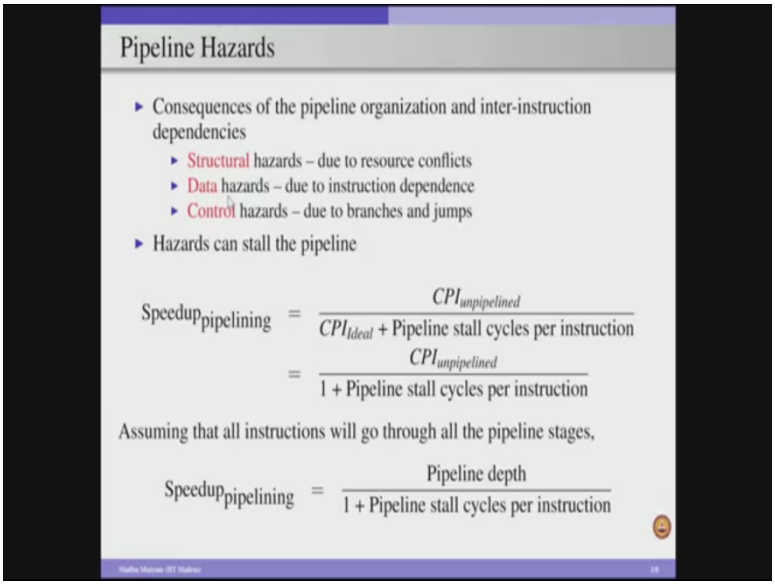
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science And Engineering
Indian Institute of Technology, Madras

Module – 05
Lecture – 15
Pipeline Hazards (part 1)

So, in the last module we discussed fundamentals of pipelining. And also we mentioned that in an ideal scenario with a k stage pipeline we can get a k fold increase in the performance, but in reality, because of several constrains, the performance through a k stage pipeline may not be a k fold. So, in this module we are going to discuss set of pipeline hazards and that actually limit the performance improvement that we can get through pipelining.

(Refer Slide Time: 00:48)



The pipeline hazards are caused because of organization of the pipelining as well as other inter-instruction dependencies. And these hazards can be classified as three categories. The first one is a structural hazard, second one is the data hazard and the third one is a control hazard. And the structural hazard is caused mainly because of the lack of resources or because of resource conflicts and whereas data hazards happen because of the instruction dependencies among the set of instructions that are processed by the pipeline.

And the control hazards are mainly because of the control instructions that are there in the program. And these control instructions are typically branches and jumps. Because of these hazards our normal flow of execution through the pipeline will be disturbed and as a result

there will be some stalls or bubbles created in the pipeline and as the stalls are created in the pipeline that is going to degrade the overall performance. So, effectively the speedup that we achieve with pipelining by considering this pipeline hazards is,

$$Speedup_{Pipeline} = \frac{CPI_{Non-pipelined}}{CPI_{Ideal} + Pipeline\ stall\ cycles\ per\ instruction}$$

And in an ideal scenario our CPI will be almost equal to 1. So, we can represent this equation as,

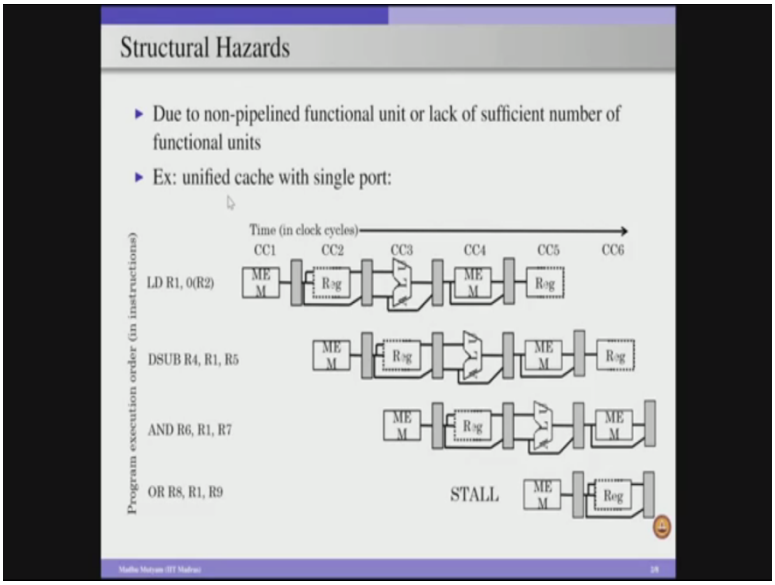
$$Speedup_{Pipeline} = \frac{CPI_{Non-pipelined}}{1 + Pipeline\ stall\ cycles\ per\ instruction}$$

