they did not do it, that is all. So, there must be frequency constants. You can phase every stage if you want, that is possible. Phasing every stage essentially what you are saying is that I am splitting the 5 stage in two halves, but really there is no latch in between, because it is a multi cycle pipe stage; that is what you need actually. Any other question

(Refer Slide Time: 22:53)

Bypass and stall logic

- What does the bypass logic look like?
 - How many forwarding paths?
 - How large are the MUXes?
- What about load interlocks?
 - Detecting possible hazards early simplifies things
 - Fixed positions of rs, rt, rd are important for RF access and hazard detection
 - In MIPS all interlocks can be implemented in ID/RF
 - Need to control IF and EX
 - MIPS R3000 does not have any hardware interlock: compiler fills the load delay slot



• Branch target and *condition* in ID/RF?

So, a little bit more on bypass and stall logic. So, what is the bypass logic look like, how many forwarding paths, how large the muxes, we have already discussed, because you work through your second assignment. these concept will become even more clearer as we would like to get to draw multiplexers on a piece of paper, to figure out what the logic should be, or if you can without drawing if you can follow it in your head; that is better, but any way point is that as you do that, it will become very clear how many of these things require.

(Refer Slide Time: 18:47)


Data hazards

- Can we always avoid stalling?


```
lw   r1, 0(r2)
sub  r4, r1, r5
and  r6, r1, r7
or   r8, r1, r9
```

lw		IF	ID	EX	MEM	WB			
sub			IF	ID	EX	MEM	WB		
and				IF	ID	EX	MEM	WB	
or					IF	ID	EX	MEM	WB

 - Need some time travel (backwards)! Not yet feasible!!
 - Hardware *pipeline interlock* to stall the *sub* by a cycle
 - Early generations of MIPS (Microprocessor without Interlocked Pipe Stages) had the compiler to fill the *load delay slot* with something independent or a NOP




What about load interlocks? So, the question is, where should I detect this interlocks. So, you can see, you can go back to this example. There are many places where I can detect that there is a problem, I can detect it as early as here, as soon as the instruction is detected. I can detect it at the in this stage that you know value is not available, and there is actually a there is a hazard, so that I can stall it here. But of course, these are the two possibilities only in this case, but if you have a deeper pipe there are many other options; however, detecting possible hazards early simplifies things, because you are not too far into the pipeline which means, you do not have too many things behind you, so you can stall the pipe early. And the fixed positions of r s r t and r d, these are the register identifiers form your MIPS encoding, are important for register file access has some hazard detection, because this simplifies matters a lot. Because you know exactly what to compare, which positions of your instruction to compare against. So, in MIPS all interlocks can be implemented in the decode stage itself. So, that is very good, that

makes it, that makes the distribute the all over the pipeline. You can just implement everything in one particular pipe stage.

(Refer Slide Time: 22:53)

Bypass and stall logic

- What does the bypass logic look like?
 - How many forwarding paths?
 - How large are the MUXes?
- What about load interlocks?
 - Detecting possible hazards early simplifies things
 - Fixed positions of rs, rt, rd are important for RF access and hazard detection
 - In MIPS all interlocks can be implemented in ID/RF
 - Need to control IF and EX
 - MIPS R3000 does not have any hardware interlock: compiler fills the load delay slot
- Branch target and *condition* in ID/RF?


MAINAK CS422
82

So, so that make sure that if you implement all the interlock in the decode stage, what you have to do. You have to tell the fetcher to stop that is one thing. Then the second thing you have to fitting NOPs into the forward direction of the pipeline. You are saying all zeroes to the execution stage. You have to keep on doing the, until the interlock resolves.

(Refer Slide Time: 18:47)


Data hazards

- Can we always avoid stalling?


```
lw   r1, 0(r2)
sub  r4, r1, r5
and  r6, r1, r7
or   r8, r1, r9
```

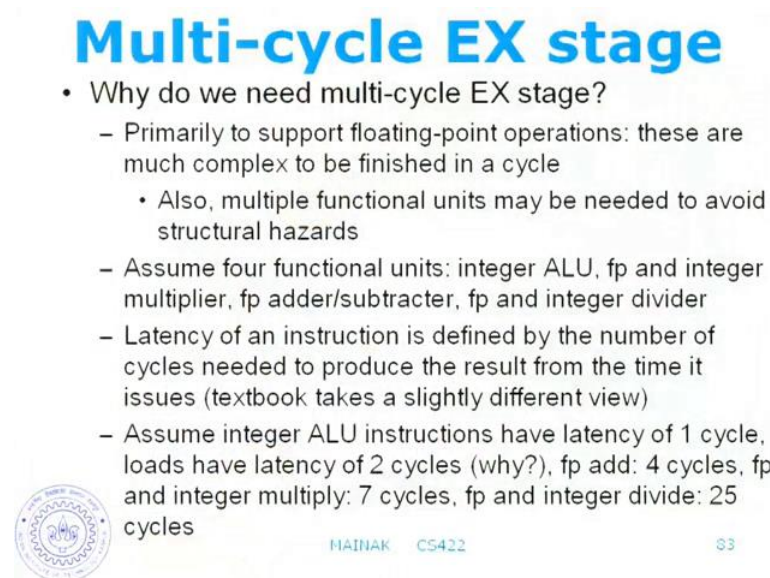
lw		IF	ID	EX	MEM	WB			
sub			IF	ID	EX	MEM	WB		
and				IF	ID	EX	MEM	WB	
or					IF	ID	EX	MEM	WB

