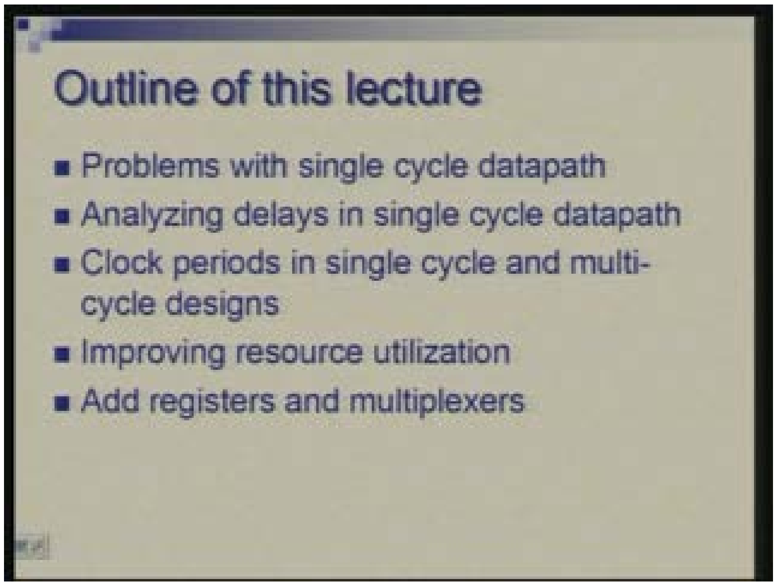
Computer Architecture

Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 20
Processor Design - Multi Cycle Approach

We have discussed a very simple form of processor design called single cycle design and in the last lecture we ended by making some observation about such a design. These observations were about its performance and inability does certain kind of instruction. So today we will introduce another type of approach called multi cycle design which tries to overcome these problems.

(Refer Slide Time: 02:21 min)

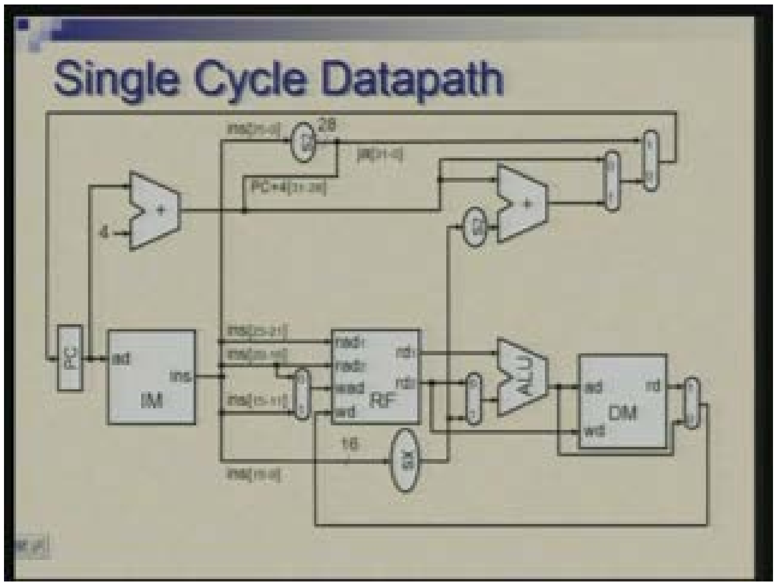


Outline of this lecture

- Problems with single cycle datapath
- Analyzing delays in single cycle datapath
- Clock periods in single cycle and multi-cycle designs
- Improving resource utilization
- Add registers and multiplexers

This is the overall lecture plan and today we will start by repeating those problems re-observing those problems which are related to single cycle data path. We will look at how, we analyze the delays and what for the difficulty which is noticed. We will see how clock period can be improved; how clock could be speeded up by using multi cycle design. We will also look at this as a way of improving the resource utilization. The resources are the main hardware components we have even in our data path and what basically we will do is we will try to share these components such as memories and ALU and to facilitate that we would need to introduced some more components but often lower costs generally these are registers and multiplexers.

