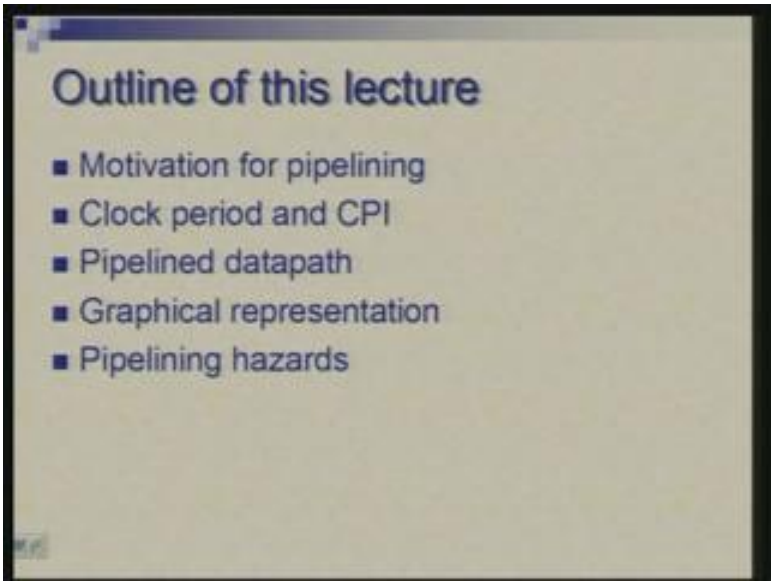
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 24
Pipelined Processor Design Basic Idea

We will start design approach to the processor called pipelined design and that is the approach followed in most modern processors. The idea of pipeline here is similar to an assembly line where you have series of operations to be performed and there is a sequence of jobs which flow through various stages or various stations. So imagine that you have a simple task of let us say doing bottling so there may be many steps involved cleaning the bottle, filling the material, capping it, putting the label and sealing and so on. So it is not that the entire assembly line resources are dedicated for one bottle and then you move to the next bottle and so on. So as the one bottle goes through one stage to the other stage to the next stage the other bottle which follow after it on a conveyor belt or in a similar mechanism. So we will talk of something similar here where sequence of instructions will flow through a series of stages of datapath and there will be many instructions at any time in flight throughout datapath.

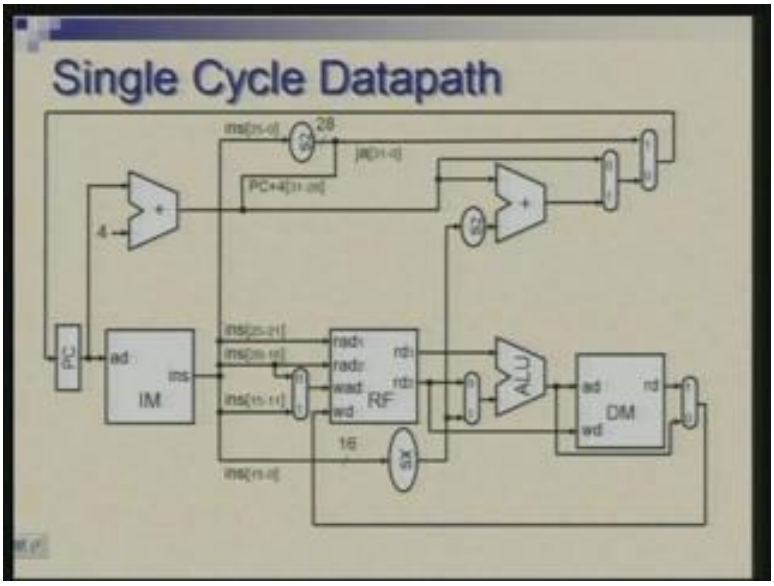
So first of all we will discuss why we want to do the pipeline why we want to go through this kind of a design what is its influence on the clock period and CPI or cycles per instruction. We will then briefly look at the datapath design for this approach how we represent it graphically for the purpose of analyzing the operation, then finally we will look at the problems we will face while trying to do so.

(Refer Slide Time: 2:46)



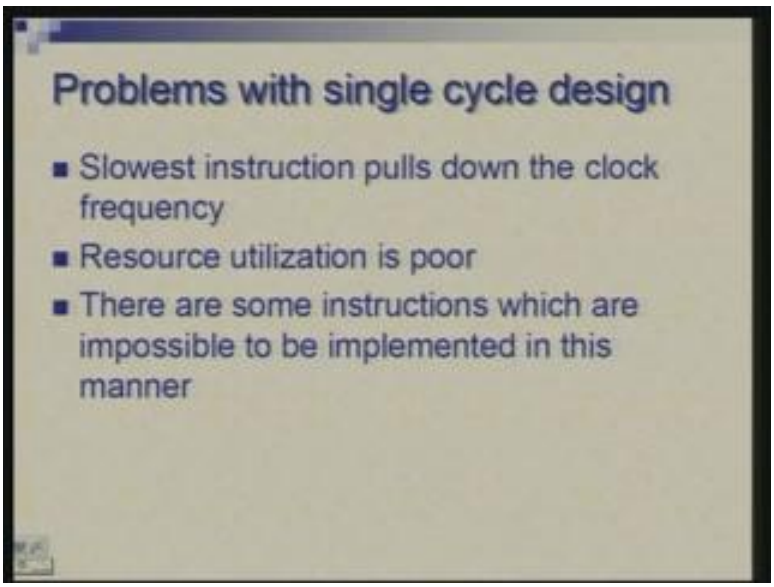
It ideally looks alright but when you look through the details there are some difficulties and that prevent us from achieving perfectly what we want to achieve.

(Refer Slide Time: 3:03)



So recall that we had started with a single cycle datapath design where once you start with an address in PC the information flows through various stages within a cycle and the instruction completes at the end of the cycle. So, for example here (Refer Slide Time: 3:28) address of the instruction in PC instruction is fetched, the operands are fetched, the operation is performed in ALU, if necessary memory is accessed, if required results are written so all that happened in the cycle; meanwhile if required new address for the next instruction will be calculated.

(Refer Slide Time: 3:52)

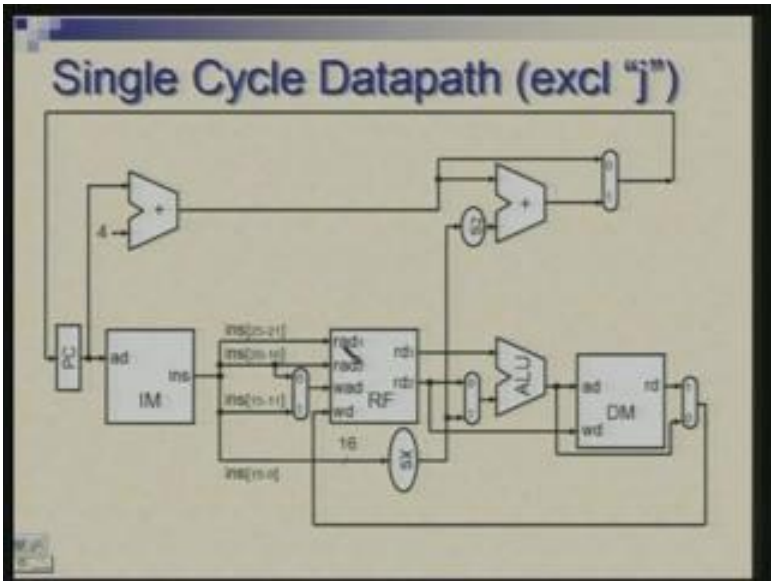


So we had a couple of problems with this. Particularly there was a difficulty in achieving fast clock rate because the slowest instruction here pulls down the clock frequency and secondly

resource utilization is poor. Although if you look at, let us say, adder, once the information has flown through it the adder is not doing anything useful but adder remains tied up; adder is engaged for the entire duration. So, all resources appear to be busy all through whether a change is occurring in them or not . So once let us say a new address comes up for register file, the operands are brought out on the output of register file but now that remains static throughout the rest of the cycle and we in a way are tying up all the resources all the time. And of course there is a fundamental limitation of this approach that there are many instructions which cannot be executed in this kind of an approach particularly when instructions involve inherently sequential operations.

