GPU Architectures and Programing

Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No 1
Review of Basic COA w.r.t. Performance

Hi everybody. So, this will be our first lecture for the course on GPU architectures and programming. So from our introductory video, we have provided a brief outline of the different topics that we are going to cover in this course. And just going through some of the related slides of that video once again for our purpose.

(Refer Slide Time: 00:49)

The classic 5-stage RISC pipeline | Instruction Level Parallelism

- Fifteen years ago, Graphics on a PC were performed by a video graphics array (VGA) controller.
- VGAs evolved to more complex hardwares : accelerating graphics functions
- Early GPUs and their associated drivers implemented the OpenGL and DirectX models (APIs) of graphics processing.
- With time, HW functionality evolved as programmable SW

GPU Architectures and Programming | Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So this is just again going back into the history as we explained that in earlier. There I mean, personal computers, we had this notion of graphics pipeline, which was functioning in the form of our video Graphics Array, which eventually got replaced by a more programmable version. That is the graphics processing unit or the GPU, and eventually people found that since is a programmable parallel processor.

It can be used for accelerating lot of general purpose workloads. Which is basically giving rise to the term gp,GPU or general purpose graphics processing unit.

(Refer Slide Time: 01:27)

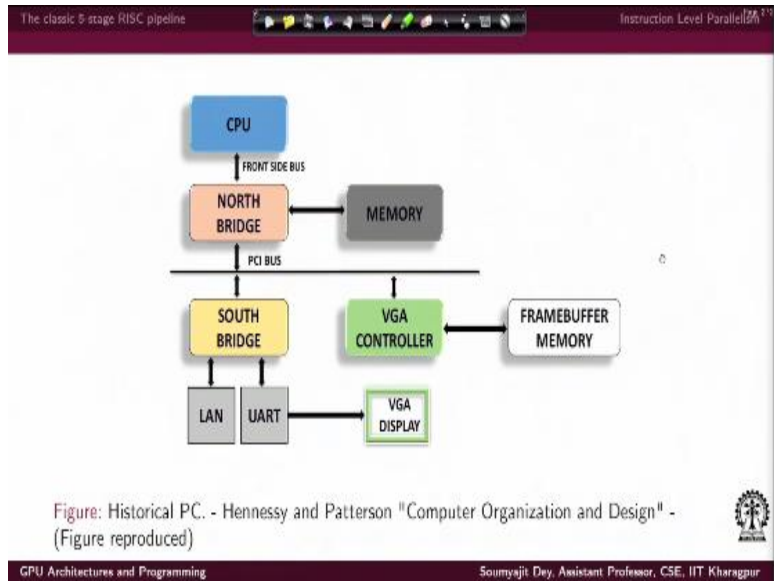


Figure: Historical PC. - Hennessy and Patterson "Computer Organization and Design" - (Figure reproduced)

And we also provided this snapshot of how our standard CPU connects with a VGA and other related peripherals.

(Refer Slide Time: 01:38)