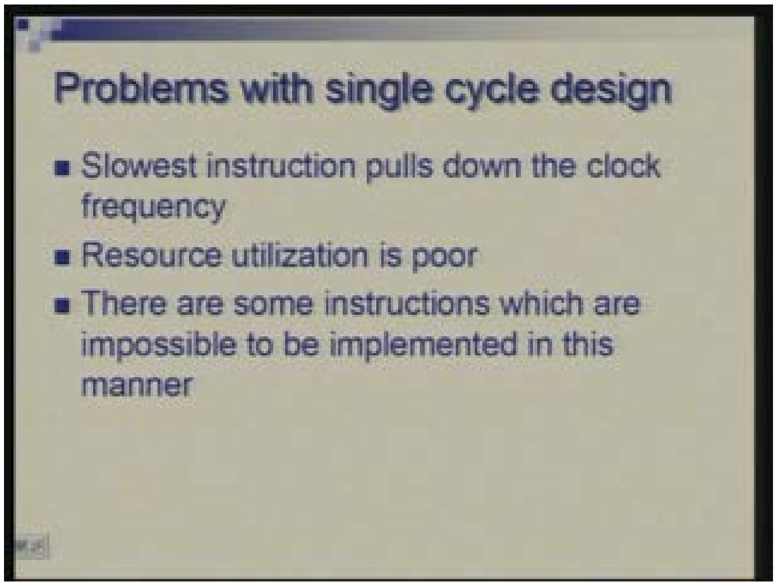
(Refer Slide Time: 02:22 min)


Single Cycle Datapath

This is the data path which was designed in the previous lectures where every instruction is completed in a single cycle. So all activities begin with a new address in PC and the cycle ends by updating the PC value as well as updating the state of the whole processor in register file and memory. So these are the three problems which we have noticed that the slowest instruction pulls down the clock frequency; if there is a wide disparity in the timings of various instructions then the whole set of instructions will be running as slow as an instruction so that is the difficulty with such a design approach.

And secondly, resource utilization is poor. Apart from having a full fledge ALU we had to use two adders and we had to keep the instruction memory and data memory separate because in that type of design it is not possible to work with single memory from where you fetch an instruction and then later on fetch data also so that we will not work out.

(Refer Slide Time: 02:50 min)

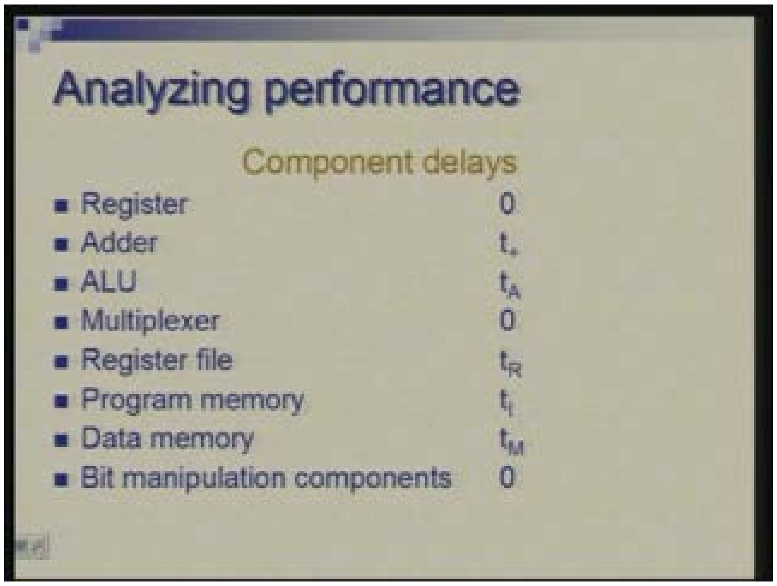


Problems with single cycle design

- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

Also, I hinted upon some kind of instruction which cannot be realized by this type of approach. So we will focus today on first two issues. Just to recall that we had analyzed the performance by taking the delays of individual components some simple ones were assigned zero delay and others had some significant delay denoted by t plus t A and so on these symbols are denoting the delays of individual components.

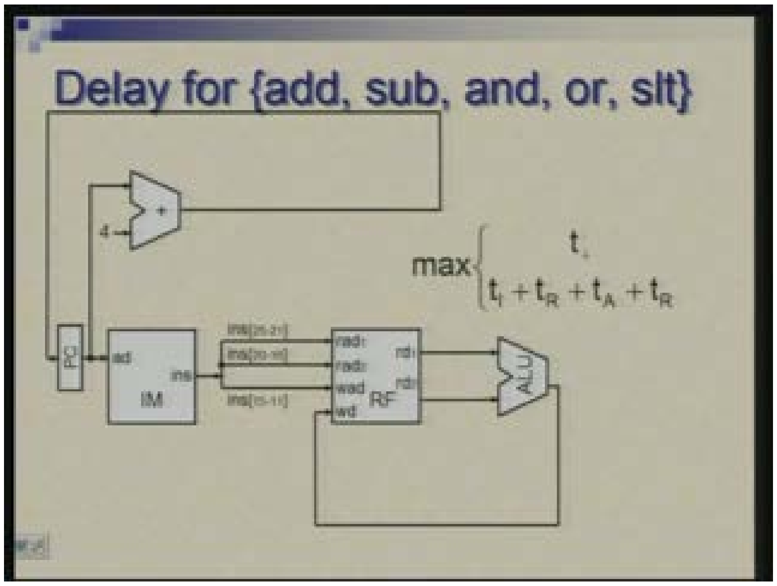
(Refer Slide Time: 03:50 min)