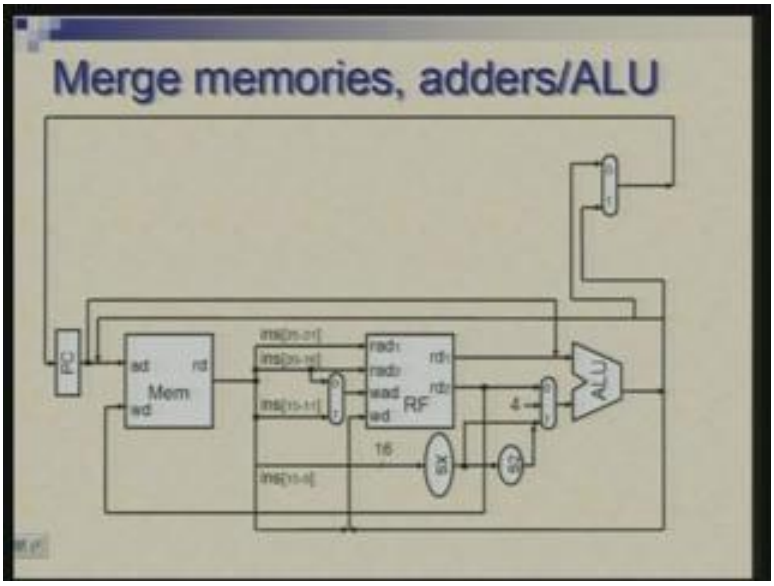
We have taken some examples of such instructions in the tutorial. So with that problem in mind we move towards a multi cycle datapath design and here I am simply drawing that same design except that just to make things look little simpler I have excluded the j instruction or the jump instruction. So some bit concatenation which is happening for jump address calculation is omitted. So here (Refer Slide Time: 5:36) what we had done was step by step we did two three things.

(Refer Slide Time: 5:39)



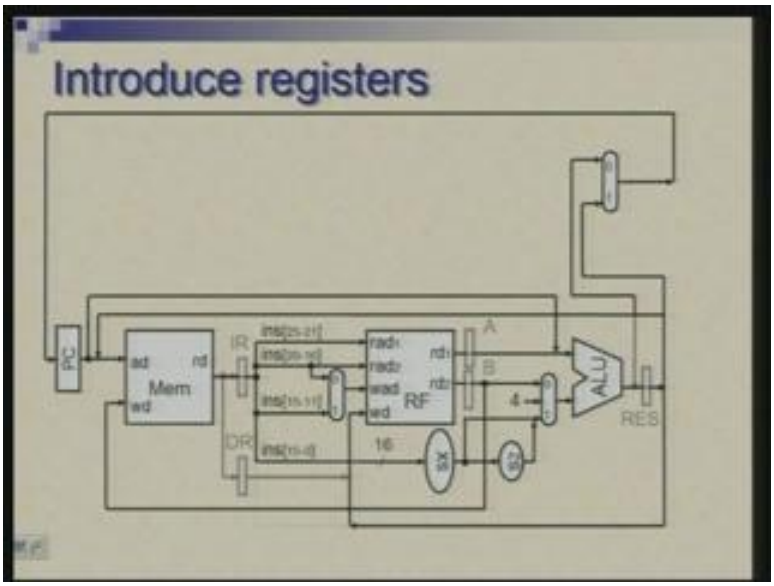
One thing was we tried to economize on the resources; wherever it is possible to merge we did that then we introduced registers to make sure that resources after one cycle become free for the next operation. So registers are actually to store the values so that the resource becomes free.

(Refer Slide Time: 6:05)

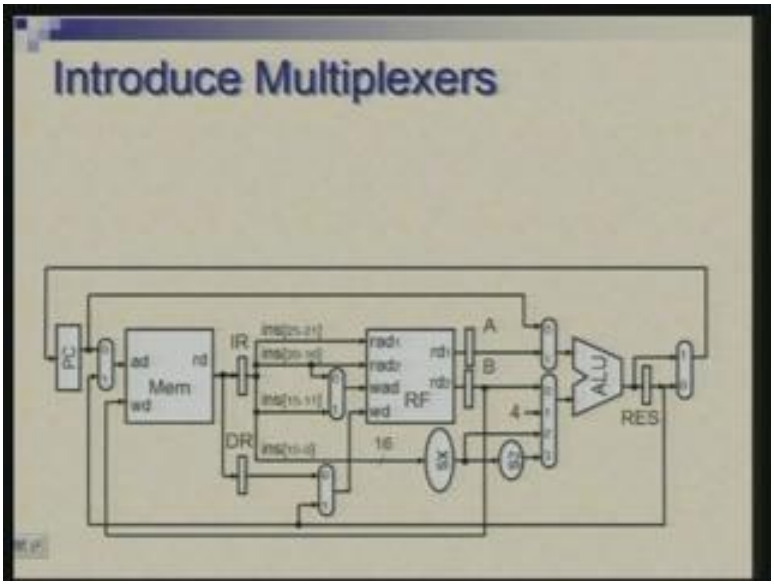


So the same datapath after merging memories and eliminating adder is shown here. We had then introduced registers so that the action of one cycle gets isolated from the next cycle and finally introduced multiplexors to complete the datapath.

(Refer Slide Time: 6:17)

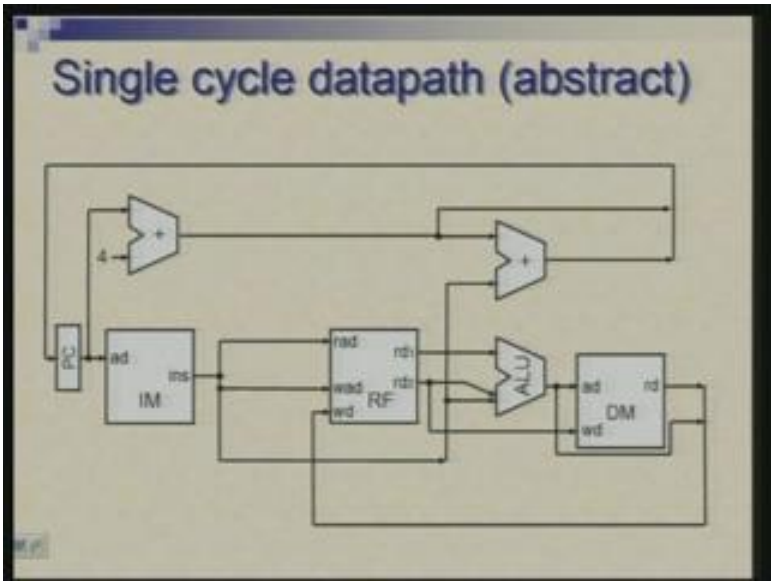


(Refer Slide Time: 6:25)



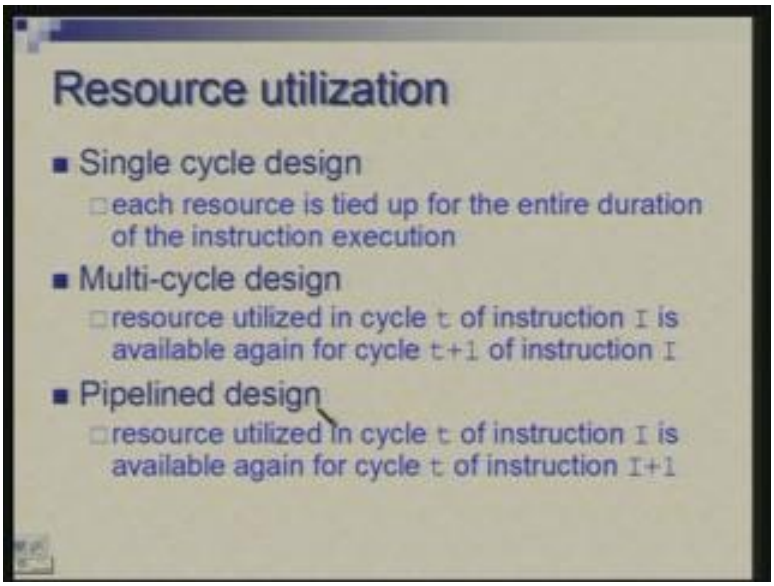
Again I have omitted jump instruction. so now as we have come from single cycle datapath to multi cycle datapath what we have got is a much faster clock, clock period needs to accommodate not the entire operation of one instruction but one specific micro operation right.

(Refer Slide Time: 6:55)



So now let us look at things little more abstractly. This is I am omitting, now I am going to omit the registers and the multiplexors and just show main key resources and the interconnections. So basically all multiplexors and registers would be gone, we have instruction memory, register file,ALU, data memory and two adders whereas in the multi cycle datapath we have a single memory register file (Refer Slide Time: 7:47) and ALU and register isolating this.

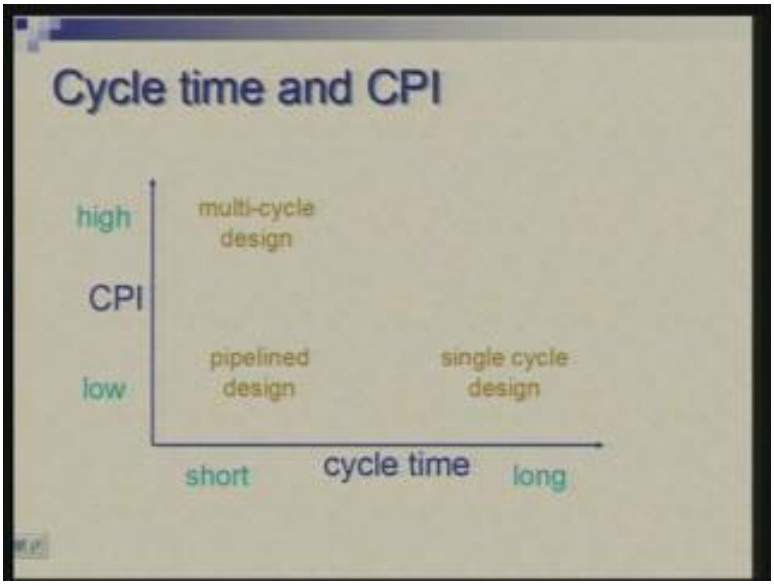
(Refer Slide Time: 7:54)



So now let us look at a more precise comparison of these two. In single cycle design each resource is tied up for the entire duration of the instruction execution whereas in multi cycle design resource utilization is better; a resource which is utilized in cycle t of instruction I is available again for cycle t plus one for the same instruction I. So a given resource can do whatever it whatever is required in different cycles for the same instruction. In pipeline design what we will try to achieve is something different that a resource which is utilized in let us say cycle t for instruction I is after that free for the cycle t for instruction I plus 1.