 - Need some time travel (backwards)! Not yet feasible!!
 - Hardware *pipeline interlock* to stall the *sub* by a cycle
 - Early generations of MIPS (Microprocessor without Interlocked Pipe Stages) had the compiler to fill the *load delay slot* with something independent or a NOP


81

So, MIPS r3k does not have any hardware interlock, as you have just mentioned, compiler fills load in the slot, but in generations ahead of that, they actually have the load interlock, implemented in the decode stage. So, in this case how we have implemented the interlock; that is very simple actually. As soon as this is decoded, you know the source of this instruction. All we have to do is, we have to check the previous instruction, which is currently sitting where, which is in which latch. Which is in this latch currently sitting here, currently executing here. So, all you have to do is, you have to take the content of this latch, and check two things, is the previous instruction a load instruction, and does the destination of the load match with one of my sources; that is it. if the answer to both is yes; that means, I have to search. So, essentially what I have to do is, I have to tell the fetcher to stall the one cycle, and I have to fitting all zeroes into this particular latch in this cycle. Next cycle onward the proper action will be given. These all we have already discussed branch target in decode r f. You can actually bring the condition execution also in this stage if you want to, that lengthen your cycle little bit, but I can read you of all branch bubbles, but anyway. So, I will not go into that, we have discussed in great detail branches and all.

(Refer Slide Time: 26:50)



Multi-cycle EX stage

- Why do we need multi-cycle EX stage?
 - Primarily to support floating-point operations: these are much complex to be finished in a cycle
 - Also, multiple functional units may be needed to avoid structural hazards
 - Assume four functional units: integer ALU, fp and integer multiplier, fp adder/subtractor, fp and integer divider
 - Latency of an instruction is defined by the number of cycles needed to produce the result from the time it issues (textbook takes a slightly different view)
 - Assume integer ALU instructions have latency of 1 cycle, loads have latency of 2 cycles (why?), fp add: 4 cycles, fp and integer multiply: 7 cycles, fp and integer divide: 25 cycles

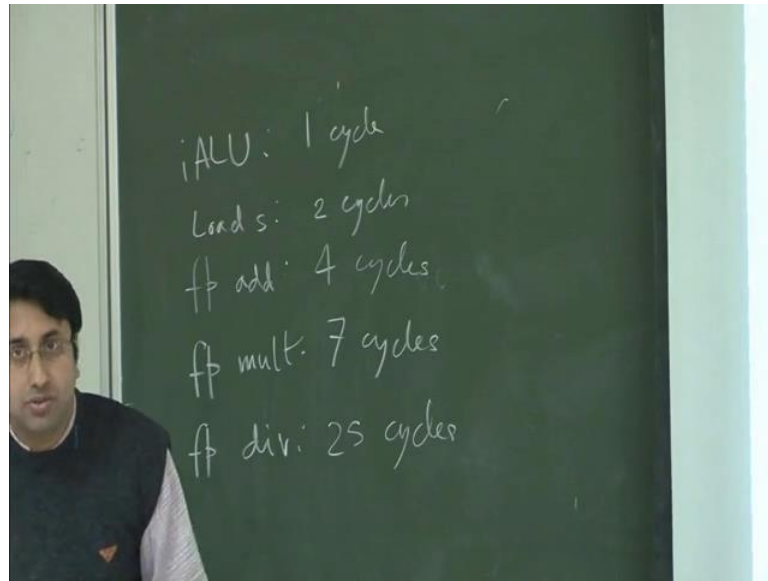
HAINAK CS422 83

So, next thing is, till now we have looked at this simple pipeline that fetch execute decode, and write back. I am sorry fetch decode execute Memory write back, where execution takes a single site, but that is far from what the reality is. There are multi cycle execute stages, and the very reason for that is, to privately support floating point

numbers. These are much complex we finish the cycle. For example if you think about a floating point multiplication, it is impossible to finish in the single cycle. Also multiple functions may be needed to avoid structural hazards. So, if you cannot pipeline these multi cycle function units, you may require multiple of these, to avoid structural hazard, because now essentially what you need to do is, there is a n cycles in a transfer unit ,which is not pipeline. The next operation cannot go in, until this configuration. So, n cycles this will be occupied. So, which means we avoid structural hazard, you may have to have multiple of these zeroes.

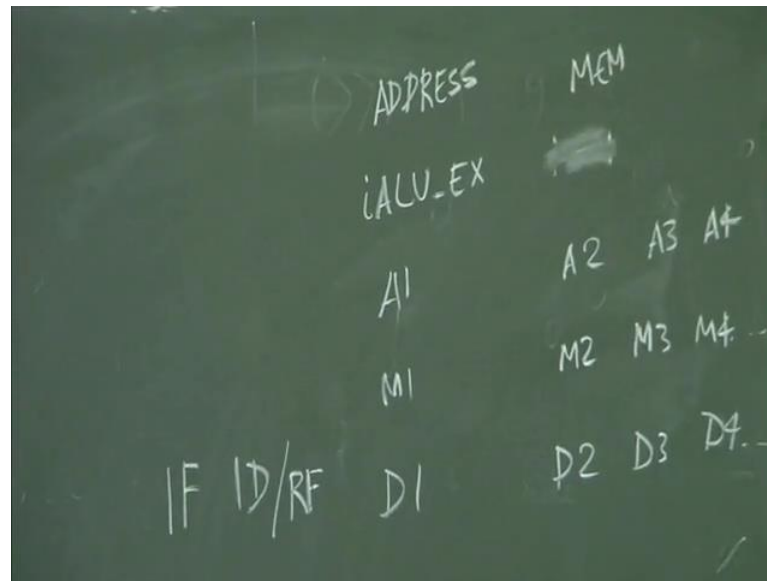
Next instruction can also do to different unit. So, for the following discussion, we will assume that we have 4 function units, which consists of one integer a l u, which we will do all the integer operations, except multiplication and division. We have a floating point in integer multipliers. So, essentially what we do is, we will carry out all of integer multiplications under floating point multiplier, which we can do without any problem. We will have a floating point adherence ad blocker, and we will have a floating point integer divider. So, all integer divisions will be carried out on the floating point divider. Is it clear, the architecture. Now, a latency of an instruction is defined by the number of cycles needed, to produce the result from the timing issues. The textbook defines it in the slightly different way, but we will follow this definition in the latency. So, do not get confused. So, assume that integer A L U instruction have latency of one cycle. So, these are simple operations. So, they can finish the cycle. Loads have a latency of 2 cycles, why is that, because we have already seen that , in execute stage it computes the address, in Memory stage it executes the load, address; that is why it is 2 cycles.

