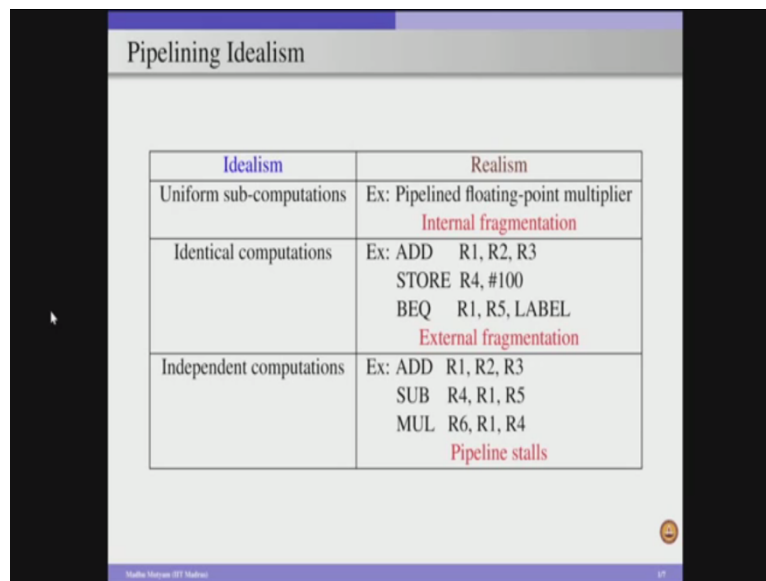


Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science And Engineering
Indian Institute of Technology, Madras

Module – 05
Lecture – 14
Fundamentals of Pipelining (Part 2)

So, in the last module, we introduced the pipelining concept and explained the pipelining with example, by considering floating point multiplication. Now, in this module we are going to look at instruction pipelining. So, in ideal scenario, we know that, when we divide a computation into k stages, we get k fold improvement in the throughput, but in reality that is not what we can get. There are several reasons for that. We will illustrate, why we are not getting k fold increase in the case stage pipeline in reality, by using different examples.

(Refer Slide Time: 36:48)



Idealism	Realism
Uniform sub-computations	Ex: Pipelined floating-point multiplier <i>Internal fragmentation</i>
Identical computations	Ex: ADD R1, R2, R3 STORE R4, #100 BEQ R1, R5, LABEL <i>External fragmentation</i>
Independent computations	Ex: ADD R1, R2, R3 SUB R4, R1, R5 MUL R6, R1, R4 <i>Pipeline stalls</i>

Now, in an ideal scenario, the pipelining assumes that we divide our computations in uniform sub-computations. In other words, when we consider a computation and in ideal scenario we assume that this computation can be divided into equal parts, there can be like k stages or n stages or whatever, but each of this sub stage will be of or taking the same amount of time as that of other stages, but in reality that is not what we can get.

For example, in the last module, we considered the floating point multiplier unit and we know that when we divided this floating point multiplication computation into three stages, the first stage is taking 125ns, second stage is taking 150ns and third stage is taking 125ns.

So effectively, so, not always it is possible to divide our computations into equal sub computations. So, as a result, we lose a performance because of these non-uniform sub computations. We know that, the cycle time of pipelining is determined by the stage that is taking maximum amount of time among all. So, in the floating point multiplication pipelining design, we know that second stage is the one which determines the overall, the pipeline stage cycle time.

So, because it is taking 150ns compared to the other two stages, those are taking 125ns. Now, effectively first and the third stage are actually losing 25ns of the time and this is called as internal fragmentation. When we have a pipeline design where each stage is taking different amounts of time, now we will get, we will waste some amount of time because of this non-uniform computations and that wastage is called as “Internal Fragmentation”.

This is nothing but the time wasted within a pipeline stage. So, this mainly happens because our sub computations are not always of equal length in time. And this is one of the reasons why we cannot get a k fold increase when we divide a computation in k pipeline stages. And the second reason, why we can k fold increase is non-identical computations. In ideal scenario, we consider that all computations are identical. We assume that, when we are performing computations on the pipeline design, we always get the same type of computations, but that assumption is not true in reality.

