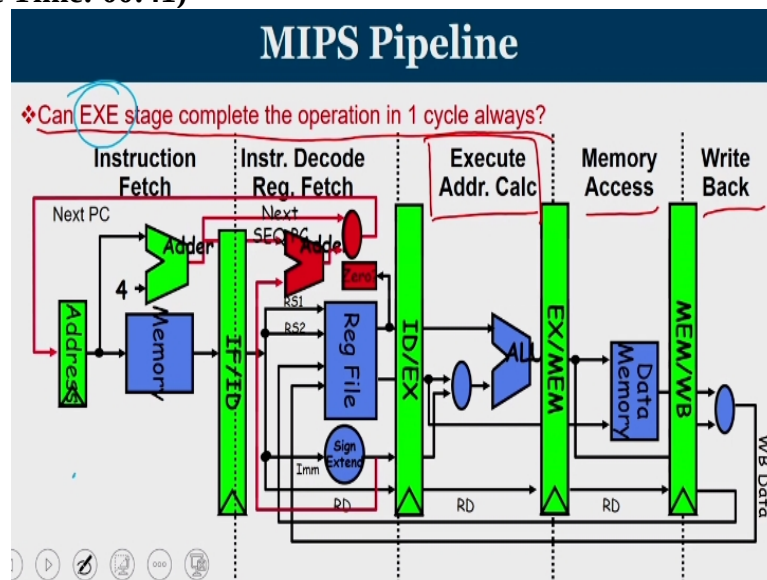


Advanced Computer Architecture
Dr. John Jose
Department of Computer Science & Engineering
Indian Institute of Technology-Guwahati

Lecture - 06
MIPS Pipeline for Multi-Cycle Operations

Welcome to lecture number six of advanced computer architecture course. In today's lecture our focus is on multi-cycle pipelines.

(Refer Slide Time: 00:41)



So this is the conventional 5-stage, MIPS pipeline that we were seeing over the last five lectures. We have seen that with one cycle the instruction fetch has to be over and in one cycle instruction decode and the register fetching should be over and in one cycle, the execution of calculation of address, memory access and right back. So we are assuming that all these five stages will take only one cycle each.

So far we did not think much deeper about this question, can everything be over in one cycle? Let us ask one specific question. Can the EXE cycle or EXE stage complete, can the EXE stage complete the operation in one cycle always. That is a challenge question that we are going to see.

(Refer Slide Time: 01:25)

Multi-cycle Operations

- ❖ Can EXE stage ^{1 cycle.} complete the operation in 1 cycle ?
- ❖ Some operations require more than 1 clock cycle to complete.
 - ❖ Floating Point/Integer Multiply
 - ❖ Floating Point/Integer Divide
 - ❖ Floating Point Add/Sub
- ❖ Dedicated hardware units are available on the processor for performing these operations.

Now, there are certain operations that requires more than one clock cycle to complete. For example, whenever you are involved in a floating point or an integer multiply operation, if you are involved in a division operation or if it is a floating point add or sub. So generally if it is an integer add or sub, then thing should be all over, the EXE stage will take only one cycle.

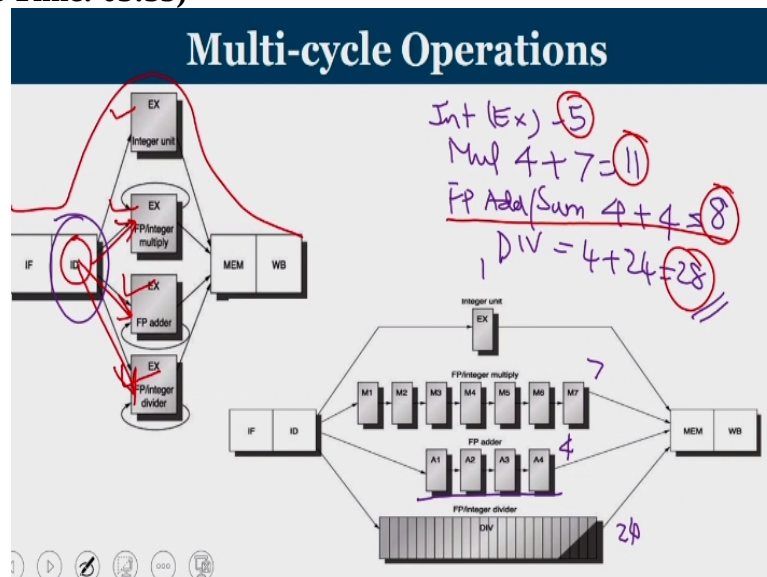
But if you are going to have a floating point adder/subtractor operation then, we know that in order to manipulate two floating point numbers in terms of adding or subtraction, first we have to adjust the exponent and then only we can add or subtract the corresponding mantissa. Similarly multiplication operation involves multiple partial summation and division operation involves multiple subtraction also.

So when it comes to floating point adding and subtraction and any sort of multiplication or division whether it is an integer multiplication or a floating point multiplication similarly, whether it is an integer or a floating point division also execution stage cannot be completed in one cycle. We know that conventionally an execution stage means your operands are ready.