(Refer Slide Time: 29:40)



So, from the time, the instructions, the load issues in the time the value is produced it takes 2 cycles; ex state, and Mem state. Floating point add is 4 cycles, floating point an integer multiplier cycles, and floating point integer divide 25 times, so that is the model of execution . So, maybe I list it here so that you can remember; integer A L U one cycle, all loads floating point integer 2 cycles, floating point add. Well as in floating point add remember that subtractions also have done the same unit. So, 4 cycles and floating point multiply, also integer multiplies same, 7 cycles, and floating point divider 25 cycles. Any question on this particular model of function unit. So, use the multiple slides to demonstrate how multi cycle ex execution state is, make you like much more complicated. So, that is one more term to define, that is repeat interval of an instruction. This is number of cycles between two instructions in the same category, that can execute without a structural hazards. So, for example, number of cycles between two floating point editions, that can execute, is a repeat interval of the floating point address. So, it depends on how the function units are pipelined. So, for example, if the arrow is not pipelined, you will have to wait for 4 cycles before fitting the next instruction.

(Refer Slide Time: 31:48)



So, let us assume that all the units other than the divider are pipeline, except this 1, these are all pipeline. So, division has a repeat interval of 25 cycles, while other instructions can issue back to back, repeat interval of one cycle. So, which means, there would be 7 difference, there could be different multiple of multiplication operations in the pipeline of the multiplier, that in a point time. So, what is the pipeline look like. So, I have a peg stage, I have a decode register file stage, and then I have different pipe stages. So, I will put it as i A L U ex 1 cycle, then I load pipeline which is just ex and Mem. Then I have a floating point adder which has 4 cycles. So, I will say a 1 a 2 a 3 a 4. And similarly I will have m 1 m 2 m 3 m 4 m 5 etcetera to m 7 and divider will have d 1 d 2 and d 25, and then n I have a write back stage. So, all the instructions between the values write back and (). So, this looks very different, somewhat previous architecture.

Now I have. So, let me look at maybe I will put. So, I know 4 different pipelines is going on, and even for the load instruction if I want I could put the separate a l u, to compute the address, maybe I will do that, hence an address stage, and a Memory stage. So, there are quite different pipeline that I have though. Then all get diverged from the decoder, some depending on the decoder outcome I will decide each pipe to send the instruction to, and finally you write back in the register file. So, here you require more pipeline latches or pipeline registers from here. So, between these stages you require pipeline registers, any other complications. So, that is all about the related slides. So, let us start with the simple 1; structural hazards. So, divider is not pipeline, that poses a big


problem, because if you have two division operation, which are separated by less than 25 cycles you will have a problem. The second instruction has to be solved, and then I can decide with the decoder itself. I want to implement all my internals in the decode stage.

(Refer Slide Time: 34:45)

New hazards

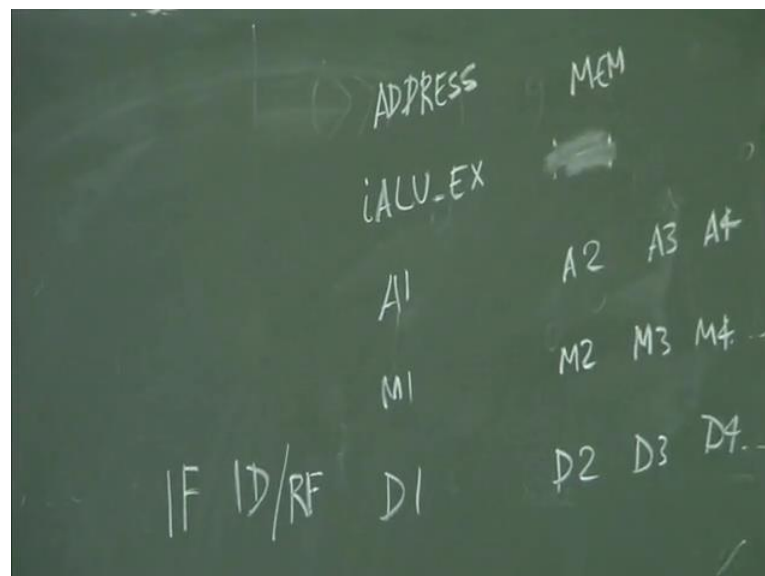
- Structural hazards
 - Divider: stall instruction issue in ID/RF
 - Any suggestion for CPI improvement? (other than pipelined divider)
 - Floating-point register write ports
mult.d, ..., ..., add.d, ..., ..., load.d
 - More write ports or hardware interlock?
 - Interlock options: detect in ID (shift register write port scheduler), detect in MEM or WB (stall which one?)
- More stalls due to RAW data hazard

load.d	\$f4, 0(\$2)
mult.d	\$f0, \$f7, \$f6
add.d	\$f2, \$f0, \$f4
store.d	\$f2, 0(\$2)


MAINAK CS422
85

So, whenever I have a division instruction I will check if, currently there is anything in the divider; that is it can be easily maintained here. So, that I can stall the instruction. So, clearly that is going to introduce bubbles in my pipeline. So, a slightly smarter thing that I could have done here, is that I can only stall the instructions.