And consider an example here, we have three instructions here, one is an add instruction, the second one is a store instruction and third one is branch equal, so branch instruction. So, in this particular example our add is going to take some number of pipeline stages, whereas store is going to take different number of pipeline stages. And similarly, branch instruction, it may take two pipeline stages. So, we are going to discuss the instruction pipelining in a short while, but in that example we can clearly see branch is going to take two cycle latency whereas, store is going to take four cycle latency and add is going to take five cycle latency, in a five stage pipelining.

In other words, add is using all the pipeline stages, store is using four pipeline stages and one pipeline stage is wasted and branch instruction is taking only two pipeline stages for

performing the branch. And as a result, now we can clearly see that different instructions may take different pipeline stages or may use different pipeline stages and some pipeline stages may be underutilized. And that is called as an “External Fragmentation”. This is nothing but some stages are not utilized because of type of instructions we are performing and if the whole stage is not used than that is called as an external fragmentation.

Whereas, in the internal fragmentation case, some amount of time within a pipeline stage is wasted. So, because of this non identical computations also, sometimes we cannot get a k fold increase using k stage pipeline. And the last reason for not getting k fold increase or not getting a k fold throughput improvement using a k stage pipeline is independent computations. Not always all the computations are independent. If all the computations are independent we can execute all this computations without any problem.

So, there would not be any pipeline stalls and the overall performance can be improved, but in reality, instructions which are executed one after other may be dependent on the previous instructions. For example, consider a scenario, where we have an add instruction, subtract instructions and a multiply instruction. So, we know that, subtract instruction requires the output generated by an add instruction. Anyway, in this particular example, we consider three operands and the first operand always represents the output or the destination and the last two represent the source.

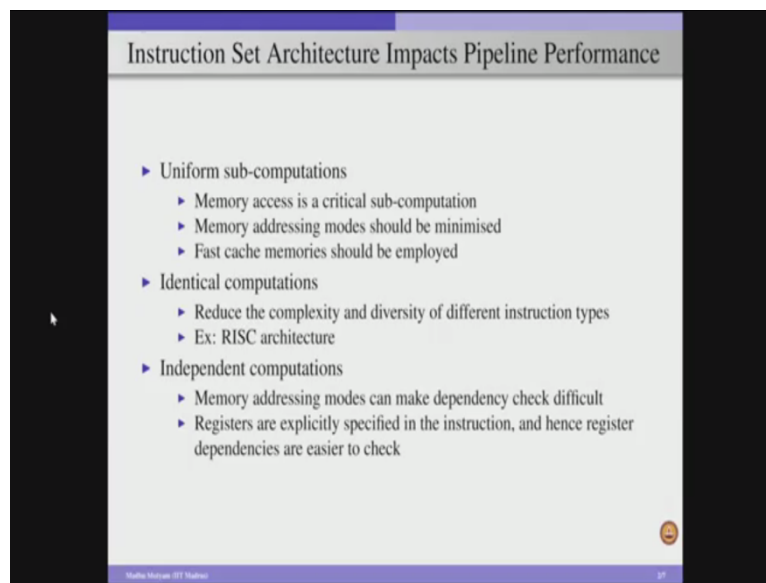
So, when we perform add operation, we read the data from two resistors R2 and R3 and the added value is stored onto R1. And when we want to perform subtract operation. So, subtract requires R1 as an input. So, effectively sub cannot be executed until add is completed, or add is writing the data to R1 till the time the sub operation has to be stalled. So, because of the dependency, sometimes some of the pipeline stages may not be proceeded and we may get pipeline stalls.

Even in this example, multiply also depends on sub to be performed. Because multiply takes an input R4 and which is produced by a subtract operation. Effectively all these three operations are dependent. Second operation depends on the first and the third is dependent on the second. So, as a result there will be pipeline stalls because of these pipeline stalls, our overall throughput may not be equal to the ideal scenario. So, effectively, because of these different scenarios, in reality, we may not get k fold increase with a k stage pipeline. So,

having discussed these three different categories where these three different categories which limit our overall performance improvement we get with the pipelining.