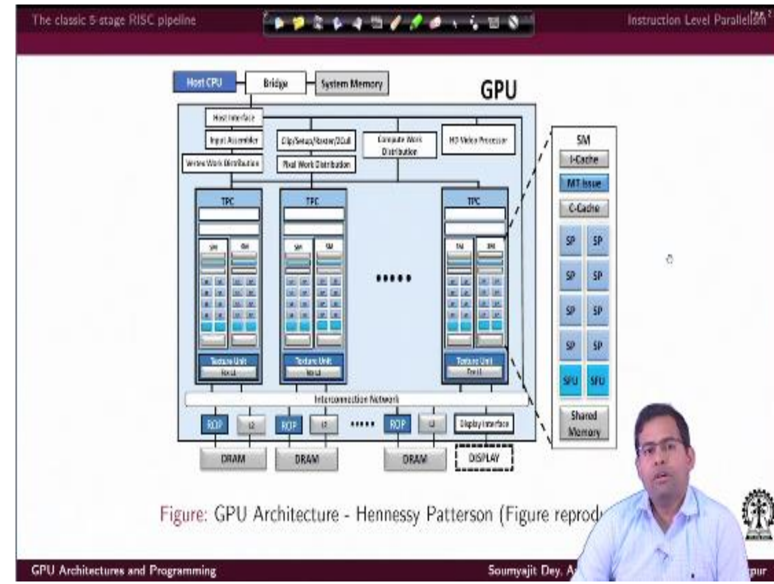
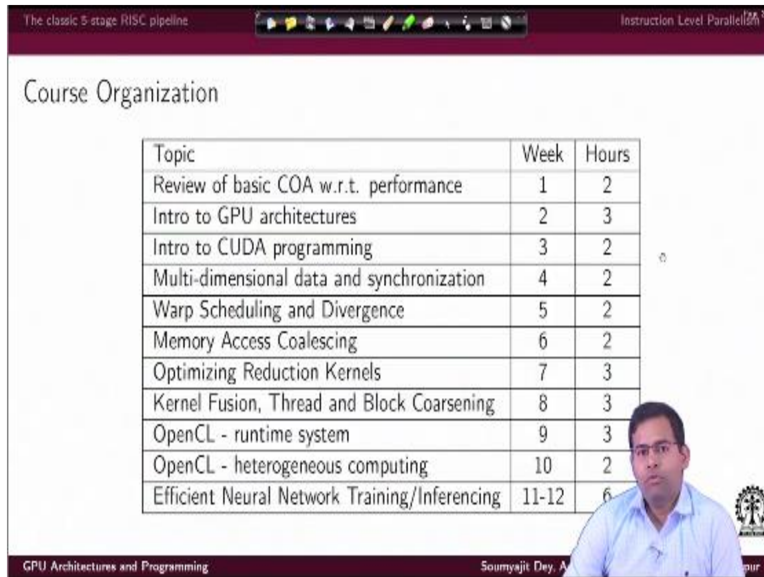


Figure: GPU Architecture - Hennessy Patterson (Figure reprod

We saw, how really the architecture of a GPU looks like with a lot of processing cores inside it, which are the SPS or the scalar processors. They just to give you an idea that I can view a CPU, as a, maybe a multi core platform. Containing a small number of compute codes, whereas a GPU, essentially contains thousands of compute codes, those simplistic, in terms of functionality, but lots of them.

(Refer Slide Time: 02:07)



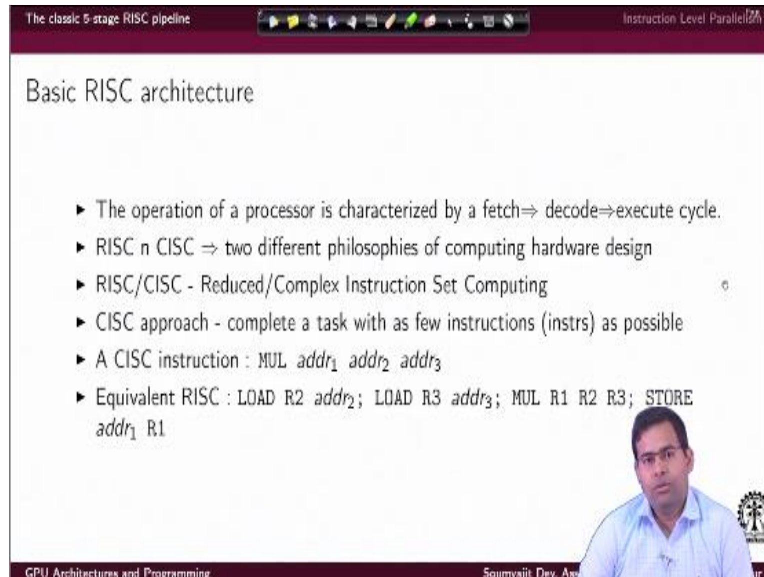
Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

With this basic background we have already introduced the course organization, Out of which. Now we will get into the review of the basic computer organization architecture, which is the topic one. So let us get started with that.

(Refer Slide Time: 02:23)

So we will start with a brief overview of what we have historically known as the classic five stage RISC pipeline.

(Refer Slide Time: 02:33)



So, what is the RISC by the way. Now, RISC is more of a computing philosophy. We which evolved as separate philosophy. The then practice philosophy of CISC, which essentially means complex instruction set computing. While RISC men, reduced instruction set computing. So what really are these things. To get into that, let us just browse up on what does the processor really do.

A processor is a digital system, which is executing a few specific operations in the loop continuously. A processor crunches instructions, which we have been knowing from our basic background as assembly instructions which are getting compiled. By which are getting assembled by assembler to a machine instructions. A processor processes, missing instructions, through a standard loop of fetching the instruction.