(Refer Slide Time: 31:48)




So, here essentially. See there are two things you can do; one is that whenever you put a divider here, you say that a 25 cycles I am not going to issue anything else, because the register file will be written up to 25 cycles. The other thing that you can do is you can say well I will continue issuing instructions, only when I get the next division, I will check and then stall. So, that is slightly better. In fact, much better than this because the here essentially what we are trying to exploit, is that we have division operation not that frequently. So, you have a lot more instructions to put into these 4 pipelines, why the divider operates. Then of course, next your much more complicated, because now let me very careful while fitting this instructions into this pipelines. You can make sure that they are independent of the running division operation.

(Refer Slide Time: 34:45)

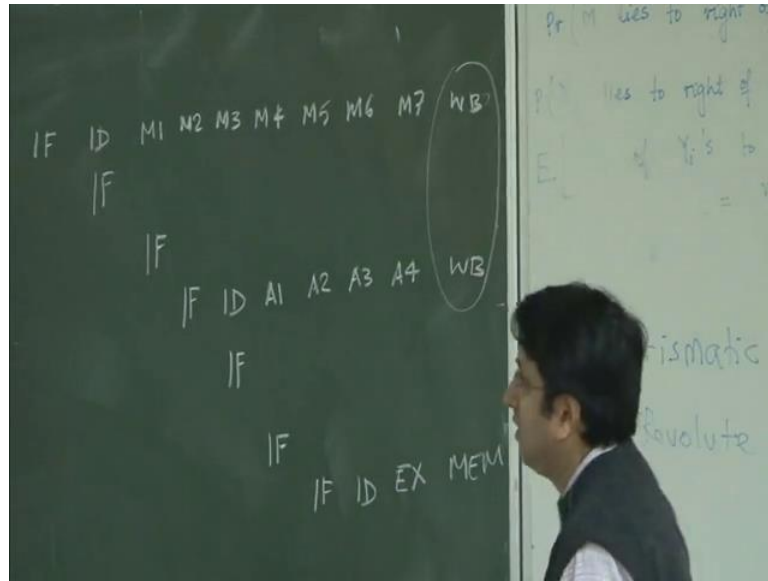
New hazards

- Structural hazards
 - Divider: stall instruction issue in ID/RF
 - Any suggestion for CPI improvement? (other than pipelined divider)
 - Floating-point register write ports
mult.d, ..., ..., add.d, ..., ..., load.d
 - More write ports or hardware interlock?
 - Interlock options: detect in ID (shift register write port scheduler), detect in MEM or WB (stall which one?)
- More stalls due to RAW data hazard

load.d	\$f4, 0(\$2)
mult.d	\$f0, \$f7, \$f6
add.d	\$f2, \$f0, \$f4
store.d	\$f2, 0(\$2)


MAINAK CS422
85

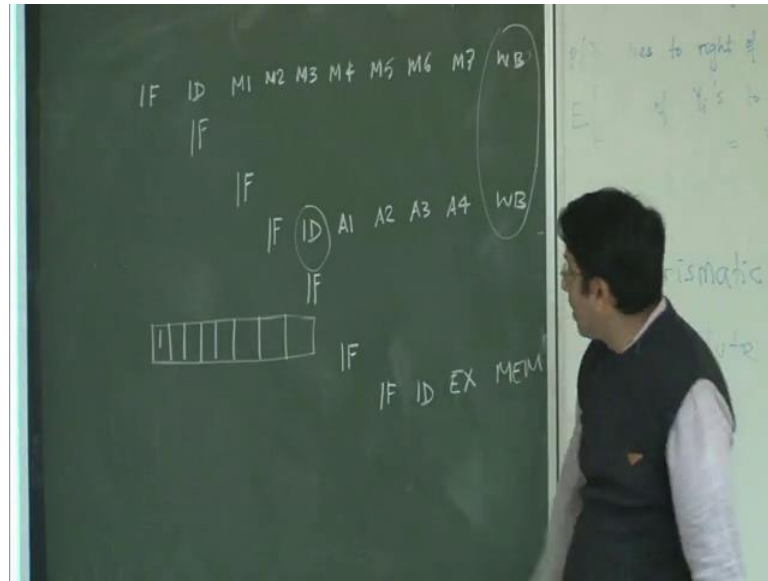
(Refer Slide Time: 36:07)



So, that is one problem about the divider. The second structure hazard arises between register write forces. So, here is an example. So, let us take that example and see what happens actually. So, I have a multiplication instruction. So, how does it execute, fetch, decode m 1 m 2 m 3 m 4 m 5 m 6 m 7 write back. There is the multiplication instruction. Then I have two more instructions which I do not care. We can fetch here, there are one add instruction. So, let us see how it executes, fetch decode a 1 a 2 a 3 a 4 write back. Then I have couple of more instructions; I think I have a load instruction, which has fetch decode ex, and then you have a write back stage there. The problem is with these two instructions; they try to access the register file in the same cycle. They call the write code in the same cycle, they may be (()) that registers, but many to separate write codes to give. So, that is another structural hazard, that comes up because of this different latency in the instructions.

