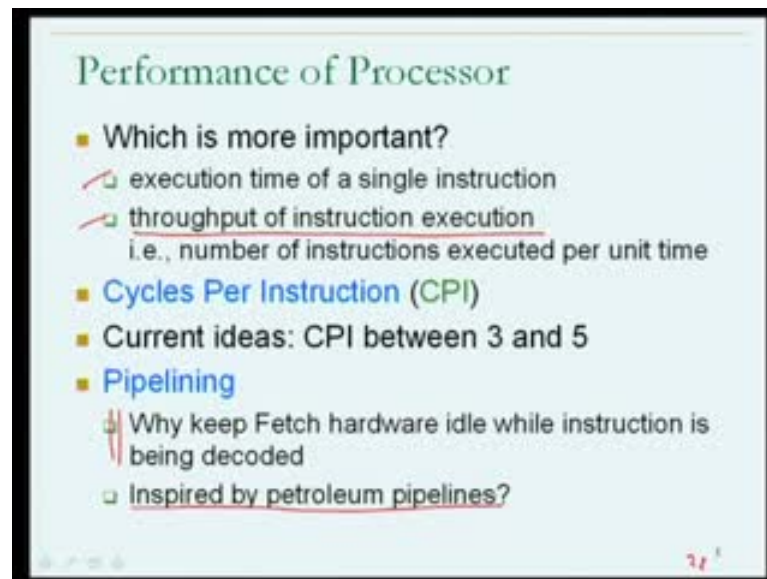


High Performance Computing
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture No. # 22

Welcome to lecture 22 of our course on high performance computing. In the previous lecture, we had started a new topic that of the architecture of pipeline processors. And let me just remind you little bit about what we had in our opening discussion. The primary prospective of pipelined architecture is that the design of the processor will not be driven so much, by requirement of reducing the execution time of the individual instructions, but rather by increasing the through put or the rate that which the instructions are executed.

(Refer Slide Time: 00:52)



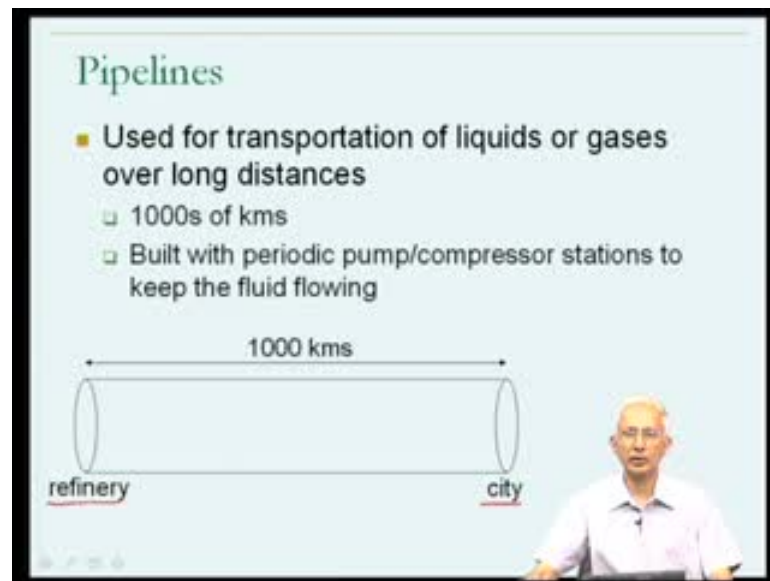
The slide, titled "Performance of Processor", contains the following text:

- Which is more important?
 - ☐ execution time of a single instruction
 - ☐ throughput of instruction execution
i.e., number of instructions executed per unit time
- Cycles Per Instruction (CPI)
- Current ideas: CPI between 3 and 5
- Pipelining
 - ☐ Why keep Fetch hardware idle while instruction is being decoded
 - ☐ Inspired by petroleum pipelines?

And the common objective of either these would have been to, so the objective is to increase **the** throughput of instruction execution; whether one had try to decrease the execution time of the single instruction or increase the throughput the rate at which instructions are executed. The common goal was really to reduce the execution time of programs which is still the case.

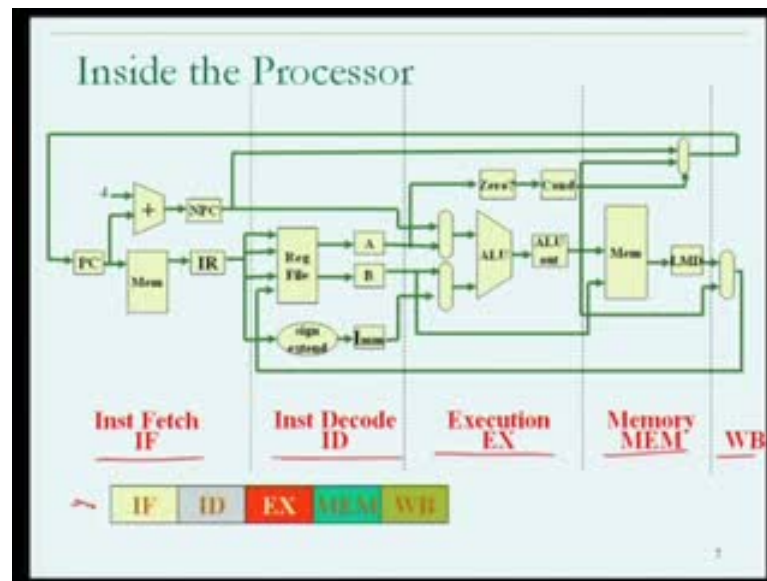
Now, in the case of pipelining, the idea is to try to keep the different pieces of hardware that are used in the execution of an instruction. As we had seen earlier, in this course, busy to the extent possible as motivated by the point down here; after instruction has been fetched, is it necessary to keep the hardware that did the fetch idle, until that instruction has been fully executed. And the answer as we briefly saw in the lecture 21 was no, one could do better.

(Refer Slide Time: 01:52)



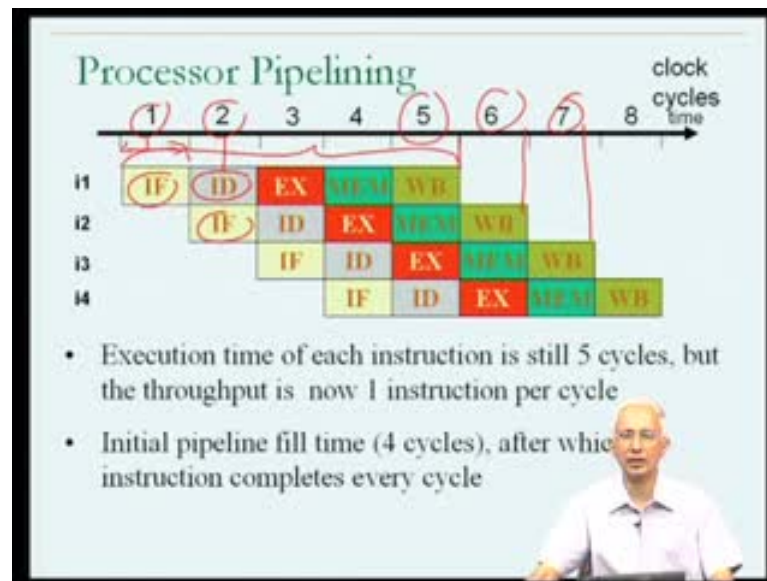
And I suggested that the design of pipeline processors may be viewed as having motivated by the idea of petroleum pipelines, which I briefly introduced with a diagram of this kind. The idea being that petroleum pipelines could be extremely long running to potentially thousands of kilometers and they are used to transfer liquids or gases over long distances, such as between, let us say a petroleum refinery and the location of the bulk of the end users.

(Refer Slide Time: 02:10)



Now, from this perspective, you will recall our **the** hardware, that we had reasoned out for the execution of an instruction, which included hardware to fetch, then decode or understand the instruction, cause the execution of the instruction, do any memory operation that may be required. And write the result of the instruction which we have, now abstracted into these five colored color coded boxes; this will be our prospective on the simple hardware, for execution of MIPS like instructions. And the simple idea of processor pipelining as we saw it last time; when we looked at the timeline, which I have now discretized in terms of clock cycles.

(Refer Slide Time: 02:42)



So, from now on, for the discussion of pipelining, when we look at the pipe the timeline, we consider cycle 1 followed by cycle 2 and are not too much concerned about the activity, the different pieces of activity that happen within the cycle; so, time the timeline has been discretized.