You have identified the opcode, the control signals are ready. The operand is ready at the input of the corresponding functional units. So far we have seen our functional unit is a ALU wherein it can perform add, subtract, and other logical operations but not advanced operations like multiplication, division, and floating point adding and subtracting.

Now our question is how will we deal with operations wherein the operations itself is taking more than one cycle in its execution stage. So some operations like what we have seen will require more number of clock cycles to complete. So you require dedicated hardware units. Such kind of dedicated hardware units are available in the processor for performing these specific operations.

(Refer Slide Time: 03:35)



So consider the case that you have an integer unit which is our conventional EX unit. And for multiplication, you have a multiplier, you have a floating point adder and you have a division unit also. So previously, this was my pipeline wherein I have IF, ID, EX, MEM and writeback and now we know that another units are been added. Whenever there is a multiplication, then after the ID stage I am feeling it to the multiplier.

Similarly, whenever there is a floating point adding or subtraction, it is going to the floating point adder and then you have a divider unit also. So essentially at the ID stage, now we are going to have a choice. Depending on what is the operation, one of these execution unit are being used. So accordingly you are going to forward the output of the decode stage into that.

Looking deeper into this, the EX stage will take only one cycle whereas, if it is a multiplier it is going to take 7 cycles exclusively for execution and 4 cycle for adding

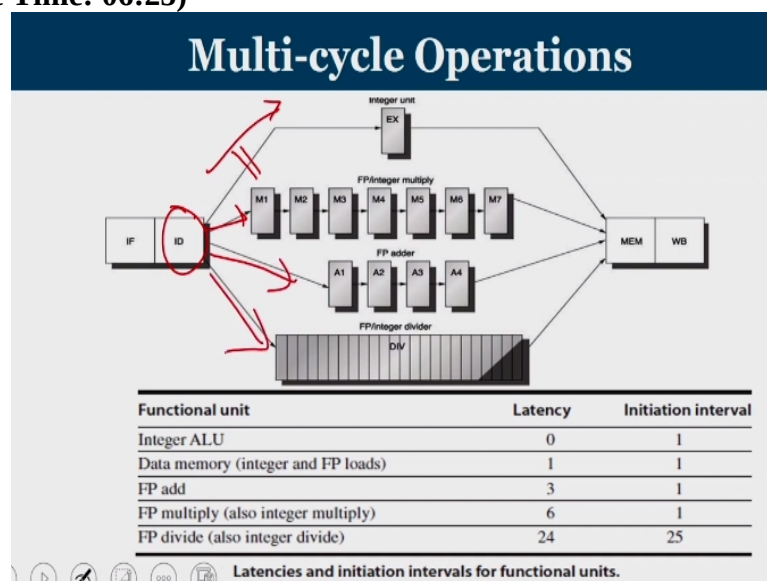
and 24 cycle for division operation. So if it is going to be a normal integer operation that make use of the EX unit in the pipeline, then it will get over in 5 cycles. If it is a multiplication operation then conventionally we have IF, ID, MEM and writeback, those 4 stages plus another 7 stage of multiplication that make it total of 11.

If it is floating point add or sub then we know that apart from the normal 4 another 4 more cycles are required and that is making it as 8. And for division operation, we have 4 + 24. So it will roughly take 28 cycles to complete. So a general integer operation is going to get over in 5 cycles. If it is a multiplication, it will take 11 and if it is a division it is going to take 28 cycles and if it is a floating point adder/subtractor operation, then you require 8 cycles.

This is happening because of making use of variable pipeline execution units. So having said this, our initial assumption was all the instruction will be using 5 cycles. And now, depending upon the operation, when you are going to deal with multiplication division and floating point operations, then it takes more number of cycles to complete the execution.

So even though we start in a sequence, the ending of the instructions may not be in order depending upon how longer it is going to take in the execution stage.

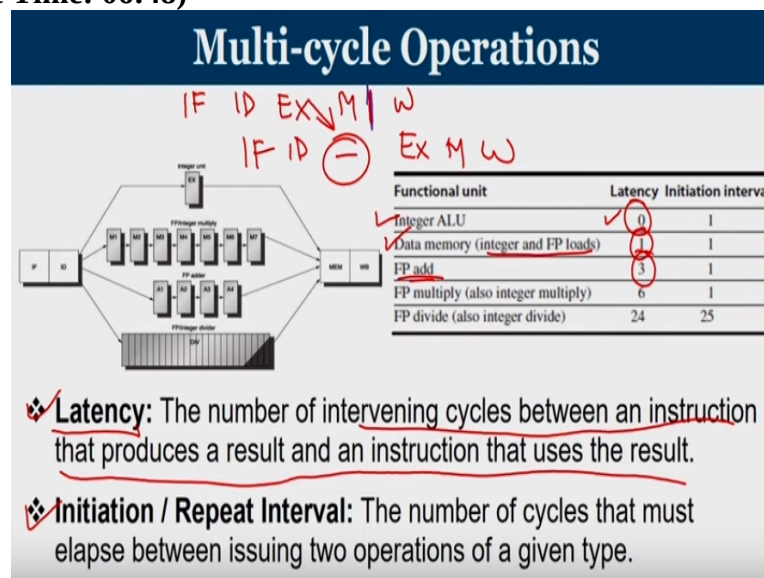
(Refer Slide Time: 06:25)



Now, coming into this multi-cycle operations, you know that there are two things that you wanted to define. So this is a multi-cycle pipeline. So the ID unit is basically

acting as a splitter depending upon the opcode, either you will be forwarding into this path or like this, like this.