And if we assume that all the instructions in our program will go through all the stages of the pipeline, then CPI of non pipelined design can be expressed as the number of pipeline stages. So, effectively the speedup that we achieve with pipelining can be expressed as,

$$Speedup_{Pipeline} = \frac{Pipeline\ Depth}{1 + Pipeline\ stall\ cycles\ per\ instruction}$$

So, effectively if more number of stalls are there then the speedup will be degraded significantly. So, if you want to come up with an efficient pipeline design, we have to ensure that the number of pipeline stall cycles is minimized. If you reduce that the pipeline stall cycles per instruction then automatically we can improve the speedup. If we assume that the number of pipeline stall cycles per instruction is 0, then our speedup that we achieve with pipelining is equal to pipeline depth, that is nothing but, if a k stage pipeline is considered then we will get the k fold increase in the speedup, but that will happen only if pipeline stall cycles become 0, but in reality because of the structural data and control hazards our pipeline stall cycles per instruction will not be 0.

(Refer Slide Time: 04:00)



So, first we will look at the structural hazards and as we mentioned earlier. The structural hazards are mainly because of resource conflicts. So, we will consider a simple scenario for example, if you have a pipeline design, where all the stages except the execute stage is taking one pipeline cycle time. And in our execute stage we have set of functional units. And for example, if our floating point multiplier unit is not pipelined and it is taking the total time equal to for example, ten times the pipeline cycle time.

Then effectively if any floating point multiplication instruction is processing through the pipelining. So, once it enters the execute stage this instruction will take ten pipeline cycles time to compute that. Then effectively this floating point multiplier will not be available for any subsequent floating point multiplication instruction. And also if the output of this pipeline, if the output of this floating point multiplication, is required for subsequent instructions those instructions also cannot be executed.

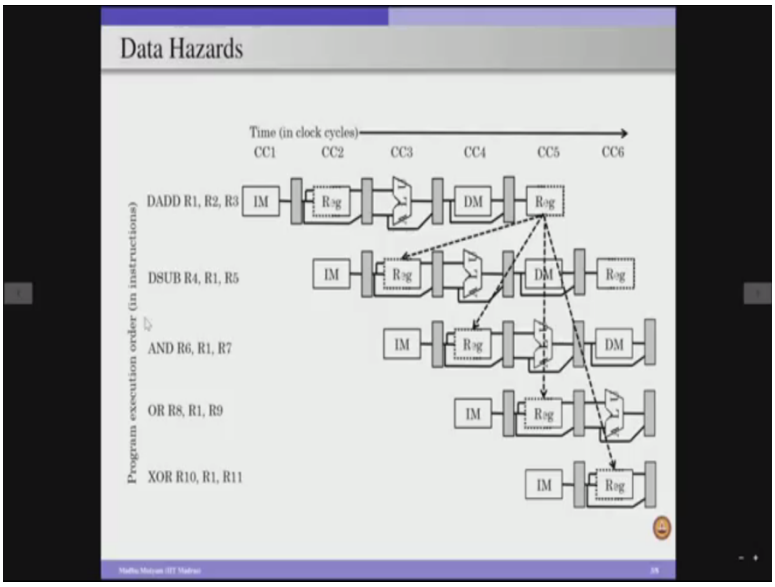
So, the second part is called as a data hazard, but the non availability of floating point multiplier unit for subsequent instructions is called as structural hazard because our floating point multiplier is not pipelined and which is taking ten pipeline cycles time. So, effectively this unit is busy for the next ten units of time. And as a result, none of the other floating point multiply instructions can use this, the functional unit. And as a result we will have structural hazard.

