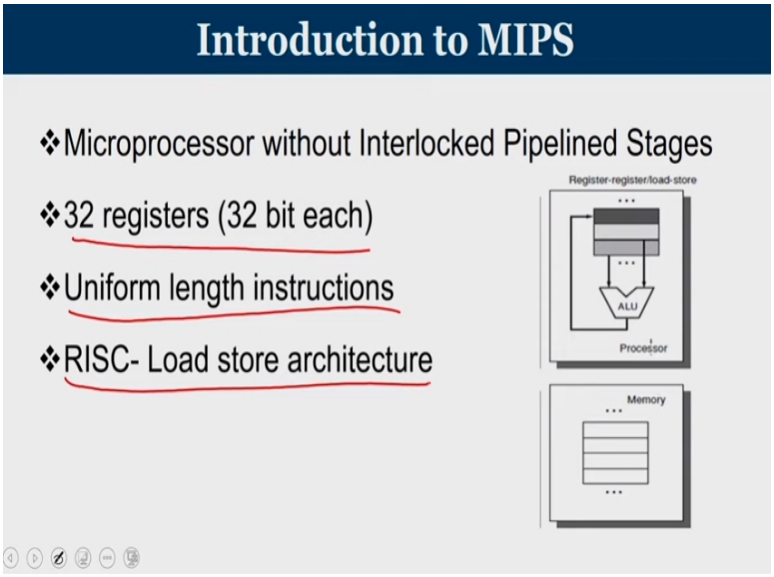
Advanced Computer Architecture
Prof. Dr.John Jose
Department of Computer Science & Engineering
Indian Institute of Technology – Guwahati, Assam

Module No # 01
Lecture No # 03
Introduction to RISC Instruction pipeline

Welcome to the third lecture of advanced computer architecture course. In the first two lectures, we try to recap the basic conception computer organization and few performance evaluation methods. This lecture onwards we are moving into the advanced architecture concepts and to begin with we will have instruction pipeline as the point of discussion for this lecture.

(Refer Slide Time: 00:58)



Moving straight into the topic in order to understand the concepts of instruction pipeline, let us first take a case study of a risk microprocessor. MIPS is the risk microprocessor that we are going to have for our case study in understanding the instruction pipeline principles. Processes are broadly classified into RISC architecture and CISC architecture based upon the implementation of various instruction set.

Let us try to understand what is the basic difference between a RISC machine and a CISC machine if all the instructions are of uniform length and they are also represented in a uniform length, instruction and coding and more or less if they take equal number of cycles to complete

then, they basically belong to the property of RISC architecture. It is known as the reduced instruction set computer where a task is represented as a sequence of simple instructions in the case of a RISC.

And when you move on to the CISC architecture, maybe with a few number of complex instruction we can represent the same task. For example, consider the case of a task wherein you how to find out square root of a number. If we have a single instruction, wherein you give the number as the operand and that will take care of finding out the square root, then, it is actually a very complex operation that is being represented by a simple instruction.

Whereas, the same square root can also be done with the help of multiple symbol operations. And that basically can be considered as a classical example of how a RISC processor works, a processor cannot support a complex instruction like a square root, but the task of finding out square root can itself be done with the help of simple smaller operations. Similarly, finding out the largest among the block of numbers it can be represented as a single instruction or it can be a sequence of simple operations.

In the case of a CISC architecture, you have instructions of variable length and they will take a variable number of cycles to complete. So, in today's lecture, what we are going to consider is a very popular RISC architecture known as the MIPS microprocessor. Let us try to understand what are the important architectural features of this MIPS microprocessor. MIPS is the abbreviation for microprocessor without interlocked pipeline stages.

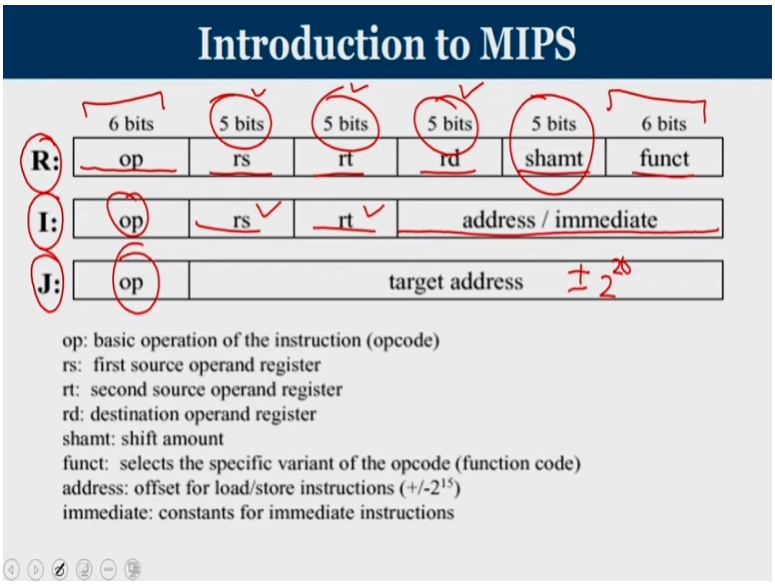
This processor has 32 registers and each register can accommodate 32 bits each the peculiarities all instructions are of uniform length and it follows a load store architecture when we had the discussion on what are the various instruction set architectures based upon where operand is located when it comes to an ALU instruction, we have seen about stack architecture, accumulator architecture, register memory architecture and register-register architecture which is also known as load store architecture.

So, the peculiarity of load store architecture is it is only a load and store instruction that is going to access the memory when it comes to ALU operation, both the operands are available inside the registers. So, we are going to have our further discussion on understanding of instruction

pipeline with the help of the MIPS RISC architecture wherein it has started to registers each of capacity 32 bit the registers are named us r0, r1, r2 etc., up to r31.

All of them are taking 4 bytes, all instructions are represented as 4 byte and they are of uniform length instruction and they follow the RISC loads to our architecture.

(Refer Slide Time: 05:19)



Now, let us try to understand how instructions are represented in the MIPS microprocessor. Like what we have already mentioned, every instruction in MIPS is represented in 32 bits, and MIPS as basically three types of instruction the R type instruction, I type and the J type instruction. R type instruction is also known as register type instruction where it the first 6 bits of the 32 bit instruction will represent the OP code and then we have the source operand rs and rt and rd is the destination operand.

And then you have the function called and the shift amount this section this 5 bit is used only in the case of a shift operation it will tell you how many bits you how to shift the operand. So, when it is an add operation for example, this will tell you it is an add and what category of add the addressing mode will be specified in that function code and these 5 bits each will tell you what is the register that you are going to talk about.

