Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 53
Pipelining the MIPS32 Data Path

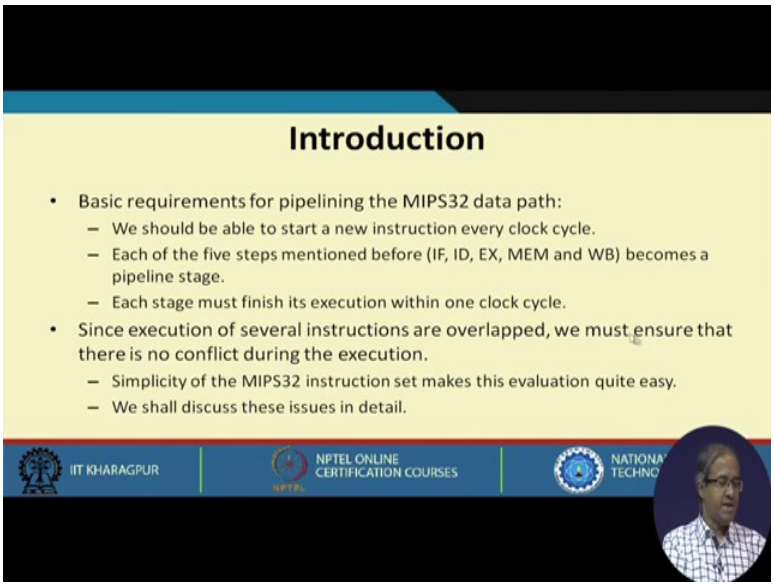
In this week we shall be starting our discussion on how we can create a pipelined version of the MIPS32 data path. Earlier we have seen how we can have a non-pipelined data path for the MIPS32 instruction set architecture, and you have seen because of the simplicity of the instruction set, the regularity, the simple instruction encoding, the required hardware was very simple. You need very simple hardware in the data path with very regular interconnections. If you recall, there are 5 steps overall to execute an instruction in the MIPS32, instruction fetch, instruction decode, execute, memory operation and write back. We start from there and initiate our discussion for this lecture.

(Refer Slide Time: 01:32)

We shall be exploring pipelining of the MIPS32 data path.

Let us look at the basic requirements first. We have seen the non pipelined data path of the MIPS32 earlier.

(Refer Slide Time: 01:42)



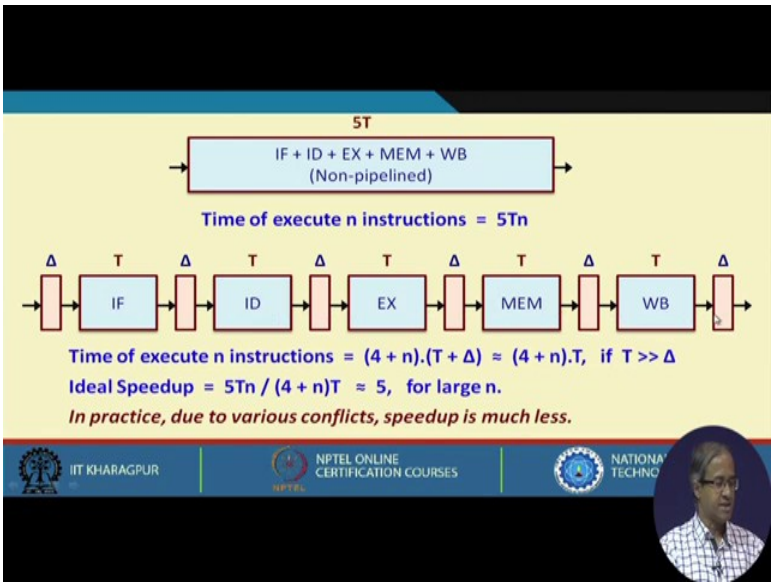
Let us now understand if we want to do pipelining, what are the basic requirements? The first requirement is we should be able to start a new instruction every clock cycle. When we talk about instruction pipelining we are saying that instructions are being pipelined, we are feeding the instructions to a pipeline one by one. So, instructions are flowing through the pipeline stages one by one, and they are getting executed. In the ideal case I would expect that one instruction would be completing every cycle, which is the beauty of the pipeline. Once the pipeline is full we are expected to get one output every clock. Because we are getting one instruction completed every cycle we should also be able to start a new instruction every cycle.

This is one requirement, and each of the 5 steps will become a pipeline stage. This is another modification we would be doing. And the stages will be such that, they must be finished within one clock cycle, because for the pipeline operations to be done in a smooth way, your clock period must be chosen to be large enough such that every stage execution must be finished by that time. This is a mandatory requirement; your clock cycle time must be large enough such that every state should finish their execution.