(Refer Slide Time: 06:48)



Now, let us try to understand two important concepts, one is called latency associated with the functional unit and second one is initiation or repeat interval associated with the functional unit. Over the last few lectures of discussion we have seen that let us say I have consecutive integer operation like add or sub whatever it is and assume that there is no data dependency between them.

Then in every cycle, I can actually fetch or I can actually issue the integer operations. That means since it is taking only one cycle in the execution stage, every cycle new, new pair of operands are coming and I am able to perform the operations as well. Let us try to see whether it is the same case, when you go to multi-cycle operations. You are going to define a term called latency associated with the functional unit.

It is defined as number of intervening cycles between an instruction that produces a result and an instruction that is going to use a result. So let us say I am going to produce a result by an integer ALU. If somebody else wanted to use it, they should not actually wait. The latency is actually zero because we know that if there is an EX stage where I have a IF, ID, EX, MEM, and writeback that next EX stage comes here and you see operand forwarding.

Even though there is a dependency between them, I can actually forward the result of the EX to the next instruction. So the second EX instruction is not delayed for any cycle by virtue of operand forwarding. So the latency associated with integer ALU is zero, because we require 0 number of intervening cycles between the instruction that produces the result.

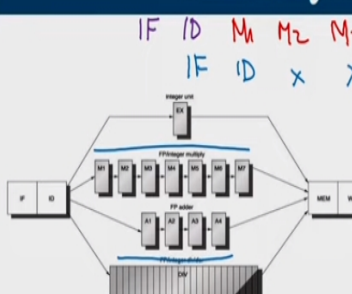
That is an integer ALU that produces the result and anybody who is going to use the result. Similarly, when you go to data memory, especially in the case of floating point loads, that means I am going to produce a value and this value is a lower operation value. So we know that load operation typically is going to reach the processor at this point. I am going to access from memory.

So if somebody wanted to use the value that is already loaded, then its EX cannot be here. The EX gets shifted by one cycle. So I will repeat the concept once again. If the first instruction is a load, then at this point only we are going to have the value ready. So the EX can be only at this point. Essentially, I need to wait for one cycle.

So whenever a value that has been produced by a load has to be used by some other instruction, then at least one intervening cycle is required. That is what is meant by the latency of a load operation. Now, consider the case that you are going to have a floating point add operation, is going to produce the result. And here the latency is shown as 3. Let us try to understand how this latency 3 is computed.

(Refer Slide Time: 10:30)

Multi-cycle Operations



Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

❖ **Latency:** The number of intervening cycles between an instruction that produces a result and an instruction that uses the result.

❖ **Initiation / Repeat Interval:** The number of cycles that must elapse between issuing two operations of a given type.

So we are going to have two floating points. So let us say there is a floating point add operation. So I have an IF, ID. Then if it is floating point add, rather than EX it is going to A 1, A 2, A 3, and A 4 and then MEM and writeback. Now assume that there is some other instruction that wanted to use this. So here it is IF, this is ID. EX cannot start at this point because the value is available.

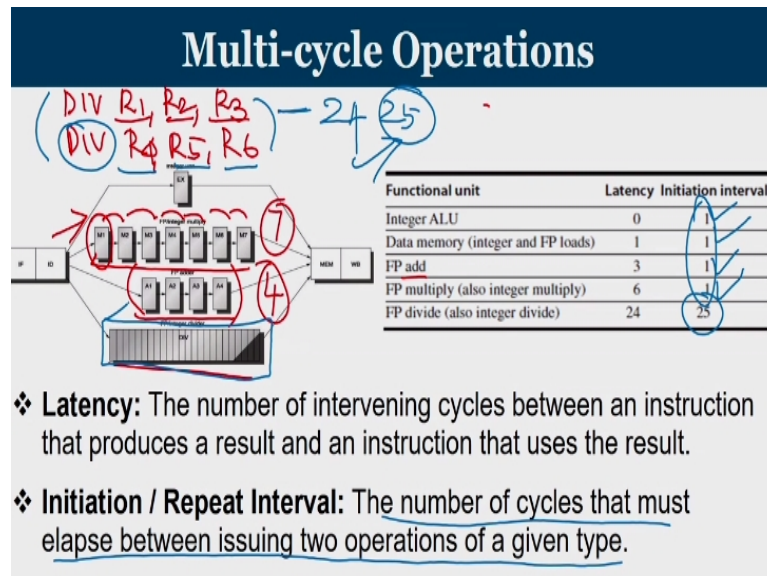
The result of the floating point add is available only at this point. So the EX can be kept only here and this is going to be MEM and writeback. Now, how many cycles are spared? You can see that here I cannot perform EX. So basically 3 cycles have to be intervened. So there is a delay of 3 cycles, if the value that is produced by a floating point add has to be used by somewhere else.