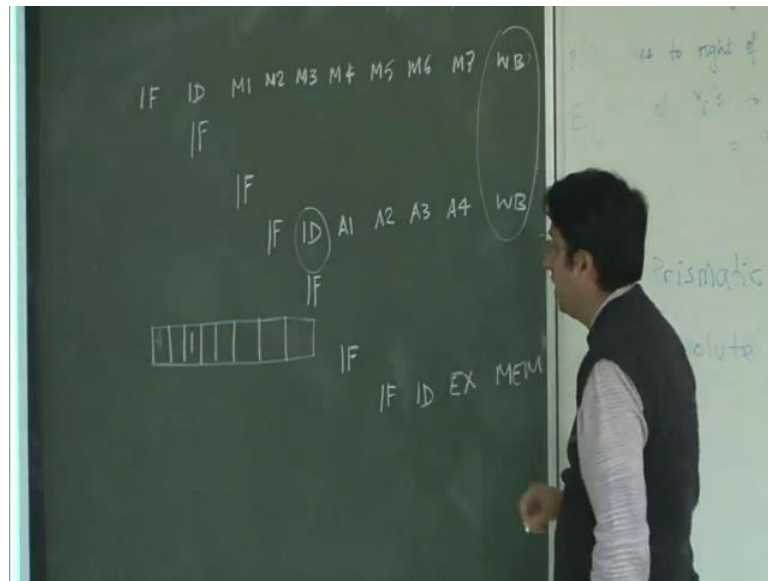
So, you have to be careful when scheduling the register. Make sure that, there is no clash in the write back stage, because unless you have no codes, you cannot really allow this to happen. So, here the solution, you may either have write ports, or you have to introduce hardware interlocks, which means stall cycles.

(Refer Slide Time: 38:29)



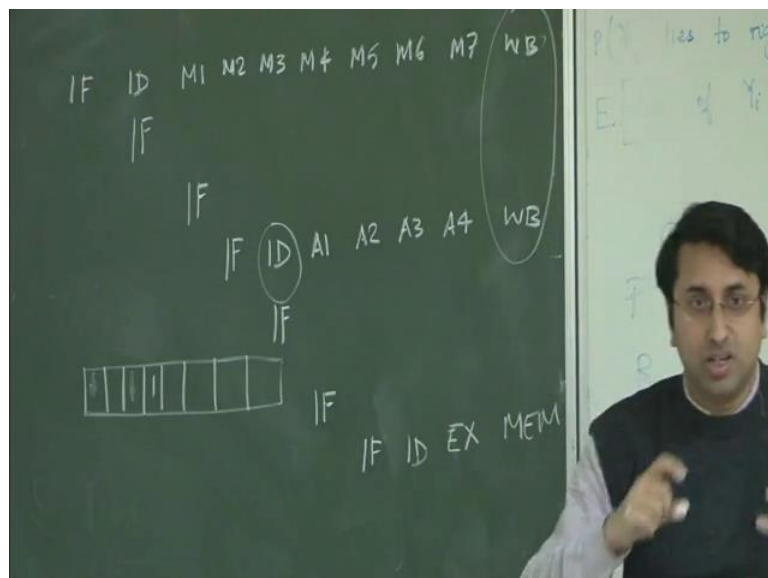
So, where would you introduce this interlock, this particular register write code interlock. So, what are the options; able to detect write here, that there is going to be a clash in the register write ports, when this instruction goes to, because I know exactly the latency of this, I know exactly the latency of this. So, I can calculate actually the decoder, sitting here that whether contains register write code or not. So, usually this is implemented in the ship register write port. So, essentially what you will do is, you make it a ship register. So, when the when this multiplication operation issues after the decode, you calculate how many cycles hence it will actually, access the register file. So, these are essentially bits in my ship register, they corresponds the cycle, so 1 2 3 4 5 6 7 8. So, in this cycle m 1 will execute, this cycle m 2 m 3 m 4 m 5 m 6 m 7 write back. So, I will mark up one here, in the write in the register, in the ship register.

(Refer Slide Time: 39:16)



So, when this instruction goes. So, every cycle I decode a new instruction I will ship this register on this cycle by one position. So, by the time I decode this instruction, this bits has moved here. So, this instruction is bit moves here, this instruction this bit moves here. So, when I decode this instruction. So, this bit is sitting here at this point. So, now, I calculate, for this one this is a 1 a 2 a 3 a 4 write back. I move this bit 1, one position where I decode this, and I find out there is a clash.

(Refer Slide Time: 39:56)



In the same slot, this instruction is going to be access the register file, and this instruction will also going to have. So, whenever there is a clash, I introduce an interlock. So, when let by 1 cycle to this interlock. So, this bit will move on by 1 cycle, and then when this guy goes, it will actually follow it. So, there will be 2 1. Subsequent instructions will check both of it. So, essentially what I am doing is, whenever I encounter a new instruction I prepare and ask, with exactly 1 bit 1 i, and it with this particular instructor. If the output of and is nonzero I know that there is a clash, I will have to stall the current instruction. How many cycles I may stall. So, we have to check by stalling by each cycle, to see that with the things until I get a 0, outcome from the and operation. So, that is one option, I can detect in the decoder. The second option is, I will let it go, and detect it in the Memory stage, or the write back stage.