Now, there are complications that will come in. See execution of several instructions will overlap now. Because now we have the IF, ID, EX, MEM, WB stages. When one instruction is in ID, next instruction will try to come into IF, and so on. We must ensure that there is no conflict of any sort during the execution of this instruction. We shall also

see that there will be conflicts, but the analysis and some solutions to avoid them become quite easy because of the simplicity of the MIPS32 RISC instruction set. We shall be discussing these issues in some detail subsequently.

(Refer Slide Time: 05:03)

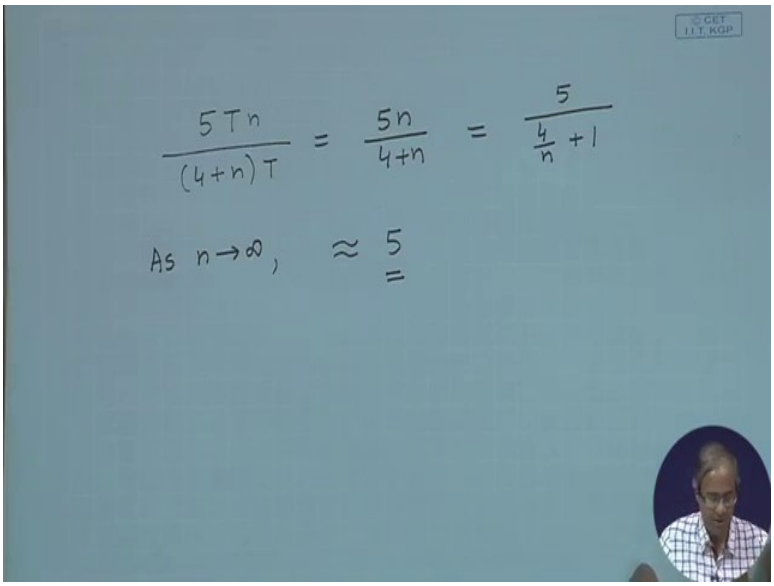


This is our non-pipelined MIPS32 data path, where all these 5 steps are together. Let us assume that the time taken by this entire thing is 5T. So, every instruction will require a time 5T to finish, then the next one comes. If there are N instructions the total time will be clearly 5T x N; this is the non-pipelined version.

Now for the pipeline version we do something like this. We break these 5 steps into 5 stages, and we insert laches between stages. Let us assume that each of the stages take time T and laches have a delay of Delta. We have already seen that how we can analyze a pipeline and its execution performance. You can similarly say that for N instructions what will be the total time, 4 clock cycles will be required to fill the pipe, and after that I will be getting one output every clock cycle.

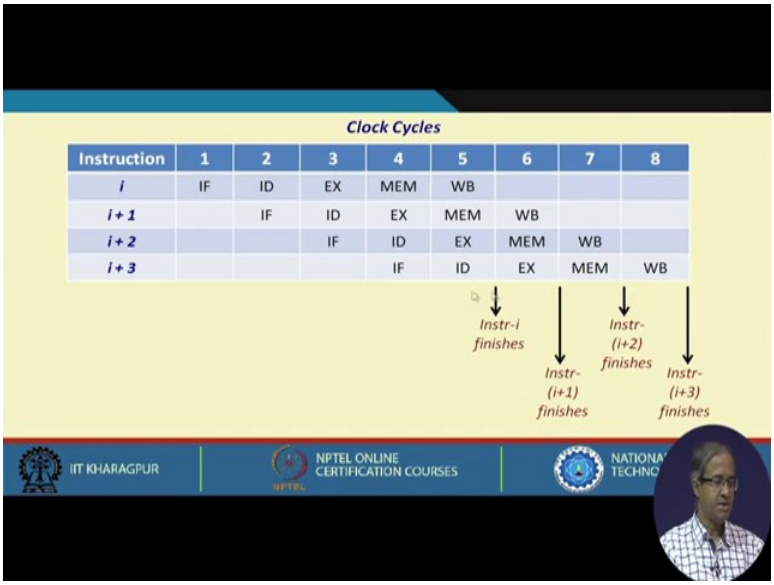
The total number of clock cycles required will be 4 + N. And what is the clock cycle time? It must be at least equal to T + Delta. Now if T is large as compared to Delta, if we ignore this Delta, this becomes approximately equal to (4 + N) x T. So, ideal speed up will be 5TN / [(4 + N) T].

(Refer Slide Time: 07:20)



Now this T cancels out. So, I have 5N / [(4 + N)]. As N becomes large, let us say N tends to infinity, 4/N will tend to 0. So, this value will be approximately equal to 5. This will be the ideal speed up as N becomes large. So, number of stages 5 is the idle speed up. So, your idle speed up will be equal to 5 when N is large, but we will see later that things are not that rosy in an instruction pipeline. There will be various kinds of conflicts that will appear, speed up will be significantly less than 5.