Now, we will see how the ISA has an impact on the pipeline performance. So, in fact the instruction set architecture has a significant impact on the overall performance improvement that we get through pipelining. Again we consider those three categories and then we will see how ISA has an impact.

(Refer Slide Time: 08:38)



So, first we start with uniform sub computations. According to uniform sub computations, our overall computation can be divided into sub computations and each sub computation takes the same amount of time as that of other sub computation. But if we use too many number of addressing modes then we may get significant performance penalty or we may not achieve very good performance through the pipelining. The reason is if you are using an memory addressing mode in our ALU operations and if one operand is defined based on memory addressing mode and the other operand is reading from a register and if you want to perform an ALU operations on this and we know that memory access latency is much higher compared to accessing a register file.

So, as a result, our overall time to read values, to read these two operands will be significant. And as a result this instruction is going to take much more time compared to any other instruction which reads operands from registers alone. So, and also we cannot prevent a memory access because the memory is the component which stores all the values and so on.

We have to read the values from the memory, but we limit the memory access by using only load and store instructions.

And only whenever we need memory data stored in the memory to be read, we will use these load and store instructions. And we will perform all ALU operations on registers. If you do that then we can minimize the impact of these non-uniform sub computations. And also, because going to the memory always, is going to take significant amount of time, we have to consider faster cache memory between the processor and the memory. And that also improves our overall the performance by reducing the memory access time.

So, effectively when we want to design an ISA, we have to limit the number of memory addressing modes to be used in our ISA. And we need to consider multiple levels of faster cache memory, once we do this then, once we do these things then, effectively we can minimize our non-uniform sub computations. And the second one is identical computations. So, in order to come up with these identical computations.

So, we need to come up with less complex instructions in our ISA and also these instructions should not be significantly divergent. So, that means like the diversity among these different instructions types should be minimized. Once we minimize diversity among different instruction types and also once we minimize the complexity of an instruction. So, all the instructions may take similar amount of time.

So, as a result, we may get identical computations among the instructions. And best example for such type of instructions are considered in RISC architecture, Reduced Instruction Set Computer Architecture, which actually provides a simpler instructions and all the ALU operations are performed on register operands and all the memory operations are performed through load and store instructions.

And because of registers are involved in performing ALU operations, all the ALU operations are going to take similar amount of time. That way like, we can minimize these non identical computations in our execution and as a result we can improve the performance through pipelining. And finally independent computations.

So, in order to know whether two instructions are dependent or independent, if we use register addressing modes we can easily identify the dependency among outputs of the

previous instructions and the inputs of the next instructions. Because all we have to do is, we have to look at these register operand and then do the dependency check easily.

But if you are going to use memory addressing modes for operands in the instructions then dependency check among these operands, which are actually specified using memory addressing mode will become very complex. And so as a result, it will be very difficult to perform this dependency check and unless you perform dependency check, we cannot proceed with the subsequent instructions. So, we have to stall the subsequent instructions until the previous instructions completed and that is going to degrade performance significantly. And so as a result, in order to exploit these independent computations to improve our performance through pipelining, we need to minimize memory addressing modes in our ISA.

And as I mentioned earlier, it is easy to check dependency among registers. So, as a result, we need to use register addressing modes for our ALU operands. And this is also another reason why RISC is actually using all ALU operations using register addressing modes. So, effectively, when we want to design our ISA, we need to consider these, all these things, into consideration. So that, when we go for pipelining of instructions, we can improve the performance. So, now we will consider the pipelining of instruction execution, by considering RISC instruction set. So, in the lifetime of an instruction execution there are several stages involved.

(Refer Slide Time: 14:32)

A Simple Implementation of a RISC Instruction Set

- ▶ Instruction fetch cycle (IF)
 - ▶ Based on PC value, fetch the instruction from memory
 - ▶ Update PC
- ▶ Instruction decode/register fetch cycle (ID)
 - ▶ Decode the instruction and register read
 - ▶ Do the equality check on the registers for a possible branch
 - ▶ Compute branch target address, if needed
- ▶ Execution/effective address cycle (EX)
 - ▶ Memory reference: Calculate the effective address (EA)
 - ▶ Register-register ALU instruction
 - ▶ Register-immediate ALU instruction

Slide Number: 01/01

The first one is, we have to fetch the instruction because we know that, our instructions are stored in an instruction cache and the data is stored in data cache. We can consider the split cache mechanism for not having interference between this data access and the instruction access. And typically instruction cache will show significant special locality. And so as a result we generally separate these caches into two separate things and this is called as a split cache mechanism.