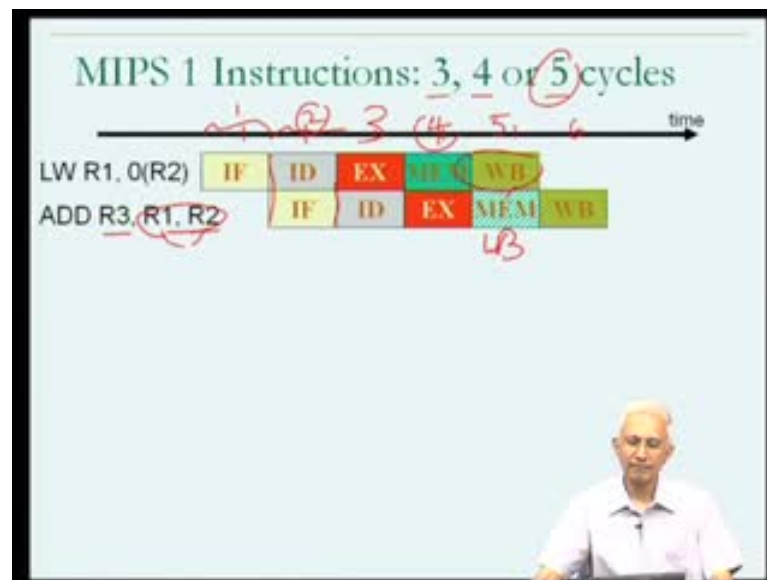
And if one instruction, the first instruction to be executed occupies the instruction fetch hardware for the first cycle; and subsequently occupies the instruction decode hardware for the second cycle and so on. We see that the second instruction to be executed instruction following; the first instruction in the program, could actually use instruction fetch hardware in the second cycle. In other words, as soon as the first cycle has finished being fetched second instruction could be fetched; thereby, allowing a much greater throughput of instruction execution.

As we had summarized towards the end of the class, even though a single instruction thus till take 5 cycles to be executed, instructions are completing at the rate of one per cycle; instruction 1 completes in cycle 5 , instruction 2 completes in cycle 6 , instruction 3 completes in cycle 7, and so on. And this is substantial improvement over an instruction completing only every 5 cycles.

So, it looks like pipelining has the substantial benefit from the performance side. I would remind you that, our discussion of pipelining opened with the question from the

performance prospective which is preferable; reducing the execution time of each instruction or improving the throughput, the rate at which instructions are execute. And here we see, that there is a great potential for an improving the performance; in other words, the speed at which programs are executed.

(Refer Slide Time: 04:26)



Now, let us look back a little bit at what our analysis of the MIPS 1 instruction requirements had, let us do. We will recall that we had analyzed, we had defined what the activity that could be done in hardware in 1 cycle. And based on that definition, we had sort of deduce that the different mixed instructions could either be executed in 3 cycles, 4 cycles or 5 cycles. And looking at our discretize timeline, let us look at a few examples. Now, once again we are concentrating on pipelining, but let us consider a situation where the first instruction which is going through the pipeline is a load word instruction; and I will remind you that in the case of the load word instruction, the number of cycles that it took to execute in our analysis of the simple processor which was not pipelined, was that it took 5 cycles to execute.

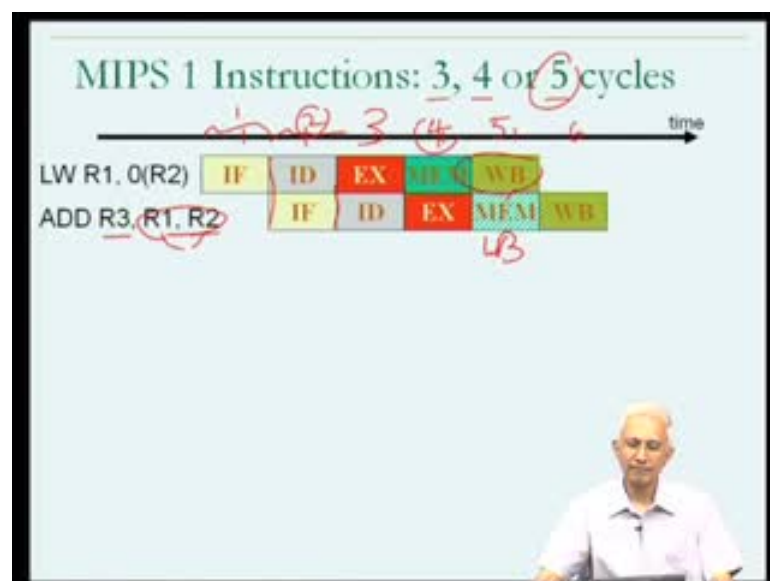
So, the instruction which took the complete, I mean which took, which actually use each of the 5 steps instruction, execution completely. And therefore, if I drew the use of the different pieces of hardware along the timeline, and from now on I may not even show you the tricks on the timeline, unless they are absolutely essential; so, just understand

that instruction 1 spends the first clock cycle in IF and so on. Time is not discretized and therefore, these boxes will provide us picture about what is happening on the timeline.

So, the load word instruction actually takes these 5 cycles to execute. Now, let us suppose that the second instruction, which is going to enter the pipeline, just after the load word instruction is an add instruction. And I will remind you that the add instruction, actually requires a cycle to be fetched, a cycle to be decoded, a cycle to execute and the cycle to write back, but does not require a cycle for any memory activities. Since it takes its operands out of registers and stores result into another register; so, it does not really require cycle in which to do a memory operation.

So, if I analyze, in a pipelined implementation of the MIPS 1 instruction set, very clearly the add instruction could be fetched in cycle 2, because the instruction fetch hardware is no longer being used in the fetching of the first instruction. Just proceeding along the timeline, it is clear that the add instruction can be decoded and its operands can be fetched in cycle number 2, which is in cycle number 3, which is why I show the ID, the grey box for the add instruction, as happening in cycle number 3. Note that the ID hardware is no longer being used by the first instruction; the ID of the first instruction had been completed in cycle number 2.

(Refer Slide Time: 07:01)

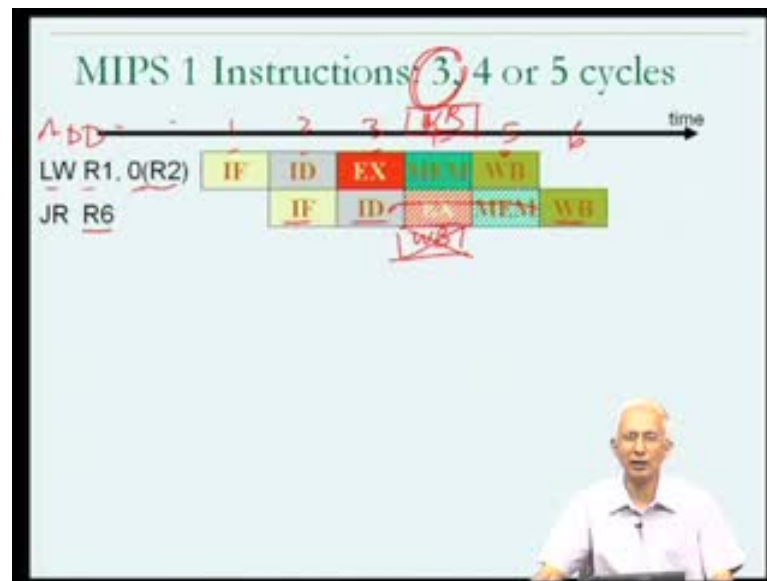


Similarly, I proceed down the timeline and I will see that the execution, in other words, the adding of the value inside the R 1 to the value inside R 2, can take place for the add instruction in cycle number 4. Now, with the question now is, the add instruction does not require the MEM stage; and therefore, I could think of actually using the write back hardware, for the add instruction in cycle number 5, but unfortunately the write-back hardware is being used in that cycle, by the load word instruction. And therefore, the earliest that the write-back hardware can be used for the add instruction would be cycle 6.

And in some sense, we understand that even though we show a blank box for what is happening as far as the add instruction is concerning in cycle number 5. In reality, we could view the cycle that particular cycle has being used, by the add instruction in the MEM stage, even though the instruction does not require the MEM stage. In other words, the MEM hardware need not be occupied doing something for the add instruction, but it is not going to be occupied anyway. And therefore, I will just show it in the hashed mode indicating that MEM hardware is in some sense, being used by the add instruction in cycle number 5; and is therefore, still in green but I am showing it in this hashed mode.

So, the bottom line was that the add instruction could not finish in 4 cycles, because it could not do the write-back, in cycle number 5; since the write-back hardware was currently being used by the previous instruction. So, there was no benefit from trying to design the process of pipeline; so that, the write-back of the add instruction happened in cycle number 5, the possibility was not there. Let us look at a few more examples of instructions sequences, to understand a little bit more about what is going to happened behind the scenes with the pipelines.

(Refer Slide Time: 08:50)



Now, second example, let me assume that the first instruction going through the pipeline is the load word instruction; and you will remember about load word instruction that it certainly has to be fetched; it certainly has to be decoded. There was an execution operation associated with it and that is because the effective address; in other words, the addition of the contents of register R 2 with the displacement must be done.

There is a need for memory operations, since this is a load word instruction. And there is a need for the write-back, because this is a, they need to update the register R 1 and this is the same as we had seen in the previous line.

Now, let us suppose that the load word instruction is followed by a jump register R 6 instruction. Now, the thing to remember of the jump register R 6 instruction, is that, it certainly requires the use of the IF stage, it has to be fetched as any instruction does. Similarly, it requires the use of the ID stage in that, it must be decoded and its operand must be fetched, its operand is R 6; the target address is contained in the register R 6.