(Refer Slide Time: 08:51)

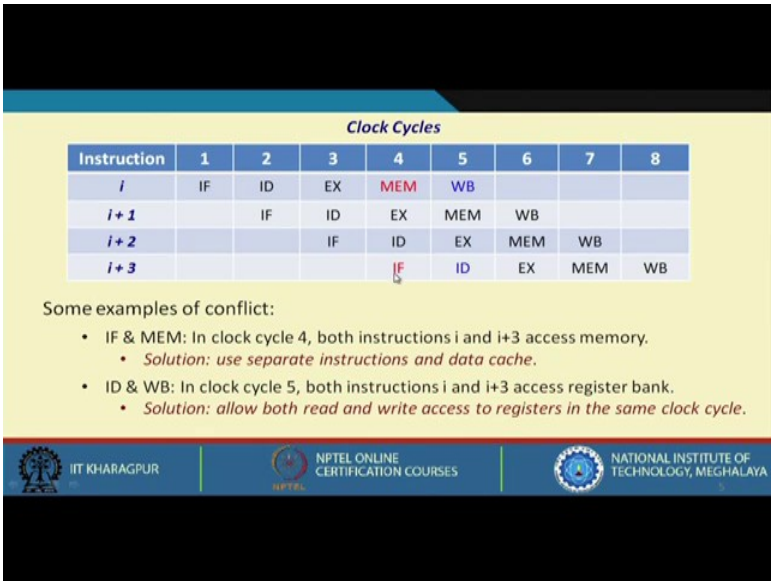


Let us see how the pipeline works. I am showing the time steps or clock cycles out here, and let us say instructions are coming one by one.

In clock cycle 1 the first instruction i enters the IF stage; that means, the instruction is fetched. So, after it is fetched it goes to the ID stage in step 2. So, while it is being decoded the next instruction can be fetched. So, there is some overlap. After this is done first instruction will go to the EX phase, second instruction to the ID phase, and third instruction can go into the IF. In this way after fourth step the pipe is full. This way it will go on. The first instruction will finish here, second instruction will finish here, third instruction will finish like this.

It is after time 5 instruction i will finish. Then time step 6 instruction i + 1 will finish, time step 7 i + 2 will finish, and time step 8 i + 3 will finish. So, after the initial delay for the pipe to get filled up, in the ideal case you will be getting one instruction completed every clock cycle. That is a beauty of pipelining.

(Refer Slide Time: 10:46)



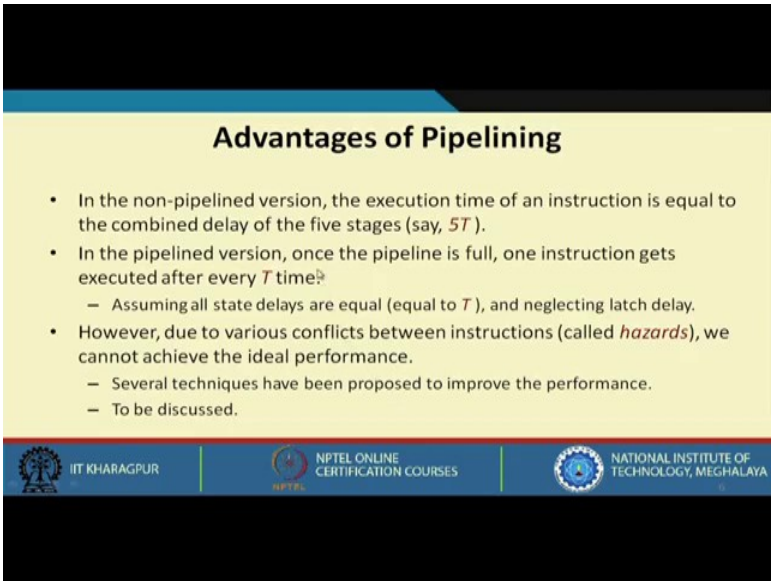
You can see that it is not that simple; there can be various kinds of conflicts. In this diagram now I am showing 2 kinds of conflicts.

First is between IF & MEM that is shown in red, suppose the first instruction was a load or store kind of instructions, load and store means during the MEM stage it will be accessing memory. But at the same time instruction i + 3 is also trying to read from

memory. It is doing an instruction fetch. So, there is a conflict for memory; both instructions i and i + 3 are trying to access memory at the same time. For this particular case we can suggest a solution. Let us use separate instruction and data caches. See now you can find a logic why people actually use separate caches for instruction and data, this is the logic actually why it is required. If we do that, then this IF will be using the instruction cache and MEM will be using the data cache. So, the conflict that was there is apparently removed or avoided.