That is why the latency is been given as 3. Similarly, you just imagine you are going to have a floating point multiplication operation and somebody else wanted to use the result. Floating point multiplication we know that it is going to take 7 cycles. So it is M 1, M 2, M 3, M 4, M 5, M 6 and M 7 followed by MEM and writeback. These are the stages.

Now, if somebody else wanted to use this result, the result is available only after M 7. So the EX should be here, EX, MEM, and writeback. So these are the cycles which we will be going to extort and that is what you see it is a 6. So floating point add even though the latency is 4 here you see, since the adjacent instruction is starting one cycle late only 3 is being felt, 3 cycles of delay.

Similarly, for multiplication, I have 6 cycles delay. Now, we got to see that the next one is floating point division which will need 24 cycles to complete the operation. So there is a latency of 24 cycles that is coming into this. That is all about latency. Now, let us try to understand what is the initiation interval, that is one more term that has been defined.

(Refer Slide Time: 13:12)



You can note from this diagram. Let us now try to understand what is initiation or repeat interval. From the diagram you can see that this multiplication unit and the adding unit are divided into smaller blocks and they are connected by a pipeline. That means, every cycle whatever be the output of M1, that will be given to M2. Output of M2 will be given to M3 in a pipelined fashion.

So M1 can take new pair of operands that has been given by ID. Or we can say that the multiplier unit is internally pipelined. Similarly, the 4 stage adder is also internally pipelined. That adder can take new values every cycle, but it takes for a set of operands for completion of floating point adding, it will take 4 cycles. So 7 cycle of multiplication and 4 cycle of floating point adder/subtractor are internally pipelined.

As long as there is no dependency between the data, every cycle I can start a new operation, whereas your division unit is an unpipelined. That means, once you supply a division operation, for example consider the case that you are going to have division operation of R2 and R3, which is to be stored on R1. Let us say the adjacent instruction is R4, R5, and R6.

If you look at these two instructions, you can see that there is no dependency between them. R1, R2, R3 are the first operands of the first division operation. Whereas, the second division is operating on R4, R5, and R6. There is no dependency between them. So the result of the first division is not needed as far as the second division is concerned.

But still, since our division unit is unpipelined, it will take 24 cycles of latency that happens. 25 cycles are required to complete the operation. And if you look at the division, I can use my division unit only after 25 cycles. That means, the number of cycles that must elapse between issuing two operations of a given type. I am going to use the same division again, two divisions has to be separated by 25 cycles because my division unit has a latency of 24.

Now if you look at this, the individual ALU has an initiation interval of 1. That means two integer ALU operation should be separated by one cycle. Similarly, data memory operations, floating point add, floating point multiplication all will take one cycles. That means in the very adjacent cycle, I can have an operation of the same type provided there is no data dependency.

But in the case of a division since the functional unit is unpipelined, for the next 25 clock cycles, the unit will not accept any operand. That is why the initiation interval of the division unit is higher. So let us try to understand what we learned in this. In the case of a multi-cycle pipeline after the ID stage, there are different functional unit. It is a duty of the ID stage to forward whatever is the decoded instruction and the corresponding operand into the appropriate functional units.

If it is an integer operation or a logical operation that has been done on integer registers forward it to EX unit. If it is a multiplier operation, irrespective of whether it is an integer or a floating point unit, it has to be supplied to the 7-stage pipelined multiplier. Now, if it is a floating point adder/subtractor operation, then the ID stage has to forward the instruction and the corresponding operands to the 4-stage pipelined floating point adder/subtractor.

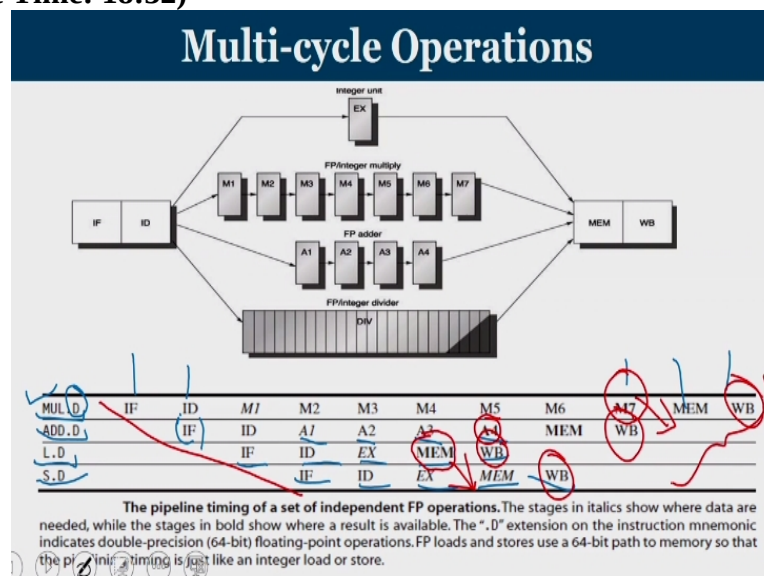
And for a division we have an unpipelined unit which will take 25 stages to complete the operation. So that is basically dealing with all these instruction. So the ID unit is now more capable. Depending on instruction it is going to forward the corresponding operands and the decoded instruction into appropriate functional units. And these functional units will take different amount of time to complete.