Since there are already 32 registers, the first 5 bit will tell you what is going to be the registered name for the first operand next 32 will tell the registered name for second operand and the last 5

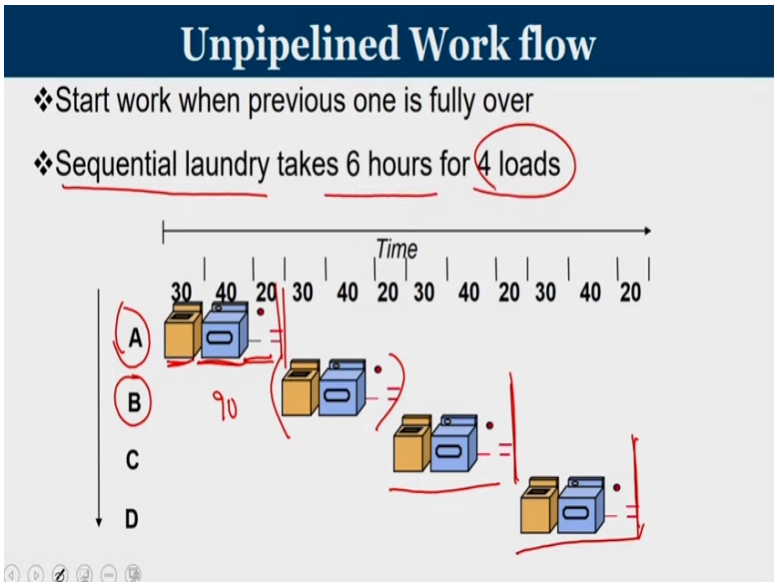
bit will tell you the registered name for the destination. Now, when it comes to I type, we will be making use of 16 bits, the last 16 bit is used to represent a portion of an address or an immediate value and the first 6 bit is op code.

And you will be having one source register and the other one is used as the base register when it comes to jump instruction you have a 6 bit op code followed by the remaining 26 bit is called target. So, any jump that is possible in

$$\pm 2 \wedge 26$$

has been permitted here. So, broadly we have three types of instruction, R type, which has 3 register operands, I type, which has 2 register operand under 16 bit immediate value or an address and the J type which has a 26 bit relative jumping address and an op code.

(Refer Slide Time: 07:31)



We will now try to understand the concept of pipelining in general. Prior to understanding about how an instruction pipeline happens, I would like to draw your attention to a laundry example. So, consider the case that we wanted to wash the clothes for the person A let us say the whole laundry operation is consisting of a washing unit what is been so represented with this color, the yellow color and the washing will take 30 minutes of time after the washing, We are going to take the clothes from the washer and going to put it in the dryer where the dryer will take 40 minutes to complete the task and then we have 20 minutes to fold the clothes. So, altogether it will take 90 minutes for the clothes of Person A to be washed. Once person A has completed the

washing, then the person B is going to start which will be taking 30 minutes, 40 minutes and 20 minutes each.

And once B done, then the work of C has been taken and then you are going to complete the work of D each of them are going to take 90 minutes each. So altogether when you are going to do the work like this it is called a sequential laundry, where the washing process of the second workload will start only if the complete work associated with the previous workload is over. That means washing of B will start this entire process will start only if A is completely over.

Similarly C will start only if B is completely over and D will start only if see is completely over. So, each of the sequential laundry is going to take 90 minutes each and then we have total of 4 such workloads. So

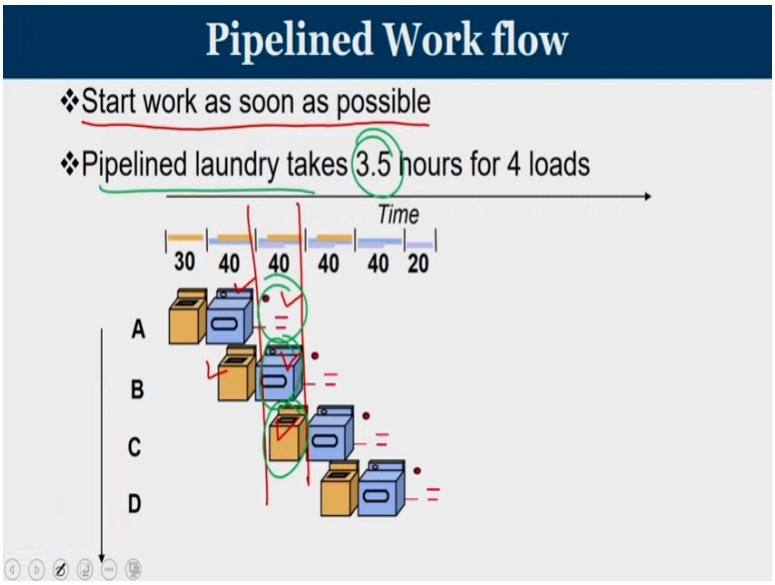
$$90 \times 4 = 360$$

that is eventually 6 hours will be taken to complete 4 loads. Now, in this example, we have to understand that the entire laundry process associated with a single task say A has to go through 3 sub units a unit called washing which will take 30 minutes to complete the washing process.

The second unit will take 40 minutes for drying and the last unit will take 20 minutes for folding. The peculiarities all these 3 tasks are been done by independent separate units and after the task is over, you how to take the clothes from 1 unit and put it in the other. Now, when the second stage that is a drying process in is in progress the washing unit is free as well as the folding unit is free.

If you wanted to pipeline the whole operations, then the idea is we wanted to make sure that every unit is busy. So when the drying of workload A is in progress, the washing unit can eventually start the washing for workload B and once the drying is over for A once the folding process starts then the washed clothes of load B can be shifted to the drying operation that is exactly what we are going to with a pipelined laundry.

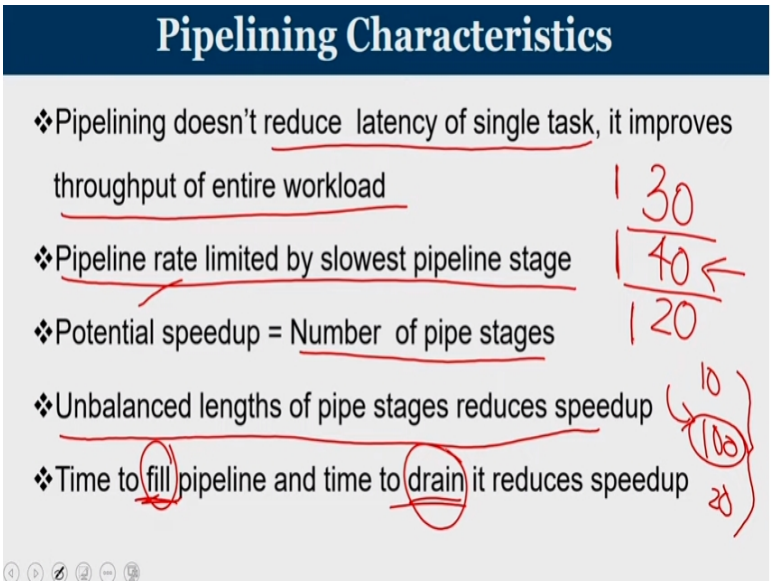
(Refer Slide Time: 11:01)



So, in the case of a pipeline laundry, the concept is start work as soon as possible. If the functional unit that is going to carry out the work is available start the work. So, when the drying is in progress, the washing for B will start when the folding is in progress for A the drying for B and washing for C is parallelly happening.

So, if you look at this particular time window, then all the 3 units are busy, the folder is busy with completion of folding of the clothes of A the dryer is taking care of the drying of B and the washing unit is taking care of washing of load C. This makes sure that every unit is itself busy and by which we can improve the throughput of our system. So the entire pipelined laundry is going to take only 3.5 hours to complete the same task. So what are the savings that we get what are the characteristics of pipeline?

(Refer Slide Time: 12:05)



Pipelining does not reduce the latency of a single task as far as A or B or C is concerned in the previous laundry example the latency associated with a single task is not reduced. But the throughput number of tasks that we can complete is significantly improved. And the pipeline rate is limited by the slowest in the pipeline stage. In the previous example, we have seen the first unit is going to take 30 units, the second one is going to take 40 units and the last one is going to take 20 units.