If addition is done in the third cycle of one instruction, that adder will do the same third cycle the addition for the next instruction. So here by t I do not mean an absolute time but I mean the second cycle of instruction or third cycle of instruction or fourth cycle of instruction. So we are trying to share resources not within an instruction but we are trying to share across the instruction. So one instruction goes past the resource and the next instruction comes in.

(Refer Slide Time: 9:34)



In terms of cycle time and CPI combination this is how we can place three designs. The first design was a single cycle design where we had a long cycle time whereas CPI was low; by definition CPI was one because we were talking of a single cycle design. On the other hand, on a multi cycle design we achieved a short cycle time right but on the other hand the CPI went high. So now if you recall how performance is determined performance is related to the product of these two quantities so you want both these to be low. You want low CPI; possibly if you can do 1 it is very good and also you want faster clock short cycle time.

So basically farther away from origin you are the worse it is and the pipeline design actually tries to get the short cycle of the multi cycle design then low CPI or CPI of 1 of single cycle design. So what it means is that ideally in every cycle we want to push in one instruction in the pipeline and in very simple case imagine that the resources are placed in a sequence one after the other and instruction will flow through these linearly.

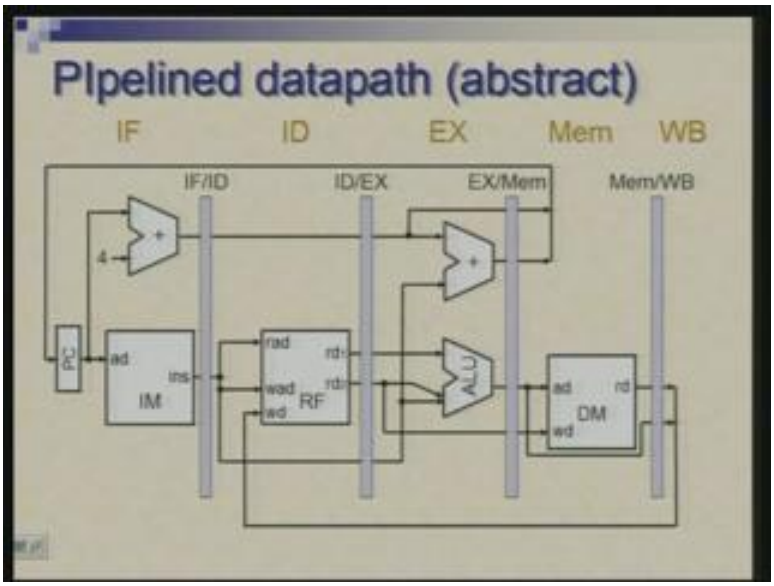
Therefore if there are no problems, there are problems which we will actually discus later but imagining that this pipeline can flow smoothly the CPI we can achieve is 1. The clock period on the other hand is still dictated by one operation and not the entire duration of the instruction. So the cycle time will be more or less similar to what you get in multi cycle design. There would be subtle differences that too may not be exact so I am showing some what a qualitative picture and not a very quantitative picture.

So one thing we must notice is that if we focus your attention on a particular instruction the instruction is still taking multiple cycles. Instruction has to go through several stages and it emerges or it completes its action only after multiple cycles but the rate at which instructions are initiated the throughput is higher so ultimately when you are executing large number of instructions it is the throughput which will matter. We are not, for a large program we are not worried about how long one instruction took individually; we are worried about the rate at which instructions get executed because that will ultimately determine the time.

So if you have of course the number of instructions is finite and at the end you have to wait for the last instruction to finish it is not sufficient to just worry about initiation of the last instruction so if let us say there are hundred instructions each instruction has to go through five cycles then the 100th instruction starts in hundredth cycle and finishes in 104th cycle. So total number of cycles will be 104 but approximately if since you are going to have not hundred but thousands and millions of instructions roughly speaking the initiation rate would determine the total time.

Now how do we design a pipelined datatpath?

(Refer Slide Time: 13:25)



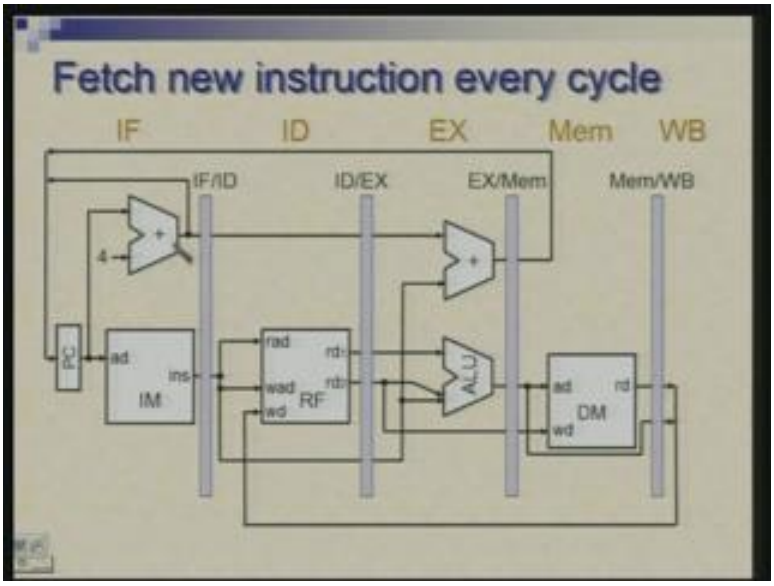
You see in this picture all that I have done is I have taken the same datapath of the single cycle design I have as I mentioned that I have I would take this (Refer Slide Time: 13:40) where I am omitting the multiplexors and some small details; just try to show broadly the flow of information. So same diagram I have taken and introduced a couple of registers. We did introduction of registers in case of multi cycle design when we went from single cycle to multi cycle design. Of course there our attempt was twofold; we were also trying to share the resources within the instruction; we are not trying to do that. By this pipelining mechanism itself the resources will get shared across instructions. So we are leaving the single cycle datapath as it is and introducing registers (Refer Slide Time: 14:27) so I have drawn these like walls vertical walls which are crossing all the forward going lines right so everything which is cut by all lines which are cut by these vertical walls are basically passing through register that means output of these gets stored, output of these get stored and is available in the next cycle in this stage. So the whole datapath is actually very clearly divided into stages; you can label these as IF stage instruction fetch stage, instruction decode or register fetch stage, execution stage, memory access stage and write back stage. So write back is actually happening into the register file but it is a stage which is afterwards so that is how I have written it.

And these registers (Refer Slide Time: 15:22) which separate the stages are labeled by the two neighboring stages. For example, IF stroke ID means the register between these two stages. So now in this most of the paths are actually forward going paths which get broken by the registers there are two backward going paths one is this the data which needs to be written back into the register file so naturally it has to go back because it is the same register file which is doing read and write and the result of this PC address calculation this also has to go back to PC. So these are the two and these do cause some complication which we will see in due course of time. So the ideal thing is that the pipeline should be so-called linear pipeline because things only move forward and do not come back because coming back in some sense is an indication of reutilization of the resources.

So, for example, if let us say if we had the same single memory (Refer Slide Time: 16:50) for instruction and data; in this stage the address will have to be routed back to this memory and then that data could be taken out further so that would definitely create some complication because by the time you have reached this stage (Refer Slide Time: 17:08) some other instructions have moved into the pipeline and they will be in conflict. I will precisely define this kind of conflict little later but at the moment we assume that pipeline is by and large a linear pipeline.

Now there is one problem here is in terms of how we are starting the next instruction in the next cycle. The next instruction depends upon the next value of PC but the way we have done is that the next address calculation is actually being calculated here (Refer Slide Time: 17:52) where you are ready to select between PC plus 4 and PC plus 4 plus offset and that is happening that is available only after.... suppose this was first cycle, second cycle, third cycle so it is only in the fourth cycle we have this value which we put back in PC. But if you do that then you cannot really run your pipeline and keep it full.