We learned about latency and initiation interval, two important terms in order to get a deeper understanding about how this functional units behave. Latency means, it is a number of intervening cycles between an instruction that produces a result and another instruction that is going to use the result. In the case of an integer operation we know that in the very adjacent cycle I can use it because of operand forwarding.

Then there is no delay actually, that is called the latency of zero. Whereas, in the case of a load operation, there is one cycle delay. So after the load is over, minimum one cycle of delay is required for another instruction to use it. When it comes to floating point add or subtraction we require minimum of 3 cycles, because the floating point adding itself is 4 stage A1, A2, A3, and A4.

There is a normal shift of one cycle in every pipelined instruction. So another 3 more cycle is required if you wanted to use the result produced by a floating point adder. In the case of a multiplier, since it is a 7-stage multiplier, latency of 6 is experienced. Whereas in the case of division, it is an unpipelined division unit. So the latency is 24 and the initiation interval is 25.

(Refer Slide Time: 18:32)



Now, let us try to see some examples and we see some challenges that are being faced while doing multi-cycle operation. So consider the case that you are going to have a multiplication operation and this D indicates we are going to operate on a double data. So double data means, basically your operand is a floating point. So when you

perform a multiplication operation on a double data, you know that IF and ID is normal and then you have 7 stages of multiplication followed by MEM and writeback.

After that I have an add operation. So add operation starts in the second cycle. And it has A 1, A 2, A 3, and A 4 and MEM and writeback will be following after that, and then you can see a load which is the third instruction, which will make use of the integer pipeline, because the EX stage is used for calculation of effective address or IF, ID, EX, MEM, and writeback.

And then you have a store which is going for IF, ID, EX, MEM, and writeback as usual. Now, what we want to understand from this, the result of the multiplication is available only at M 7, at the end of M 7. Result of add operation is available at A 4. Whereas result of a load operation is available at MEM. So any kind, if somebody is going to use the result of multiplication, it has to be after this.

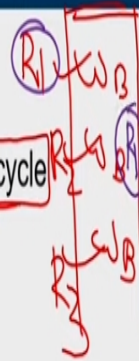
If somebody is going to be using the result of a load operation, it has to be after this. And look at one pattern. We started one-one instruction in every cycle. But when you look at the time at which they are completing, the writeback stage is happening for multiplication very late, whereas your add is complete a little bit early, load even still early and then the store. So this is called out of order completion.

If you look at the order in which the instructions are completing, it is not in the same order, the way which instruction have started fetching. This is happening because of variable number of cycles needed at the execution stage. Having understood the basic of what is a multi-cycle pipeline, now let us dig deeper into what are the issues that an architect face during the design of such multi-cycle pipelines.

(Refer Slide Time: 20:46)

Issues in Longer Latency Pipelines

- ❖ Since divide unit is not-pipelined – structural hazard
- ❖ Instruction have varying runtimes – more register writes/cycle
- ❖ WAW hazards possible



The first one is your division unit is not pipelined. So that can lead to a structural hazard. Let us try to recap what you mean by structural hazard. Structural hazard means when two instructions try to use the same hardware then that is known as a structural hazard. So think of a case I have a division operation to be done on, let us say, these floating point registers.

My F indicates they are floating point registers and R is indicating integer registers. Now, another one is another instruction. So division of F2, F3, F4. Division of F6, F7, and F8. If these two instructions come in adjacent cycles, the second instruction has to wait until the first one is completed by the division unit. This is because the division unit will not accept any operand until the previous division is over.

So this is not due to any dependency. You can see that from this pair of instructions, there is no data dependency at all and this is what is known as a structural hazard. Since the division unit is not pipelined, all subsequent instructions, all subsequent division instructions rather have to wait because the division unit is already doing the division of a previous operation.

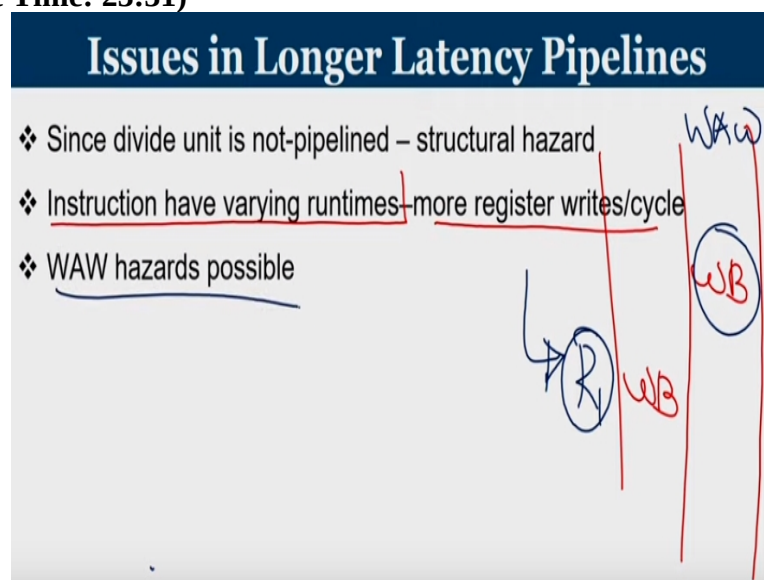
Now, the second design issue is instructions have varying run times. So this can lead to a scenario where multiple instructions read the write back backstage at the same clock cycle. Instructions of varying run time, this will produce more register writes. So I need to have facilities which will permit me to write into different registers. Let us

assume the first one is going to write to R1, second instruction is to R2 and third instruction is to R3.