So, considering this I can start new work only at intervals of 40 because it is a second unit that is the slowest. The pipeline rate is limited to these slowest pipeline stage. So, in this case, at regular intervals for 40 minutes new work can be in shifted and the speed up that we are going to get from a pipeline is equal to the number of stages here we have 3 stages or roughly the speed of this 3 times.

Now, when you have an unbalanced length of pipeline, for example, here we have 20 here we have 30 the other one is 40 it is not uniform length, this is slightly creating some performance issues, what if you are going to consider something like 10, 100 and 20 let us say these are the 3 stages of the pipeline, one stage is having very high delay whereas others are having very slow delay, here also we can initiate new task only once in 100 unit of time.

So, this is an unbalanced, this is example of an unbalanced pipeline, these kind of cases should be avoided or your pipeline will achieve its full performance only if all the stages are having

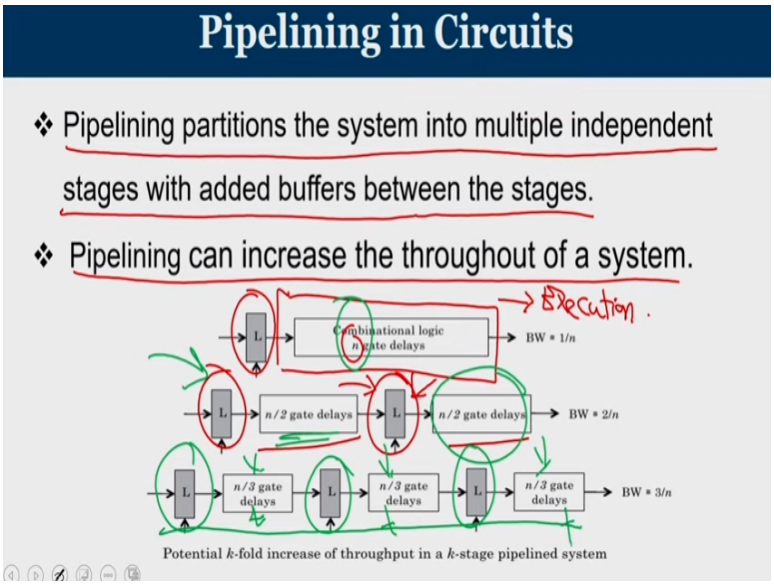
almost an equal pipeline latency. And then the last aspect of pipeline is from 1 unit of pipeline from 1 sub component of the pipeline, I how to take up the task and then feed it to the next one. So that is an extra overhead time to fill the pipeline and time to drain the pipeline.

Once the task is done inside the pipeline unit it has to be drained out from that unit and it has to be fed into the new one. So, this feeling and draining is going to consume little bit extra overhead. Having understood the concept of pipeline with an example of a laundry now, we will try to correlate the concepts that we have discussed just now, in the case of pipeline circuits. We already mentioned that with respect to execution of an instruction, there are various sub stages like instruction fetching, decoding, executing the task, storing result and all.

All these are done with the help of logic gates. So, if you consider the entire operation associated with execution of an instruction like fetching followed by that we have the decoding operation and then we have the operand fetch or execution all these stages if you consider all these as one unit we have a single monolithic unit which will take. So, this particular combination system so this circuitry will be having an address as its input and then the output is the instruction that is present in the address is fetched, decoded, executed and the result is being stored.

That is what we have if it is a single monolithic design. Now, that is not a modular structure. From that can I sub divide entire monolithic big design into smaller sub components where each of this component is taking care of one aspect of the execution of the instruction, we are going to divide the entire circuit into multiple smaller components.

(Refer Slide Time: 15:58)



So, when you applied pipelining, pipelining partition the system into multiple independent stages with added buffers between the stage. As we are already discussed, pipelining can increase the throughput of a system. Consider this diagram wherein just imagine that this is the big circuit that will take care of execution of an instruction. So, you are going to give the address this address is been fed into this logic and this logic is going into memory bringing the instruction decoding it and doing all operations needed with it,

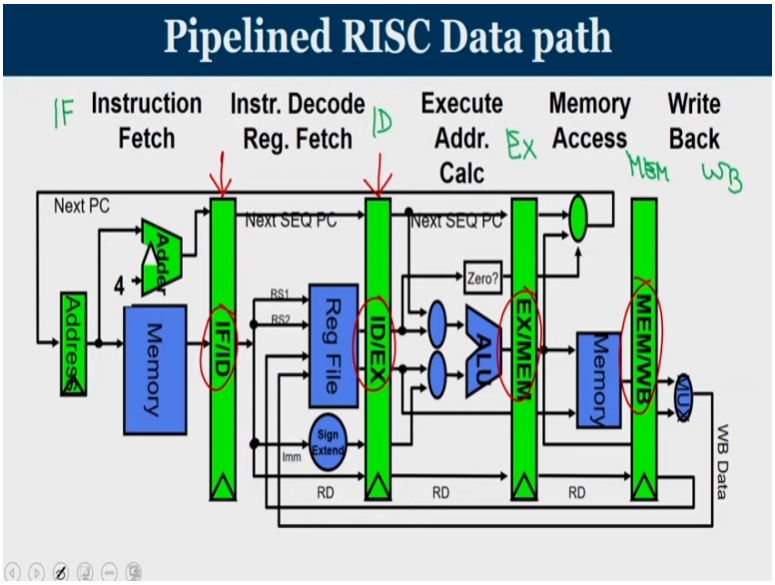
At the end, we are going to have a completion of the execution that is been done. Now, the second line tells that rather than considering the entire logic, as a big logic with n gates, can I divide them into roughly 2 half each with n/2 gates and providing a buffer in between them. So, the same task now I am going to do it as two sub task. So, the first the left side sub task consists of roughly n/2 gates, it will do some kind of an operation and whatever partial operation that is been done, the result is kept in the latch.

And after that, when it has been triggered, the second half is going to take the spatially processed result from this L and do the operation. When the second unit is taking care of the operation, when this unit is busy doing the operation, the first unit is ideally free, it can take up a new task. Similarly, I can divide further also and this whole approach of dividing the actual circuit which was n-logic gates into smaller sub components and to interface with them with latches is called pipelining inside the circuits.

Essentially this is the same thing what we are going to look into, can I divide my entire execution into smaller components like instruction fetch, instruction decode, execute like that. Now, I would like to draw your attention to the MIPS microprocessor which we have just introduced and incorporating the pipeline principle that we discussed. So, in an un-pipelined RISC MIPS microprocessor.

There were 5 different components which will take care of the execution of a single instruction, these are instruction fetch, followed by instruction decode and register read the third stage is execution or address calculation, forth one is memory access. And the fifth one is right back into the registers.