This is a scenario where if our functional unit is not pipelined, but there are other scenarios also. Such as like, if our numbers of functional units are not enough to perform the operations then also we will get a structural hazard. For example, we have a unified cache memory and assume that this unified cache memory has a single port for read operations. So, now once you have a single ported unified cache and when we are performing a read operation for an instruction from this unified cache, we cannot use this cache for data read operation.

So, as a result there is a resource conflict and that also creates a structural hazard. So, this is an example where we consider unified cache. So, the meaning of unified cache is like this cache can store both the instructions as well as the data. And also we consider this unified cache as a single port. So, now you can see here in the fourth cycle, we are actually performing a load operation through this unified cache. So, because the first instructions are load instructions which wants to load the data from the unified cache, that is effectively CC4, cycle four, we are going to use this the unified cache for data read operation, but while we are performing that, then we cannot use this cache for fetching the fourth instruction that is an R instruction.

So, effectively there is a stall our fourth instruction will be delayed by one cycle and in the cycle five the fetch operation for this fourth instruction will start. And that unnecessarily creates a bubble and that degrades the performance. In order to reduce the structural hazards, we can encourage the number of functional units or we can pipeline a functional unit.

(Refer Slide Time: 08:01)



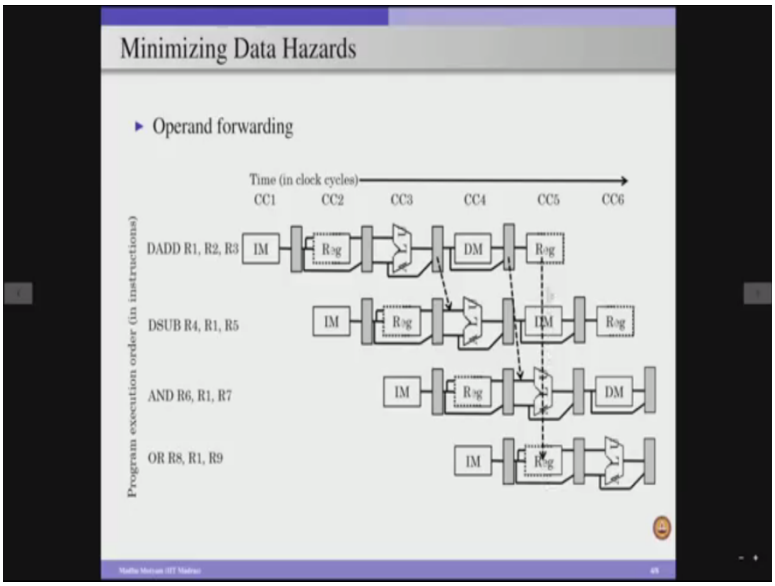
So, next we will look at the data hazard. As I mentioned earlier the data hazard is because of the dependency between the instructions. So, for example, consider a scenario where we have five different instructions that we want to execute through pipelining. And we can see here the first instruction a add instruction which produces the result onto a resistor R1 and subsequent instructions such as subtract instruction, AND instruction, OR instruction and XOR instructions require the output from this register R1 effectively.

All this four subsequent instructions are requiring the output generated by this add instruction. So, in other words, unless the add instruction produces the value into register R1, this four subsequent instructions cannot execute. And that is actually creates a data dependency between these instructions. And as a result our subsequent instructions cannot proceed further in the pipeline unless the add instruction is completed.

So, that is what is shown in a pictorially in this foil. So, here the register, we are writing to a register in a register file in clock cycle five, but that is required in clock cycle three for subtract instruction, clock cycle four for AND instruction, clock cycle five for OR instruction and clock cycle six for XOR, but because the value will be written at the end of the fifth cycle or half of the first half of the fifth cycle. So, as a result there would not be any problem for OR instruction as well as XOR instruction.

We can supply the data for these two instructions, but for subtract and the AND instructions. So, we have to come up with other mechanisms to minimize the penalties associated with this data dependency. Otherwise, we have to stall the sub and AND instruction. We can minimize the penalty associated with the data hazards by forwarding the required operands to subsequent instructions. That is called as operand forwarding.

(Refer Slide Time: 10:33)



So, now you can see here, we know that, though we are going to read R1 in cycle three for subtract instruction, but actual value is given to functional unit or ALU only in cycle four. So, as a result, even without reading the value from the register as soon as the previous instruction, that is add instruction, computes the, performs the add operation from this temporary register associated with the ALU unit or the pipeline register associated with ALU functional unit, can forward the computed value directly to the ALU input of subtract operation.