All these three instructions are going to reach the WB stage, the write back backstage exactly at the same time. This is because different instruction will take different number of cycles in its execution. So somebody who is starting very late can complete early. So we should expect these kinds of scenarios. So that is a challenge. How will you address this?

Now, we can have WAW hazards that is been possible. So how will you get WAW hazard? So think of a case that this is R1. Let us assume the second instruction is also going to write into R1. Now, we will try to see what are the next problems that are being faced. One of the important problem that we face in multi-cycle pipeline is the WAW hazard.

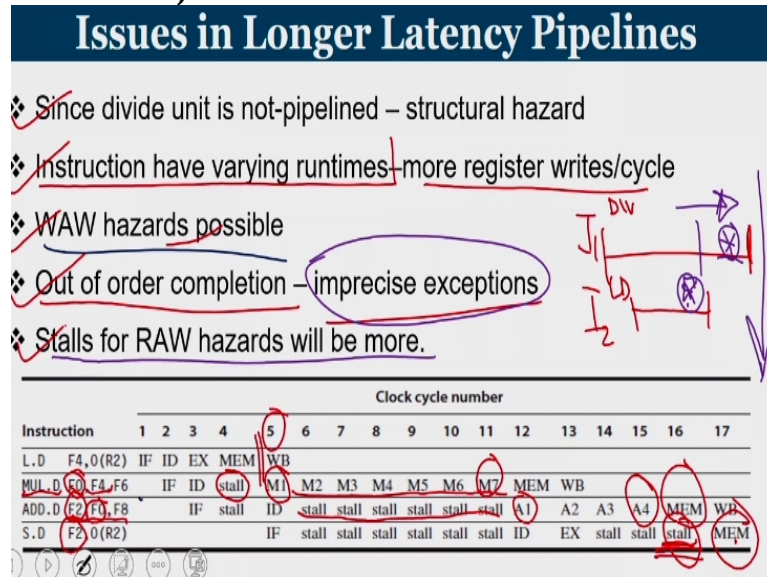
(Refer Slide Time: 23:31)



So what exactly is the WAW hazard. Think of a case that you are going to complete your writeback at some clock cycle. Let us say some other instruction is going to complete before you even though it is started later than the first instruction, it can sometimes complete before the first instruction. Now imagine that both the instructions are going to write into the register R1.

So the second instruction will write into R1 before the first instruction now written. That is exactly write after write hazard. So WAW hazard is actually possible in these kind of pipelines.

(Refer Slide Time: 24:16)



So we need to have mechanisms that will help us to address these WAW hazards. Then another important problem is out of order completion can sometime lead to imprecise exceptions. Here I wanted to draw your attention to understand what is a concept of an imprecise exception. So exception is an unusual behavior that can happen. For example, division by zero.

As far as fetching is concerned no problem. During decoding you try to take the values of two registers and when you divide one over other let us assume that the divisor is zero, that can lead to an exception. So think of a case that your memory got corrupted. So you fetch an instruction and during decode phase you happen to see an opcode which is not defined, that is an exception.

So similarly there are many exceptions are there. Now when you have out of order execution that can lead to imprecise exceptions. An imprecise exception means an exception is occurring on an instruction that is going to complete before some instruction that are issued before it. So think of a case. Let us say I have I1 that is going to complete only at this point.

Let us say there is I2 which started after I1 and it is going to get over before I1. For example, your I1 can be a division operation. Your I2 can be a normal load operation. Now think of a case that you are going to have an exception in I2. Do not try to handle the exception because you are not even sure that whether I2 will be really executed or not because the previous I1 is still under execution.

So there can still be possibility that I1 may face an exception at some point later. The exception handling rule tell that exceptions has to be handled only in order. First before handling the exception created by I2, I have to be sure that there is no exception that will be creating by any instruction proceeding I2. So if we permit instruction to complete out of order, then that can lead to creation of imprecise exceptions.

Now, stall for RAW hazards will be very more. Generally we know that in our normal instruction pipeline, we know that your stall is very minimal. If there is an ALU instruction immediately after a load, there will be a stall between them. But if it is a sequence of ALU instruction even though there is a data dependency between them by the technique of operand forwarding we will be able to handle that.

But when it comes to multi-cycle pipelines, it may not be the case always. So look at this example, where you are going to load an instruction. See here this R indicates integer registers and F indicate double registers or floating point registers. So consider this load instruction which is going to get over at clock cycle number 5 and from the value loaded, you are going to have a multiplication operation.

So F4 is the resultant of the load instruction and this F4 act as a source operand for this multiplication. This multiplication is operating on a double data. So when you are going to have, your multiplication can start only after the MEM stage because only after the MEM stage the value to be loaded is available and that is operand forwarded. So there is one cycle stall.