Similarly, there is another conflict here, suppose first instruction is a register type instruction, say ADD. After everything is done, it will be writing the result into the register in the WB phase. Now if we recall the micro operations that we have seen earlier for the different stages, you remember that in the ID phase the register operands are pre fetched. Again here marked in blue there is a conflict in time cycle 5. The instruction i + 3 is trying to read from the register bank, instruction i is trying to write into the register. This is also some kind of a conflict. So, what is our solution here? We can read from the registers also you can write into the register in the same clock cycle. Well you may think that this is a little funny because how you can write or read from a particular register in the same cycle, but we will see that it is possible to do that.

(Refer Slide Time: 13:54)

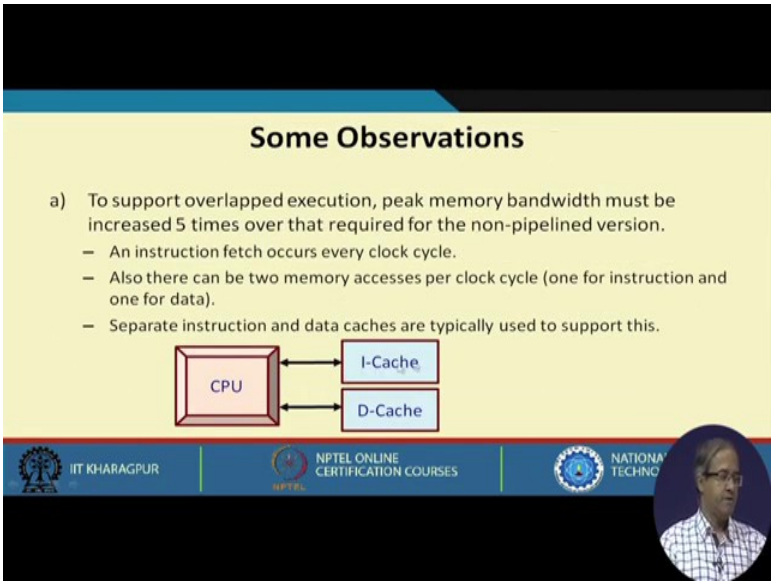


Coming back to the advantages of pipeline that you have already, in the non-pipelined version of the MIPS the execution time of a instruction is equal to the combined delay of

the 5 stages which we had shown in that example 5T, but in a pipeline version in the ideal case, one instruction gets executed after every time T. Of course, you are assuming that all stage delays are equal, and equal to T, and we are neglecting the delays of the laches.

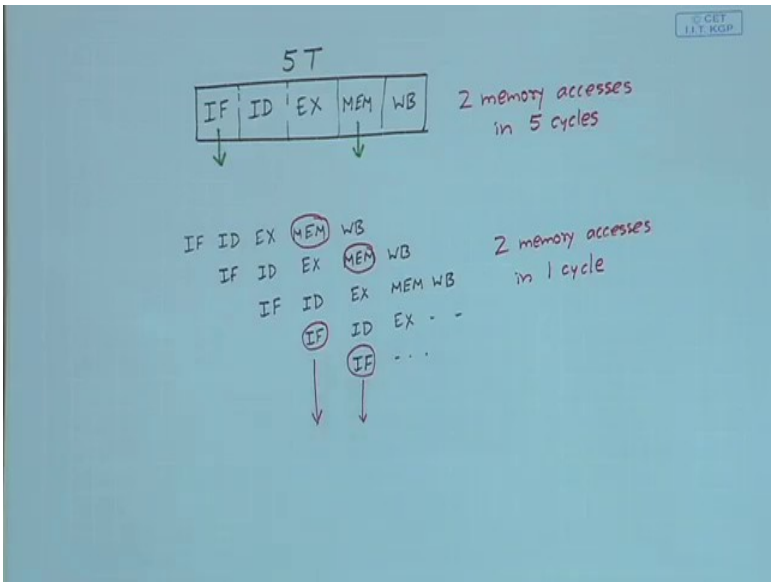
But we shall see later that it is not so simple in the practical scenario because of various conflicts that can arise between instructions, which are commonly known as hazards. We cannot achieve this ideal performance, because in the ideal case we have seen that our speed up can approach 5, but because of hazards it can be less than 5. However, various techniques have been proposed to improve the performance to the extent possible. We shall be looking at several of the techniques.

(Refer Slide Time: 15:27)



Fine, let us look at some of the observations we discussed a while back. Let us look at the memory constraint. Earlier, when you had a non-pipelined version of a MIPS processor, an instruction was getting executed in 5 steps, that is 5T time.