Decoding the meaning of the instruction and finally executing the instruction. And then again jumping to the next instruction how a processor really goes about this business is where it is the computing philosophies differ. So fundamentally, this style of whether, follow RISC style of computing. Or CISC style of computing is based on the basic notion of instruction set of a processor.

That is, what are the different types of machine level instructions, a processor is going to support. Are those instructions going to be very simplistic activities? Or are the instructions

individually powerful enough to do lot of things by just one specific instruction itself? So this is where this philosophies differ. That is why RISC is reduced instruction set computing and CISC means complex instruction set computing.

So coming to the CISC approach, the philosophy of CISC is essentially, that a processor should try and complete a processing task, using as few instructions as possible. That means, when a program is compiled to generate code for a CISC processor. The number of the, one of the one of the basic objectives of the code generation process would be to represent. The program in terms of assembly instructions and thereafter in terms of machine instructions.

Using as few instructions as possible.That means each of the instructions are going to do a whole lot of work. For example, let us take a CISC instruction, as we have given here that we have a multiplication operation MUL with three operators. Address 1, address 2 and address 3, Essentially we are meaning that while executing this instruction. The processor shall fetch data from the memory address 2 and memory address 3.

Multiply them, and store the result back to memory address 1. Now of course, as we can see that this is quite a lot of job. It is going to load load data from the main memory is going to multiply those data points, and is going to store the result back to the main memory. All done by executing a single instruction. So in terms of hardware design that means we are thinking of a hardware which is complex enough to do all those things by executing a single instruction.

The RISC philosophy is diametrically opposite to this. So in an equivalent RISC world. We do not have this kind of a multiply instruction. Instead that do not get replaced by a sequence of these instructions. So there will be a load type instruction, using a load command. One should be able to load the data from memory address 2 to some register R2. This will be followed by another load command.

Through which one should be able to load the data from address 3 to the registrar R3. And then we have a multiply instruction. So as you can see, is multiply instruction takes as arguments contents of registers R2 and R3 multiplies them and stores the result in the register R1. Now this

will be substituted by the store instruction, which was told the result from R1 back to the memory address, address 1.

So essentially, this kind of summarizes the RISC philosophy that all kinds of operations specifically arithmetic logic operations will have to have going to. They have to happen over data values that are present in the register the result of such operations shall also be stored in registers. There is no operation that is going to happen on value stored in the memory. So, the only instructions which are going to write or read data from memory.

Or the load and store instructions. So, this is one way to represent the difference between a complex instruction, and the equivalent case of a sequence of simple instructions. So when we say that if the RISC processor is a processor. Where the instruction set comprises only this kind of very simple instructions. When we say it is a CISC processor, the instruction set is arbitrarily complex.

There are instructions which are very powerful by themselves doing a lot of stuff.

(Refer Slide Time: 08:28)

The classic 5-stage RISC pipeline
Instruction Level Parallelism

CISC vs RISC

CISC features
- Older ISA
- Multi-cycle instructions, HW intensive design
- Efficient RAM usage
- Instructions - complex and variable length, lots of them
- Micro-code support
- Compound addressing modes

RISC features
- Ideas emerged in 1980s
- Single-cycle instructions, SW intensive design
- Heavy RAM usage, Large Register file
- Small no. of simple fixed length instructions
- Less no. of addressing mo

GPU Architectures and Programming
Soumyajit Dey, Assi

Overall, to summarize the difference. The CISC features the features of a CISC kind of system. The instruction set architecture of a CISC processor is older, the philosophy is quite old. So

CISC processor came much earlier RISC was an idea which emerged in the 80s earlier to that. It was mostly CISC instructions, used to be multi cycle.

That means, the processor is driven by a master clock, and each instructions execution shall consume multiple clock cycles. Of course, because each instruction is going to do a lot of work. So it is a hardware intensive design. In the sense, it may be the case that for some powerful instructions. You need some specific hardware you need built into the system. RISC feature is that all the instructions of single cycle.

Every instruction will do something very simple, but at the same time since it is going to do something very simple, it will consume a single clock cycle and get the job done. Which would mean for the same functionality the number of instructions in CISC keys lists, the number of instructions in RISC or more, but each of them, execute faster. So, overall, the timing for the final task may not differ much is how about, the system goes about achieving the job.