There is the need for in fact, in the case of a jump register instruction there is not really any need for the EX stage, because the target address is immediately available in R 6 and there is no need for the MEM stage; MEM stage either because there is no competition to be done.

And therefore, the next piece of activity as far as the jump register instruction is the updating of the program counter in the write-back stage. And once again we see that there are the possibility of, actually this is an example of an instruction which would conceivably finish in 3 cycles, since it needs only the IF hardware, the ID hardware and the write-back hardware. And the possibility which we may have considered is that the write-back hardware could be used by the jump register instruction in cycle number 4, before it has used by the load word instruction.

So, the load word instruction is going to use the write-back hardware in cycle number 5. Now, we need to sort of look at this in a slightly moralistic prospective. Now, we have started by assuming that load word instruction was the first instruction in the pipeline, but remember, **the** during program execution, the load word instruction followed by the jump register instruction, may have followed other instructions; in other words, they could have been an instruction proceeding the load word, the load word instruction such as an add instruction. If they had been, then the add instruction would have been using the write-back hardware in cycle number 4; in other words, the instruction before the load word instruction.

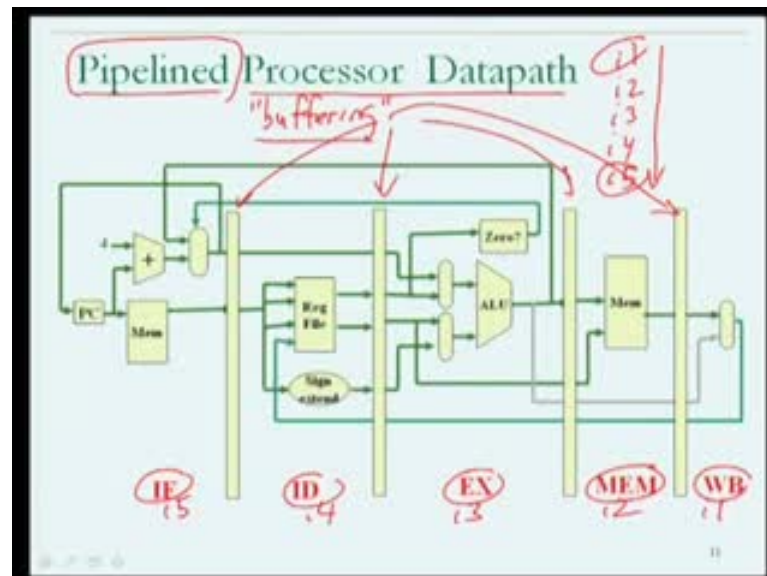
Hence, for the same reason that we argued with the previous example. We can now argue that the jump register instruction cannot use the write-back hardware in cycle number 4, because it would conceivably be in use by the instruction, before the load word instruction in that cycle; cannot be used in cycle number 5 either. And therefore, the earliest that we can use the write-back hardware for the jump register instruction would be cycle number 6.

So, once again since cycle number 4 and 5, in during cycles number 4 and 5, no useful activity as far as the jump register instruction are taking place, but what I will do is just assume that the EX hardware and the MEM hardware, in this hashed mode are sort of pass through or not being used for the execution of the steps of the instruction, but **are in** some sense being occupied by the jump register instruction in cycle 4 and cycle number 5; prior to the write-back hardware being used from cycle number 6.

And again one way to view this is that, in the hardware that we had outlined, the information was flowing from the IF stage to the write-back stage; Information pertaining to any one instruction. And therefore, the EX stage and the MEM stage are

providing the path, for the flow of the information from the ID hardware, through the write-back hardware and this will keep things nice and simple. If you once again just view the jump register instruction as though conceivably taking only 3 cycles, actually occupying all the 5 cycles; in other words, all the 5 pieces of hardware associated with instruction execution.

(Refer Slide Time: 12:46)



So, if I look at the pipeline from this perspective, we now has this picture of the 5 pieces of hardware associated with instruction execution exactly as they were before, but there is this notion of 1 instruction, let us say instruction i 5 being fetched, while instruction i 4 is being decoded, while instruction i 3 is being executed and instruction i 2 is using the MEM hardware and instruction i 1 is being return back.

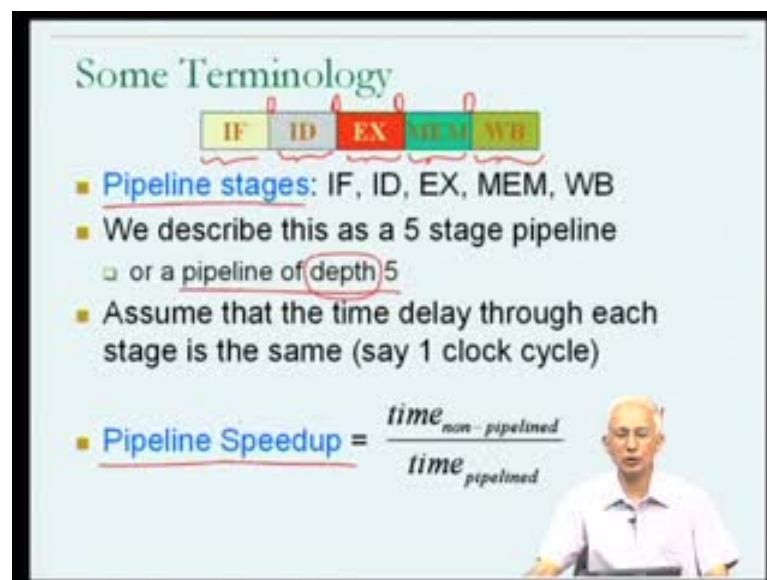
And I write the instructions in that order, assuming that instruction i one is the first among them to be executed in program order. So, this is the order in which your program is executing the instructions and if look at the hardware within the processor, I might find instruction i 1, in other words, the first of them occupying the write-back hardware; at the same time, there instruction i 5, the last of them is occupying the instruction fetch hardware.

Now, to make this work, it may be necessary to remember some of the information that is being passed between the IF stage and the ID stage, we are talked about some special

purpose registers, in our discussion of the pipeline. And therefore, what I am going to assume, in talking about a pipeline implementation of the processor, is that, there may be some a need for some buffering of that kind or some memory of that kind and that is what will be indicated by these yellow boxes, yellow rectangles which are now shown between the 5 pieces of hardware.

So, they are these 4 rectangular boxes and we could think of them as containing some kind of buffering or some kind of memory, required to transfer information from the previous piece of hardware to the next piece of hardware. And in some sense we had seen this idea in the form of some registers, but this is just to allow for that requirement of pipelining the processor; some additional information may have to be remembered. So, the bottom line is that we are going to assume that without a great deal of additional complication being added to the hardware, that we have already seen. The ordinary processor that we had looked at which we now refer to as the processor data part; the path ways through which data and information and instructions flow through the processor for execution and the pipeline equivalent may not be that much different. We will sort of work with that assumption, some additional buffering, some additional small amount control may be required.


(Refer Slide Time: 15:08)



Some Terminology

IF ID EX MEM WB

- **Pipeline stages:** IF, ID, EX, MEM, WB
- We describe this as a 5 stage pipeline
 - or a pipeline of depth 5
- Assume that the time delay through each stage is the same (say 1 clock cycle)
- **Pipeline Speedup** = $\frac{\text{time}_{\text{non-pipelined}}}{\text{time}_{\text{pipelined}}}$



Now, for in terms of some of the terminology that we are going to use; after now I have talked about the 5 pieces of hardware that we had reasoned out as being necessary for the execution of an individual instruction.

We have now put them together, with some amount of buffering in between them which I am not generally going to show in the diagrams, but in referring to any one of these pieces of hardware is often, the most common term which is use to prefer to them as the pipeline stages or the stages of the pipeline.

So, I will talk about the I F stage or the instruction fetch stage, the write-back stage or the WB stage of a pipeline. Now, in this particular pipeline has 5 stages and I would i could therefore describe it as a 5 stage pipeline. One could imagine that there could be pipelines which are design to have a smaller or a larger number of stages and you could imagine at along these lines. In our discussion of different steps involved in executing an instruction, we had package to the different steps in to these 5 pieces of hardware, but we could conceivable have package them into let us say - ten smaller pieces of hardware and then we would have now been talking about 10 pipeline stages with different names; and We would be talking about ten stage pipelines. Another way of describing this is that some people talk about a pipeline of depth 5, rather than talking about a 5 stage pipeline; so, this is the notion of the depth of a pipeline.

Now, let us think about pipelines little bit, until now when we talk about the 5 pieces of hardware, we were working with some rough understanding of how much work can be done by hardware in 1 clock cycle; the clock cycle is defined by some hardware clock associated with the processor. And we were working towards having about the same amount of work being done in each of the 5 pipeline stages. And we will continue with that assumption, I will assume that takes 1 clock cycle, to finish the work of any one of these pipeline stages.

So, there equal us for us the time delay through this stage is concerned. I can therefore, think about concept of speed up. Now, the principle behind trying to come up with this speed up is that we understand that by using a pipelined processor rather, than by using a pipelined process which is not pipelined; in other words, one without the buffering one in which single instruction is completely executed, before the next instruction is fetched.