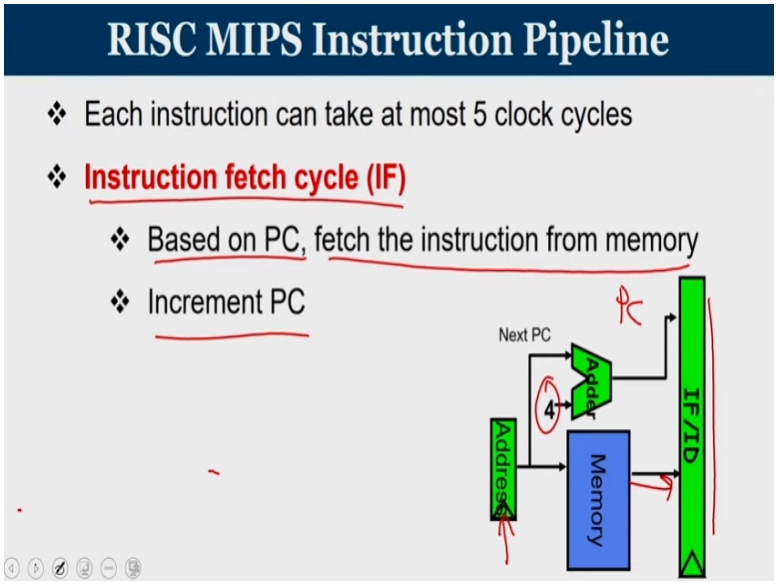
(Refer Slide Time: 19:03)



This is roughly what you can see from this diagram, there exists some units which will take care of the instruction fetching operation. And then we have the instruction decode and the register fetch operation then it is execute or address calculation followed by memory access and then at the end, it is the writing back. Now, when you are going to pipeline this, then with whatever you have seen the various 5 stages between 2 stage, Let us stage 1 and stage 2, we are going to add interfacing in between them and that is called the pipeline registers that is kept between various stages. We are going to give names for each of the stage they are called IF, second stage is ID, third stage is EX, fourth stage is MEM, fifth stages write back. So, depending on the stages, this pipeline registers are also given a name ask for this, this is a pipeline register that is in between IF and ID. So, we call it as IF / ID register, this is a register that is between ID and EX. So the

name is ID / EX. Similarly EX / M, MEM / WB are the name of the pipeline registers. We will now try to see each of these sub stages in detail and try to understand what are the operations that are being done.

(Refer Slide Time: 20:38)



So, the first one is known as instruction Fetch, this is the circuitry associated with the instruction fetch. What it does is based on the program counter value fetch the instruction from memory. So, this is the address that has been supplied by the program counter, it goes to the instruction memory or memory in which the instruction is been stored and the contents of that is been taken and that has been stored in the pipeline that IS / ID.

In the meantime, we are going to update the PC such that we will come to know from where the next instruction is to be brought. Since it is a RISC architecture and all instructions are taking uniform length the next program counter value can be easily obtained by finding the

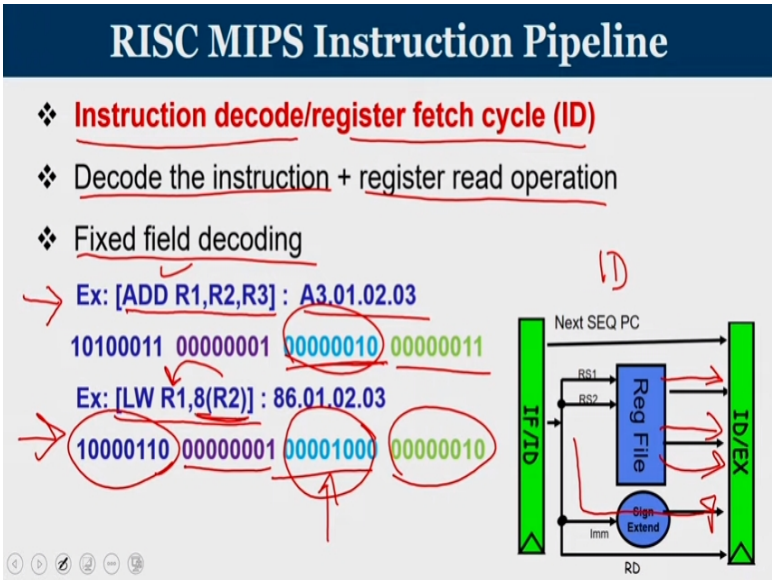
$$PC = PC + 4$$

So the whatever is the new PC that is been added to 4 so it is very easy as long as all instructions are have uniform length, updation of PC is a relatively simple step.

If the instructions were not of uniform, like what we see in CISC, some of the instructions are 1 byte some are 2 bytes, some are 8 bytes like that, then after bringing the instruction, we have to understand how long the instruction is and only then we can increment the value of PC. So the basic operation that is been done in instruction fetch one is, based on the program counter value

go to memory, bring the instruction and update the program counter value by adding 4 to it. This match is done in the instruction fetch state.

(Refer Slide Time: 22:14)



Now, moving on to the second stage, the second stage is also known as instruction decode as well as register fetch cycle which is abbreviated as ID. So, whatever you have done in the instruction fetch stage, an instruction is available in the IF/ID register. And from there, you do some operation at the end of this unit the ID/EX register will contain the resultant of the decoding and register fetch operation.

The first task associated with ID stage is decode the instruction and if at all there are any register values that are to be read prior to execution operation, then read the register content as well. It is also known as fixed field decoding. For example, consider the case that you wanted to perform an operation add R1, R2, R3 let us say the binary representation of it in hexadecimal is A3.01.02 and 03. As an example, consider the case that this is the 32 bit value which correspond to an add R1, R2, R3 operation. Let us imagine that this represents op code and this represents R1 another one represents R2 and this represents R3, R3 this is R2 and this is R1. So, when we get 32 bit instruction, the decode unit can cut this 32 into 4 components each of 8 bit each. And the first 8 bit is analyzed to understand what the op code is.

We are already seen that op code means what is the operation and the remaining three 8 bit units will tell what are the source operands this can be done parallel. So, I should not even wait for

what the opcode is, I can go and find it out the 8 bit values that will tell you what is the name of the registers. This process is not us fixed field decoding as the number of bits required to represent the op code and represent the operands are been predefined.

It can be also a case that if the instruction is going to be a load word instruction, we see that this is a RISC microprocessor and RISC use load and store instructions in order to transfer contents from memory. So load word R1, 8(R2) let us say it is instruction whose meaning is, I have to transfer the contents from a memory location into a register, let R1 be the destination register. So, the contents of memory location whose address is R2 + 8.

R2 is another registered, go to R2, find its 32 bit value it is the content of R2, add 8 to it, that is the absolute address. Go to the memory location, transfer the contents into R1, this is called load word. Now, if it is a load word instruction, then this portion will tell you that it is a load word, this will tell it is R1 this is going to tell it is 8 and this is telling it is R2. So, in this case content of R2 is to be read and this should not be treated as r8.