(Refer Slide Time: 18:43)

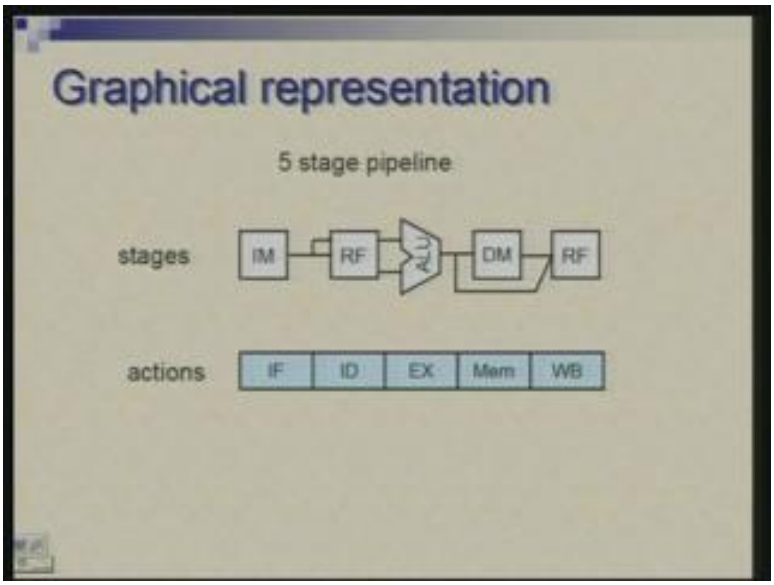


We need to quickly be ready with PC plus 4 in the next cycle itself and therefore I carry out a slight modification of the datapath. It is that the multiplexor, of course I am not showing the

multiplexor here but the multiplexoer which selects between PC plus 4 and PC plus 4 plus offset is actually placed within the first stage so that is very important and the complication here is that since there are many instructions in the pipeline there will be some instruction here and there will be another different instruction in this stage or in this stage (Refer Slide Time: 19:04). So we are trying to mix-up things. But at the moment let us ignore that we will we will come to understand the implications of that but right now our focus is on quickly calculating PC plus 4.

And for the moment let us imagine that we do not have beq let us just turn away our eyes from that and imagine that we have instructions like load store, add subtract and slt and so on and for quite some time we do not have beq instruction. As long as we keep that away we have a possibility of being ready with PC plus 4 in every cycle. So therefore this part is really now geared to pump in a new instruction every cycle.

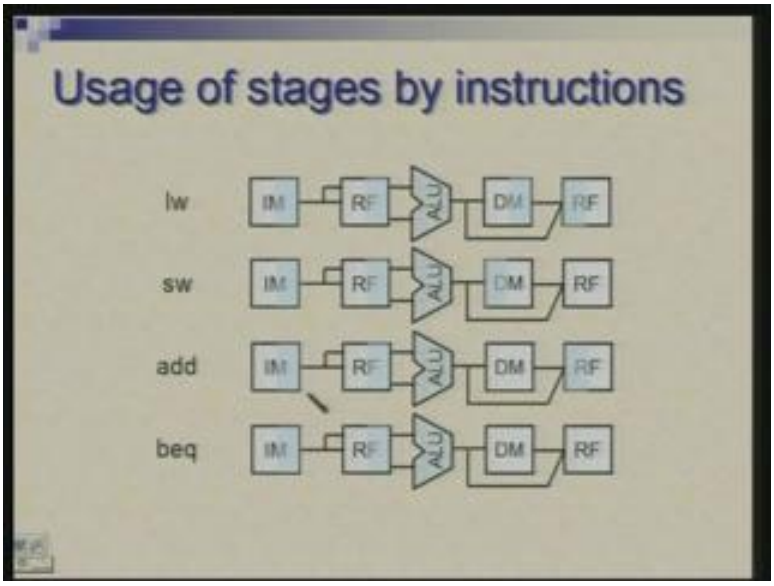
(Refer Slide Time: 20:05)



Now let us try to analyze things little bit. For that we will not carry that big diagram and I abstract it further. So you can either show very symbolically the five stages which we have; instruction memory, register file, ALU, data memory and again register file. So RF I am placing twice because there are two different functions being performed here (Refer Slide Time: 20:34) you are reading here and you are writing here so it is the same register file but logically I want to conceptually show it as a linear pipeline. Or I can show in terms of actions. The action happening here is instruction fetch, instruction decode, execute, memory access and write back (Refer Slide Time: 20:56). So now I could look at this may be try to put multiple instructions in the pipeline and see how it happens.

And there are two ways we can do it. Before I go for that we can show usage of different stages by different instructions by properly shading.

(Refer Slide Time: 21:21)

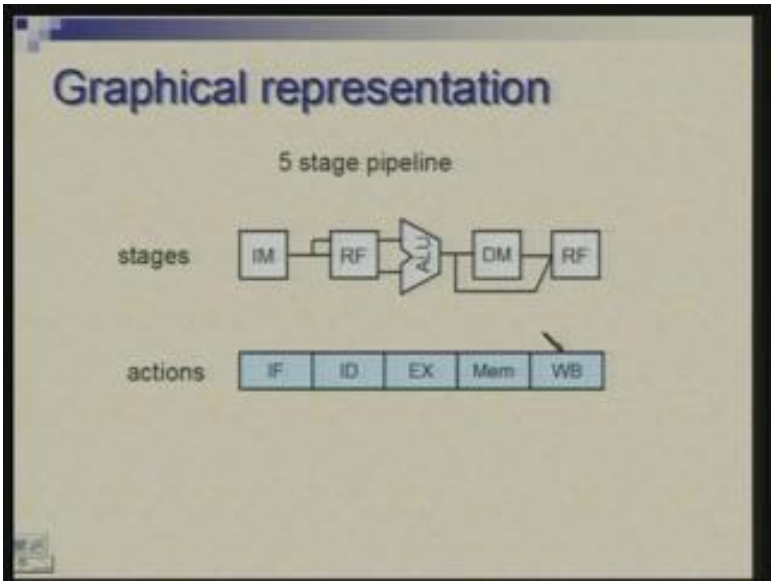


I want to just check if it is very light but are these shades visible? Yes or no. So you would notice that these boxes I have in some cases I have shaded and in some cases they are half shaded. Let us say devices like register file if I am shading the right half it actually means I am reading because you know I am shading the active part closer to the output ports and shading the left half means that I am indicating a write action so I am shading the portion close to the write port so similarly for memories. ALU of course gets fully utilized so either it is fully shaded or not shaded in this you will see all those shaded here.

IM always gets shaded in the right half, DM may get shaded in the left half or right half. So there are different instructions: here add will represent the entire class, for load let us see, IM is reading register file, utilizing ALU then reading from memory, writing into register file. For store these first three steps are same, the fourth step is that you are writing into data memory. For add instruction memory operation is skipped and there is a bypass path (Refer Slide Time: 22:52) so after that I am doing write into register file. For beq I need only first three stages.

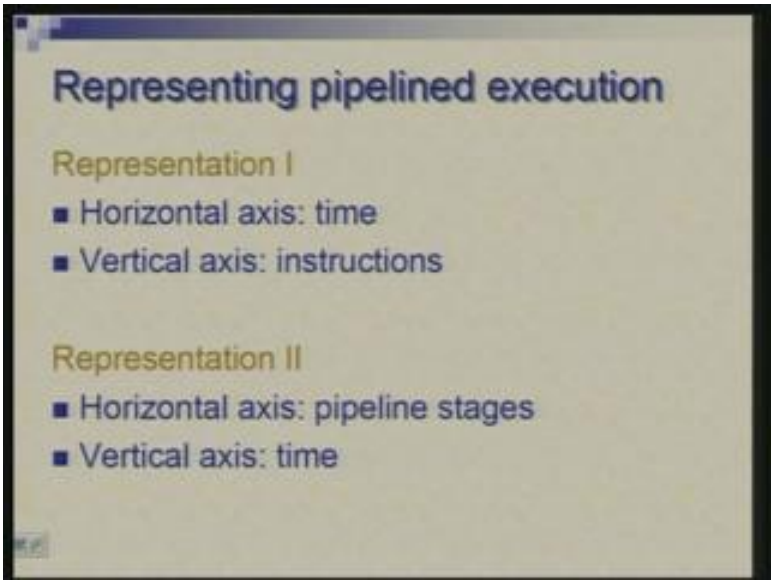
Now, also imagine that whether an instruction requires five stages or less. We will we will not worry about that part of it. So for example, add instruction will do register writing in fifth cycle only and therefore we will be wasting a cycle in between as far as add instruction is concerned. And this does not bother us because as I mentioned what is of concern to us is the throughput. So by trying to do register write in fourth cycle we might create complications for instructions before or after so we do not do that. We have sort of fixed as shown here (Refer Slide Time: 23:53) fixed what is done in which cycle and at which stage. So write back is always fifth cycle right whether you need something in between or not.

(Refer Slide Time: 24:10)



Here the deviation from this are not too many but in a realistic situation when you have many instructions the differences between the instructions may be more varied and therefore you will find that in the utilization of stages there will be more holes. But again this uniformity helps in organizing the pipeline.