Analyzing performance

	Component delays
Register	0
Adder	t_+
ALU	t_A
Multiplexer	0
Register file	t_R
Program memory	t_I
Data memory	t_M
Bit manipulation components	0

(Refer Slide Time: 04:03 min)

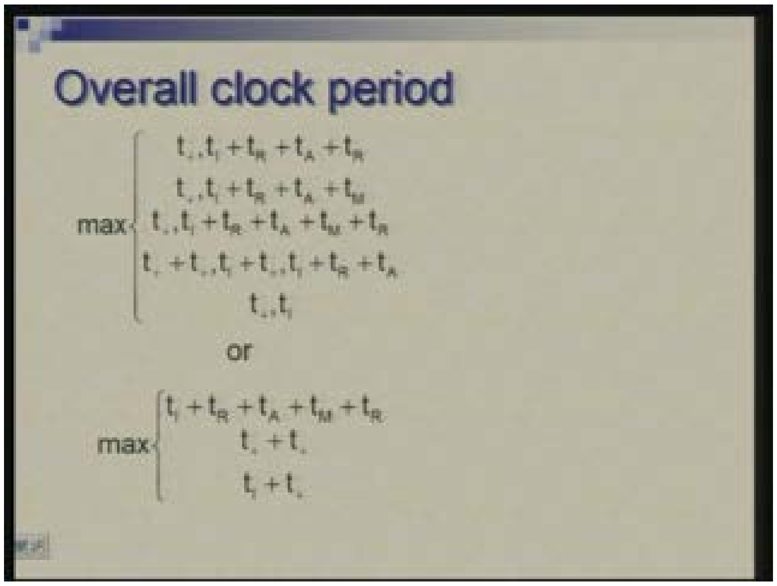


Delay for {add, sub, and, or, slt}

$$\max\begin{cases} t_{+} \\ t_I + t_R + t_A + t_R \end{cases}$$

And for each instruction or a group of instruction we enumerated various paths from storage element to storage element through which the data has to flow in the instruction and identified the possible expressions which would decide the overall clock period.

(Refer Slide Time: 04:22 min)



Overall clock period

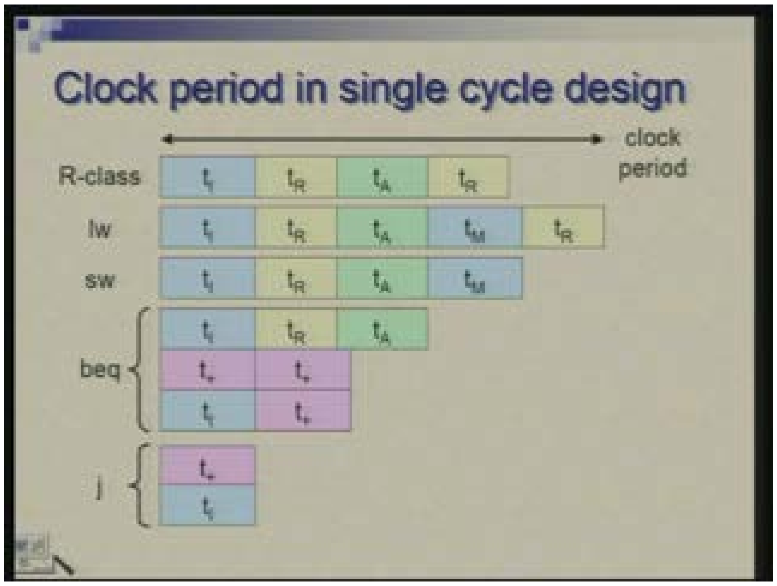
$$\max\begin{cases} t_{+}, t_I + t_R + t_A + t_R \\ t_{+}, t_I + t_R + t_A + t_M \\ t_{+}, t_I + t_R + t_A + t_M + t_R \\ t_{+} + t_{+}, t_I + t_{+}, t_I + t_R + t_A \\ t_{+}, t_I \end{cases}$$

or

$$\max\begin{cases} t_I + t_R + t_A + t_M + t_R \\ t_{+} + t_{+} \\ t_I + t_{+} \end{cases}$$

So this was the final expression we had. The one at the bottom is in a simplified form and it could be noticed that the largest of these sub-expressions will actually decide the clock period. To illustrate this further let us look at some values. Again I am not putting these values in numeric sense but trying to depict it pictorially; so the horizontal x here is a time axis and for each instruction I am trying to show the delays which are involved in the flow of information along the data path.