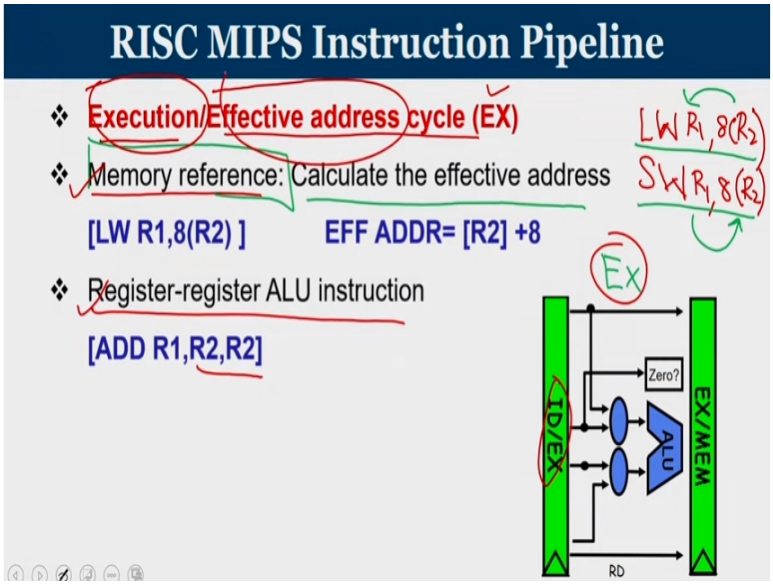
The immediate value of 8 has to go. So, if the instruction is of an add type, then this has to be R2 if the instruction is of load type, then rather than considering this as R8 it should be considered as an immediate value. To make things simple, without decoding the second stage would not know whether it is an add operation or it is load operation. If it is an add operation, then it has to be understood as R2, if it is a load operation, it has to be understood as scalar value 8.

So, both the possibilities are being considered and once you understand what is operand, in this case, the value of R3 as well as the content of R2 and R3, both are read from the register file and they are brought into the ID/EX register. So, let us try to understand what this operation is been done.

In the stage, in an ID stage be decoding is done to understand what op code and operand is and after identifying the operand names it has been gone into the register file and the content of the source operands are being fetched from the register file under kept the ID/EX register. Once we come to know that it is a load or a store operation, then a displacement value a 16 bit displacement value is coming as well as the name of the register is coming.

So, then the content of the base register only is going to come so that is been supplied. So, depending on the op code, the activity that is being carried out in the ID stage is going to vary either it will find out the values of two of the source operand let us say R2 and R3 and bring its contents to ID/EX register or it will transfer a 16 bit immediate value and the content of register when it happens in the case of a load word.

(Refer Slide Time: 27:43)



In any case, the values will reach the ID/EX register. Now, the third stage is known as execution stage, it is also known as effective address stage. The abbreviation given to the stage is EX. The operation that is carried out in the EX will vary depending upon what is the operation? Consider the case that the operation is a memory reference that it can be a load word operation or a store word operation.

We know that in the case of a load word or a store word, there is a registered name that is associated. And so consider the case that we are going to take an example where the load word and store word is been shown like this, where the load word is

LW R1, 8(R2)

or if it is a

SW R1, 8(R2).

What does it mean, in the case of an

LW R1,8(R2)

the address is given by R2 + 8 that is been transferred to R1.

It is a copy from memory to processor, that is a load operation whereas in the case of a store the content from R1 is moved to memory whose address is given by 8 of R2. In both these cases that is if it is a memory reference operation, then in the execution stage, in the third stage of the instruction pipeline content of R2 is added with the immediate value 8 that is calculation of the effective address is what happens in this cycle.

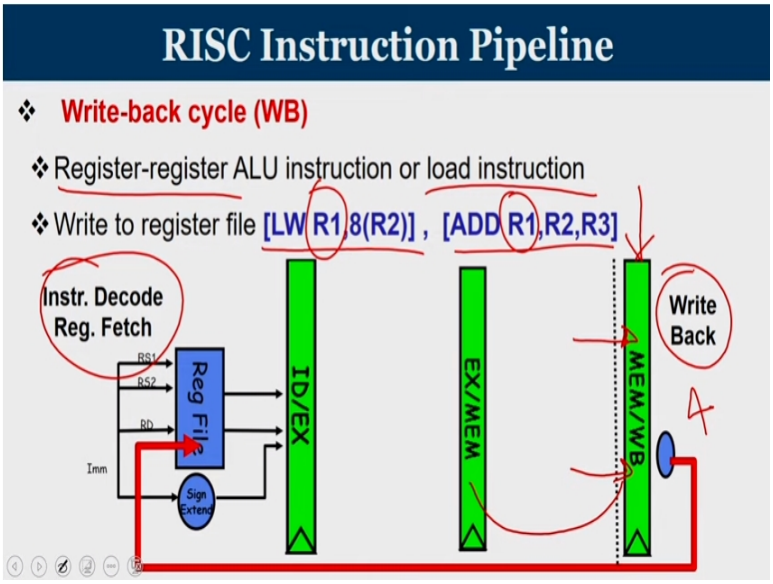
Whereas, if the operation is a register-register ALU operation like what we have seen,

ADD R1, R2, R3

then it is actually adding of R2 is going to happen here. The content of R2 is added with the content of R2. So, if it is an arithmetic logic cooperation, wherein both operands are registers, then the actual execution is carried out in the EX stage, whereas if it is a memory reference, then effective address is computed.

So, if you look at the name, either the execution take place or the effective address calculation takes place in third stage.

(Refer Slide Time: 30:09)



The next is the fourth stage, it is also known as memory access cycle called MEM stage this stage is used only by load and store instruction if it is a ALU instruction, then no operation is carried out in the MEM stage, whatever was the result that was available in EX/MEM register, that is simply by passing into the MEM right back register, if it is a ALU operation, if it is a load

operation, whatever is the address that is calculated in the previous stage from the address the contents are loaded.

If it is a store operation, then whatever is the address and whatever is a data that is available in the EX/MEM register that is being written into memory. So, if it is a store, it gets over here, if it is a load, then the value is taken from memory. Just to summarize, the MEM stage is the fourth stage by the time you reach the MEM stage, if it is ALU operation already the execution is over. If it is a memory operation, the address is computed for memory operations using the address the memory access takes place.

If it is a load operation, then with respect to the effective address that is calculated in the previous cycle, go to memory access the location and take the value and the value is now stored in this MEM write back register. For a store instruction already the operand the value is available at the EX/MEM register as well as the address is also available. So with that address with the data, go to memory and complete the store operation.

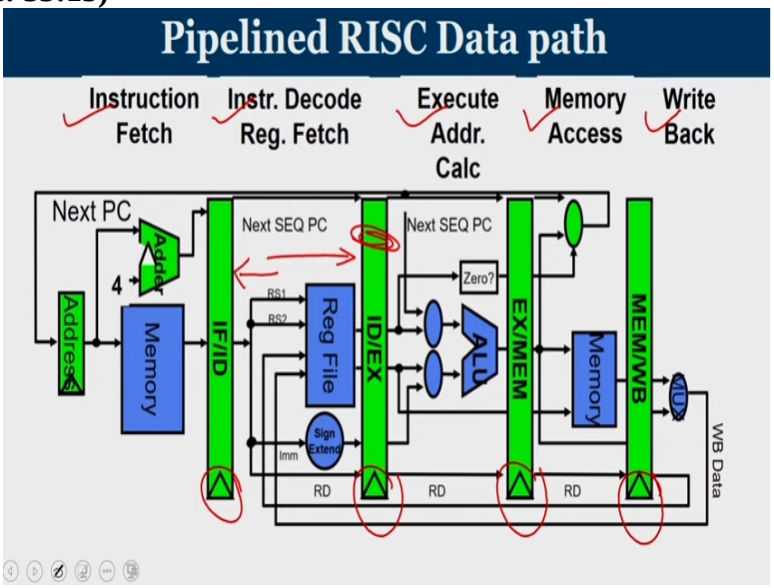
So store operation is going to get all over in memory and nothing comes out to the MEM write back register. And the last stage is known as write back, any write operation that needs to be performed into a register then it is the fifth stage that will take care of it. So for register-register ALU operations or load operations, the destination is a register. For example, LW R1, 8(R2) or add R1, R2, R3 in both these cases.