So, I could compare the performance of a pipeline processor, with the performance of an equivalent non-pipelined processor, using this measure called pipeline speed up; pipeline speed up is defined as a time that takes to execute, let us say a particular program with a particular input data, on the non-pipelined implementation of the processor divided by the time that it takes to execute that same program, on the same input on a pipeline version of the program, on the pipeline version of the processor. So, assuming the same instruction set, we are assuming the same program, the same input data, so you run it you build a pipeline version of the processor which can execute those instructions; you build a non- pipelined version, you measure the time that it takes on each. And the ratio of the two you notice that, we hope that the denominator, the time on, the time on the pipeline processor will be less than the time on the non-pipeline processor; and the pipeline speed up should be hope be greater than 1, that is our hope.

And this is the terms, speed up is use in other contexts as well which why it qualified by the term pipeline, in front, we are trying to measure how many times faster a program can become the execution by using a pipelined implementation.

(Refer Slide Time: 18:56)

Pipeline Speedup IF ID EX MEM WB

- For 5 stage pipeline taking 1 cycle per stage
 - Let us compute the speedup over a non-pipelined processor that takes 5 cycles for every instruction
 - Calculate how much time each of these processors takes to run a program involving the execution of n instructions
 - Non-pipelined processor: $5n$ cycles
 - Pipelined processor: $4 + n$ cycles
 - Speedup = $\frac{5n}{4 + n} \rightarrow 5$ as $n \rightarrow \infty$

Now, let us try to calculate the pipe potential pipeline speed up, for our 5 stage pipeline or a pipeline of depth 5.

The way we will do this is we will start by stating as assumption, I am going to assume that it takes one stage each stage of the pipeline takes 1 cycle; and I am going to compute the speed up over a non-pipeline processor that takes 5 cycles for every instruction. Now, I know that I could build a non-pipeline processor which takes 5 cycles for some instructions, 4 cycles for other and 3 cycles for yet others.

But I will compute the speed up, in terms of a pipeline processor which takes 5 cycles for every instruction; and the reason is it will be a little difficult to talk about the number of cycles that it takes for an instruction in the absence of the program. If I have a program which contains only jump register instructions, then very clearly, every instruction takes 3 cycles in the non-pipeline version. So, this is just an assumption to allow us to talk in terms of programs, in general rather than specific programs.

Now, what we will do is we need to calculate how much time it will take to run a program, on the pipeline processor; and we will have to calculate how much time it takes to execute the same program, on a non-pipeline processor. And what I am going to start off is I am going to assume that the execution of the program, involve the execution of n instructions; n could be ten thousand, n could be a million it does not matter, we just going to use this notation that the number of instructions which are going to be executed, in order for the program to run completely is n .

Now, we can reason about how many cycles it will take, as far as execution on the non-pipeline processor is concerned. Now, we know that, on the non-pipeline processor, it takes 5 cycles for every instruction, if there are n instructions then the total time to execute this program will be $5n$ cycles; so, that is easy to see.

What about the case of the pipeline processor? Now, I know that in the case of pipeline processor, when the program starts executing in the first cycle, no instruction will complete it is a 5 stage pipeline. So, in the first cycle the instruction i_1 the first instruction will get fetched; in the second cycle instruction i_1 will get decoded and instruction i_2 will be fetched and so on; and I know that for the first 4 cycles no instruction will finish execution.

However, every cycle after that if I am lucky and instruction will finish execution; therefore, I can argue that the total number of cycles required to execute the program of n instructions and the pipeline processor will be $4 + n$.

So, for the first 4 cycles the pipeline fills up and for every cycle after that hopefully an instruction will finish execution; and therefore, the n instructions will take $4 + n$ cycles; and therefore, I can compute this speed up is $5n$ divided by $4 + n$.

Now, this is interesting, it does not give me any particular insight, but I could, then think about the scenario where n becomes larger and larger, you know that the $5n$ divided by $4 + n$ is not very interesting when n is equal to 1, because when n is equal to 1 in other words if the program takes 1 instruction, the programs execution involve only 1 instruction, then the speed up is going to be 5 divided by 5 which is 1; if there only 2 instructions, then this is 10 divided by 6 which is 1 point something and so on.

But what if n is very large, so for example what if n is many millions or many billions; so, the typical case that we are concerned about is where there are large programs in execution, where the more concern about the speed up for large programs, then the speed for the small programs; because small programs take a small amount of time anyway. So, one could actually look at this expression for speed up, in the limit as n tends to infinity and get to, get some field for the significance of the speed up and if you take the limit as n tends to infinity, you will notice that this tends to 5.

(Refer Slide Time: 23:14)

Pipeline Speedup

- A pipeline with p stages could give a speedup of p (compared to a non-pipelined processor that takes p cycles for each instruction)

(Refer Slide Time: 23:21)

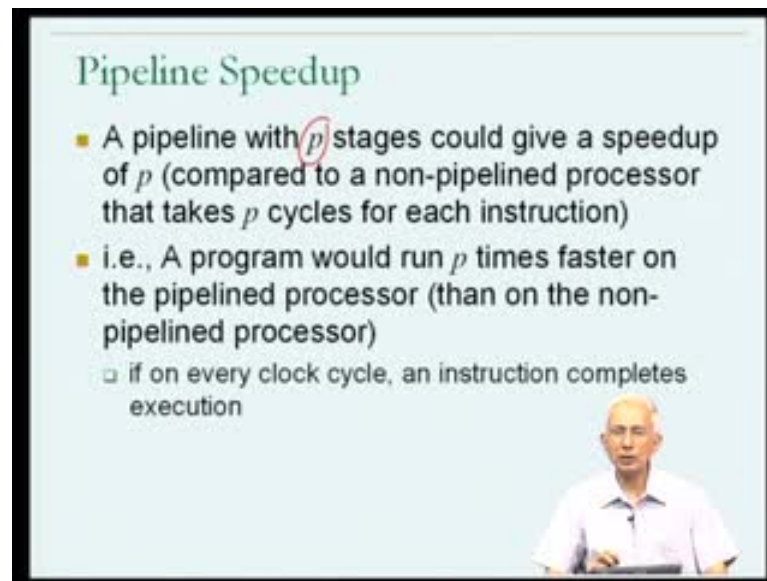
Pipeline Speedup

IF ID EX MEM WB

- For 5 stage pipeline taking 1 cycle per stage
 - Let us compute the speedup over a non-pipelined processor that takes 5 cycles for every instruction
 - Calculate how much time each of these processors takes to run a program involving the execution of n instructions
 - Non-pipelined processor: $5n$ cycles
 - Pipelined processor: $4 + n$ cycles
 - Speedup = $\frac{5n}{4 + n} \rightarrow 5$ as $n \rightarrow \infty$

So, the thing to know it here is that for extremely large programs, I could actually expect that the program would run 5 times faster on the pipeline processor than would have run on the non-pipeline processor. Now, this is interesting, because we note that 5 comes from, if you look carefully at the derivation, the 5, in this result is coming from this term and that term is coming from the number of cycles that it takes for every instruction to execute on the non-pipelined implementation, which was actually coming from the number of stages in the pipelined processor.

(Refer Slide Time: 23:45)



Pipeline Speedup

- A pipeline with p stages could give a speedup of p (compared to a non-pipelined processor that takes p cycles for each instruction)
- i.e., A program would run p times faster on the pipelined processor (than on the non-pipelined processor)
 - if on every clock cycle, an instruction completes execution

Therefore, rather than thinking about the case where 5 stages in the pipelined processor, if I now try to generalize to the case where there are p stages in the pipeline processor, let us say 20, 30 whatever. Then I could argue along the same lines as in the previous slide, then I could expect the speed up of p , if I am comparing the pipeline processor with a non-pipeline processor, they takes p cycles to execute each instruction.

In other words, if I had a pipeline with 20 stages in it, I might expect that pipeline processor would be capable of executing an instruction 20 times faster than the non-pipelined equivalent. So, this is now becoming an interesting situation, the 5 stage pipeline with the speed up of 5 maybe not but with the 20 stage pipeline with the speed up of 20; capable of executing programs 20 times faster, thanks to the paying attention to the rate at which instructions are executed, rather than the time for the execution of any one instruction.

So, the way to interrupt the speed up of p is, a program would conceivably run p times faster on the pipeline processor, then on the non-pipeline processor. And once again for large values of p this is an extremely good result. We have done something very positive by thinking about pipeline processors, from the prospective of the performance of the processor; the satisfaction of the percent running programs on the processor.