But we know that any operation after a load will consume a stall. Now you have 7 cycles of multiplication operation and you look at the sequence your add operation has an F 0, which is same as the result of the previous multiplication. Meaning the add

instruction is dependent on the result of multiplication. So my A1 can start only if M7 is complete. Now you look at the number of stalls.

The number of RAW stalls have drastically increased. Similarly, the result F 2, whatever is the result of the add instruction that is needed for the store operation. Essentially, the result of add operation that is to be stored in F 2 has to be stored back to memory. So your store operation can happen only after the add operation is complete. Now you may ask why MEM is here. MEM can be here also.

Two memory operations cannot happen in the same cycle. That is where I get one more stall over there. So to conclude, we have problems of structural hazard when it comes to division, because division unit is unpipelined. We have problems of multiple write operations per cycle due to varying run times. Sometimes there can be the possibility of WAW hazard.

This out of order completion can sometime lead to imprecise exceptions that has to be handled and we need to address many number of stalled cycles or whenever you go for multi-cycle pipeline, RAW hazards will be having more number of stalls.

(Refer Slide Time: 29:30)

Issues in Longer Latency Pipelines											
Instruction	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		ID	ID	EX	MEM	WB					
ADD.D F2, F4, F6			IF	ID	EX	MEM	WB				
...				ID	ID	EX	MEM	WB			
...					IF	ID	EX	MEM	WB		
L.D F2, 0(R2)						IF	ID	EX	MEM	WB	

- ❖ Single write port (Serialize completion) vs multiple write ports
- ❖ Resolve write port conflicts in ID stage and stall issue by 1
- ❖ Stall either of the instruction (priority basis) at MEM / WB stage
- ❖ Stall at MEM will force a stall to trickle at EX, M7, A4 stages

Now issues in longer latency pipelines. So consider this multiplication instruction and then you have some time you have add operation, then you are going to have a load operation. If you look at the sequence, you can see that they all are going to reach the

writeback stage exactly at the same clock cycle number 11. So if you have only single write ports, then I cannot write them.

So even though this is writing to, one is writing to F0 whereas the other two are writing to F2. So I have to write a value into F0. At the same time, I have to write a value into F2 and it has to happen at the same time. If you have only one single write port, then I have to make sure that serializing the completion. If you keep multiple write port then the overhead of managing and sometimes we may not get cases like this and that can lead to wastage of the resources.

Another issue is how can you resolve this. We have to make sure that never this kind of a scenario exist. So resolve write port conflicts in the ID stage itself. When I perform an ID stage here, for a multiplication instruction, I know that there are another 7 more cycles for the multiplication operation and then you have a MEM and writeback. So if ID happens at 2 for a multiplication operation, I know that there is a writeback that happens at 11.

Similarly, when I have an add instructions whose ID is at 5 I know that this add instruction also is going to reach the writeback stage at 11. If we have the information that is available inside the control unit, which tells that an instruction previously issued at clock cycle number 2 will also reach the writeback stage exactly at the same time, then I am not supposed to complete my decoding at clock cycle number 5, either I should delay.

That is called resolving write port conflicts in the ID stage itself. Every instruction upon reaching the ID stage knows by what time they are going to reach the writeback stage and try to look at the past and see whether any other previously issued instruction also will reach the writeback stage exactly at the same clock cycle. If so the second instruction or the later instruction has to be delayed appropriately if you wanted to make sure that no instruction is reaching the writeback stage together.

Or stall either of the instruction on a priority basis at MEM writeback stage. So once you reach the MEM writeback stage you know that there is a previous instruction that is also going to perform a write. So you have to incur a stall at the MEM stage. If

there is a stall at this MEM writeback stage, then the stall is going to trickle down to EX, M 7, A 4 like that.

So whenever I encounter a stall at the end of the MEM stage, then the instruction that is reaching at A 4 that has to be stopped. Instruction that is there at M 7, those have to be stopped. That is called trickling back effect. So here in this slide, we have seen, there can be possibility that multiple write operations have to be performed at the same clock cycle.

Either you keep multiple write ports or use serializing of the write operation in the same clock cycle. Or else we have to resolve such kind of scenarios by looking early, sufficiently early itself by using a look ahead mechanism. The moment you perform ID stage you will know by what time this instruction is going to reach the writeback and appropriate actions has to be taken in order to delay certain instruction such that they would not reach the writeback stage exactly at the same time.

You can even stall the instruction later in the MEM writeback stage appropriately trickling back into earlier stages of pipeline will also happen.

(Refer Slide Time: 33:20)

Issues in Longer Latency Pipelines											
Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
ADD.D F2,F4,F6			IF	ID	EX	MEM	WB				
...				IF	ID	A1	A2	A3	A4	MEM	WB
L.D F2,0(R2)					IF	ID	EX	MEM	WB		

❖ WAW hazard at register F2

❖ Delay issue (ID → EX) of L.D until ADD.D enters MEM stage

❖ Keep the result of ADD.D and give it to needed instruction.

❖ Hence only L.D will write on F2.