In the case of add R1 is a destination, in the case of this load word R1 is a destination, Writing into R1. So, what is to be returning to r1 that will be available in the MEM write back register. In the case of a load, the loaded value is available here. In the case of an add, the resultant of the add is transferred from EX/MEM to MEM write back this writing happens at the fifth stage. So any reading from a register that is happening in the ID stage and any writing on the register that is happening in the write back stage.

So here logically it is being shown with this red color arrow that moves on the write back into the register file that has been accessing instruction decode, it does not mean that the register file is present inside instruction decode the instruction decode stage will access the register file and

read the values whereas the writeback stage will access the register file for writing the result into it.

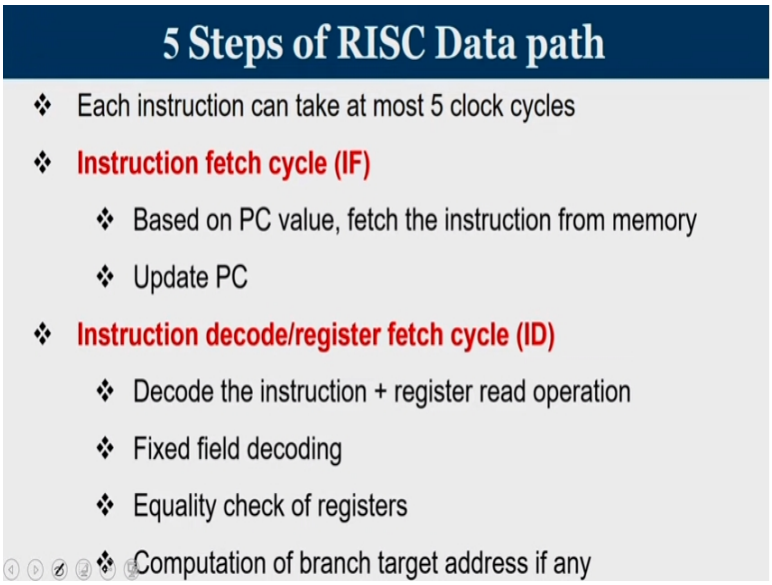
(Refer Slide Time: 33:15)



So that constitutes the 5 stages of instruction pipeline you fetch the instruction decode and read the registers execute or calculate effective address for load store perform memory access and write the result back into registers. So, we can see that all these pipeline stages are having the pipeline register and they are connected to the same clock. Once the clock is ticking, whatever is available in your input pipeline register that access the input to the combination block perform the operation right the result into the output pipeline register.

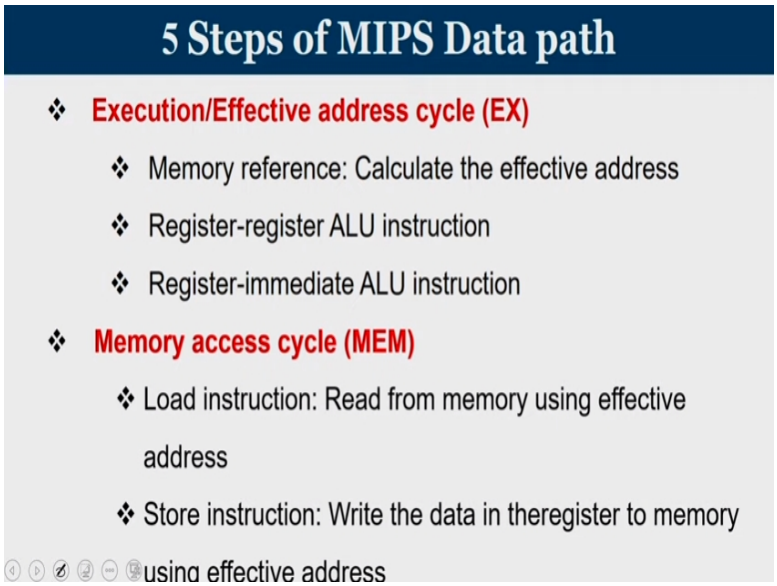
So, the output pipeline register this is the output pipeline register for the second stage and this is the input pipeline register for the second stage, the output pipeline register for the second stage is acting as the input pipeline register for the third stage and so on. So, every unit at the beginning of the clock cycle will accept the input from the input pipeline register, perform the operation using its combinational block and write the result in its output pipeline register, which in turn needs to be used by the next subsequent stage.

(Refer Slide Time: 34:26)



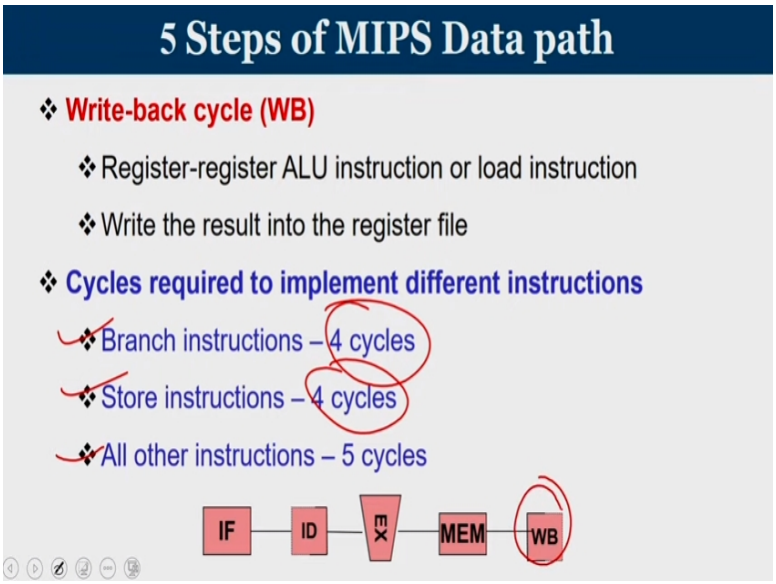
So, we will try to summarize the 5 steps of RISC data path you have an instruction fetch, were in PC, based on PC value their location has been fetch from memory and updation of PC happen. Instruction decode, decode the instruction and read the register contents, it is a fixed filled decoding, if you wanted to have equality check off two registers, let us say R1 = R2 that also is done in the stage and computation of branch target address also happen in the ID stage.

(Refer Slide Time: 34:54)



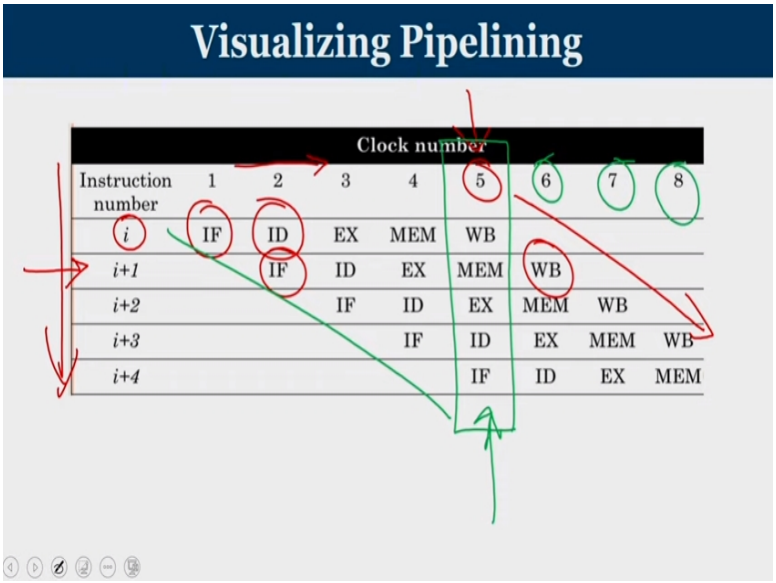
The EX stage take care of effective address calculation for data access instruction and for any other ALU operation, the actual instruction is being carried out here for MEM access, accessing the memory using the effective address that is computed using the EX stage.