(Refer Slide Time: 25:10)

Pipeline Speedup

IF ID EX MEM WB

- For 5 stage pipeline taking 1 cycle per stage
 - Let us compute the speedup over a non-pipelined processor that takes 5 cycles for every instruction
 - Calculate how much time each of these processors takes to run a program involving the execution of n instructions
 - Non-pipelined processor: $5n$ cycles
 - Pipelined processor: $4 + n$ cycles
 - Speedup = $\frac{5n}{4 + n} \rightarrow 5$ as $n \rightarrow \infty$

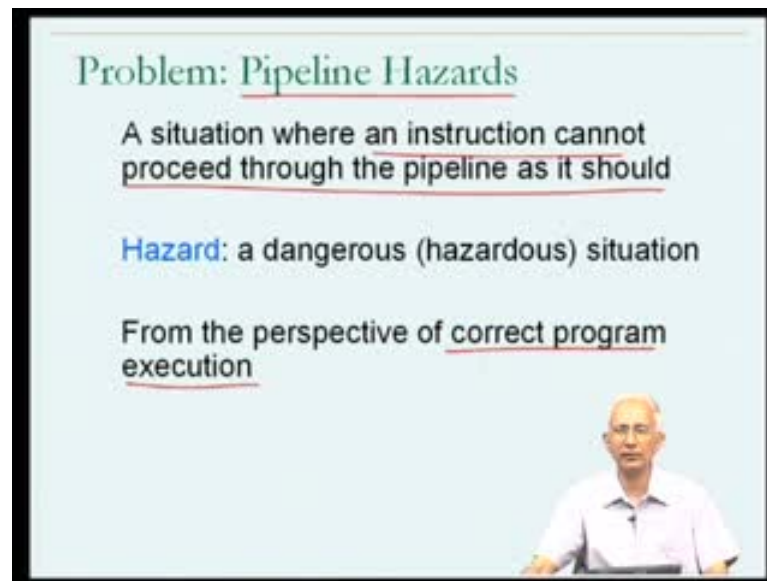
Of course, this is built under the assumption that on every clock cycle an instruction completes execution. If you look back at the derivation, I had over here, the time for the pipeline process was the pipeline filled time which was p and the current example is going to be p minus 1. For the 5 stage pipeline, there was a fill time of 4; for the p stage pipeline, they will be a fill time of p minus 1. And then, I was assuming that every cycle after that an instruction could complete; so, that is the caviar that I have written over here.

(Refer Slide Time: 25:36)

Pipeline Speedup

- A pipeline with p stages could give a speedup of p (compared to a non-pipelined processor that takes p cycles for each instruction)
- i.e., A program would run p times faster on the pipelined processor (than on the non-pipelined processor)
 - if on every clock cycle, an instruction completes execution

(Refer Slide Time: 25:48)



So, the speed up of p will result, if on every clock cycle an instruction completes execution, and the question is whether this is going to be achievable in practice. Now, we are going to next, spend some time looking at one of the problems with programs because of which, this is not typically going to happen; so, it so this is not always going to be the case that every clock cycle in instruction is going to complete.

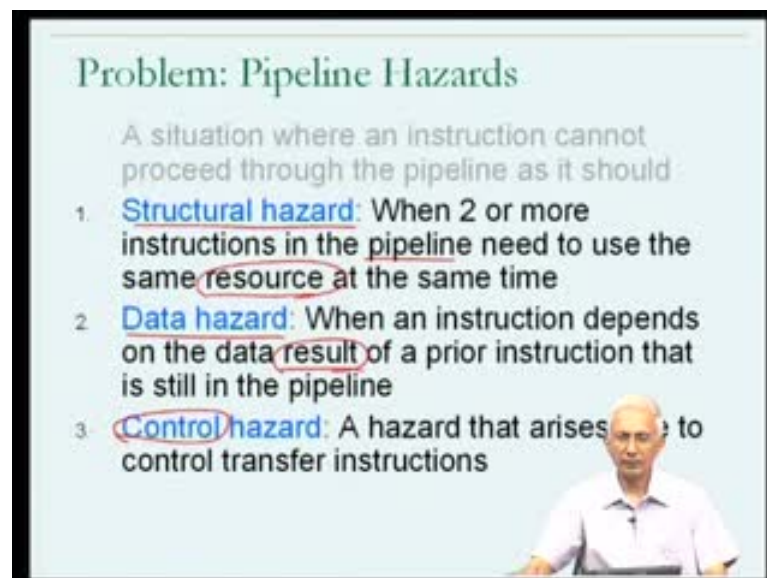
Now, the situations in which it is not possible for instruction to complete every cycle, arise out of problems within dynamically problems, within the pipeline due to different kinds of instructions; these problems are known as pipeline hazards. Basically, a pipeline hazard is a situation, where an instruction cannot proceed through the pipeline as it should; and if an instruction cannot proceed through the pipeline, as it should; in other words, from IF to ID, to EX, to MEM, to write-back, during consecutive cycles, then it is not going to be possible, for an instruction to complete every cycle.

So, the question is what are the different kinds of hazards that could arise, in the kind of pipelines? That we have seen for the kinds of the programs that we expect we may have to write. Before that, let me just to explain, the term hazard in general, means, a dangerous hazardous situation; and in the case of the pipeline hazards, the danger arises not from some physical danger, but from the fact, that if the instruction is allowed to proceed through the pipeline, stepping through stages of the pipeline in consecutive

cycles, then what could happen is the correctness of the program could be compromised.

So, this is hazard that one has to worry about from the perspective of designing the hardware. So, the term pipeline hazard is of concern to the people design the hardware, they have to make sure the programs run without these hazardous situations, resulting in incorrect program execution; correct program execution is essential, otherwise the processor is not doing what the programmer intended his instructions comparison his program to do.

(Refer Slide Time: 27:48)



Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

1. **Structural hazard:** When 2 or more instructions in the pipeline need to use the same resource at the same time
2. **Data hazard:** When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. **Control hazard:** A hazard that arises due to control transfer instructions

The slide features a light blue background with a black border. A small inset image of a man in a white shirt is visible in the bottom right corner of the slide content.

So, with this understanding of what hazard is, let us let me outline these different kinds of pipeline hazards, that one has to worry about are. Now, the first kind of pipeline hazard that architects worry about is something called the structural hazard; and the term structural hazard is used for situations, where instructions in the pipeline need to use this same resource of the pipeline at the same time; in other words, it is because two of them need to use the same resource that the same time.

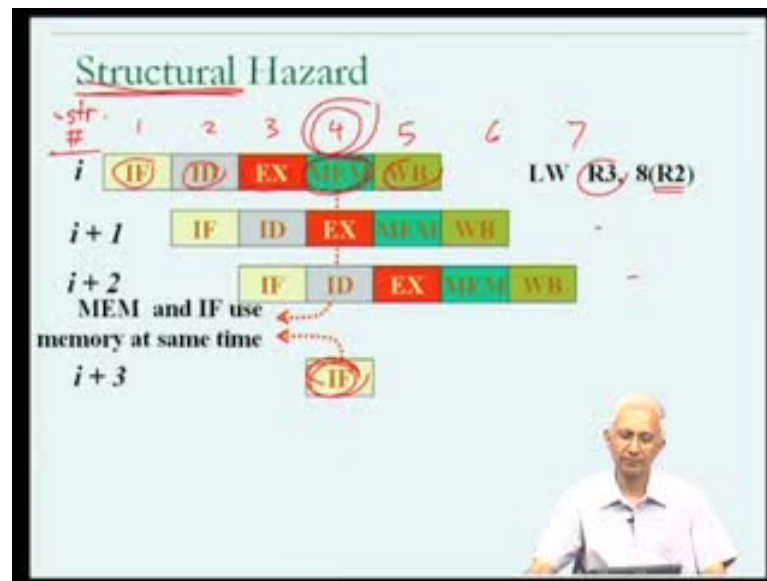
Hardware resource of the pipeline that it is not safe for them to proceed through the pipeline as they should, stepping to the next stage on the pipeline in the next cycle. therefore, something to do with pipeline resources, the hardware resources of the pipeline that what is the structural hazard is.

We see more about structural hazards shortly. The second kind of hazard which the architect and the programmer has to worry about there is something called the data hazard, it relates to the data that the instructions of the program are manipulating; and it describes a situation where an instruction depends on the data result of a prior instruction that is still in the pipeline. In other words, there is situation where there is one instruction which is somewhere in the beginning of the pipeline, there is another instruction which is somewhere towards the middle or the end of the pipeline; and it is the second instruction that is producing a result that is of relevance, to the instruction earlier in the pipeline.

So, there is some kind of a data relationship between the instructions; as I was result to which it is not safe for the instruction may be earlier in the pipeline, to proceed through the pipeline as it should; so, that is known as a data hazard note that, this is distinct from the structural hazard. Structural hazard arises because two instructions require the same piece of hardware, at the same time. The data hazard arises because an instruction needs a piece of data that another instruction in the pipeline is supposed to produce.

So, it is some kind of a producer consumer type of a problem. If the producer has not yet produced the piece of data, then the consumer is not going to be in a position to proceed through the pipeline as it should, until the data has been produced. Now, the last kind of hazardous situation that I am going to talk about there is something called the control hazard; and as a name suggests, this is a kind of a hazardous situation that arises due to control transfer instructions. And we see each of these in a few slides, in the discussion to follow.

(Refer Slide Time: 30:34)



Let us start by looking at the structural hazard. So, this is the situation where two or more instructions in the pipeline, need to use the same resource, at the same time. The best way to understand the structural hazard will be to look at an example; so, let us proceed to look at the situation that could arise as instructions are using the hardware resources, in other words, the IF, ID, EX, MEM, write-back pieces of hardware comprising the pipeline. So, let us suppose that the first instruction of interest to us which I will describe as instruction i is a load word instruction.