So, our instructions are stored in the instruction cache and the PC is the program counter which specifies the next instruction to be executed. So, whenever we want to start executing an instruction, we first consult the PC and PC is going to specify an address and based on the address stored in the PC, we will go to the instruction cache and fetch the instruction from the cache memory or the main memory if you do not have the cache memory at all.

Once we fetch, once while we are fetching the instruction from the instruction cache, we update PC and PC is incremented by instruction width. Let us say, if each instruction is taking 32 bits then PC is incremented by 4 bytes. So, that it is now pointing to next instruction in the sequence so, because, this updating of PC and accessing instructions from the instruction cache can be performed simultaneously, so that, we will do these operations parallelly.

So, as a result at the end of an instruction fetch cycle, we have an instruction, which is represented in terms of stream of bits and this stream of bits are stored in the register called as instruction register also represented as IR. So, once instruction is stored in an instruction register, now we need to see, what type of instruction this particular the instruction whatever we fetched just now because we know that, when we fetch instruction from an instruction cache, it consists of stream of bits.

Unless we decode this stream of bits, we cannot say what their duty or what is the role of particular instruction. Whether this instruction is going to perform read operation, write operation or an ALU operation or something else. So, that means to know the purpose of that instruction we have to decode. And so we will give the contents of this IR instruction registers to the instruction decoder and the instruction decoder is going to decode this instruction by using different mechanisms.

Typically, instruction is represented in terms of set of fields and the first field is represented as the opcode, the next sets of fields are going to specify the operands. So, we give corresponding field to decoder and the decoder is going to specify what type of instruction

this particular bit of streams, bit stream is representing. And once we know that we will perform the operation. But also this instruction represents the operands and operands also represented by using bit stream.

And so once we decode these operands then we know, what is the addressing mode these operands are represented by and based on that either we will go to register file to read the instructions or we may have to go to the memory. But to go to the memory then we will perform the other operation that is called as the Execute Stage. And also in our simple five stage pipeline, we consider that if the instruction is a branch instruction we also perform the branch condition check in the second stage itself. That means this the second stage ID, instruction decode stage, not only decodes the instructions and if it is a branch instruction we also perform branch condition check.

So, as a result by the end of a second stage of the five stage pipeline, our branch instruction completes its execution. And also we will compute the branch target address in this particular stage. Remember when we have an ALU operation in the ID stage itself, instruction decode stage itself, we will perform the register read. So, effectively our ID stage consists of decoding an instruction, to know what is the type of instruction and also we know the register addresses from that instruction.

And once we have the register addresses we go to the register file and read the register contents from the specified locations in the register file. That is called as a register fetch cycle. So, register read also will be performed in the id stage. And the next stage, that is called as an execute stage where we actually perform actual execution of an instruction. If it is an ALU operation because this is a RISC instruction set architecture. So, we will perform the ALU operations on register contents.