(Refer Slide Time: 16:01)



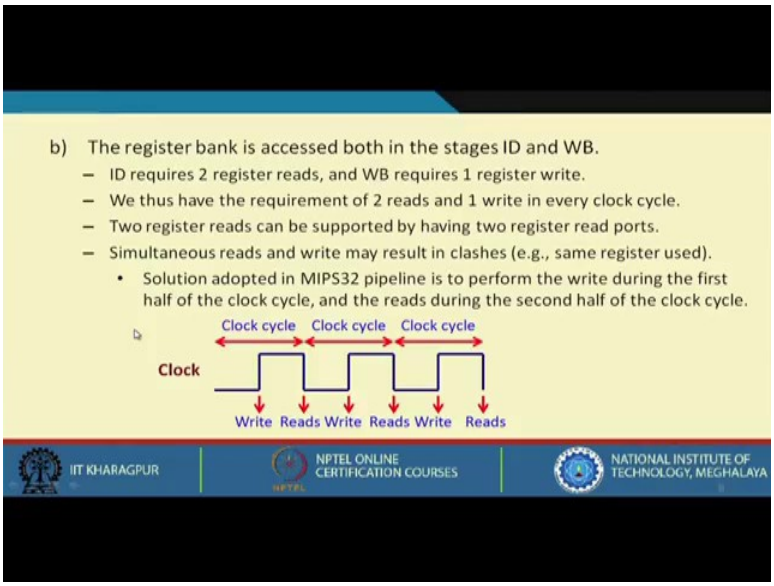
The total time was 5T for non-pipelined version, there were 5 steps of execution, IF, ID, EX, MEM and WB. Now if you consider memory access in this 5T time, there was one memory access here and there can be another memory access here. Of course, all instructions are not load and stores, but in the worst case there will be two memory accesses. For the non-pipelined version there will be 2 memory accesses maximum in 5 cycles. This is for the non-pipelined case.

But for the pipeline case, the instructions will be overlapping. What happens here? In the worst case in every cycle there can be potentially 2 memory accesses. Earlier there were 2 memory accesses in 5 cycles, but now for the pipeline version there can be maximum 2 memory accesses in one cycle. So, what does that mean? It mean simply that my memory bandwidth requirement is increasing 5 times.

This is a big issue. To support overlapped execution the peak memory bandwidth must be increased 5 times over that required for non-pipelined version. One solution is that in the L1 cache we use separate instruction and data caches. Most of the time because of the locality of reference data reference by the CPU will be found in the caches.

So, there will be no conflict, one instruction can fetch from I-cache other instruction can read or write from the D-cache. But of course, when there is a cache miss there will be delay, but this is a simple solution and this explains why we have separate instruction and data caches.

(Refer Slide Time: 20:28)



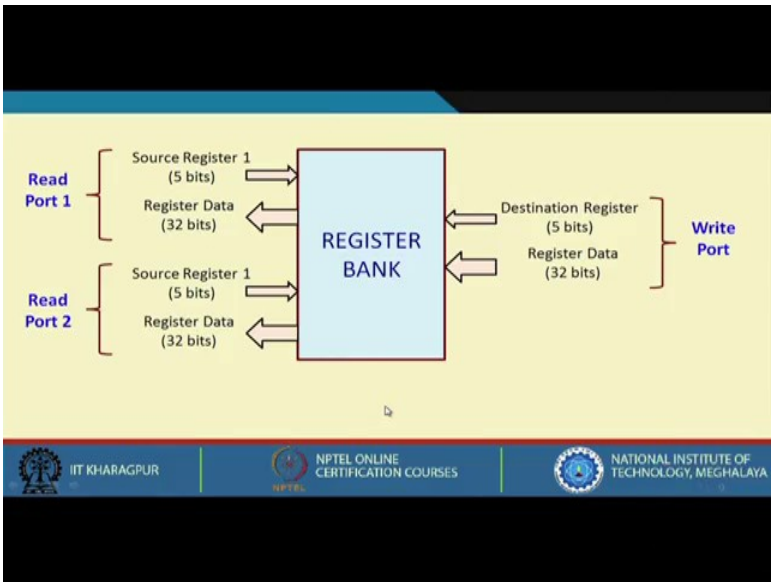
This second is the register bank issue. We saw earlier that some instruction can read a register during ID, some other instruction can write into a register during WB. We just made a loose statement that we shall allow both read and write to happen in the same clock, but how is that really possible? Let us let us find a solution here. In the ID stage what we are doing? We are pre etching both the register operands, source 1 and source 2. So, ID requires 2 register reads.

And in WB we require at most one register write. So, in a pipeline our requirement is that in every clock cycle we should support 2 register reads and one register write. How we can have 2 register reads? If our register bank has 2 separate read ports, then reading 2 registers concurrently is not that much of problem, but the trouble will be someone is trying to read register r1, but someone else is trying to write in to register r1. Simultaneous reading and writing will result in clash if same register is used. Now here a simple trick to solve this problem.