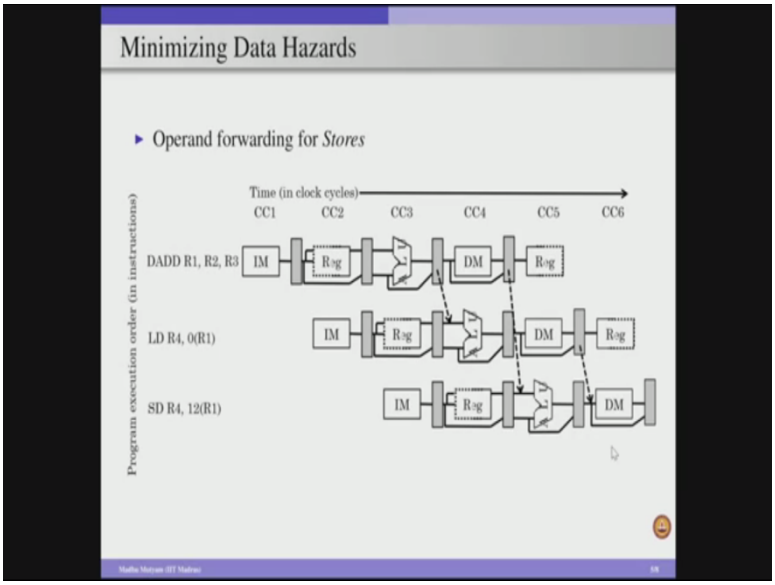
So, effectively the output from the adder is forwarded to the input of subtract functional unit. So, we are forwarding this operand from the previous functional unit to next functional unit. As a result, we do not have to worry whether read operation is performed for R1 in cycle three or not. So, once we forward this, our subtract operation can proceed without any problem. So, as a result, subtract operation does not require any stall. And similarly, for the AND instruction because AND instruction requires R1 at cycle five. As I mentioned earlier, by the time the functional unit is going to execute, if we supply the value then there would not be any problem.

So, as a result, we do not have to read the value in the previous cycle to be supplied to the functional unit and we can just supply the data as and when the functional unit is requiring. So, the functional unit is requiring value in cycle five for AND instruction. So, we just

forward the value of R1, which is not yet written to the register, but it is still there in the pipeline register associated with the stage between memory access as well as write back.

So, this pipeline register is going to supply the value of R1 and that will be forwarded to the input of the AND function unit. So, as a result AND instruction also will be executed without any stall. And similarly, the OR instruction, by the time OR is requiring the data a register read. So, already the value is written to the register in the fifth cycle for ADD instruction. So, here we can clearly see the first half of the cycle five, we are writing the content to the register file. And the second half of the cycle we are going to read from that particular thing. So, we can just forward that and then as a result OR can be performed. Even otherwise also we can forward this value directly to the ALU unit, but effectively by using this operand forwarding we can eliminate or we can minimize the pipeline stalls that occur because of data dependency.

(Refer Slide Time: 13:46)



But this operand forwarding can also happen for stores. See in the case of stores, we can see here in this example. So, we have an ADD instruction which is supplying, which is producing the output in register R1 and after the ADD instruction we have a load instruction and which takes this R1 as the source to calculate the effective address in the cache memory. And we will go to the cache memory and get the data stored in the location and write that onto register R4 and finally, we store this value that is stored in register R4 to some other memory location in the cache.

So, to do that effectively, load cannot perform unless R1 is produced. And similarly, store cannot start its execution unless load is completed. So, effectively there is dependency between these three instructions. So, now using this operand forwarding concept that we discussed in the previous foil, we can forward R1 value to load instruction that is from the output of ALU will be given to input of ALU of load instruction.