Now, here in this case, you can see that your load instruction is going to write into F2 and instruction before that is also going to write into F2. As per the normal sequence an add instruction that is fetched in the fourth cycle will reach the writeback only in

the 11th cycle. Whereas a load instruction that is issued in the 6th cycle will complete its writeback in the 10th cycle. So what exactly happens?

The register, the destination register of these two operations are same. So load will write the value at clock cycle number 10. Whereas the add is going to write the value at clock cycle number 11. So any instruction that comes after the load, if they are going to read the value of F2, ideally we want after this point, whoever is going to read the value of F2 it is the value that is been loaded by this load instruction.

But in this case, after the value is loaded at clock cycle number 10, clock cycle number 11 the register F2 is overwritten meaning any instruction that is coming after here they are not seeing the value done by the load instruction, rather they are seeing the value that is being written by the add instruction. So that is what is known as the write after write hazard on register F2. This has to be addressed.

This also can be done by a delay issue in ID/EX of the load until add has reached the MEM stage. So you have to make sure that the add is reaching the MEM stage. Once it reaches the MEM stage then it is defined. Because within one cycle you will reach the writeback. So delay all other subsequent instruction such that they all will reach the writeback at the same time.

Or keep the result of ADD.D and give it to the needed instruction. That is yet another possibility. If your add instruction is going to update the register later, do not update F2. Whatever is the operation, at the end of the operation, the add has to forward the result only to those which is going to use it. In that case, only load is going to write into F2. So these are some of the methods by which we address this WAW hazard.

(Refer Slide Time: 35:33)

How to handle Issues in Longer Pipelines

- ❖ Check for structural hazard in DIV.D and write ports EX MEM
- ❖ Check for RAW data hazard at ID stage: If the source of an instruction in ID is F_i then F_i should be there as the name of destination of instruction in ID/A1, A1/A2, A2/A3 and ID/M1, M1/M2, ..., M6/M7
- ❖ Check for WAW data hazard: If any instruction in A1, ..., A4, M1, ..., M7 has the same destination as the instruction in ID and the time at which they reach WB is same, delay issue by 1 cycle and repeat.
- ❖ Perform operand forwarding from EX/MEM, A4/MEM, M7/MEM, D/MEM, MEM/WB

Now how to handle issues in longer latency pipeline. So the first one is you have to check for structural hazard if any. Check for structural hazard in terms of structural hazard can happen in two ways. First is multiple writing in the register file, and second one is adjacent division operations. Since the previous division is not yet over second instruction cannot proceed.

Similarly when multiple writes happen in the same clock cycle, number of write port is not sufficient. Then you have to check for RAW data hazard. Read after write hazard at the ID stage. So how will you do? Or how can you essentially find it out? There is a RAW hazard between a pair of instruction. So this is possible if the source of an instruction is in ID.

So how can you actually check a RAW data hazard? If the source of an instruction in the ID stage is F_5 ; let us say the name of the register is F_5 , then F_5 should be there as the name of destination register of some other instruction which is there in ID/A1, A1/A2, A2/A3 or ID/M1, ID/M2 or ID/M1, M1/M2 like that. So if I am the source of an instruction, then that instruction name, the register name should not occur in any of the pipeline register after me.

So I am in the ID stage. After that it is ID/A1 or ID/M1 pipeline register. So this is the pipeline register between ID stage and the floating point adder and this is the pipeline register between the ID stage and the multiplier. Similarly, you have pipeline register

A1/A2. This is the pipeline register between the first stage of adder and the second stage of adder. Similarly I have M1/M2. I can have M6/M7 also.

So this number indicates what are the pipeline stages and this is actually a buffer. So in short, how are you looking for data hazards? The answer is simple. If the source of an instruction which is there in the ID stage is similar to or it is exactly same as the name of the destination of an instruction which is already there in ID/A1, A1/A2 or A2/A3 like that. Then there is a RAW hazard.

In that case appropriate forwarding has to be done. Similarly, how are you going to check for WAW hazard? If any instruction in A 1 to A 4 or M 1 to M 7 has the same destination as an instruction in ID and the time at which they reach WB is same then delay this instruction by one cycle. And then you repeat the same operation whether it has been resolved or not.

This is the way how are you going to check for data, check for data hazard that is RAW hazard as well as WAW hazard. In the case of WAW hazard, any instruction that is there in A1 to A4 or M1 to M7 should not have the same destination as that of an ID. And perform operand forwarding. So previously we have seen that operand forwarding generally happens from output of ALU to its input or from the output of MEM stage to the input of ALU.

And now, this EX stage itself is of multiple cycles. So operand forwarding should be there from EX/MEM, A4/MEM, M7/MEM, D/MEM and MEM/WB all into the input of the corresponding unit. Let us say A1, M1 or division unit. So with this we come to the end of this lecture. In this lecture we were trying to see about those pipelines where the execution takes more than one clock cycle.

We have seen the normal integer pipeline which takes only one cycle in the EX stage, a four stage pipelined floating point adder multiplier, another seven stage multiplier, and then you have seen a unpipelined division unit as well. So these functional units, have varying run times and because of that there is a chance of out of order completion of instruction, which can lead to WAW hazard, multiple write operations

and structural hazard and we have seen how basically these things are addressed.
Thank you.