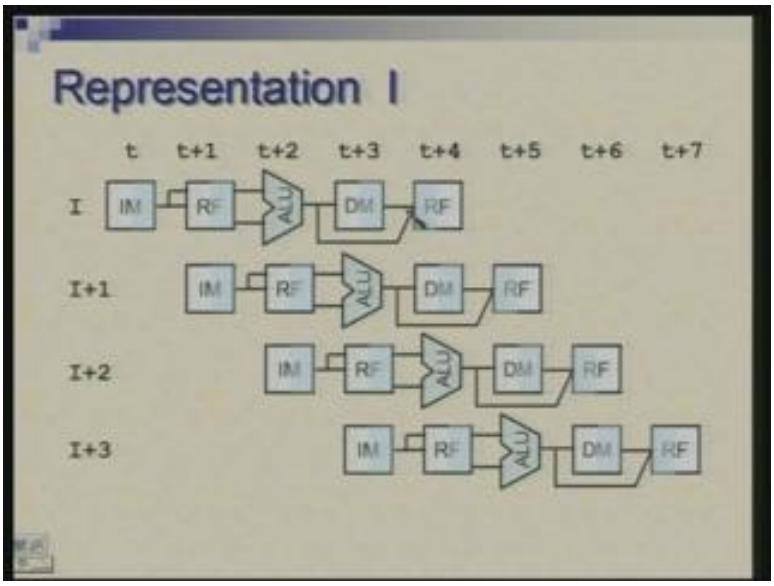
(Refer Slide Time: 24:31)



So now how do we represent execution of multiple instructions over the pipeline? There are two approaches which we can follow; one representation is that on the horizontal axis we represent time and on the vertical axis we represent different instructions; show a sequence of instructions.

Alternatively we can show pipeline stages on the horizontal axis and on the vertical axis we show time so let me illustrate both of these.

(Refer Slide Time: 25:03)

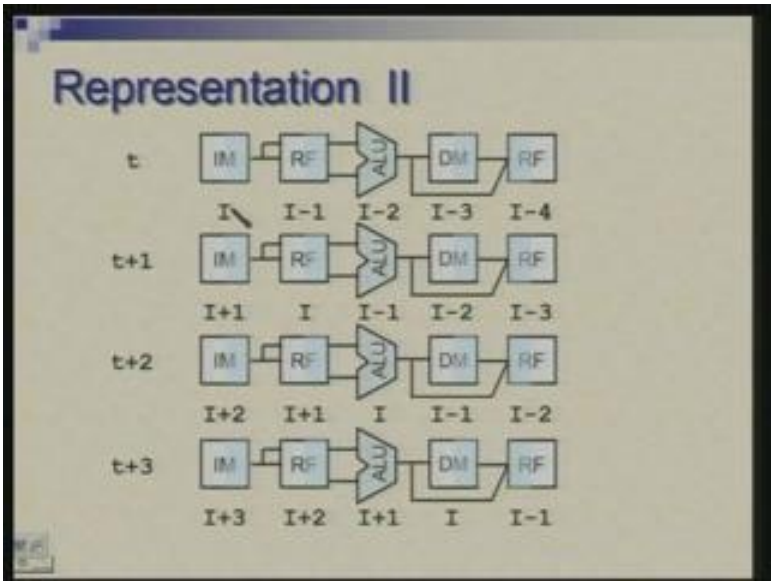


So here is the first approach. The horizontal line is the timeline now and each row would represent how an instruction is progressing. So instruction I progresses through the five stages in time... here time is in terms of clock cycles. I am not counting time in nanoseconds or microseconds it is in terms of clock cycles. So cycle t it does instruction fetch and cycle t plus 1 does decode and so on; it goes through the pipeline in the times cycles t to t plus 4.

Next instruction enters the pipeline in time t plus 1 and leaves at t plus 5 and so on. So you have instruction after instruction which are entering the pipeline and you can show multiple instructions by showing them staggered in terms of their stages like this.

So now if you want to look at snapshot at a particular time for example let us say time t plus 3 you have instruction I in DM stage, instruction I plus 1 in ALU stage, instruction I plus 2 at RF stage and instruction I plus 3 in IM stage. So the second representation which I mentioned will try to capture this.

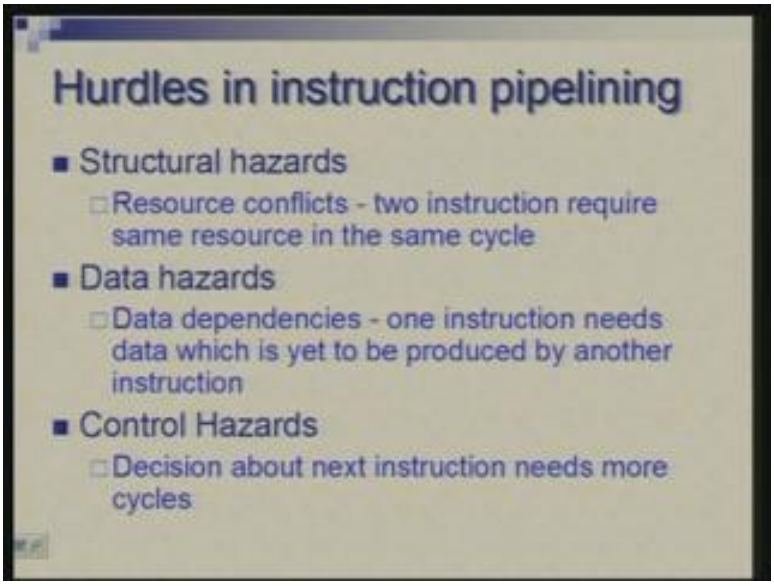
(Refer Slide Time: 26:25)



So here is the pipeline so in a sense we are keeping the pipeline stationary and we are trying to see how instructions move through those. So at time t these are the instructions filled in the pipeline in different stages (Refer Slide Time: 26:43). In the next time cycle these instructions shift one stage right and you have I plus 1 here, I here and so on. So again you can focus your attention on a particular stage and you will see that how instructions are going past this stage in different time cycles; you can focus your attention on any of these and you can see how instructions are flowing through each of these stages.

So basically whether we go for this representation (Refer Slide Time: 27:24) or this representation Refer Slide Time: 27:28) they are trying to capture same information just from two different angles and the idea in any of these is to be able to see how the instructions are placed to each other relatively because there might be some interaction which may have to take place between instructions and for that purpose you need to see several of these together.

(Refer Slide Time: 27:51)



So now if the instructions can flow the way I have shown in the last two slides everything is wonderful and you achieve the CPI of 1. But in practice this may not always happen and there are some obstructions or hurdles in that happening so these hurdles can be classified into these three hazards which are called structural hazards, data hazards and control hazards.

Structural hazards are basically resource conflicts where two instructions are trying to utilize the same resource at the same time. If one instruction is utilizing ALU in one cycle and in the same cycle another is utilizing the register file there is no conflict. But if something happens that two instructions want the same thing then there is a conflict and the pipeline gets choked.

Why would this happen?
This could happen if the pipeline is not a linear pipeline. We have shown a very ideal linear pipeline where nothing comes back the only place we are coming back in our design is that we are sending the address back to the pc but that of course does not cause a problem because it is address for the new instruction and the new instruction gets initiated by that. The other places that the data which is to be written back into the register file is actually coming back so but fortunately it does not conflict. Let us go back and see it here for example (Refer Slide Time: 29:46).

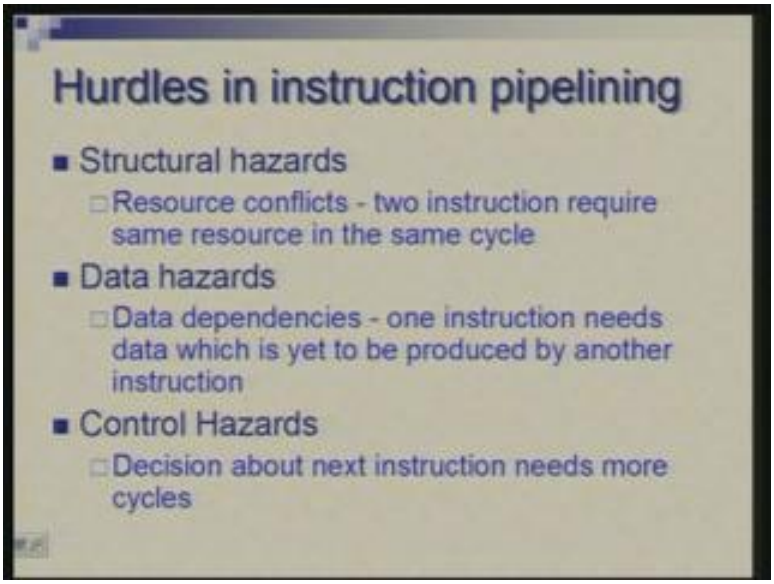
Instruction I is writing into register file in cycle t plus 4 whereas instruction I plus 3 is reading from register file in the same cycle. The kind of design we have assumed for register file we have said that it can do reading and writing simultaneously; it has two read ports which will be utilized in this example by instruction I plus 3 at the same time the write port will be accessed by instruction I. So there is no conflict because the resource has that ability to cater for two instructions at the same time right. On the other hand, the way we have chosen our memory is that it can either do read or write at the same time. So therefore trying to share DM and IM would not have worked out. For example, you would notice that in cycle t plus three I is using DM and I plus three is using IM and if this memory is not capable of doing one read here and

one read there (Refer Slide Time: 30:59) then there would have been a problem. If you do want to share and you can put a multiport memory this could work. But practically these memories are not multiport and therefore it makes sense to keep this structural hazard away by having two separate memories.