So, the load word instruction will be fetched in the first cycle; it will be decoded fully understood in the second cycle and its operand the value of R 2 will be fetched. In the third cycle the value of 8 plus, the value inside R 2 will be calculated; and in the fourth cycle the cache memory will be the address which was computed in the third cycle will be sent of the cache memory where the data will be read; finally, in the fifth cycle register R3 will be updated.

So, we know exactly what happens as far as the load word instruction is concerned, it occupies all the 5 cycles. Now, let us suppose that there is an instruction following the load word instruction and I had talked about the load word instruction as being instruction number I ; so, the instruction that follows it is the instruction number I plus 1. And I am not too concerned about what that instruction is in fact, let us suppose that the instruction following instruction i plus 1 which we call instruction i plus 2, is also not

may not too concerned about **the** what the instruction is. So, we have a situation where the load word instruction was executing between instructions 1 and 5; the instructions that follows it was from cycles 2 to 6, instructions that follows that was from cycles 3 to 7 and so on. Let us comes, consider the next instruction, the instruction which will call I plus 3.

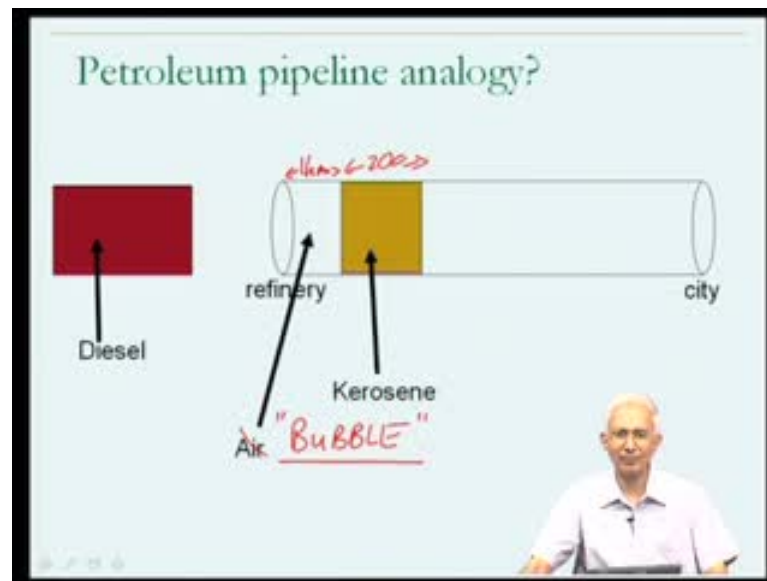
Now, that instruction is supposed to be fetched in cycle number 4. And if you look back and see what the other instructions are doing in cycle number 4, instruction $i + 3$ is using the IF hardware, instruction $i + 2$ is using the ID hardware, instruction $i + 1$ is using the EX hardware, instruction i is using the MEM hardware. It looks like none of them are trying to use to the same piece of hardware at the same time; and you may be wandering what kind of a structural problem is there. Each of them is using a different piece of hardware, nothing could go wrong, but let us think it a little about what is happening as far as instruction i is concerned and instruction $i + 3$ are concerned.

Now, in what instruction i is doing the instruction i , the load word instruction is doing a memory operation, which means, that it is using the cache in cycle number 4. What is instruction $i + 3$ doing? It is being fetched. And once again we understand that in order to be fetched if main memory is not going to be used, then instruction $i + 3$ must be using the cache memory.

So, here, we have maybe called a structural hazard, we have a situation where in cycle number 4, in other words, at the same time instruction i and instruction $i + 3$, both need to use memory, both have to access memory. One in order to be the piece of data, the other in order to access an instruction, but the factor means that both of them need to use the same hardware resource memory or cache in this particular case, at the same time. And this is a situation where the same piece of hardware, and therefore, we refer to this as a structural hazard.

In cycle number 4, instruction i and instruction $i + 3$, both need to use memory or cache, in our, based on our assumptions about how the processor operates; and therefore, this has to be classified as a hazardous situation. These two, the same hardware resource cannot be used by two instructions at the same time.

(Refer Slide Time: 34:24)



Now, let us just go to our petroleum pipeline analogy; I, since I have raised the analogy of the petroleum pipeline, I thought I should use it in some point or the other and is there any kind of equivalent for hazardous situations, as far as the petroleum pipeline is concerned.

Now, let me just remind you, the pipeline is this pipe, very long pipe, which runs let us say from a refinery to a city, and it can be used for example, for transporting the different products of crude petroleum from the refinery; the refinery takes in crude petroleum and breaks it up into the different products, like kerosene, diesel airline fuel, etcetera and then transported through the pipeline to the end user.

So, let us suppose that currently, let us suppose this is a 1000 kilometer pipeline and currently we are using it to transfer some volume of kerosene, let us suppose that it is about 200 kilometers within the pipeline; we are using it to transport some volume of kerosene from the refinery to the city.

So, that is what the pipeline is currently being used for and is going to be used the way that this happens as we saw is by pumping the kerosene through the pipeline; It is not progressing through the pipeline by some gradient by the fact that, the pipeline is sloping down progressively, because it is explicitly being pumped.

Now, let us suppose at this point in time, the kerosene fraction has is occupying the pipeline and we need to start off, we actually want to also transport, let us say looks like, looks like about 400 kilometers, in terms of the pipeline, some large volume of diesel from the refinery to the city.

Now, the situation that we have is we could immediately start pumping diesel into the pipeline as soon as the 200 kilometers of kerosene had been pumped into the pipeline; we could start pumping the diesel into the pipeline. But the problem that could arise now, is that the kerosene and the diesel would mix could using something, which is neither kerosene nor diesel and end user will not be very happy.

Therefore, this is something like a hazardous situation, in the instruction execution pipeline, where there instructions that are interfering with each other. And the question is in order to solve the pipeline in the instruction execution scenario, can derive some kind of inside from the petroleum pipeline scenario.

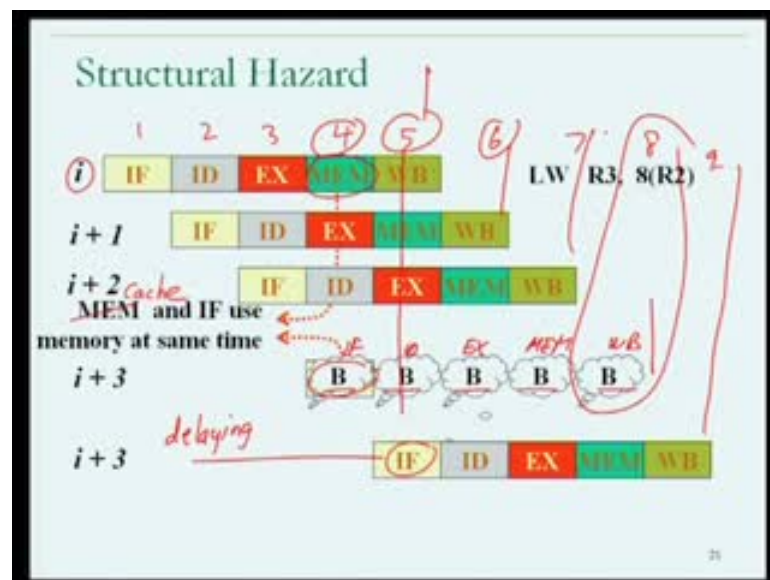
So, when we have these two volumes of fluid which interfere with each other, if they, when they are going through the pipeline. How can we remove the problem as far as the petroleum pipeline is concerned? And the answer to the question which may arise in your mind, in terms of the petroleum pipeline is that before pumping the diesel, the diesel volume into the pipeline, if I actually pump some air into the pipeline, in other words, the kerosene is progressing is being pump through the pipeline from the refinery to the city, but before pumping the diesel into the pipeline, if I pump some air into the pipeline and then I pump the diesel into the pipeline, then there is no danger of the kerosene and the diesel interfering with each other.

So, if I say that - for the 200 kilometers of kerosene in the pipeline, I pump 1 kilometer of air into the pipeline, and i am not an expert in pipeline technology, so i do not know if 1 kilometer is even necessary, and not even sure, if this is how they resolve these problems in pipelines, but it seems like a satisfying solution. And subsequently, we pump the large volume of diesel through the pipeline, then the diesel and the kerosene will not bother each other.

So, there is this notion of pumping air into the pipeline, the petroleum pipeline and technically instead of talking about it as air, I could talk about it as inserting a bubble of air into the petroleum pipeline.

So, a large bubble maybe 1 kilometer in size, just to separate the kerosene from the diesel. And we could try to think of using exactly the same kind of an idea, in the case of the instruction execution pipeline; to separate instructions from being from doing dangerous things, we could pump air into the pipeline they must be an analogy of what it means to pump air into the pipeline.

(Refer Slide Time: 38:26)



Going back to our structural hazard, we had the situation where in cycle number 4, the instruction i and instruction i plus 3, both needed to use the same hardware resource; and for the moment I am referring to it as memory, but could think of it as cache, without loss of generality.