(Refer Slide Time: 5:02)

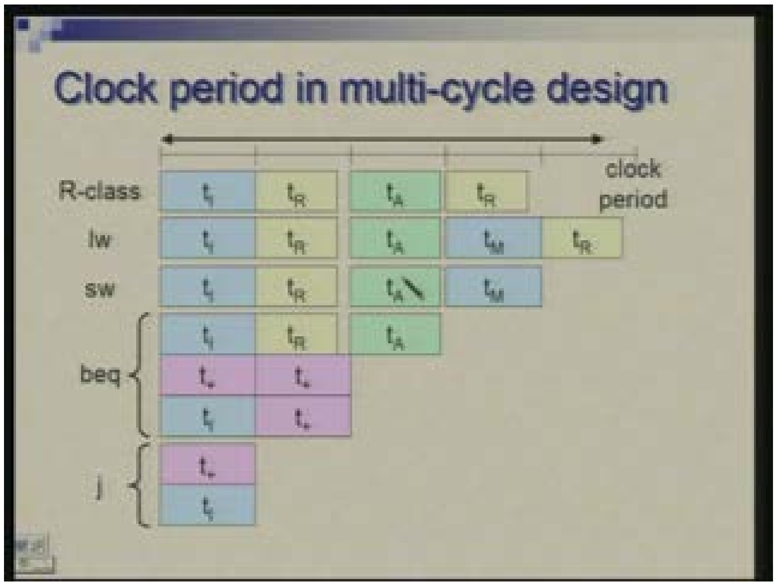


For R class instruction there is a delay involved in fetching the instruction, fetching the operands from the register file then doing the arithmetic operation and finally storing the result in register file and so on for each instruction I have indicated the possible path which may dictate the clock period. So I have taken nearly equal values for all these terms except that there are slight differences; t i and t M I have taken same and in fact t plus is also same; t A is slightly less than that and t R is even little less than that.

So, from first three cases I have dropped the t plus term because that is invariably that is less than all these and we do not really need to look at that. So now you can very clearly see that lw with this kind of values lw will dominate and dictate the clock periods. So clock period would be from this point to this point (Refer Slide Time: 6:09); remember that horizontal axis is the time axis I am showing along the time line. but once clock period is fixed all instructions are taking same time and you could see so much of dead time in other instructions in R class, sw, in beq and most in j.

Now when we introduce multi cycle design basically what we try to do is divide execution of instruction into multiple cycles. So we need to decide what gets done in first cycle, what gets done in second cycle and so on and there are lots of choices. So we will take one simple choice here which tries to do one major action in one clock cycle. This is only one of the possibilities and what it means is that now looking at each individual times we take max of these and take that as a clock period. So in this case t i, t M or t plus all are equivalent and the largest among all so that decides the clock period. The R class instruction now gets done in four clocks: lw in 5, sw in 4, beq in 3 and j in 1 so there are still small dead times because of inequality because all these times are not exactly equal there is some differences. But you would notice that the overall improvement in performance will be there because the overall wastage in time is much smaller.

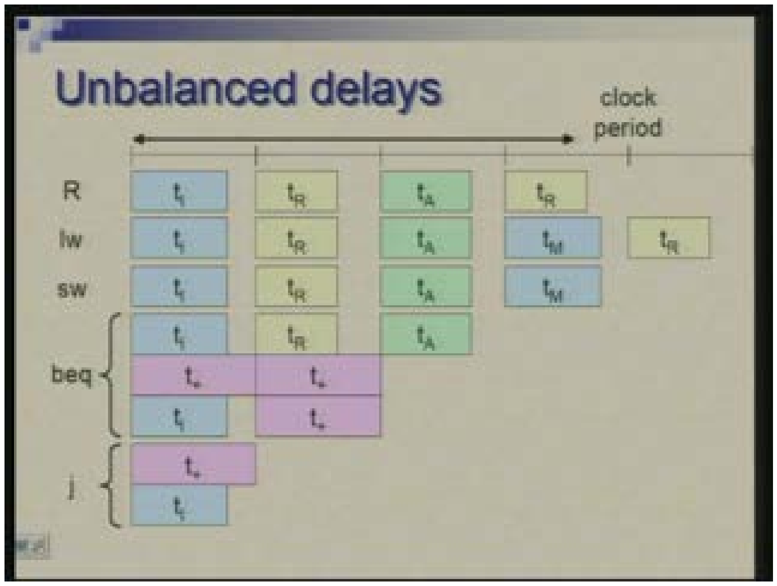
(Refer Slide Time: 7:47)



One more thing you must notice is that the total time lw takes now is more than what it was taking earlier. Earlier it was taking one clock which was from here to here, now it is taking five clocks which goes from here to here (Refer Slide Time: 8:04). So because there is some wastage of time here also but on the whole everything put together this approach would still give you better performance or save time.