So, essentially Memory write back in this case is the last stage. So, essentially I will say well I will delay until I reach that, and then I will detect, we just fine also. The only problem is, we have to introduce stalls, you may have to introduce stalls to multiple different pipelines. Here I am showing only two instructions, that have a clash, there may be more. For example, if I did not have this instruction here, if I have the load here, this will also clash like for example, if this instruction was a load. This will actually have a write back here. So, you may have to stall multiple pipelines, if you give a (()) pipeline.

(Refer Slide Time: 34:45)

New hazards

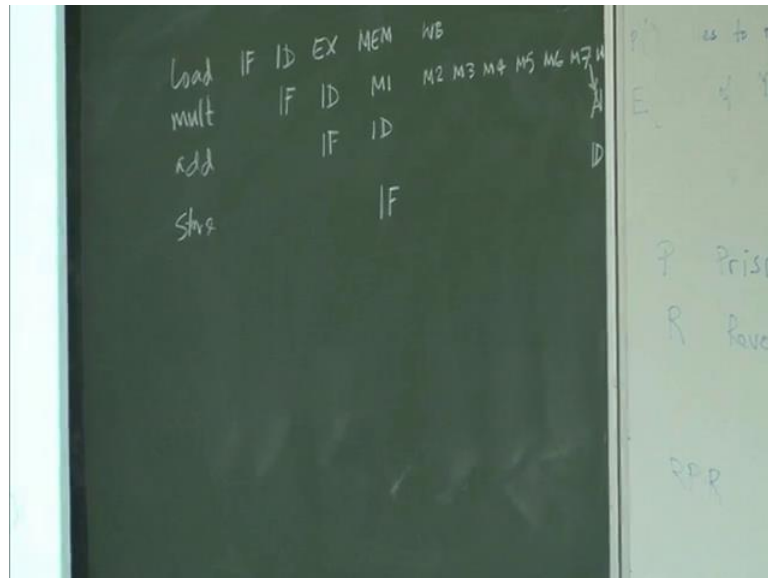
- Structural hazards
 - Divider: stall instruction issue in ID/RF
 - Any suggestion for CPI improvement? (other than pipelined divider)
 - Floating-point register write ports
mult.d, ..., ..., add.d, ..., ..., load.d
 - More write ports or hardware interlock?
 - Interlock options: detect in ID (shift register write port scheduler), detect in MEM or WB (stall which one?)
- More stalls due to RAW data hazard

load.d	\$f4, 0(\$2)
mult.d	\$f0, \$f7, \$f6
add.d	\$f2, \$f0, \$f4
store.d	\$f2, 0(\$2)



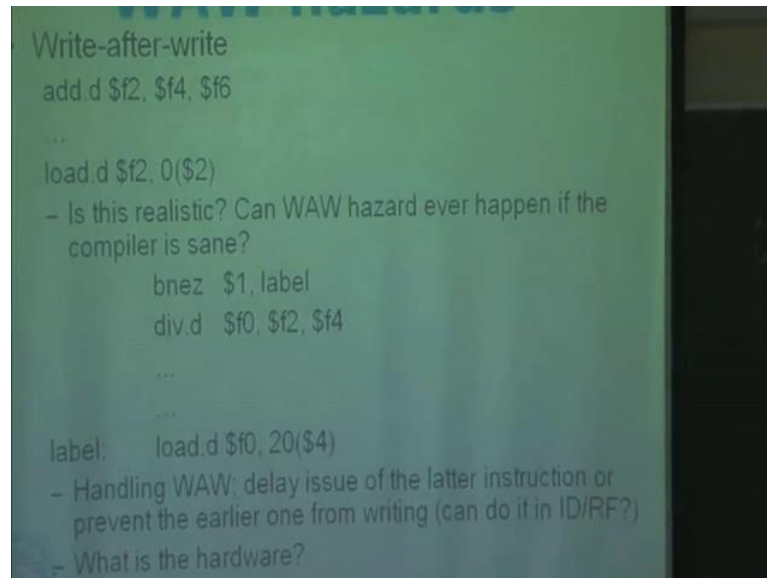
Stall which 1. So, I have these two instructions many more which one should I stall. Everybody accept the first one. The oldest instruction will be an, others will be stalled. Is this clear to everybody. This two structure hazards; one involving register write port and other involving divider. So, any non-pipeline unit, we will have a hazard. And your write ports may have a hazard depending on resolve the scheduling. You will have new raw hazards. So, here is one more example lets what.

(Refer Slide Time: 42:25)