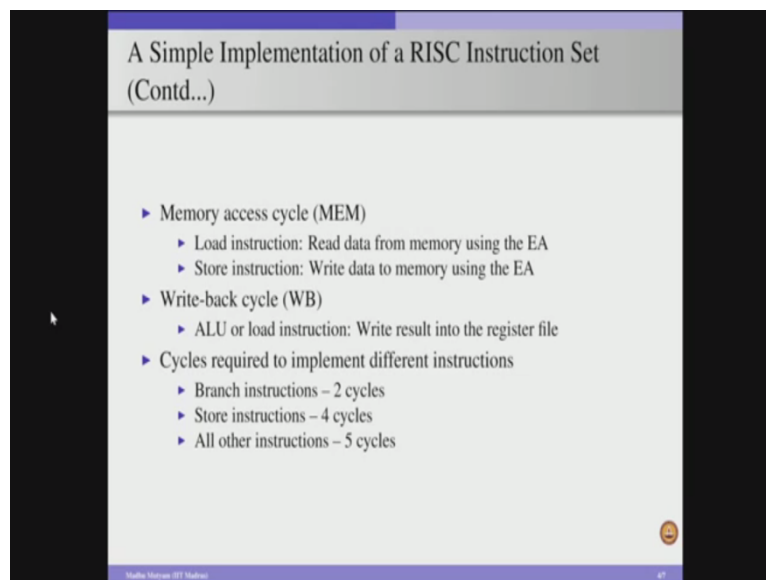
So, already register contents are read in the previous stage that is ID stage. And we give these register contents or register operands to functional unit and we perform the actual operation. And if it is a register immediate ALU instruction so, one operand is already available in the instruction itself and the other operand is available from the register file which is read in previous stage and we can perform ALU operation. And to know what functional unit to be accessed, we already decoded the opcode type in the ID stage and we used that.

And give the signal to appropriate functional unit and that functional unit is going to perform the ALU operations. And now if we are going to consider memory operations such as load

and store. So, load or store is specified in the previous stage by using the ID. ID stage, which is specifying what type of memory operations we are going to perform.

And if it is a memory operation, load or store, now we need to compute the effective address because in our instructions the memory address may not be specified directly. It will be specified as the base plus offset. So, we need to perform the add of this base address and the offset. So, as a result we use an adder to perform this addition operation and as a part of this Execute stage, to deal with the load or store instruction we are going to compute the effective address. Once we have an effective address the next stage we are going to the memory and then perform the actual operation. And that is called as memory access cycle.

(Refer Slide Time: 21:40)

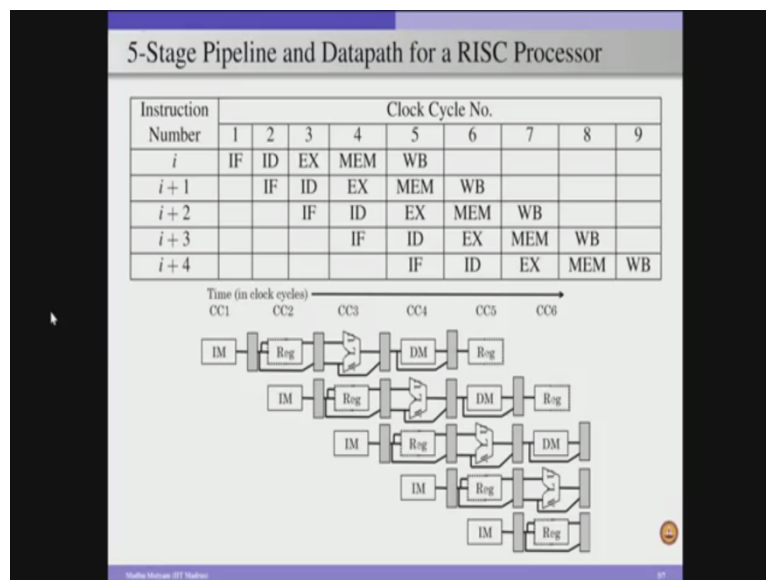


In the case of a load instruction, we go to the memory and access appropriate location specified by our effective address and read the content. Whereas, in case of store instruction. So, we go to the memory and write the contents whatever is specified in our instruction to the effective address location in the memory. So, remember in the case of memory access cycle when we are performing the load operation, we just read the data from the memory.

And we will write to the destination register only in the next stage, that is called as write-back cycle. And this write-back cycle will perform the actual write to the register for both the ALU operations as well as the load instructions. And the ALU operation, we know that the execution of ALU operation is performed in execute stage, but the actual writing to the register will be in write-back stage. So, effectively in this five stage instruction pipeline for

RISC instruction set architecture, branch instruction takes two cycles, store instruction takes four cycles and all other instruction take five cycles, for our simple five stage pipeline of RISC instruction set. Having discussed this, the pipelining of the instruction execution, now we will see how each of these stages will follow.