Now things look quite here quite good here because these different time parameters are nearly balanced. I deliberately took these values which are only slightly differing from each other and therefore the wastage you see is very little. But suppose there was a vast disparity in this time we will be again having a problem situation. For example, suppose for some reason the adders which we had for address calculation for branch and jump instruction sorry not jump branch instruction and doing PC plus 4 they were pretty slow.

(Refer Slide Time: 8:58)


Unbalanced delays

Let us say t plus becomes the bottleneck and this will hold up the clock. So now clock will get dictated by this time and we will still follow the same approach; R instruction takes four cycles, lw five cycles, sw four, beq three and j single cycle. But now notice that the clock period is larger and it has to be sufficient to accommodate each of these any of these individual activities. So t plus is dominating and you would notice that now even these instructions although they are taking four cycles but they are taking longer than what they were taking in the single cycle approach.

Of course lw was in any case taking longer than a single cycle, sw is also taking longer than single cycle and in beq there is a little bit of saving there is of course still significant saving in jump instruction
but there are very few jumps anyway in the whole program. So if you add the cycle per instruction CPI of all these now R has CPI f 4 has 5 and so on we know the CPI of individual instruction or class of instruction and we have seen how we can calculate every CPI depending upon the frequency of occurrence of certain kind of instruction in a program you can find an average. So, the total time a program would take could in fact here be larger than what it would take in a single cycle case.

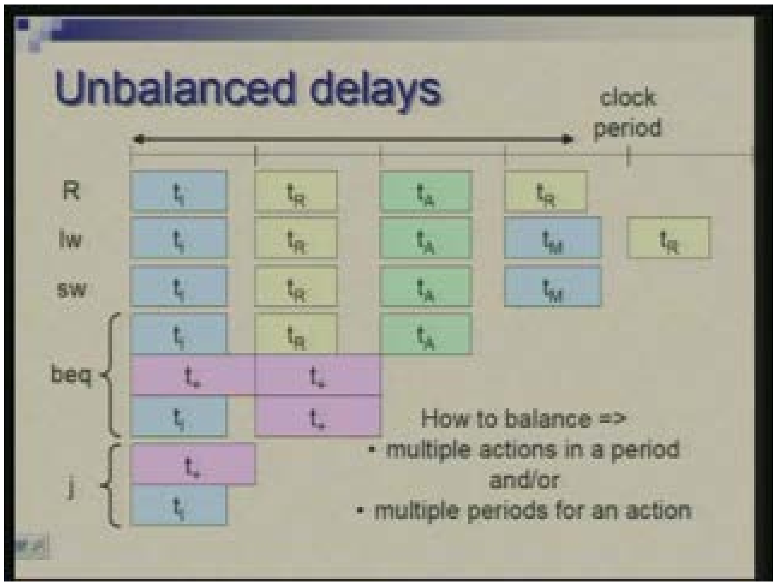
If there is an imbalance of this nature then this approach the way we have implemented multiple cycle design could be counterproductive. So what can be done in such a case is we have to do something so that there is a balance. What we are doing in a clock cycle is uniformly true uniformly same that means we identify the activity to be performed within a clock cycle which is generally balanced. it should not happen that, for example, here in this clock cycle we are doing little very little and wasting lot of time (Refer Slide Time: 11:22) whereas this cycle here is packed that is unbalanced.

How to balance?

Possibilities are that you can have multiple actions in a period. If there are two let us say two actions which are taking very little time you can do two in a clock. Earlier we were doing a everything in a single clock so that kind of idea could still be retained. We need not say that one action in one clock cycle although that is the simplest thing but this may not be always very beneficial.

Alternatively or in addition to this you could have multiple periods for an action. For example, it is not necessary that t plus if it is slow has to be done this action has to be done within a single clock. You might do everything else in a single clock but may be reserve two clocks for this so you might still have an overall better performance.