In real practice you actually have single main memory but what you have separate for instruction and data are the caches. So sometimes you may have multiple caches…. multiple level caches; the first level cache which is closer to CPU plays actually this role (Refer Slide Time: 31:46) a separate instruction and data cache there.

Another situation which can actually lead to structural hazard is that if your ALU has to do very complex operations which inherently take multiple cycles. So far we have been talking of add, subtract, compare and so on but suppose we were to extend our design to include multiply and divide we have discussed designs which are single cycle designs as well as multiple cycle designs particularly for divider it is possible to have faster divider but let us say we know only how to divide in multiple cycles. Then a divide instruction will be in this stage ALU stage for several cycles and what happens to the instruction which are following it they will get held up so the pipeline will see a structural hazard or a resource conflict either when two instructions are trying to utilize a resource because of non-linearity in the pipeline or because some stages take longer and some stages take less because of that imbalance also you may end up in one instruction still busy with the resource whereas the second instruction has reached that point and demanding the same resource.

(Refer Slide Time: 33:16)



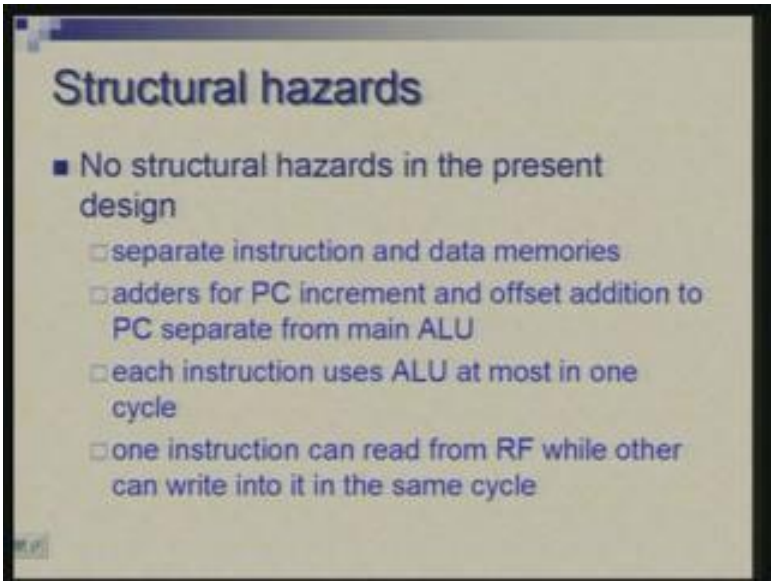
So now this is a hazard in our design; by design we have kept it out for the subset of instruction we are working we have a datapath where there is actually no structural hazard. The data hazard on the other hand is not entirely a function of the hardware but it is something inherent in the computation you are trying to do and it comes because of dependency among the instruction. So one instruction produces a result which is required by the next instruction so that is a

fundamental computation requirement and you cannot let one instruction go past another instruction or follow very closely another instruction because of this dependency. So only when the result has been produced it can be consumed by the others that is a fundamental limit and we will see what are the implications of that.

Finally control hazards are because of break in flow of control. The last category of hazard is the control hazard which come because of branch or jump instruction where we alter the flow of control.

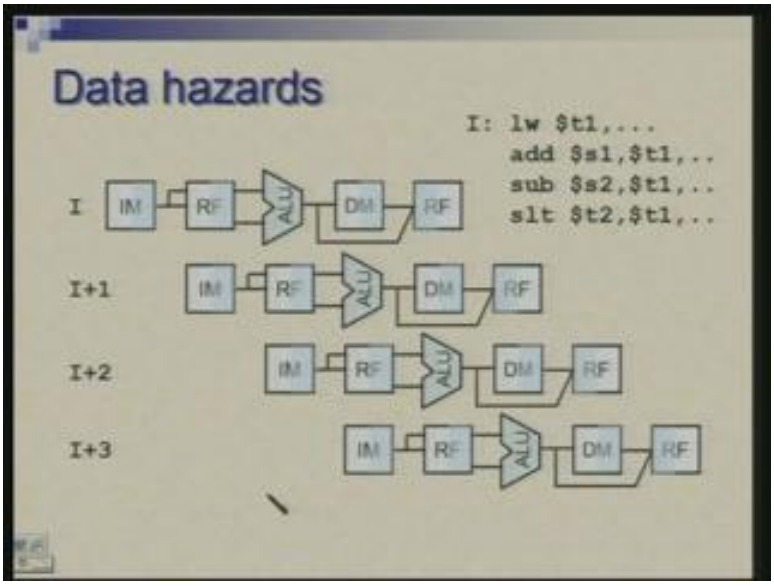
Recall that we have made an arrangement to calculate the next instruction address PC plus 4 every cycle so that we can keep the pipeline full. But these instructions mean that you have to take a decision and the decision may be taken in a later cycle say third cycle for example. So till that happens you do not know what is the next instruction to be included and that is again a fundamental difficulty in trying to keep the pipeline full.

(Refer Slide Time: 35:30)



So, elaborating further on this the structural hazard as I mentioned is because of the resource conflict. So if we do not have separate instruction data memory we will have conflict. We have retained the multiple adders which we had in the single cycle design so no resource conflicts arise. If you do not have that adder, many instructions would need to utilized ALU in multiple cycles and then we will have difficulty because the way we have designed the pipeline we want a resource to be free for the next instruction. So each instruction at the moment use ALU utmost in one cycle right they are instructions like jump which do not use but others which use ALU is not more than one cycle that is true for all the resources. And as I mentioned that ability of RF to allow read and write separately keeps the structural hazard away.

(Refer Slide Time: 36:34)



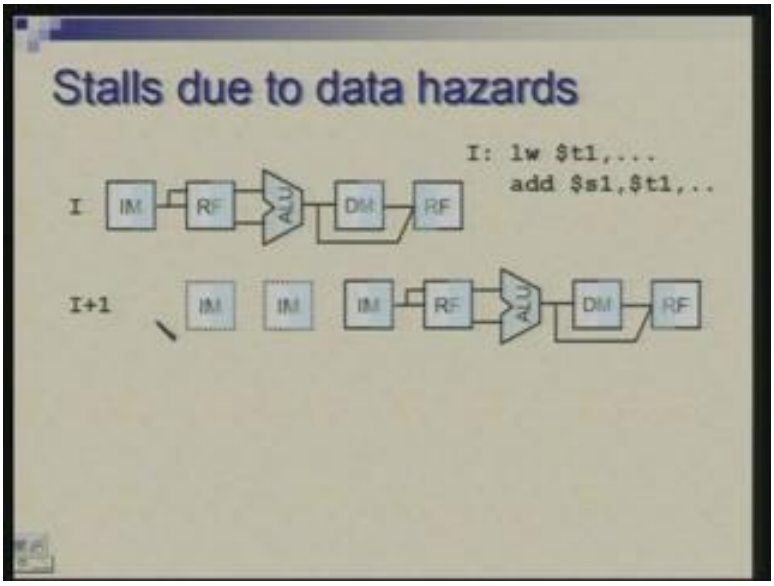
Let us look at the impact of data hazards. So suppose you have a sequence of instructions load word followed by add, subtract, slt so the key thing to be noticed here is that this instruction is putting something in register t1 which is utilized by instructions which are following it. Now let us try to understand where will this value be put in t one and where will it be utilized.

Instruction I will have this information brought from memory (Refer Slide Time; 37:15) and put back in the register file in the time cycle. So you notice that in this representation this horizontal axis is the time axis. So this represents a one particular time slot and the data which this instruction produces is available after this cycle.

Now, whereas the next instruction which is add instruction it would normally like to read the operands in this cycle in the register file and this is in time this is happening earlier so there is a problem here. We have to position we have to sort of retime these things so that the register read here of this instruction happens afterwards so it should actually happen after a gap of these two cycles it should happen somewhere here right so whole thing had has to be delayed.

If you look at this (Refer Slide Time: 38:15) because there could be multiple instructions which are dependent this value is being also read in instruction I plus 2 and I plus 2 is trying to read it here and earliest it can read is here and so on so I plus 3 also would need to be delayed. So depending upon which instruction is trying to read data from which instruction you may have to introduce delays and the impact of these delays is that your pipeline is not full always and throughput rate or the CPI becomes more than 1 or throughput becomes less than 1 instruction per cycle.