(Refer Slide Time: 35:10)



And the write back stage writes the result of an operation that has to be reflected inside a register. So, in short branches will typically get over in 4 cycles, stores also will get over in 4 cycles or other instruction will go to the fifth cycle where in a write back is happening.

(Refer Slide Time: 35:34)



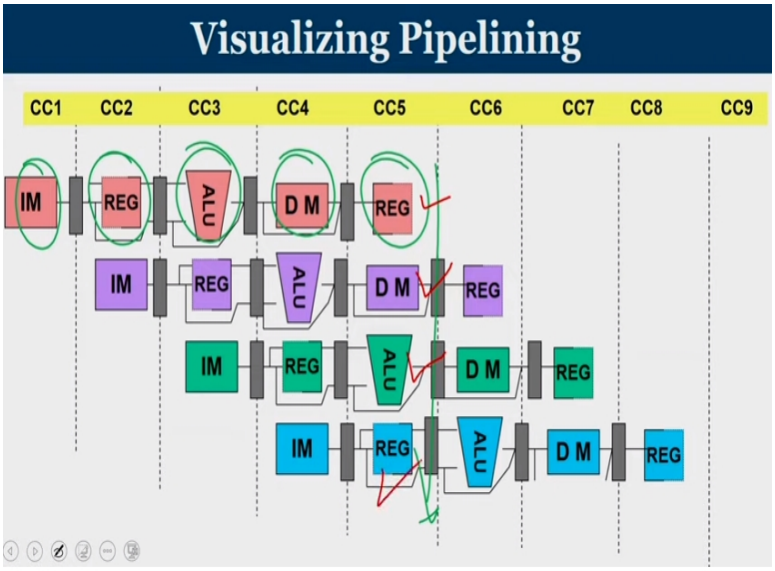
So, if you try to visualize a pipeline, consider the case wherein on X axis the clock cycle numbers are being represented and on the Y axis we have the instruction counts each of the instruction has to pass through 5 stages. So consider instruction number i where on the first clock cycle the instruction fetching is happening. And then second clock cycle perform decoding then followed by EX/MEM and right back at the end of the fifth clock cycle., the ith instruction is reaching it is write back stage or it is completing the instructions execution.

When we applied pipeline principal in i + 1th instruction goes to instruction fetching in the second clock cycle, when the first instruction is in the decode stage, the second instruction is in the fetch stage like that, the second instruction will get over in the sixth clock cycle. So, after the initial delay, the first instruction is getting over at the fifth clock cycle thereafter, in every clock cycle one-one new instruction is reaching the write backstage or the instruction is getting completed.

If you look at the fifth clock cycle, you can see that that WB stage the write back stage is dealing with the write back operations of the first instruction, the MEM stage of the instruction pipeline is taking care of data memory access of i +1th instruction, the EX stage is taking care of the execution stage of i+2 instruction, the ID stage is taking care of the decode and register reading of the i + third instruction and IF stage is taking care of the fetching of the i + 4 instruction.

So, all the 5 units of instruction pipeline are busy handling their respective operation and they are busy with 5 different instruction. So we can see that multiple instructions are already there in the pipeline. And this will improve the performance because the throughput has substantially increased for every clock cycle one-one instruction is getting over.

(Refer Slide Time: 37:46)



This is yet another visualization of the pipeline. Previously we were telling what are the operations to be done here this diagram shows which are the functional unit that are busy. First

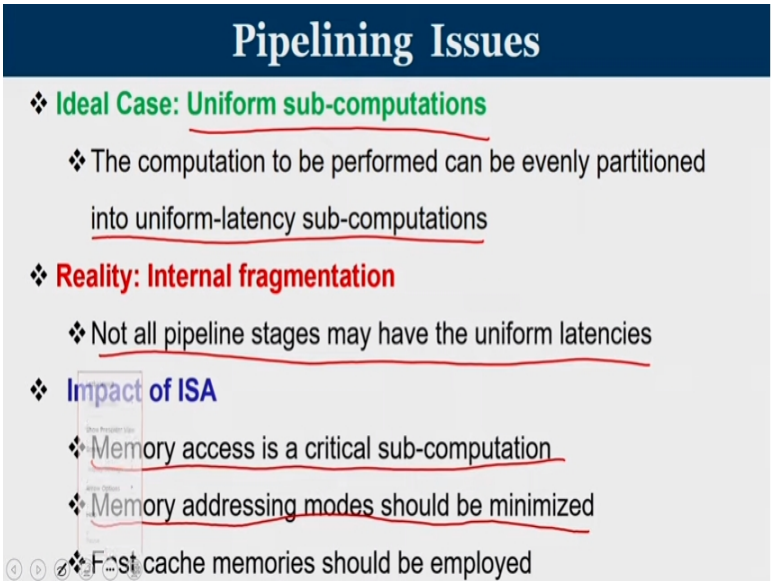
one is the instruction memory, so you fetch the instruction from the instruction memory and then you decode and read from the registers use ALU for address calculation or for arithmetic operation, then you have the data memory and then you write the result into registers.

So, if you look at down the register file is being busy at clock cycle number file, the register file is been used by the first instruction for the write operation, the data memory is used by the second instruction for a memory access, the ALU is used by the third instruction for an ALU operation or for an address calculation. And the register file is again used to for a read stage in order for the register read for forth instruction and so on.

So, we have seen what are the 5 different stages associated with the instruction pipeline of a RISC microprocessor. So, this shows the ideal case, wherein at every cycle, we should be able to start a new instruction and after the initial latencies is over for every cycle one-one instruction needs to be completed. And we make sure that every instruction has to be represented as a sequence of 5 sub operations.

It is a bit challenging design so having said all the rosy things or things the promising things about an instruction pipeline, let us know try to understand what are the challenges in implementing this.