(Refer Slide Time: 12:34)



Unbalanced delays

How to balance =>

- multiple actions in a period

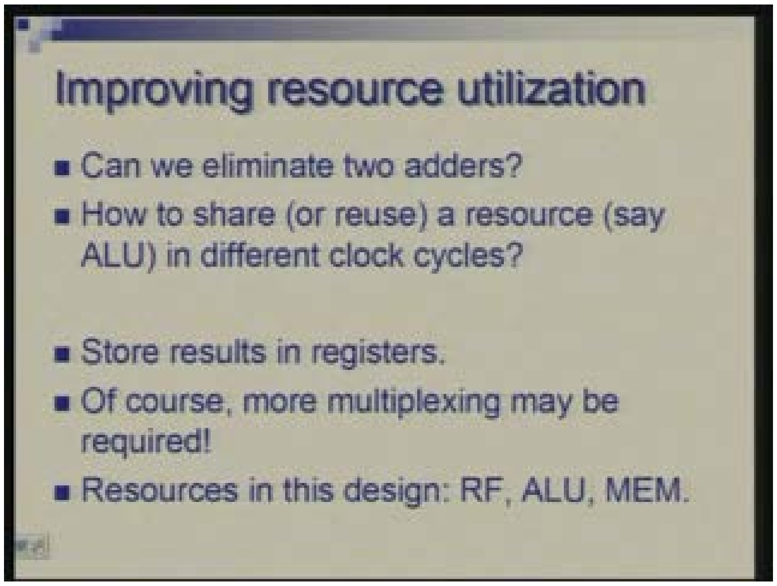
and/or

- multiple periods for an action

Now, once you bring this into picture the number of possibilities becomes very large and it is not a very straightforward solution but the point here is that one could find a suitable clock period so that the dead times or the wastage of time which is due to quantization of time by clock is minimized. So we will not going into further details of that. We will just resume that we have identified the major actions for each instruction and each can be put in one clock cycle. So we will follow that approach keeping at the back of our mind that there is a problem this is the direction in which we need to look into.

The second issue was improving the resource utilization.

(Refer Slide Time: 13:19)



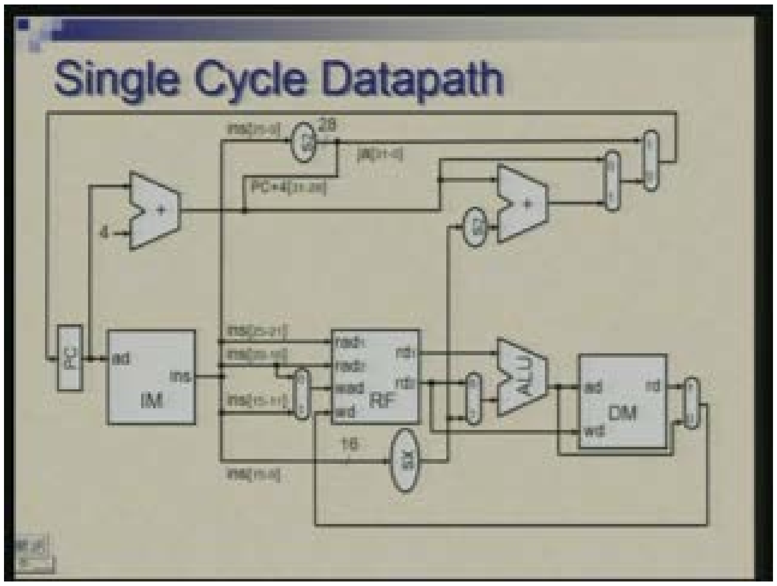
Improving resource utilization

- Can we eliminate two adders?
- How to share (or reuse) a resource (say ALU) in different clock cycles?
- Store results in registers.
- Of course, more multiplexing may be required!
- Resources in this design: RF, ALU, MEM.

Can we eliminate two adders? Can we just manage with a single ALU; that is one question. In general, how do we share resources across cycles? You are using adder in one cycle to do something, in another cycle you can do something else with the same adder so the solution lies in having the results of one operation stored at the end of a cycle in some register. You recall two designs of multiplier we discussed two different types of design; one was array multiplier where we had cascade of adders with no storage in between and the partial sum flew through flows through all those adders.

So now, as data flows through those you can notice that each one is not getting fully utilized each would be active and signals are propagating through that to only part of the duration. On the other hand, the sequential multiplier we had we did something with the adder, stored the results in register and reused that adder again so the key thing is that between two usages you have to store one result so you do something store the result and then the resource adder in this case is free to do the next operation. So same idea we will apply here and the three key resources we will keep in mind is register file, ALU which will do all arithmetic and logical operations now and the two memories will be clubbed into a single memory; it will store program as well as data.