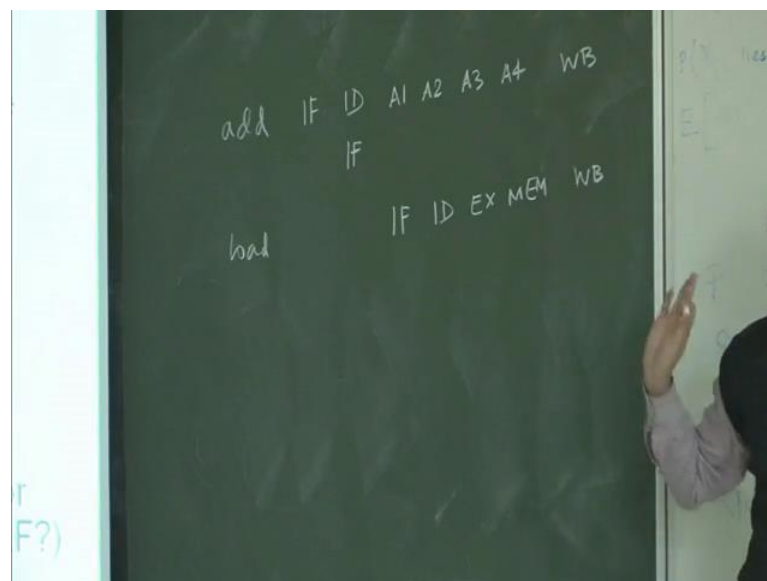


So, by looking at this you can see that. Let us see what are the dependencies this, add instruction consumes the result of the multiplication, and it also consumes the result of the node. So, both of the sources depending on two previous instructions. So, this is my load, fetch decode execute Memory write back ,traditional pipeline, mult fetch decode. does it have any dependency, no; m 1 m 2 m 3 m 4 m 5 m 6 m 7 write back. Add fetch decode when looking at 0. So, it is produced only here. You have long stall many, cannot start a 1 until this cycle, where you can actually have a bypass m 7 to m 1 ; however, the load instruction will complete by then, so there is no problem as such. So, you can have the value bypassed, you know in this particular cycle.

(Refer Slide Time: 44:22)



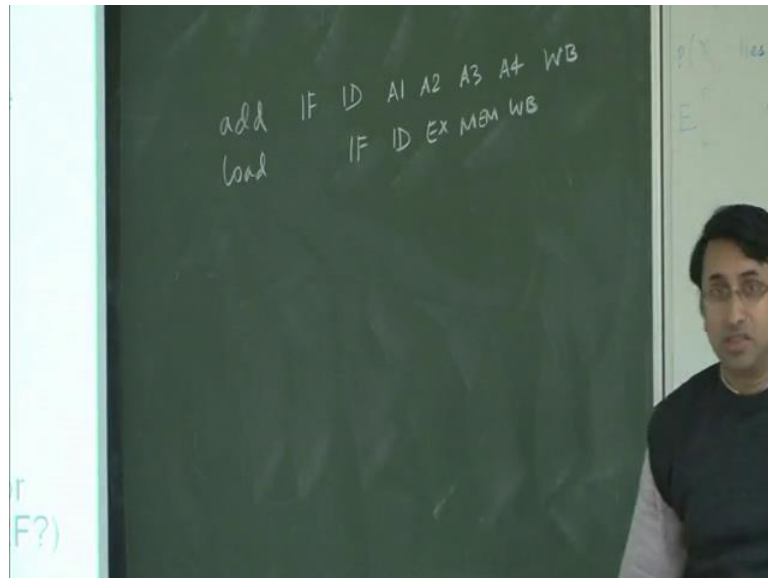
(Refer Slide Time: 44:25)



And then I have the stall fetch, this is stall cannot do anything. You can only decode here and you continue that. So, you have now more hazards, that would take care. Now there is a new type of hazard, which is writing after write. So, if you have an add instruction, which produces dollar f 2, you have some instruction and then you have a load instruction which also produces dollar f 2. So, let us try and see what happens in the pipeline. I think I have been skipping the Mem stage altogether here that cannot be skipped. Then I have the load. So, fetch decode ex Mem stage actually skips, so there is now problem. So, you can see that there is a problem. In this case, both the instructions

are writing into the same register in the same cycle. So, if you already implemented in the write code scheduler interlock, we will have resolve this. So, this instruction will essentially get delayed by 1 cycle, the decoder I will move I will write to the register file with no problem. Then I move the node instruction cycle I have.

(Refer Slide Time: 45:54)



So, there is no write code scheduling clash , but there is a problem, what is that.

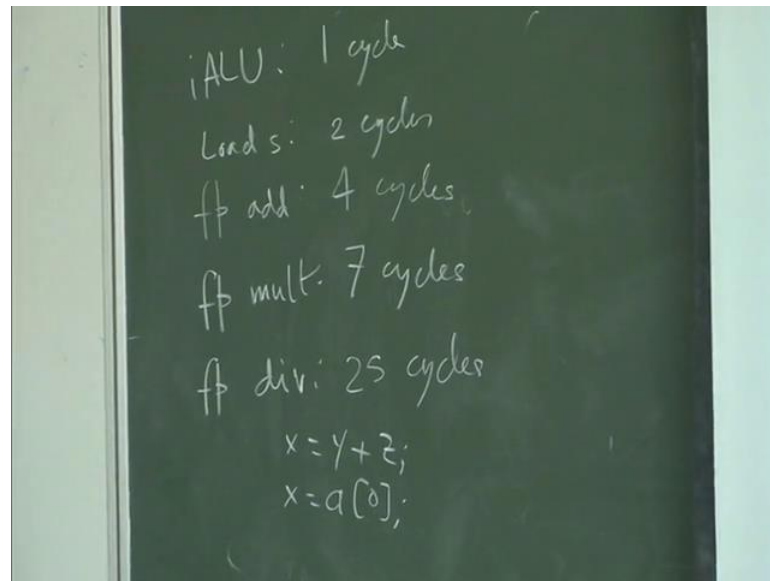
Student: (())

Exactly. So, what is the final value of dollar f 2 that will survive.

Student: (())

Add instruction, which is not correct. The dollar f 2 should have the value of the load instruction at the end of this sequence of instructions. So, this is called the write after write hazard. We have the same destination, and the earlier instruction is writing greater than the next instruction. Of course, this is a problem, but before that let us try to ask the question that, is this realistic at all.