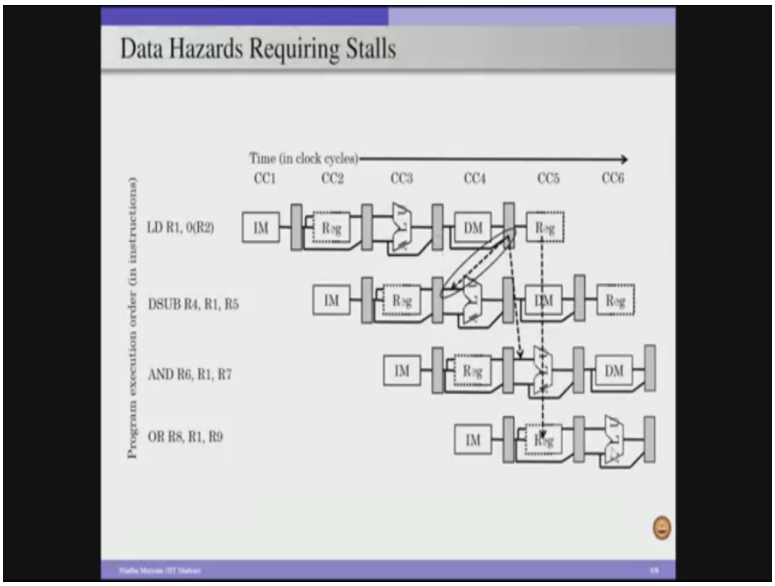
So, that there is forwarding you do not have to wait for this actual write back for R1 that happens only in cycle five, but by forwarding this value from the output of the ALU associated with this add instruction, which happens in third cycle. And we just forward that to the ALU unit in the fourth cycle. So, that the load computes its effective address by the end of fourth cycle our effective address is calculated. Now, we have to go to the memory or the cache memory and then we read the value that is located in that particular address specified by this effective address.

So, that will happen at the end of fifth cycle, for load instruction. At the end of fifth cycle the load instruction reads the value from the cache memory or data memory and by that time actually this store instruction requires this data to be written to the memory. So, we already know that store instruction requires R1 value as well as R4. R1 is already available through operand forwarding from the first instruction, ADD instruction. So, as a result like this pipeline register which is between memory access and the write back stage they will supply R4 content to the functional unit associated with the store instruction and in the fifth cycle.

In the sixth cycle, so load is actually supplying the read value, whatever the value it read from the data memory to the store instruction. So, effectively at the end of fourth cycle, we get R1 value forwarded to operand forwarding. At the end of fifth cycle we forward the load value that is obtained for this load instruction to the store. So, by the end of sixth cycle we perform store operation successfully.

So, effectively we are not incurring any stall for this store instruction. So, in summary data hazards also create penalty in the pipeline performance. And to minimize that we exploit operand forwarding concept and this operand forwarding concept can be applied for ALU outputs as well as the load values that we read from the memory. So, using this operand forwarding concept we can minimize the data hazards penalty, but there are some scenarios where data hazards cannot be eliminated or the stalls incurred because of data hazards cannot be eliminated.

(Refer Slide Time: 17:49)



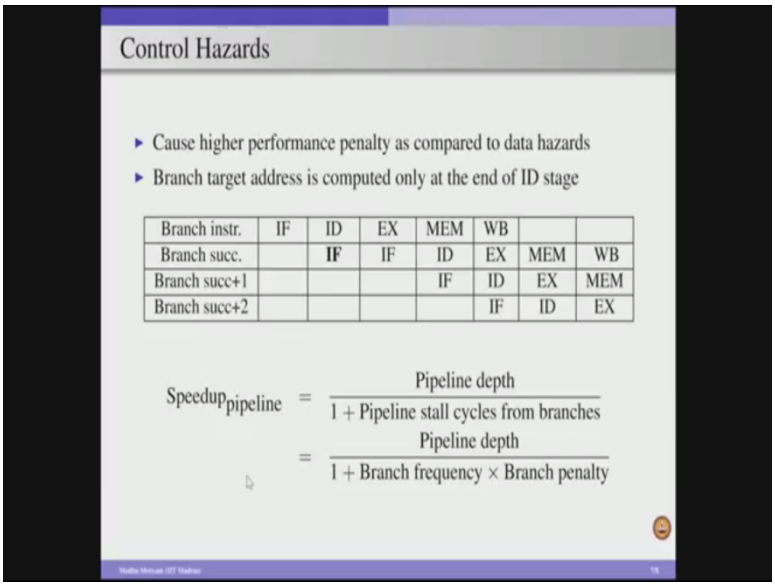
For example, here we can see subtract operation requires R1, but R1 is produced by the load instruction, but we know that load instruction produces the value from or load instruction reads the value from the data memory only at the end of fourth cycle in a five stage pipeline, but our subtract operation is issued one cycle delay with respect to the load. So, effectively R1 is required for subtract operation at least by the cycle four, the start of cycle four, but the load is supplying this memory read operation only at the end of cycle four. So, effectively we incur one cycle penalty because the subtract operation cannot be proceeded unless the value is read from the memory for the previous load instruction.