(Refer Slide Time: 23:16)



So, when an instruction I is issued, the cycle one it will be in the instruction fetch stage. The cycle two it will go to the ID stage. So, that now instruction fetch stage is free. So, we can issue the next instruction to this, the instruction fetch stage because we know that instruction fetch performs independent operations as compared to instruction decode which performs independent computations as compared to execute stage, which performs independent computations with respect to memory stage and all these four stages perform independent computations with respect to write back.

Effectively, all these five pipeline stages perform independent computations with respect to each other. As a result, when we have one instruction in one pipeline stage we can send another instruction to a different pipeline stage. And also we know that ID stage will be followed only after IF stage is completed. And similarly, execute stage will continue processing something which is already processed by ID stage.

And similarly, memory is going to process something which is processed by execute stage and other things. So, effectively when an instruction I is issued, in the cycle one, it will be in IF stage the cycle two it will go to ID stage. So, that IF stage is free and in this IF stage now

we issue the next instruction. So, effectively now in the second stage, in the second instruction I+1 instruction will be in the IF stage in the cycle two whereas, instruction I is there in the ID stage in the cycle two.

When we are in the cycle three the first instruction, instruction I is in the execute stage. Instruction I+1 is in the ID stage and instruction I+2 is in the instruction fetch stage. And in the fourth cycle, our first instruction is the memory stage, second instruction is in the execute stage, third instruction is in the ID stage and the fourth instruction is in the IF stage. And this will continue.

And in an ideal state scenario, so all these instructions if they are independent and all these pipeline stages take the same amount of time, so there would not be any bubbles in our pipeline. So, effectively after first five cycles, our first instruction is completed and after that every cycle, every pipeline cycle, will compute one instruction. So, effectively our throughput will be one instruction per pipeline stage or pipeline cycle time.

So, as a result, with five stage pipeline we can get, in an ideal scenario, a 5x throughput improvement as compared to non-pipeline design. And for this pipelining if we see the data path associated with our processor, it consists of several components. We know that instruction fetch requires the data to be read or instruction to be read from instruction memory. That is what this IM, IM is the hardware component, instruction memory, which supplies the instruction for our address specified in, the PC, program counter. So, for instruction fetch we are going to use instruction memory component.

And after we fetch, in the next stage, we are going to go for decoding the instruction and also register read. Effectively to perform a register read, we are going to read operands from register file. Effectively we are going to use register file component in the second stage. While we are performing register read from a register file our instruction memory is free. So, we can perform read operations from instruction memory. As a result, we can see in CC2, that is the clock cycle two, we can perform operations on these two components simultaneously.

One is reading from register file and the other one is reading from instruction memory. And once we read operands from the register file and also once we identified that the operation is an ALU operation, the cycle three ALU operation can be performed simultaneously with a register read as well as the memory read. So, effectively in cycle three all these three components can be used for performing different tasks. So, in the cycle four, if our instruction

is a store operation we have to write that to the data memory. So, we will access the data memory in that cycle.

And while we are performing operation on data memory our ALU unit is free. So, we can perform an ALU operation using an ALU functional unit. Also we can perform a read from register file. Similarly, we can read an instruction from instruction memory. Effectively in cycle four we can perform operations, four different operations on four different components. And also again, if you are considering a single cache memory to store both the data and the instructions and if the instruction, if the cache is having a single port, then we cannot perform both data write operation and the instruction read operation from the cache, because the cache is a single cache which stores both the instructions and the data.

And also it has a single port, a read port. So, as a result there is a constraint, resource constraint, and that degrades the overall performance, but whereas, if I consider the cache as a split cache where instructions are stored in a instruction cache memory and the data is stored in the data cache memory then these two caches are independent and we can perform read operations simultaneously from these two caches. And that will improve our overall performance. Effectively, in other words in this particular example, we consider a split cache mechanism so that data read and the instruction read operations are performed simultaneously on the respective memory components.