(Refer Slide Time: 39:24)

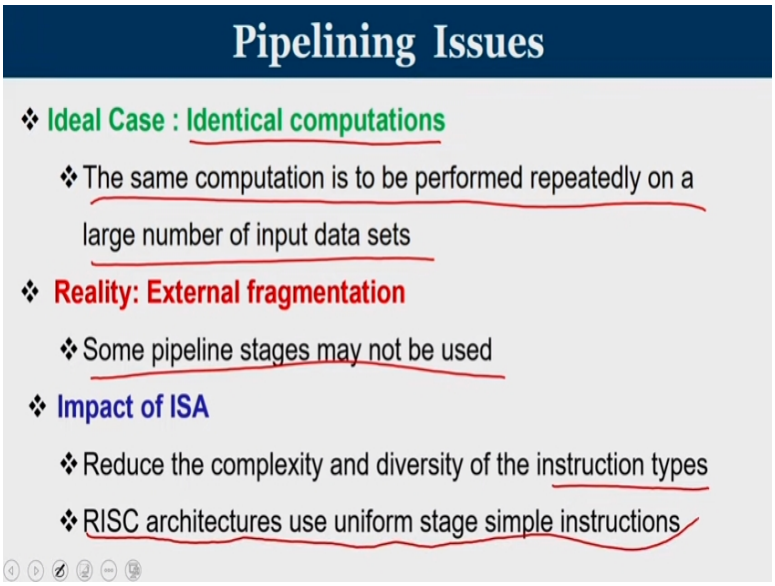


Filing to pipeline issues, the ideal case we feel that uniforms sub computation. The computation to be performed can be evenly partitioned into uniform latencies sub-computations, we have seen 5 sub computations that are associated with execution and our assumption that all these 5 are going to take same amount of time. But in reality, we will have internal fragmentation not all pipeline stages may how uniform latency.

So, how can you resolve it wherever possible, memory access is a critical sub computation when you are going to access memory. Since memory is a slower device, and processor is much faster, any kind of access to memories is going to have take more amount of time, unless we heap very high speed cache memories, but it is not possible always to accommodate all your instruction and data in cache memory due to its limited size.

So, wherever possible, try to reduce memory computation. But we are in the case of a RISC architecture, we use only load and store instruction that take care of memory. So, memory addressing modes wherever possible, should be minimized. faster cache memories has to be employed, these are the implication of instruction at architecture.

(Refer Slide Time: 40:38)



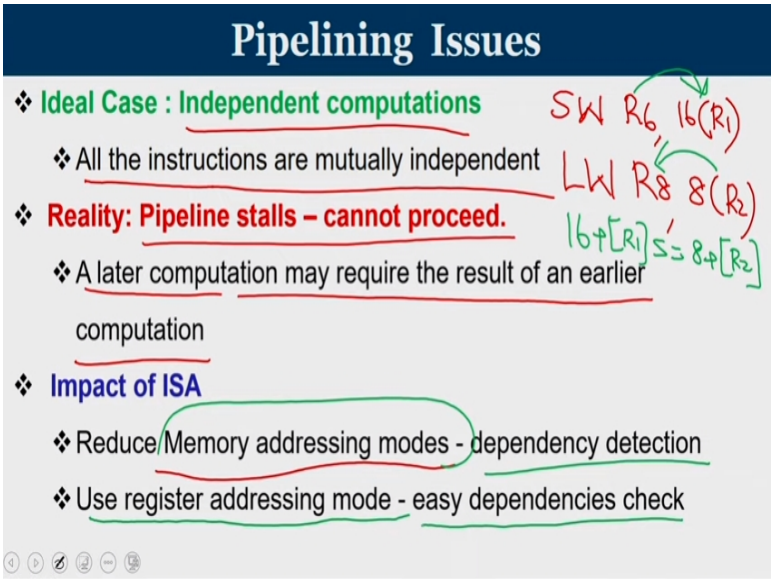
If you look into the second issue of pipelining is the ideal case say that identical computation, the same computation is to be repeatedly performed on a large number of input data sets. So, what we feel is we have 5 different stages and these 5 stages have to be done for every instruction that

this processor is doing. In reality, we know that some pipeline stages are not used by every instruction.

For example, the MEM stage is not used by ALU instruction, why we keep this we keep them very simple, what is the impact of instruction at architecture, you reduce the complexity and diversity of instruction do not go for a wide variety of instruction, keep all instructions of uniform type. So that we can accommodate all these instruction in various 5 stages of an instruction pipeline.

So reduce the diversity of instruction make all instruction look similar. So that I can make use of all of them to pass through the common set of stages. The second point is the RISC architecture use uniform stage simple instruction that is one reason that your RISC architecture will beautifully fit into this 5 stage instruction pipeline.

(Refer Slide Time: 41:52)



And the third pipeline issue is independent computations. Our idealistic assumption is all instructions are mutually independent. What do you mean by mutually independent? I can run an instruction without waiting for its predecessors to get over. So, if an instruction is dependent on some other previous instruction, only if the previous instruction is complete, then only we can run this new instruction.

So, the whole concept of pipeline what we have seen is that in every clock cycle, one new instruction is being fetched without looking into the outcome of the previous instruction. So, what is the reality if you assume this real time case, the reality is pipeline stalls will be created, we cannot proceed. Sometimes we may have scenarios in which a later computation may require the result of an earlier computation that is called dependency between instructions.

And what is the impact of instructions set architecture wherever possible, try to reduce memory addressing mode you can have dependency, but if the dependencies between registers, so consider the case that you have an add instruction with R1 as the destination operand R2 and R3 are the source operands. Now, let us consider another instruction let us say

SUB R4, R1, R5

So, we can see that as per the convention that has been used. This R1 is the place where I write the result of R2 + R3 and here the same R1 is used as the source.

So the subtraction instruction is dependent on the add instruction, because the result of an instruction R1 is needed for subtraction to begin operation. So but this is easy to understand because both of the operands that is the from the name R1 it is easy for us to understand that there is a dependency. So subtraction instruction should not run. But sometimes we may get certain scenarios in which it is difficult for us to find out dependency.

Here the dependencies easy to find out because of the similarity in the name of the registers. Now, we can have dependency when we use memory addressing modes. So consider the case that I am going to store a word the content of R6 is going to be stored into 16(R1). So go to R1 and then add 16 to it that is a memory address. And to that address content of R6 is been written. Now let us imagine there is a another load word. So consider these two instruction.

If you look at them, there is no operand that is common. The first to store word has one of the register is R6 the other register is R1. The second instruction has R8 and R2 are the register, there is no similarity between them. So if generally we ask is there a dependency between this instruction, we do not find the dependencies straightforward, but there can be a case that

16 + content of R1=8 + content of R2

which will we will come to not only at runtime.

If R2 + 8 is a same as R1 + 16 then actually store is going to write into a location from which the load is going to read. So, even though there is no dependencies that we can detect from by looking at similarity of registers still there exists dependency. So wherever possible, try to reduce memory addressing mode, because dependency detection is very difficult use register addressing more wherever possible, because it will help the compiler in easy detection of registers.

(Refer Slide Time: 46:03)

So, the these are the three pipeline issues that we have seen. And that concludes our lecture 3 so before going to wind up, let us have a quick recap. Today, we were trying to understand what is a basic concept of a pipeline. And we have analyzed the example of a sequential laundry versus a pipelined laundry and drawing conclusions from that. We try to understand how pipeline can be applied on circuits. Be familiarized with the RISC MIPS microprocessor, wherein we have 32 registers there all the instructions are represented using 4 bytes.

Uniform length instruction, it is a load store architecture. And the instruction pipeline consists of 5 independent subcomponent IF, ID, EX, MEM and write back and we have seen that what are the functionalities involved in each of these stages. And having said all the interesting things about pipeline, we were trying to understand what are the practical limitations in implementing the pipeline.

Starting from uniforms sub-computations, then similar sub-computation that has to go through and independent computations between instructions. So, try to read the textbook, get familiarized with the concept. And we are posting few assignment questions for you to get a deeper grip on the topics that is been covered in the first 3 lectures try to work on it. Thank you.