(Refer Slide Time: 47:24)



Can this really happen that there are back to back two instructions, they write to the same register. What is the meaning of this? Do you think about a program? I do an edition operation, write to variable ex. In a very next instruction, I say ex equal to some value, so this can be equivalent to a c decode which says ex equal to y plus z, and then I say ex equal to a 0.

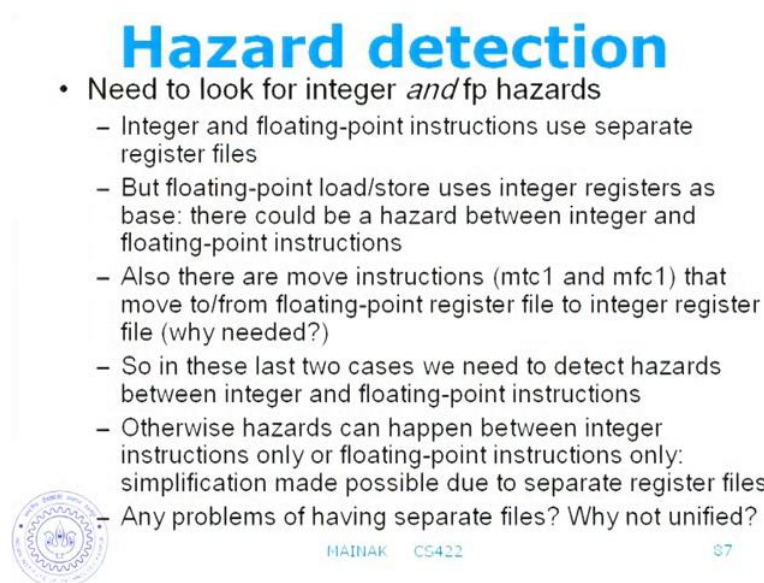
Why did I do this array operation actually? I probably use that result of that, that is make sense. So, question really is that, can this really happen, and here is a code which is straight from you know MIPS compiler. So, of course, there could not be any other situation where this can happen.

For example, if you instead of doing an addition multiplication here. We will have a much longer pipeline, and even if you have put a few instructions in between, before the load will have the same problem. So, here what happens is that, the compiler build in the branch delay slot, to an instruction you target f0. So, this instruction will always execute, and then the branch jumps to the level where we actually load a new value into this register. There is a problem in code, there is no problem in this, and this will lead to the same problem. So, the question is that how do you really handle this. So, again we will take the same procedure; that is we will introduce interlocks. Delay issue of the latter instruction, or prevent the earlier one form writing. there are two things you can do one is that; you can figure out that this load instructions write, is not going to be of any use.

These add instructions write is not going to be use. So, you can nullify that write back, and the second option is that, you can delay this particular instruction, then how a cycle is needed, to make sure that the hazard goes away.


And again we can do this in the decode stage. again with the help of ship register circuitry, in the same may be slightly different one, which will essentially look for, not only just clash in the write port, but will also check that the register destinations are also important, and in that case it will introduce this.

(Refer Slide Time: 49:58)



Hazard detection

- Need to look for integer *and* fp hazards
 - Integer and floating-point instructions use separate register files
 - But floating-point load/store uses integer registers as base: there could be a hazard between integer and floating-point instructions
 - Also there are move instructions (mtc1 and mfc1) that move to/from floating-point register file to integer register file (why needed?)
 - So in these last two cases we need to detect hazards between integer and floating-point instructions
 - Otherwise hazards can happen between integer instructions only or floating-point instructions only: simplification made possible due to separate register files
 - Any problems of having separate files? Why not unified?

 HAINAK CS422 87

So, what is the hardware you can do with this ship register. So, for hazard detection you need to look for integer as well as floating point hazards; that is the essentially; that is what we have just seen here. Just focusing on the integer side, integer floating point instruction you separate register files. So, there are two separate register files for these 2. So, you might wonder you know, could there be a bypass coming from an integer instruction going to a floating point instruction, can that be possible. Then the answer is yes even though the operant on two different register files. So, we will look at examples very soon. Floating point load store uses integer registers as well, so there is one example, where a floating point instruction may require a value to be bypassed form integer instruction, like for example here.

If dollar 2 was getting computed, in the integer pipeline you have a, you might require a bypass, from that instruction to this particular mode, even though this load is actually a

floating point load. So, there could be hazard between integer and floating point instructions. Also there are move instructions $m t c 1$ and $m f c 1$ we have discussed this earlier. We discussed in the MIPS (()), that move to or from floating point register file to integer register file. So, why you need we have also discussed this. Does anyone remember, why would I need to move values from floating point register file to integer register file or vice versa.

Student: Pipeline.

Exactly; so, if you want to cast a floating point value to an integer value, you make a move from floating point register file to integer point register file, and the other will require the move in the other direction. So, in these last two cases, we need to detect hazard between integer and floating point instruction, because $m t c 1$ that is moved to floating point file, essentially we will have a floating point register destination, and an integer register source. And that source may get produced by a new instruction. So, there will be hazard from the newer instruction to this one. Similarly, $m f c 1$ will have a target integer register, and a source floating point register.

So, that may actually start a bypass to an integer destination, because the destination is integer register here. Otherwise hazards can happen between integer instructions only, or floating point instructions only. Simplification may possible, because due to separate register files. So, any problems of having separate register files. So, this is the good thing that simplifies your bypass network. So, if you did not have, you would actually have a lot more cases in a bypass, if you share the register file. So, that is one why you want register file. Other problem is that, you probably require bigger register file, which may actually slow down your pipeline. So, that will make sense to divide them.