That is why we say CISC is a hardware intensive design because they are maybe more specialized hardware for specific kind of instructions. And RISC is more of a software intensive design because I am breaking every task into small atomic instructions, which would mean for every task, I will generate a lot of instructions in software terms. Now since CISC I have much less number of instructions to execute.

The RAM usage is much more efficient, whereas for the equivalent functionality since I have a big code to execute the RAM usage is much heavier. And also, since all RISC operations are to be carried over operands from the register file, specifically the arithmetic and logical operations. I need the support of a much larger register file, when compared with CISC. Other important thing in CISC the instructions are complex, they are multi cycle, which also means the number of cycles required for executing each of the instructions may vary.

Whereas as we have discussed in RISC, all the instructions of single cycle. Now another very important thing is, since there is no regularity in terms of the intensiveness of CISC instructions, some instructions may be very complex take multiple operents, take multiple cycles to execute.

And due to its inherent complexity in terms of specification as well as number of opponents. The instructions including length may be very big.

Whereas for some other instructions some other CISC instructions, the instructions including length may be small. Overall, the point is the CISC instructions may vary in length, whereas for CISC were for RISC. The instructions, all of them are very regular, they are going to be over same fixed length. Because they are quite generic. They are all single cycle. They are intensiveness in terms of hardware research will be kind of similar.

So they will all be of similar exactly same, of course, length. Another very important CISC feature is compound addressing modes, there will be a lot of different support for multiple possible different addressing modes in a CISC ISA are instruction set architecture whereas in RISC. There will be much less number of addressing modes that will be supported. To get into more details about these features is best.

That one should consult any well known undergraduate book on basic computer architecture. But these are the important points, which we wanted to summarize, as the difference between these two very well known computing styles and their adoption in basic processor design. In modern days possibly apart from x86 architecture, all other processors. Most, and more specifically in the embedded world, the processors used in mobile platforms.

They are mostly following their RISC computing philosophy. In the other terms, that means in the x86 world into Intel has still been able to hold on to the CISC philosophy, but internally. There is also a question of translation of these instructions. But that those are like, advanced topics, which we may touch upon later on.

(Refer Slide Time: 13:40)

The classic 5-stage RISC pipeline

Instruction Level Parallelism

Elementary CPU Datapath

- The datapath 'fetches' instruction, 'decodes' and 'executes' it
- Control logic generates suitable activation signals
- Executes different instructions with variable delays

GPU Architectures and Programming

Just an example of a CPU data path, so what's the data path? This is basically a drawing, which is trying to summarize how the different functional units inside the processor, are going to be connected. As we can see, there is a block, which represents the instructions and their storage in memory. There is a block, which represents data and their storage in memory. Now of course in a general processor instructions and data will be resident in the same memory.

For a one new man architecture. This block highlights the register file, which will be containing all the CPU registers. And these are block, which is more representative of the ALU of this kind of very simplistic CPU data path, which is trying to convey the basic operations that a CPU is going to execute. Like we have said the CPUs data path is going to execute three basic operations for now.

Fetching the instruction, decoding the instruction and executing it. Now these are the functional operations, which means it will take one instruction. Understand the semantic meaning of the instruction and do exactly what the instruction wants to do in terms of functionality. but they are why there is also the question of reading and writing operants from the memory.As well as updating values and registers.

Now, these will also give rise to some more functions which will see soon. Now, coming back to the question of data path. So, the data path, essentially represents the flow of data. Among

different functional units as captured in this picture, like how exactly that, that the data flows from each of the functional units, what are their dependencies. And the more important thing is, how is the flow of data control.

So you can see that there are certain signal lines here, which are marked in red. Now these are the lines which are expecting some digital comments, based on which they decide how this digital system would operate. Fundamentally speaking that data flowing in from a hardware unit should be used for what kind of functionality inside the hardware unit, and should be routed as an output to which hardware unit.

This is what is dictated by this signals marked in red, and they are known as the control signals. The control signals get generated through a separate logic called the control logic.

(Refer Slide Time: 16:31)

The classic 5-stage RISC pipeline

Instruction Level Parallelism

Single cycle implementation of datapath

- The choice of clock rate is limited by the instruction with maximum delay
- Options : choose the clock period more than latency of 'slowest' instruction or,
- choose variable periods for diff instructions – not practical !
- Alternate possibility - break the instruction execution cycle into a series of basic steps
- Basic steps have less delay, choose a fast clock and use it to execute one basic step at a time

GPU Architectures and Programming

Soumyajit Dey

Now as we have discussed earlier, that in RISC, the philosophies that every instruction executes in a single cycle. Now, why is that a good thing. Now let us note the important thing that even for RISC style instructions, each instruction may have different timing delay. When the functionality is implemented in terms of hardware. that is why we can say that instructions execute with variable delays in a CPU data path.