What is done in the MIPS pipeline is we divide the clock cycle time into two halves. See our clock cycle starts from here, leading edge, falling edge, this is our total clock cycle. We are saying that in the leading edge of the clock we do all the register writes, and in falling edge of the clock we do all the register reads, because you see register access is pretty fast. And clock cycle time is pretty long because all the stage executions must finish by that. Within that clock cycle time we are dividing into 2 halves; in the first half

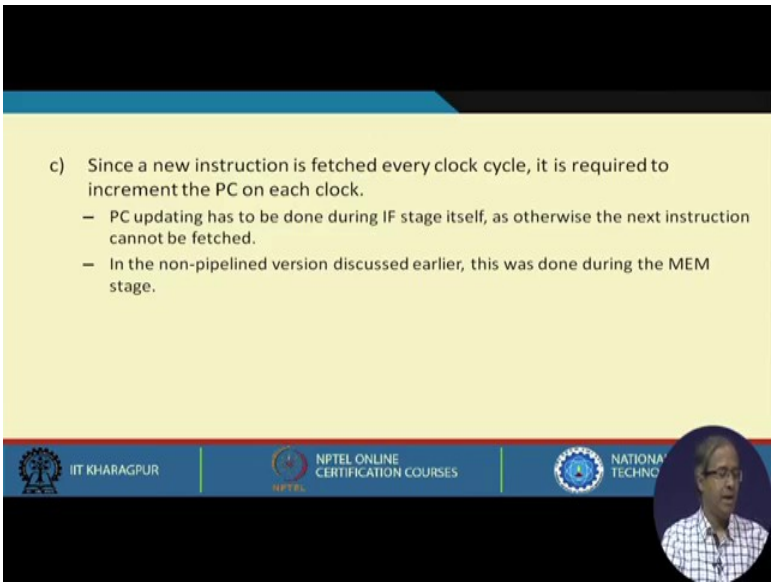
we are doing the write in the second half we are doing the reads. In case that kind of a thing happens that someone is writing into r1 and someone some other instruction is reading from r1, then first that write will happen then that read of that new data will happen.

(Refer Slide Time: 23:52)



So, our register bank will look like this. There will be 2 read ports. Source register numbers will be fed as register address 5 bits, 5 bits ;and the 2 register values will be read out just like reading memory. And on the other side there will be a write port that also will specify the register number where to write, and also the data that is to be written. As I had said write will happen in the first half of the clock and the 2 reads will happen during the second half of the clock. Using this simple convention and this modification to the register bank we can avoid this conflict.

(Refer Slide Time: 24:42)

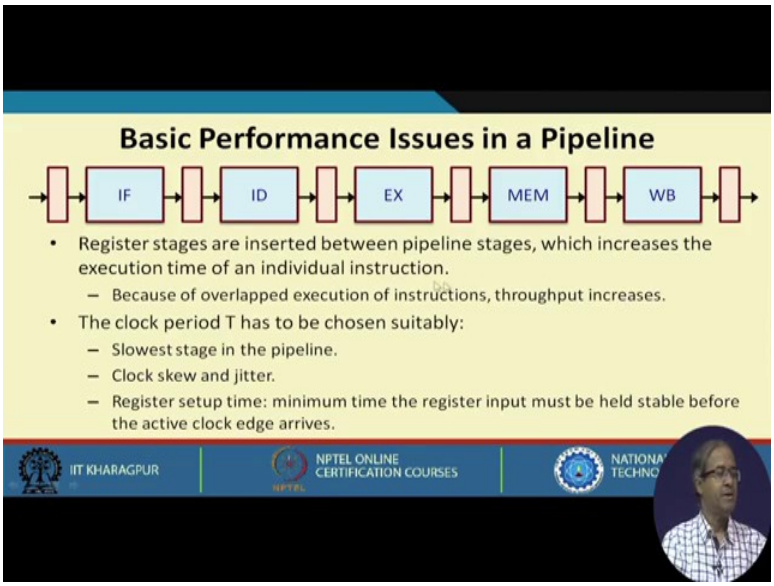


The third important issue is that, if you recall the micro-operations for the non-pipelined MIPS, there in the IF stage we were not incrementing the PC; we were using a temporary register called NPC, we were storing PC + 4 into NPC. Later on during the MEM stage we were transferring that NPC into PC if it was not a branch.

That was happening in a non-pipelined version, but in a pipeline can we afford to do that? Because in every clock I need to fetch an instruction. My PC should be ready with the address of the next instruction at the end of every clock. So, at the end of every clock, in IF itself I have to increment my PC.