So, what could be done to prevent this hazardous situation from resulting in something going wrong? Now, very clearly, in cycle number 4, since the hardware resource is currently being used by instruction number i , it cannot be used by instruction i plus 3. And therefore, we could draw the analogy of the petroleum pipeline, and say there as far as instruction i plus 3 is concerned, it will have to instead of pumping instruction i plus 3 into the pipeline in cycle number 4, I actually might need to pump some air into the

pipeline, in cycle number 4 instead of instruction $i + 3$. And I will actually show it by a bubble of air going into the pipeline, and this arise the need for pumping this bubble of air into the pipeline is, that in cycle number 4 instruction $i + 3$ cannot use the resource namely the cache, because in cycle number 4 that particular resource is being used by another instruction; hence, use the petroleum pipeline solution and pump air into the pipeline.

Now, what does it mean to pump air into this electronic pipeline? Very clearly, pumping air using a pump or a fan or something is not going to help the correct execution of instructions. So, we just drawing an analogy, but what seems to be the case in the case of the petroleum pipeline, what was the purpose of pumping the air? The purpose of pumping the air was to postponed the arrival of the diesel into the pipeline, and that is exactly what we will view the purpose of the air over here.

What we mean by the bubble of air in the pipeline is that we are postponing the fetching of instruction $i + 3$, from cycle number 4 until cycle number 5 possibly; which is why I show instruction $i + 3$, as being fetched in cycle number five and if I look back at the other activities in cycle number 5, it could well happen that this no other use of the cache memory. And the therefore, this is the safe use of the instruction fetch hardware as far as $i + 3$ is concerned.

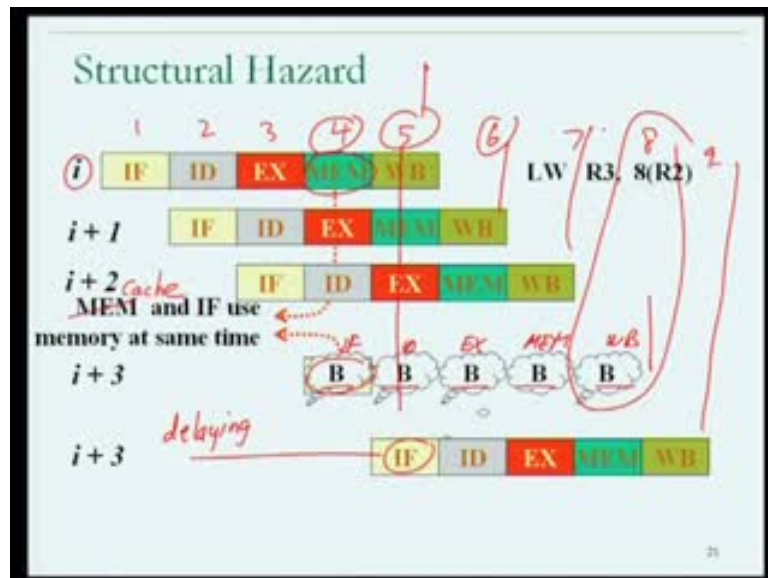
Therefore, by pumping a bubble of air into the pipeline in cycle number 4, instead of using that cycle for fetching instruction $i + 3$, I have avoided the problem; it is no longer the case that instruction i and instruction $i + 3$ are using the cache at the same time. Notice that instruction i is using the cache in cycle number 4, and instruction $i + 3$ is using the cache in cycle number 5; and therefore, there is no structural problem, it is not a situation where two instructions are using the same hardware resource at the same time.

Now, what happens to this bubble, you will notice that I have sort of conceptually shown this bubble, as progressing through the pipeline, just like an instruction would, because if you think about it, if the instruction fetch hardware was not being used in cycle number 4 but only in cycle number 5. Then what will happen to the instruction decode hardware, in the next cycle? The answer is they will be no instruction which needs to be decoded; and

therefore, they will actually be a bubble of air, in the instruction decode hardware in the next cycle.

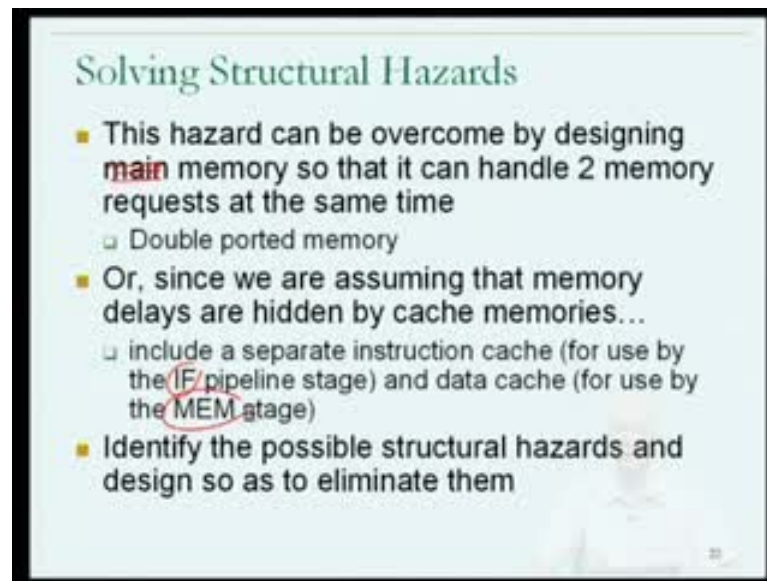
Similarly, there will be a bubble of air in the instruction EX hardware in the next cycle and so on. So, one can actually view the bubble of air as progressing through the pipeline just, like an instruction. So, the net result was that by delaying instruction i plus 3, by 1 cycle we were able to overcome the structural hazard; you notice that there is a performance consequence of doing this. You will notice that, in instruction 5, instruction in cycle 5 one instruction completed, in cycle 6 another instruction completed, in cycle 7 and another instruction completed.

(Refer Slide Time: 42:28)



But in cycle 8, no instruction completed, it was only in cycle 9 that the next instruction completed. And therefore, by taking this approach to dealing with the structural hazard, we are actually causing there to be 1 cycle in which no instruction completes; and therefore, our assumption about 1 instruction completing every cycle through which we calculated the pipeline speed up, is actually not going to result, if we use this kind of approach to dealing with structural hazards.

(Refer Slide Time: 42:59)



Solving Structural Hazards

- This hazard can be overcome by designing **main** memory so that it can handle 2 memory requests at the same time
 - Double ported memory
- Or, since we are assuming that memory delays are hidden by cache memories...
 - include a separate instruction cache (for use by the **IF** pipeline stage) and data cache (for use by the **MEM** stage)
- Identify the possible structural hazards and design so as to eliminate them

Now, there are clearly going to be alternatives and the alternative which I outline in this slide is that it should be possible to design the hardware, so that if we can identify all the situations where there could be two instructions which require the same hardware resource at the same time; maybe we could design the hardware, so that you can handle the multiple requirements at the same time.

A simple example is, we had this scenario where both instruction i and instruction $i + 3$ required the use of memory in cycle number 4; and therefore, I could conceivably have designed memory, so that it could actually handle two requests at the same time; and that kind of memory is actually not uncommon, it is called people do design memories, to able to handle multiple requests at the same time; and therefore, it is not an feasible idea.

But in our particular setting, we could do even better, we could say, since we know that main memory is not going to enter the picture too often from the perspective of instruction fetches or data fetches or stores, but most of the time this would happen through cache memories, we have not understood fully what a cache memory is, but we could handle the problem that we have seen, by having a separate cache for data, a separate cache for instructions; and by a separating the cache into two caches,

we actually diffused the structural hazard problem that had a reason, in the example that we have. Therefore, in general, one could argue that wherever structural hazards can be anticipated by the person who designed the pipeline.

The structural hazard can also be resolved by suitably designing the hardware; so that, the structural hazard does not become a situation where a bubble of air, where a cycle of non-activity has to be pushed through the pipeline **and so in the slide I outline how**. If you have separate instruction, a separate cache for instruction and a separate cache for data, then the instruction cache would be used by the IF pipeline stage and the data cache will be used by the MEM pipeline stage. And therefore, they would not be a situation where two different instructions required the same piece of hardware, the same cache, at this, in this same cycle. And again for any other kind of structural hazardous situation, the hardware, the person designing the hardware could use reasoning to diffuse the situation by suitable design of the hardware. So, in some sense we could assume that in modern processes structural hazards are likely to be handled in this way.

In other words, the architect would identify the possible structural hazards and do the design of the pipeline hardware and its resources so as to eliminate them.