And in the cycle five, if it is an ALU operation or a load operation, we have to write contents to a register and that will be performed on register file. So, effectively we are going to perform register write operation in this fifth cycle or register file. And also because the register file is independent of the data cache, we can perform data write operation on the data cache. And also we can perform ALU operation simultaneously with other two computations.

And now we can also perform a read operation for some instruction which requires data to be read from the register file, but we know that we are already performing data write operation for one instruction which is writing to the register, that is the write back stage. So, when we are going to perform a write operation and a read operation. So, generally we are going to perform write first and then we perform a read operation. And if both read and write to the same register then automatically we have to apply some optimizations like the operand forwarding technique that, we are going to discuss when we come to the appropriate module.

Also while we are performing all these things we can fetch a new instruction from our instruction cache memory. So, that is through the instruction cache that is IM and similarly, we can continue. Effectively when we are dealing with this five stage the pipeline, we will involve different hardware components in our processor to perform these operations, such as instruction cache memory, data cache memory, register file, ALU units and so on.

And if at any stage, if a component is not available then we are going to get pipeline stall and these are called as hazards and those things we are going to discuss in the next module. As long as there is no resource conflict. As long as there is no dependency among instructions and as long as all the stages of the pipeline are going to take the same amount of time through a k-stage pipeline, we can get a k fold increase or improvement in our the pipeline performance as compared to a non-pipeline design. Now, we will consider an example to see what is the improvement that we can get through a pipeline design as compared to a non-pipeline design?

(Refer Slide Time: 32:50)

An Example

- Assume that a non-pipelined processor has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, resp. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in instruction execution rate will we gain from a pipeline?

$$\begin{aligned} \text{Avg. Execution Time}_{NP} &= \text{Clock Cycle Time} \times \text{Avg. CPI} \\ &= 1\text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\ &= 4.4\text{ ns} \end{aligned}$$

$$\begin{aligned} \text{Speedup from pipelining} &= \frac{\text{Avg. Execution Time}_{NP}}{\text{Avg. Execution Time}_P} \\ &= \frac{4.4\text{ ns}}{1.2\text{ ns}} \\ &= 3.7 \times \end{aligned}$$

So, the problem statement is we consider a non-pipeline processor that has a 1 ns clock cycle and it uses four cycles for ALU operations. And branches also take 4 cycles and for memory operations it takes 5 cycles. And also we are going to execute some application in which we have 40% of the operations are the ALU operations and 20% of the operations are branch operations and 40% of the operations are memory operations.

And also we assume that because of the clock skew and the setup so, this pipeline registers are going to take 0.2ns. So, ignoring any latency impact, so, what is the speed up we get, if the instructions are executed through pipelining? So, if we are considering a non-pipeline design, we know that there are 40% of instructions are ALU operations and which are going to take 4 cycle latency. And 20% of the instructions are branch instructions which are also going to take 4 cycle latency.

And 40% of instructions are memory operations which are going to take 5 clock cycles. And we know that each clock cycle is taking 1ns. Effectively the average execution time for our given application with these characteristics, take 4.4ns, to perform, to execute a single instruction.

Whereas, if we are considering a pipeline design. We know that in a pipeline design and also we are also considering a scenario where. So, there are no dependency and other things. So, if we are considering such idealistic scenario and we know that each clock cycle is going to take 1ns. And also each stage is having an overhead of 0.2ns because of this pipeline registers. Effectively we can perform execution of an instruction with a delay of 1.2ns if we are considering a pipeline design.

So, effectively, we can perform an instruction execution with 1.2ns if we are considering a pipeline design whereas, 4.4ns, if we are not using a pipeline design for this given application. So, as a result the speed up because of the pipelining is 3.7 times. So, though we are considering the 5 stage pipeline we are not getting a 5x improvement because of the overhead associated with the pipeline registers as well as non-uniform the computations or otherwise like because of this external fragmentation because some instructions or some types are taking 4 pipeline stages.

Some are taking 5 stage pipeline altogether and because of that we get overall speed up of 3.7 times as compared to a non-pipeline design. So, with that, we conclude this module on fundamentals of pipelining. And in next module we are going to discuss, what type of hazards we get with the pipelining and how to overcome those hazards.

Thank you.