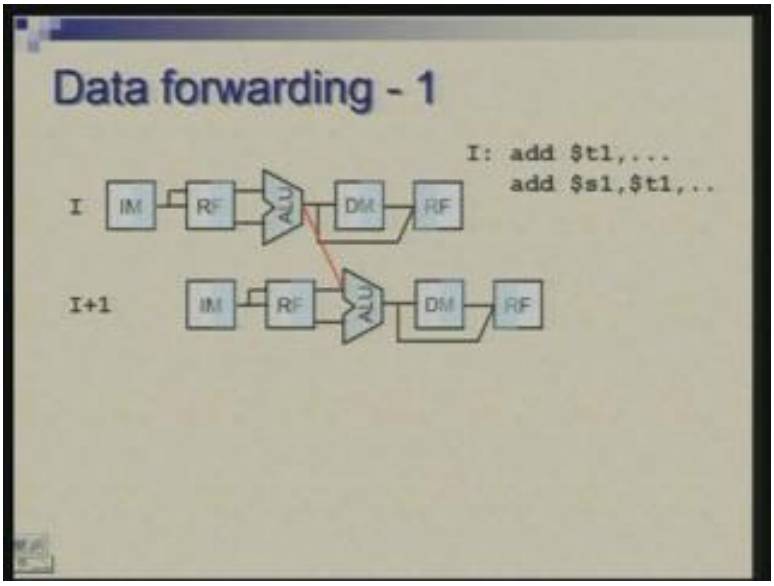
(Refer Slide Time: 39:00)



So here I show how you need to stall or delay an instruction because of this hazard. So this instruction which would have normally started here in this cycle needs to be start little late so that register reading takes place after writing has taken place. Now I have done something cleverly here that actually I am doing this reading not after this cycle but in the same cycle. This is possible if you can do reading in the later half of the cycle and writing in the earlier half of the cycle.

So assuming that the register file is inherently little faster compared to ALU and memories and the time for clock cycle is dictated by the other units... just as an example suppose register file read write time was 1 nanosecond and ALU time was 2 nanoseconds, instruction memory and data memory time was also 2 nanoseconds so then I keep a cock period of two nanoseconds and within this two nanoseconds as far as register file is concerned I can spend first half of the cycle or one nanosecond in writing and later half in reading. So that gives me little advantage here and otherwise I would have to delay it by three cycles and I am now managing by delay of just two cycles. So if you do not do that then there will be more delay. Now whether we can do it or not actually depends upon the timing. So the example of timing I chose it is possible it may not always be possible so one has to know that in general the delay would depend upon where you can schedule write and where you can schedule read.

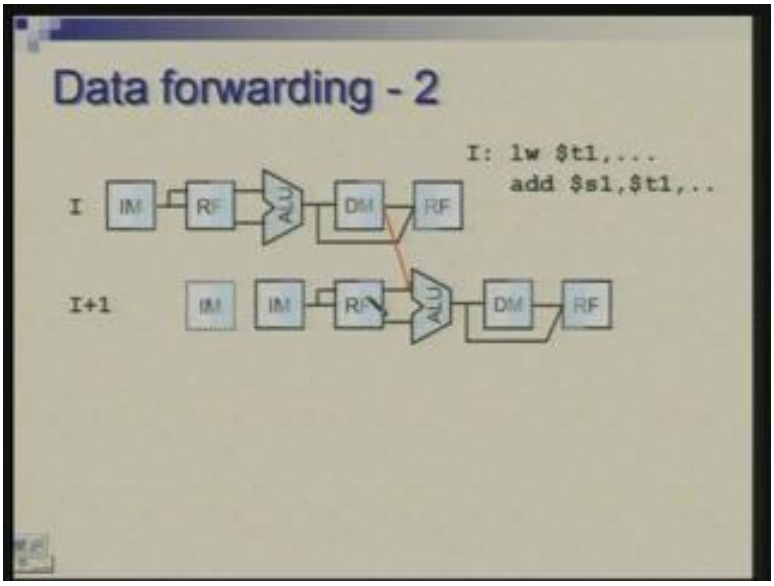
(Refer Slide Time: 41:07)



So stalling is the simplest response to data hazard but it is possible to do something better. What you do is data forwarding where we try to make the data go from one instruction to other instruction as soon as it can. So, instead of letting the data first be written into register file and then read from register file we can create extra paths in the datapath that the result of first instruction for example which are available after ALU cycle and are required by ALU cycle of the next instruction are somehow passed directly. So that means here we are seeing forwarding in the logical sense in terms of instruction to instruction but in terms of datapath if you notice what it means is that the output of ALU is also being fed back to the input of ALU it means that the we would need some multiplexors to carry this additional path. We will get to those details a little later.

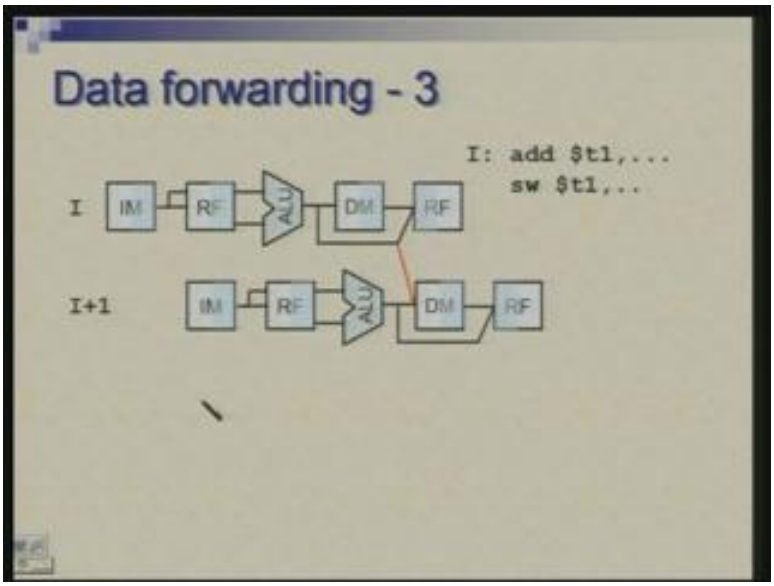
I would like you to understand the basic idea that if we let this information be forwarded in this manner (Refer Slide Time: 42:36) then you notice that we do not require to stall the instruction. The action of writing it into register file will still take place because may be some instructions down below also need that so they are not immediately reading; the instruction which is following and this would require some analysis by the hardware. The controller will basically have to see if there are two consecutive instructions of this kind (Refer Slide Time: 43:00) where the destination of one is the source of other then we simply activate that path. There are many other possibilities and in the subsequent slide let me quickly show the other possibilities.

(Refer Slide Time: 43:18)



If the first instruction is load and the next instruction is let us say add load instruction has the required value available only after memory cycle. So here we are forwarding from this memory cycle from data memory output to ALU input and here you need to introduce one delay but it is still better than two delays. Here we have not eliminated the delays but we have reduced the delays.

(Refer Slide Time: 43:49)

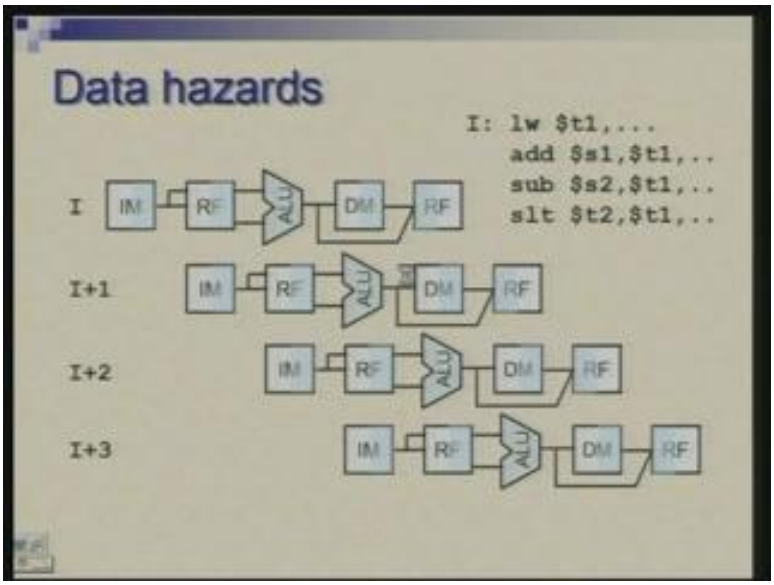


This is another possibility now here that next instruction is the store instruction. Recall that store instruction has two different uses of two registers: one register it uses for address calculation in third cycle and another register it requires for transferring data to memory in the fourth cycle. In

the previous two examples in this diagram (Refer Slide Time: 44:18) and in this diagram the second instruction is an instruction which requires data in the third cycle. So here I am looking at instruction which required data in the fourth cycle. So it is this register sw will have another register here as part of the address specification so that gets utilized in the third cycle but this (Refer Slide Time: 44:48) gets utilized in the fourth cycle and we need that data here so whether it is add instruction we can forward from the output of this stage.