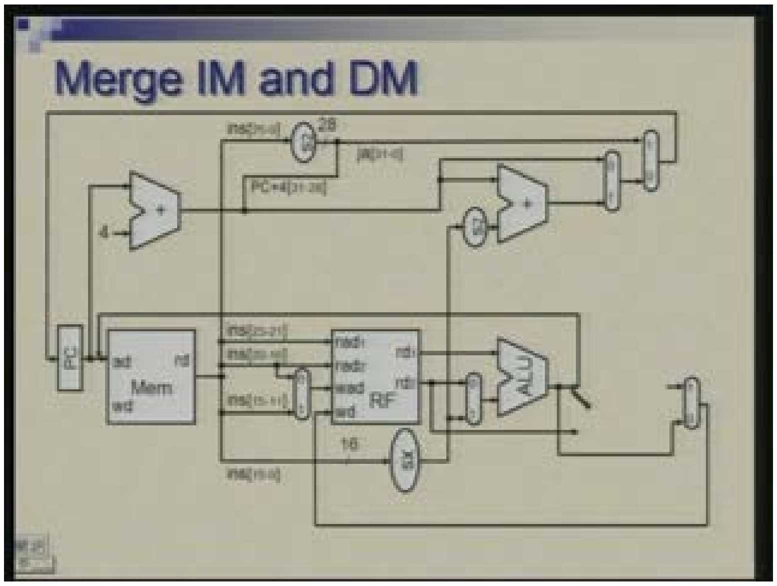
(Refer Slide Time: 15:08 min)



Single Cycle Datapath

Now let us take this is starting point which is a single cycle design and in this we will see what changes are required if we have to share these resources and we know that we are going to do a multi cycle design so each major action would be done in a separate clock cycle. Firstly, let us merge instruction memory and the data memory; replace both these by a single memory. So we remove these and replace them with a single block which is doing read as well as write and I have placed it here where program memory was kept but what we will do is we will route its inputs and outputs back to the same memory that means the ALU which was supplying the address (Refer Slide Time: 16:13) will go back and supply an alternative address; this was the data input this will go back to this point and the data which is being read will actually now come out here. So these inputs and outputs which were connected to these will now be rerouted and brought back to this particular block.

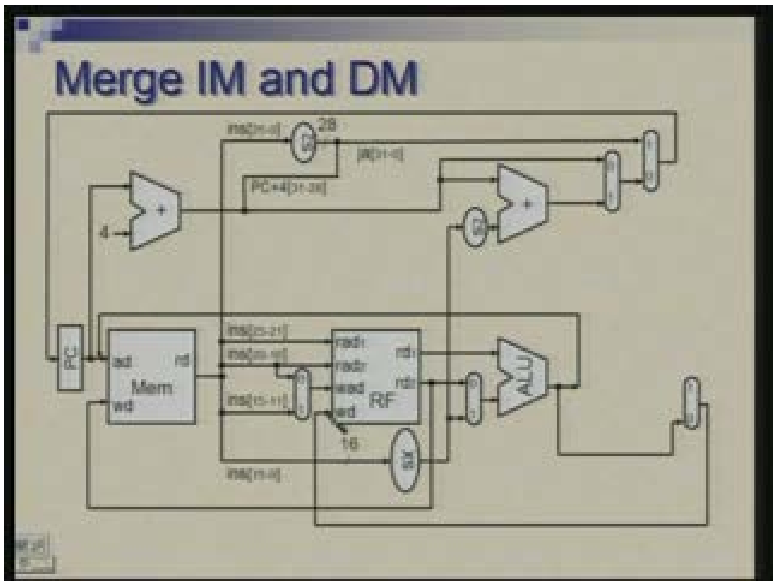
(Refer Slide Time: 16:40)