(Refer Slide Time: 45:40)

Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

- 1 ✓ **Structural hazard:** When 2 or more instructions in the pipeline need to use the same resource at the same time
- 2 **Data hazard:** When an instruction depends on the data result of a prior instruction that is still in the pipeline
- 3 **Control hazard:** A hazard that arises due to control transfer instructions

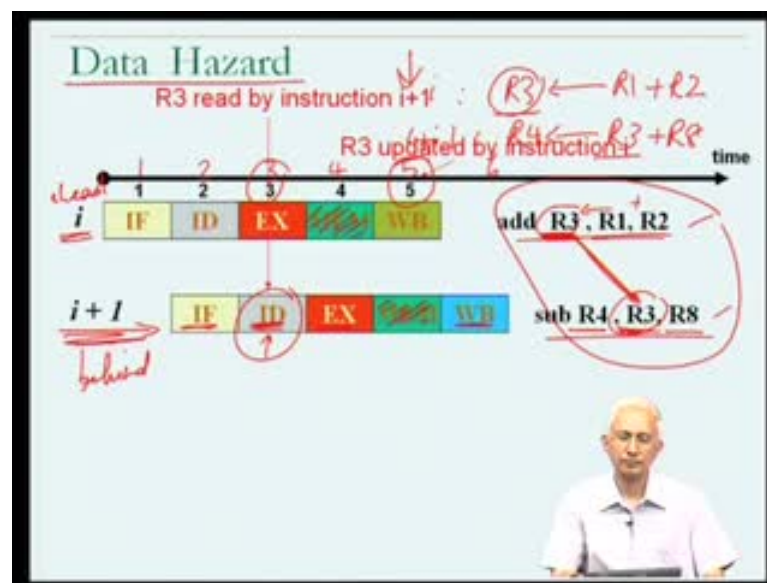
29

Now moving right along, now that we have seen what a structural hazard is, we have seen how a structural hazard could be handled, in fact, we saw two ways that a structural

hazard could be handled, one using the pipeline analogy of pumping air through the pipeline, the other that of designing the pipeline, so that structural hazards are diffused by suitable design of the components, the resources of the pipeline.

We will move on to the next kind of pipeline hazard, which I had labeled as the data hazard, a situation where an instruction depends on the data result of a prior instruction that is still in the pipeline. So, once again, let us try to understand what a data hazard would look like by looking at an example.

(Refer Slide Time: 46:26)



Now, in this example, once again I have instruction i which is fetched in cycle 1, decoded, executed, written MEM and write-back. So, you will note that in the first cycle it is fetched, in the second cycle it is understood, and it is operands $R1$ and $R2$ are fetched, in the third cycle the addition of $R1$ and $R2$ is done by the ALU, no real activity happens in the fourth cycle.

So, this is one of those hashed MEM stage boxes; finally, in the fifth cycle register $R3$ is updated with the sum, so you know exactly what happens as far as instruction i the add instruction is concerned.

Now, let us suppose that the next instruction, instruction $i+1$ is a subtract instruction. In particular, let us suppose that the subtract instruction takes as its destination register $R4$ and a source operands are registers $R3$ and $R8$.

Now what will happen to this instruction, it will get fetched in cycle 2, it will get decoded and fetches operands in cycle 3; it will execute the subtraction in cycle 4. The MEM is once again hashed, no useful no real activity happening and it will write-back the result, the result of subtracting the contents of R 3, and R 8 into R 4 in cycle number 6.

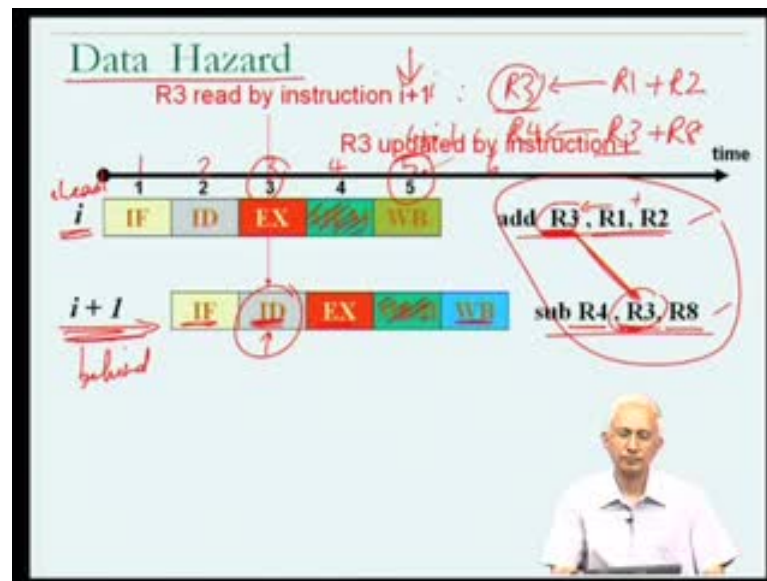
Now, what is the problem with these two instructions? If you look at the instructions carefully, you will notice that the first instruction has R 3 as its destination; so, this is instruction I. And the second instruction $i + 1$ is modifying R 4 and has R 3 as a source; in other words, the second instruction is using as a source the value that is going to be computed by the first instruction.

Now, this is the situation where there is a data relationship between these two instructions. The second instruction is dependent on the first instruction, so this might be call the data dependence or data dependency, but this is what we should note this arrow which are drawn over here, is a very notable, something which we have to pay attention to, because this is situation that this likely to lead to this hazard.

Now, how do we reason about what is happening in the pipeline as far as the relationship between these two instructions is concerned. Now, just as we understood clearly what happens in each of the steps of the pipeline as far as instruction i is concerned; let us recall what will happen as far as instruction $i + 1$ is concerned, it gets fetched in cycle 2, it reads the values in registers R 3 and R 8 in cycle 3, it does the subtraction in cycle 4 and it write-back the result into R 4 in cycle 8.

So, what is the problem? The problem is that if we look at the point in time at which R 3 is updated by instruction I, in other words, when does the value R 3 become updated due to the activity of instruction i . We notice that happens in cycle number 5, in the write-back stage.

(Refer Slide Time: 49:37)



On the other hand when does instruction i plus 1 read the value from R 3, and we know that happens in cycle number 3; in other words, instruction i plus 1 is reading the value of R 3 in cycle number 3, and instruction i is writing the desired value into R 3 only in cycle number 5.

So, what is being read by instruction i plus 1, is not the value which is being written by instruction i . And this is a hazardous situation because, clearly the value that will be read by instruction i plus 1, must be an old value the previous value that was in register R 3; and not the value which first put into register R 3 by the add instruction i .

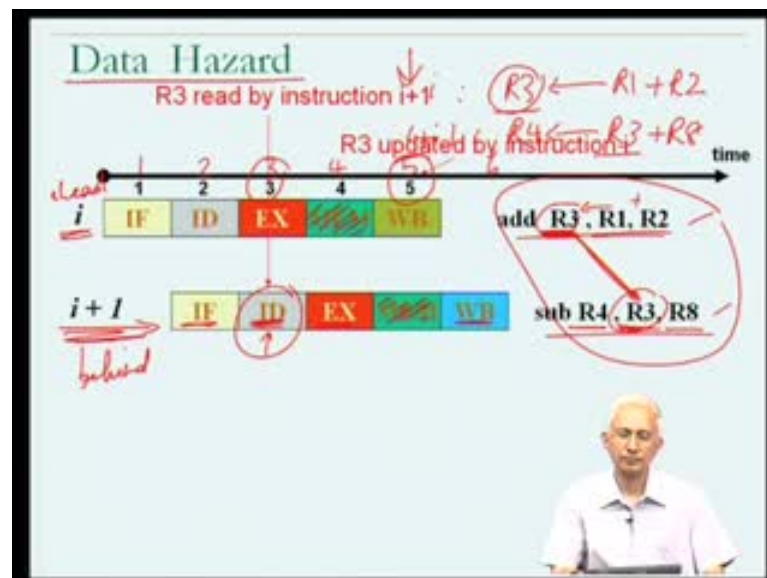
Therefore, when if the instruction i plus 1 is allow to proceed through the pipeline as per the plan shown along the timeline, then the subtract instruction will get the wrong value of register R 3 and we produce the wrong result in register R 4 and the program will not produce the correct result, in other words the program will be in error.

So, correct program execution would not result, if we use this implementation of the pipeline. The fault is not with the programmer, the programmer wrote a well-defined program in which there are two instructions, clearly indicating that the first instruction writes a result which is to be used by the second instruction. The problem is with the hardware, the hardware is not been properly designed to safely transfer the value

computed by instruction i to instruction $i + 1$ and that is why this is a hazardous situation.

Now, let just to outline this more clearly, the hazardous situation is evident from the fact that there are two instructions one which is earlier in the pipeline, and one which is, one which is ahead in the pipeline, one which is behind in the pipeline; and the instruction which is behind in the pipeline, requires a value which is yet to be computed or yet to be made available by the one which is ahead in the pipeline. And therefore, if it read the value at the point in time, that there were supposed to would result in a wrong result; that is why this is defined as a hazardous situation.

(Refer Slide Time: 51:52)



Now, we need to understand this hazard well, because very clearly this is the kind of situation that may arise in programs that we write; and we recognize that this is a legal sequence of instructions and that in writing an instruction like, instruction i followed by instruction $i + 1$. We clearly understand that we intended the value computed by instruction i to be used by instruction $i + 1$. And that therefore, there must be mechanisms designed into pipelines, through which this kind of a hazard is resolved. And we will look at this problem and multiple solutions to this problem, in the next lecture.

Thank you.