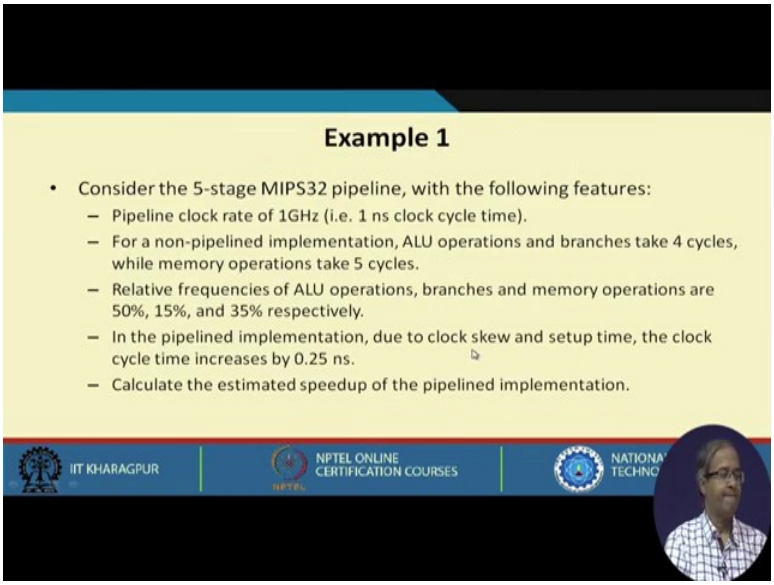
If I do not do that there will be trouble. PC updating has to be done during the IF stage itself, because if we do not do that you cannot fetch the next instruction in the next clock. As I had said in the non-pipelined version this was done in MEM.

(Refer Slide Time: 26:17)



Some basic performance issues can be discussed here. I am showing the MIPS pipeline, 5 stages with the laches. Register or latch stages are inserted for correct pipelined operation, because if you do not use latch stages then the output of one stage may disturb the input of the next stage. So the computation can become wrong. And again the clock period has to be chosen suitably, this was also discussed earlier. Because clock period T will depend on the slowest stage of the pipeline plus the clock skew and jitter and the setup time of the laches. All these times taken together will determine the minimum clock period that will ensure correct operation of the circuit.

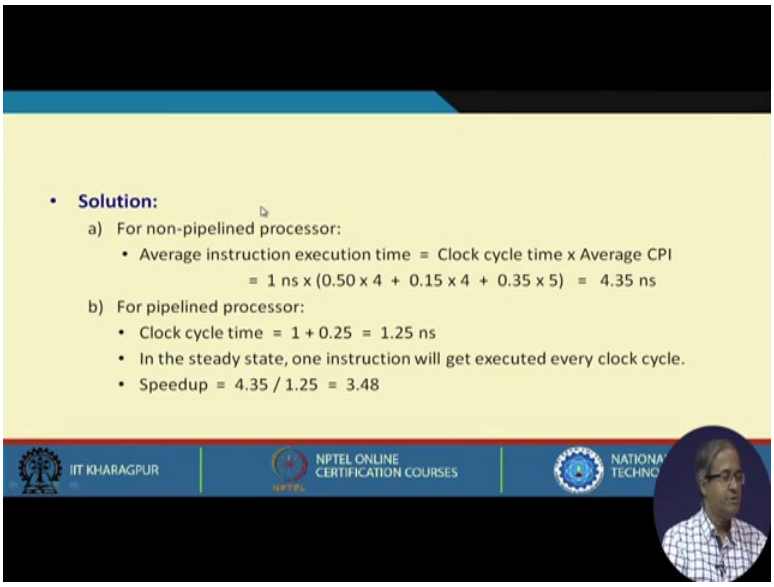
(Refer Slide Time: 27:26).



Let us take an example. We considered a 5 stage MIPS pipeline with clock rate of 1 GHz, which means 1 nanosecond clock cycle time.

In a non-pipelined implementation let us assume that ALU and branch operations take 4 cycles to finish, while memory operations because they also need memory access we will take 5 cycles. Also assume that the frequencies of this ALU, branch and memory operations are 50% 15% and 35% respectively. We also have a equivalent pipeline implementation where because of the register stages, clock skew, etc., the clock cycle time which was 1 nanosecond is increasing by 0.25 nanosecond. The question is what will be the estimated speed up of the pipeline implementation?

(Refer Slide Time: 28:48).