Now, the question is, when I have implemented such a data both in hardware. These are digital data path. It needs to be clocked by a suitable clock signal. And I have to choose a suitable frequency or period of the clock. Now, in as a standard in any digital design. The clock period is chosen by looking at the critical path of the cycle, that is the. What is the situation in which the data path, or this specific digital circuit will incur the maximum delay.

As we have said that even for instructions which are all lightweight and atomic as in RISC. Their executions may create different possible delays in data path right? So then does it mean, then the clock signals frequency should be different for different execution of instructions. So that is definitely not possible. That means we have to find out execution of which instruction takes the most amount of time.

And choose the clock period to be equal to or greater than the latency of that slowest instruction, right? Now, even if we do that, we still have a problem, right? Because then we may have a system where I have multiple possible instructions to execute. Some of them are very slow, in terms of hardware latency, some of their executions is very fast in terms of hardware latency. But since I have chosen the clock period, so that it can execute the slowest instruction.

Even for the faster instructions and their execution. I suffered the delay of the slowest instruction. So that leads to a very conservative design. So how do I get out of this problem and increase a processors, or CPU data path throughput. The way to go about it to be that look at the instructions execution path. What are the basic functionalities that each instruction is supposedly going to do, and break them into a set of basic operations or basic steps.

Now why do we do that, because if I can break this execution of an instruction into certain basic steps. With respect to the flow of data and control and processing in this data path. Then I can now clock. Each of these basic steps by a fast clock. How does it help the basic steps are simple, so they can be clocked by a fast clock for some instructions I may need the sequence of all the basic steps for some instructions, they may not need them.

We should when. Now, are you really have this system, driven by a fast clock with some instructions completing fast. And some instructions, completing slow, based on what is the requirement of the instruction in terms of these basic steps.

(Refer Slide Time: 20:32)

The classic 5-stage RISC pipeline

Instruction Level Parallelism

Multi-cycle instructions

A basic stage represents one of the following states in the execution of an instruction

- Fetch (IF): IR ⇐ Memory[PC]; PC=PC+4
- Decode (ID): Understand instruction semantics
- Execute (EX): based on instruction type
  - Arithmetic/logical operation, Mem address / Branch condition computation
- Memory (MEM): For load/store Instr, read/write data from/to memory
- Writeback (WB): Update register file

GPU Architectures and Programming

Now, if we get into this issue of identifying the basic steps required for executing an instruction. Earlier we have been telling like an instruction needs to be brought from the memory it needs to be. The meaning of the instruction needs to be brought out by the deco. By decoding the instruction and finally executing the instruction, and at the same time we also said, Okay, this is fine, but there is also this issue of bringing data from the memory.

Updating the data in the register file and all that used to fight and all that right? So bringing all these issues together, we can say that the execution of any instruction in a RISC kind of system. can be broken down into these five basic steps. The first stage would be, what we call the instruction fetch stage where the value of where. Actually, we check the content in the memory for the location that is loaded into the program counter mark here by this PC right?

And load the content into this specific register, which is called IR, the instruction register right. So, I hope everybody will be familiar with this notions of problem counter is basically a specific register in your CPU. Which loads specific memory address from which you are supposed to

fetch the instructions and execute them. So this is noted down here in terms of the standard mnemonics that you can find, that the content of the memory location.

That is pointed to by the program counter is getting loaded into the instruction register, followed by the program counter getting implemented by 4, why? Essentially we are saying thinking that memory is byte addressable, and the memory word length are 32 bits. Assuming that we should get the next program counter value by incrementing the current program counter value by 4 right? Now what is the next stage.

So the next stage would be this decode stage. That means an instruction has been fetched and instruction is nothing but a sequence of 1s and 0s in our assumption here is a 32 bit instruction. And this sequence of 1s and 1s need to be understood by the CPU. So there is this decode stage, where based on the instructions encoding and its operations. The CPU should understand most of the data paths should understand what does it instruction, want to do.