Basically it is the output of ALU which will simply just pass through this stage without any modification will be available at the end of this stage and then passed on to the input of the data memory right and if it is load instruction which is the first instruction then of course from output of data memory you can pass into input of data memory. So let me approximately indicate in datapath where these paths are actually.

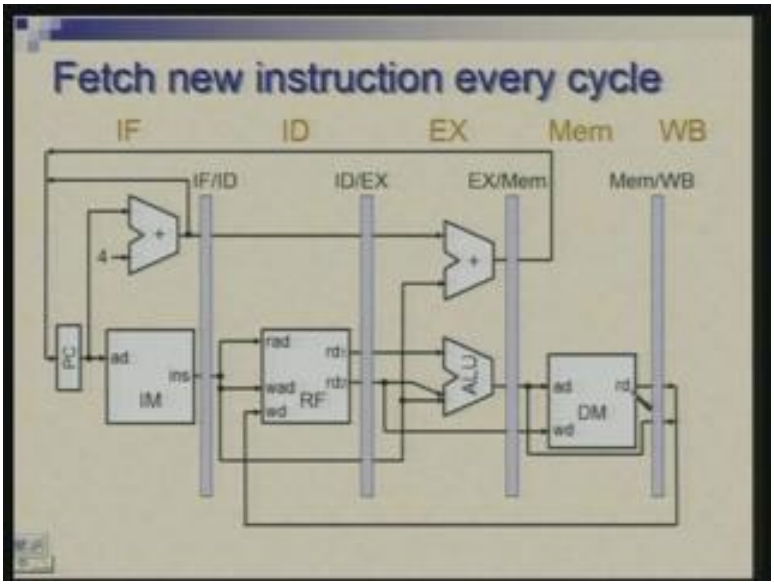
(Refer Slide Time: 45:39)



So we talked about paths going from ALU output to ALU input. So it will not be strictly speaking ALU output immediately but after the register. After this stage the data is available at this point (Refer Slide Time: 45:52) and that is fed back and brought in through multiplexors to this input or that input of the ALU. Or it could be output of memory which is picked up from here and brought to the input of ALU.

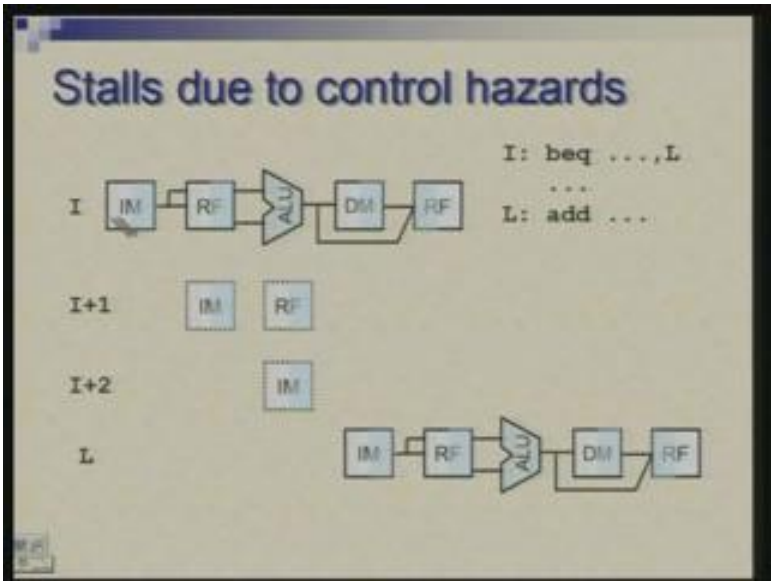
For store instruction we need to bring it here either from this point or from this point. For an add instruction let us say there is an add instruction followed by store instruction. So it is this data which was output of ALU but which simply idled here and came here and this will be brought back to this through a multiplexor. So we will see those details little more deeply later on but the basic idea is that we incur these additional expenditure of more paths and more multiplexors and as a result we can possibly reduce some delays and in some case also eliminate.

(Refer Slide Time: 46:35)



What happens due to branch instruction now? So as I mentioned earlier that when we talked about initiating every cycle one second we said let us keep branch away. Now let us face this consequence of having branch instruction.

(Refer Slide Time: 47:20)



Suppose this instruction I is a branch instruction and here if some condition is true we are branching into L which is not a consecutive instruction it is somewhere else. So after this instruction there will be a tendency to start I plus 1 or I plus 2 but the actual address is only known after this stage (Refer Slide Time: 47:52) after you have done the comparison and you have also calculated the next address. So, if the branch was to be taken it is in this cycle that you

can initiate instruction L. Now you have two choices that either the moment you notice branch instruction you freeze the pipeline, don't do anything allow just that instruction to go up to a stage where decision is known and then get the next instruction and proceed. So in the process you would have lost a few cycles but that cannot be avoided.

On the other hand, you might say that ok I will wake up only when I will know but meanwhile let it continue with the instruction in a sequence. So the hardware will pick up the next instruction because PC plus 4 is already there started, in the next cycle take I plus 2 instruction and push that in the pipeline and here you realize that probably you have gone wrong and then you abandon this; flush it out of the pipeline and start something which was actually required.

So here here you are doing something in anticipation. If condition turns out to be false then you have actually utilized this cycle. If condition turns out to be true then you did something in good faith but now you have to abandon it so it does not matter and there is no way you can start earlier than this. So this is one particular one way. So just to revise one was to not do anything, wait till decision is done and then start the right instruction. So the choice is between I plus 1 and L. So even if it is I plus 1 is to be done you start it here (Refer Slide Time: 50:04).

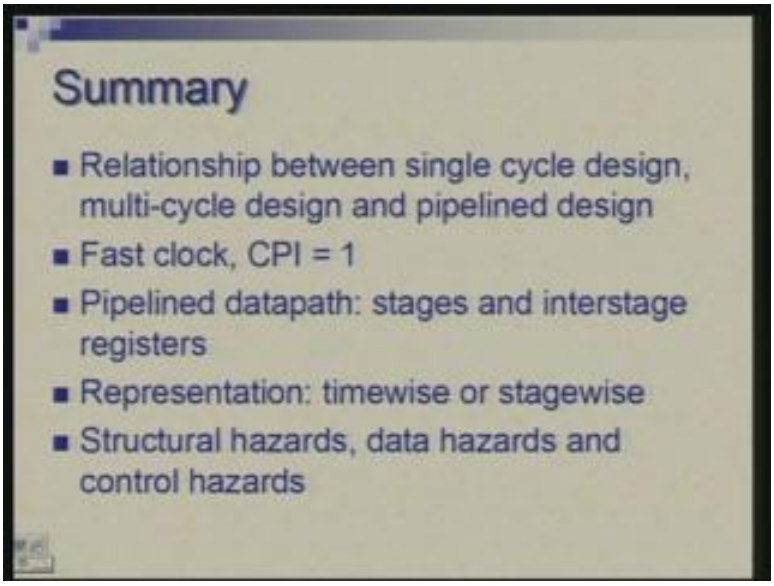
The other approach says that start something in anticipation if it is I plus 1 you have gained something if it is L you have not lost anything. You have lost something which you have to be lost in any case.

In fact this can be made little more clever that somehow here you try to guess the outcome of the branch; this is called branch prediction and you try to do since you are doing something in anticipation why not make little extra effort and try to predict whether I plus one is the most likely consequence or L. So you somehow do that prediction and try to start one of the two.

Again there is a possibility that you may since after all it is a prediction unless the numbers get compared and you know finally whether it is true or false you may not have made the right decision. In case you have made the right decision you have gained; in case you have not you have lost something which you would have probably lost even otherwise. So branch prediction I have stated in a very simple term but there are many issues involved many ways of doing it can be done statically done dynamically and you know to do it dynamically it would require lot of hardware support and so on.

We will probably look at that little more in detail. At the moment what we notice is that branch potentially causes these stalls in the pipeline (Refer Slide Time: 51:49).

(Refer Slide Time: 51:52)



So I have discussed the basic ideas of the pipeline design. We began by looking at the relationship between a single cycle design and then multi cycle design and then by bringing a pipeline into the same framework and seeing how they compare in terms of clock period and CPI. We have seen that pipelining approach gives you fast clock similar to multi cycle design and low CPI similar to single cycle design.

Pipeline datapath is obtained by introducing registers in between the stages in the forward paths. For analysis purpose we had a symbolic representation which tries to show various instructions time-wise or stage-wise. And finally we talked about three different types of hazards: structural hazards, data hazards and control hazards. We have seen that in the simplified case we are talking of it has been possible to remove structural hazards altogether; data hazard also we can cut down to quite some extent by using forwarding. The control hazards are most nasty among these. We can do something about these but not get rid of them fully. So in subsequent lectures we will see more details about the design, the datapath aspects and the control, thank you