So, as a result of this stall cannot be eliminated in our pipelining design. Even when we use operand forwarding because we cannot exploit operand forwarding in this particular example especially for subtract operation, but for all other operations that is AND and OR. So, there is no problem with the R1 forwarding because like R1 is available at the end of fourth cycle and AND and OR requires these values only at the start of fifth cycle. So, as a result we can peacefully forward this R1 value for these two instructions, but for the subtract instruction we cannot do.

So, as a result there are some scenarios where we cannot eliminate these stalls and that actually degrade the performance. So, in other words, whenever we have set of instructions to be executed through pipeline, we have to see which instructions can be benefitted from this operand forwarding. And if there are some instructions which can benefit from operand

forwarding we have to apply this operand forwarding technique and minimize the stalls associated with these dependencies. And in those cases where we cannot exploit operand forwarding concepts then we have to incur penalty associated with this data dependencies.

(Refer Slide Time: 20:04)



And finally, we will consider that control hazards. We know that the control hazards are mainly because of the branch instructions; branch or jump instructions which are actually alter the execution of the normal flow. So, as compared to the data hazards the control hazards are going to incur significant penalty. So, our overall performance can be degraded significantly, if you are not taking care of this control hazards. So, because the control hazards are occur mainly due to this branch instructions and jump instructions. And we know that in our simple five stage pipeline a branch target address is computed at the end of second stage, that is instruction decode stage.

Unless we calculate the effective target address, we cannot proceed with our subsequent instruction fetch. And if we assume that branch is not going to take place and then we fetch subsequent instruction, but after the branch is computed we know that our prediction is wrong, then we have to flush and then we have to re-fetch the new instruction. So, that is effectively, for example, if we see in this example the branch instruction is proceeded in this order.

The first cycle we fetch the instruction and in the decode stage we identified that this is the branch instruction and in the same cycle we are going to compute the target address, but

while we are doing all these things the normal scenario we can start fetching the subsequent instruction to this previous instruction. So, we fetch the instruction, but at the end of this stage we identified that the control is jumped to a different location not in the program sequence order.

So, as a result we have to re-fetch the new instruction. So, as a result of this cycle is wasted. In other words, this one cycle stall is incurred in our, the pipeline because of the target address is not known at the start of the second cycle. And after that the subsequent instructions can be executed in a normal flow. So, effectively so, unless we know the target address for the branch instruction, we are not sure whether we have to proceed with the subsequent instruction or we have to proceed with some other instruction which is located at some other address.

So, that means like we need to find mechanism to effectively find or predict the target address and minimize stalls associated with this control hazards. As I mentioned previously

$$Speedup_{Pipeline} = \frac{Pipeline\ depth}{1 + Pipeline\ cycles\ per\ instruction}$$

If you do not have any stalls because of structural hazard or data hazard and all our stalls are mainly because of control hazards. So, effectively we will compute the number of stall cycles that happen because of the branches.

$$Speedup_{Pipeline} = \frac{Pipeline\ depth}{1 + Pipeline\ \text{stall cycles from branches}}$$

$$Speedup_{Pipeline} = \frac{Pipeline\ depth}{1 + Branch\ frequency * Branch\ penalty}$$

So, if we minimize the number of stall cycles from the branches then we can improve the overall speed up with the pipelining concept. So, the pipeline stall cycles from branches can be expressed as the product of branch frequency and the branch penalty. So, effectively, so in order to minimize the penalty associated with the control hazards, we have to minimize the penalty associated with the branches. We cannot reduce the branch frequency in an application, but we can minimize the penalty associated with the branches. So, as part of next module we are going to discuss set of techniques that minimizes the branch penalty and with that I am concluding this module.

Thank you.