Is an load instruction, is an stored instruction is an add instruction, what does it really want to do. Based on this understanding, it should activate suitable functional units in the next stage That is the execute stage, and it should perform the required operation, depending on whether it is an arithmetic or logical operation, whether it is the computation of a memory address for a branch, or whether it is the computation for a branch condition itself.

So these are the computational requirements, which, if there is any will be computed in the execute stage. Now this execute stage has to be followed by another stage called the memory stage. What this for? In case the instruction is a load store type instruction. It needs to fetch data from a memory location, and that data has to be loaded into the register file right? So this fetching of data from the memory will be performed in this memory stage.

So, basically this is the stage, which is going to handle all read, writes from and to the memory. Now we have the last stage in the pipeline, which is called the write-back stage. So, any kind of update that is going to happen on the register file has to happen here. For example, if I take the

example of a load instruction. So suppose I am loading the value from some memory location x to some register R1, the access of memory location x.

And fetching the data from that location to some specific register called memory data register should happen in the MEM stage. And then writing this data from the memory data register to the specific register R1 in the register file this update of the register file shall happen in the write-back stage. So these are the five stages, which, more or less characterize the execution sequence to be followed in a basic RISC processor.

That is why we call it the classic five stages RISC pipeline. Now of course, in a real processor. There are much, much more complex functionalities present in a basic RISC processor.There may be many more pipeline stages present there. But, for our purpose, we stick to this discussion of a pipeline. With our understanding of this basic five stages of execution.

(Refer Slide Time: 26:01)

The classic 5-stage RISC pipeline — Instruction Level Parallelism

Pipelining

- Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- Every basic stage is always processing some instruction
- In every clock cycle, one instruction completes - ideal scenario
- Practical issues - pipeline hazards

GPU Architectures and Programming — Soumyajit Dey

So, overall when this pipeline operates every instruction has to go through the sequence of five pipeline stages. From instruction fetch, instruction decode, execute, memory and write-back. It depends whether it is how the instructions will really need something significant to be done in that stage or not. But all these stages will be active. Since the performance, the functionality of each of these stages, is very basic.

There can be a faster clock clocking all these stages.So that, depending on the instructions execution. I mean, we will have a fast execution of all the instructions, whether or not it really needs executing in that stage or not. Now, once we pipeline. The operations in this kind of a processor. I can always have these basic stages, always processing some instruction, which means, suppose I have instruction 1 entering the pipeline that is it has been fetched.

And it has been decoded, so when the instruction 1 is going to the decode stage, I can have instruction2 being fetched. And when instruction 1 is getting executed instruction 2 can get decoded and instruction 3 can be fetched. So that would mean, if I look at the pipeline from its endpoint. I always see one instruction, getting completed right. That means, in that way I can say that.

Overall, the throughput of this pipeline is one instruction getting executed in each clock cycles getting finished with deceptive execution in any in each of the clock cycles. So this is where the optimization really helps. To summarize, from the earlier points. Had we been sticking with a single cycle implementation of our CPU data path. The single cycle implementation would have required different amount of time for executing each of the instructions.

Depending on instructions functionality. In that case the choice of clock will be based on what is the first instruction. And then what would happen. The lighter instructions will suffer the same latency, as the slowest instructions. In order to alleviate that issue. We broke the execution of instructions into these kind of basic stages. Since each of the basic stages are very simple, we are able to plug the five pipeline, with a very first clock.

Since that is the case, and I can keep each of these stages always active, I can keep on filling instructions at the front of the pipeline. And I can see instructions getting completed at the end of the pipeline, one in every clock cycle. So overall, using this idea of pipelining, starting from multi cycle execution of instructions, we are able to see every instruction. I mean, the pipeline is able to complete one instruction per clock cycle.

So in that way I can say that every instruction requires 1 clock cycle. But that is not really the case right every instruction is requesting 5 clock cycles with respect to the basic stages. But since the pipeline is always active, it is interleaving the insert, execution of instructions. I can say that finally from a user's point of view, every rec in RISC instruction. I mean, when it executes to the pipeline. I have one instruction completing in one clock cycle.

Now, this would really be the case. In an ideal scenario.So what is a non ideal scenario, non ideal scenario means that while executing an instruction in a stage of the pipeline, it may so happen that immediately the instruction is not able to execute in the next clock cycle. In the next stage of the pipeline. These are the issues known as pipeline hazards, which we will be covering. Thank you.