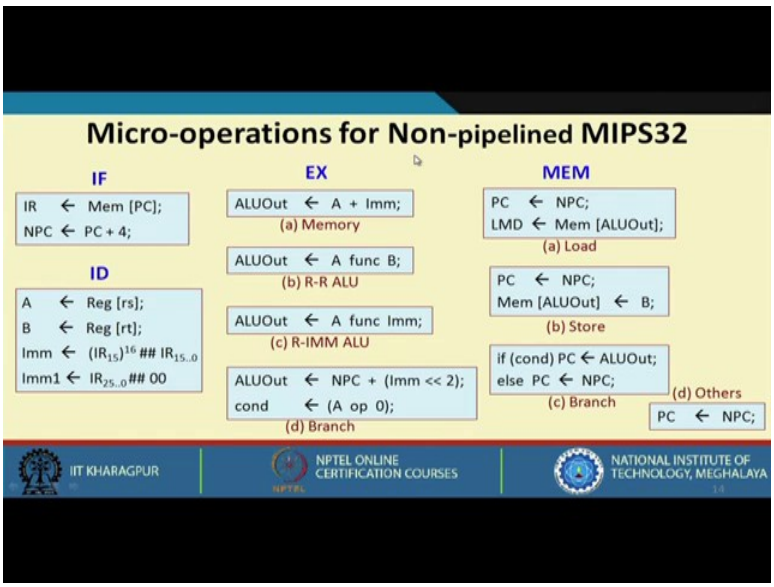


For the non-pipelined processor what we do? We can calculate the average instruction execution time by multiplying the clock cycle time with the average cycles per instruction. Now clock cycle system it was 1 GHz; that means, one nanosecond and average cycle time you see it was mentioned 50% are ALU, 15% are branch 35% are memory, and they take 4, 4, and 5 cycles respectively.

So, 0.4 x 4, 0.15 x 4, 0.35 x 5. If you calculate it becomes 4.35, but for the pipeline processor as I had said the clock cycle time is slowing down to 1.25 nsec. Now assuming that there are large number of instructions, in the steady state one instruction gets executed per clock cycle. So, the average instruction execution time will be would be equal to the clock cycle time 1.25 nanosecond. So, speed up will be 4.35 / 1.25. You see

this is based on some realistic figures, our speed up is coming to about 3.48 not exactly 5.

(Refer Slide Time: 30:12).



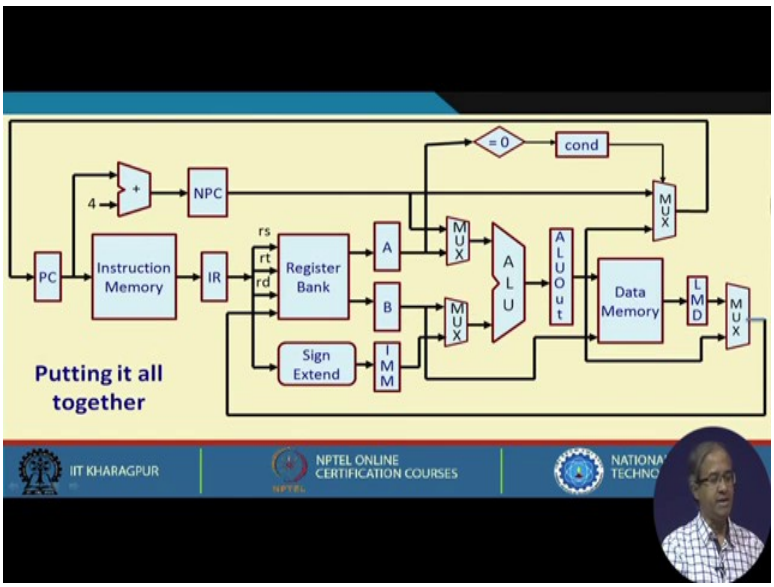
Let us let us have a quick recap on the non-pipelined MIPS32 micro-operations. During IF in the non-pipelined version we are fetching the instruction, incrementing the PC, but storing it in NPC. NPC to PC was transferred only during MEM. In ID we are fetching the registers, we are also prefetching the Immediate data and doing sign extension. In the EX stage we are doing some arithmetic operation, but depending on the type of instructions it can vary. For memory we are just adding this Imm to A to calculate the effective address of the memory; for ALU operation we simply operate on A and B; for Immediate operation also the similar thing A some function Imm; for branch operations we are calculating the branch target address, and also we are evaluating the branch condition whether it is a taken branch or non taken branch A. And again during MEM, if it is a load instruction we do a memory read, if it is a store instruction we do a memory write. If it is a branch instruction we are simply checking the condition, based on that we are updating the PC with ALUout or with NPC; but if it is any other instruction simply PC = NPC.

So, PC is getting updated much later.

(Refer Slide Time: 32:07).

And in the WB stage we are storing either the ALU output or for load instruction the output from the memory into the register.

(Refer Slide Time: 32:20).



We have also seen that all the things taken together the data path looks like this. Next what we will see is that if I want to make this into a pipeline, what are the changes that are required, what are the modifications I need to do. Like for example, one immediate thing I we can see that this PC is getting modified based on a decision, which is taken in the MEM stage. I cannot wait till that long, because I have to fetch a new instruction

every clock cycle. So, I need to increment my PC in the IF stage itself. These are a few things we shall be seeing in our subsequent lectures.

Thank you.