Merge IM and DM

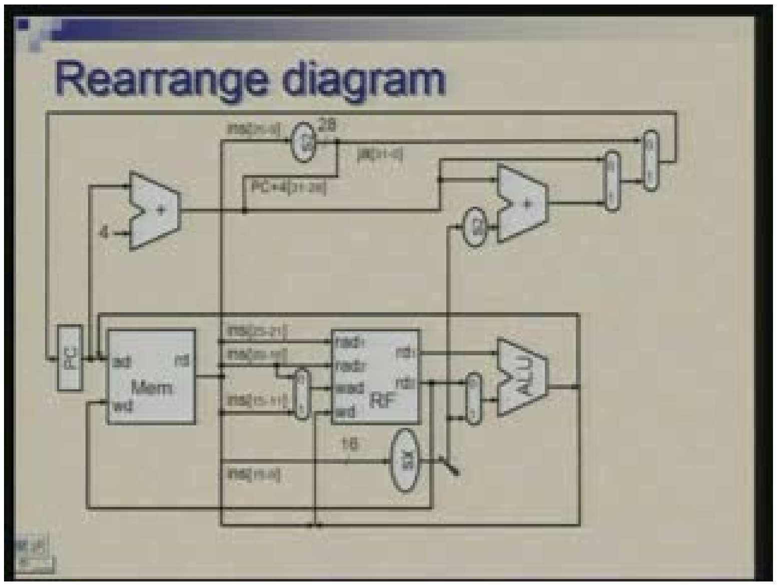
Therefore, first of all, this output of ALU which is the address for memory is connected back. Now we are bringing in a conflict here but we know how to resolve that. We will introduce a multiplexer here to take care of this but that is a problem we will tackle later.
So, first of all let us bring all connections here. I have taken care of address input, next the data input which is going here will be now brought to this (Refer Slide Time: 17:07); I have removed the old connection and it gets connected at this point. Similarly, the data which is coming out of memory and was eventually through this multiplexer going to register file will now come from this so that is removed and that is yeah it is connected here let me see it again. So the old output of the memory is removed and the new one will get connected into this. So it comes out of this and gets connected. Again we are bringing a conflict here; we know that the data needs to be brought from ALU or from memory for load instruction and for arithmetic instruction. So this is a conflict point we will resolve with a multiplexer again.

(Refer Slide Time: 17:30 min)


Merge IM and DM

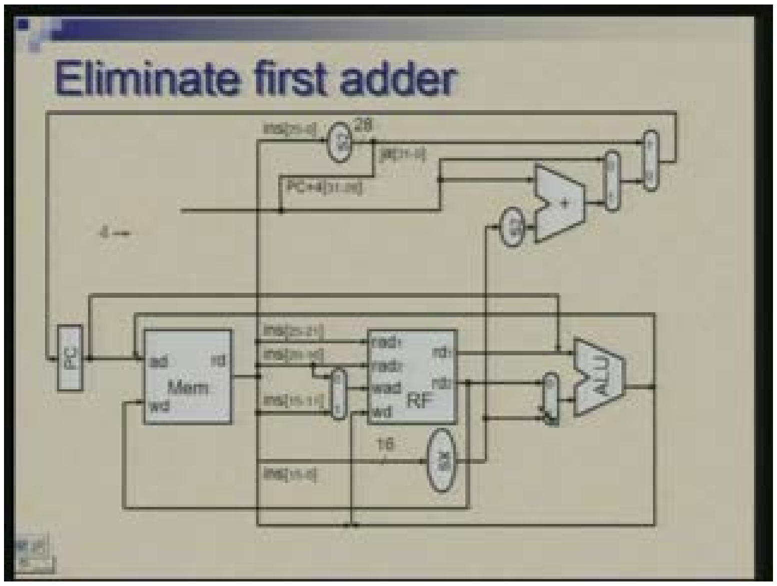
What we will do is for the moment we will remove this multiplexer and reorganize it here (Refer Slide Time: 18:40) later on. So, output of ALU is directly connected here for the moment but we know that a multiplexer is required at this point. We have gotten rid of one extra memory now gradually we will also collapse the two adders on to this ALU.

(Refer Slide Time: 19:10 min)


Rearrange diagram

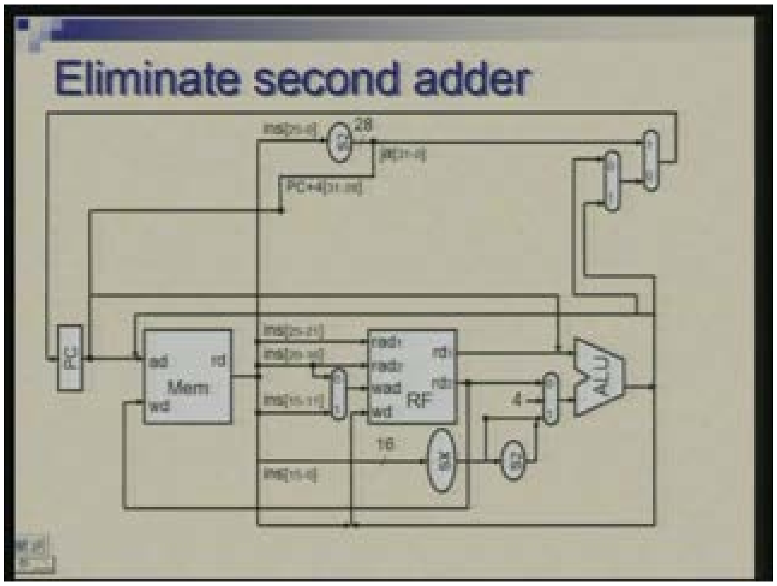
So first let me make space let me just shift things around so that there is space for carrying out an interconnection. Just the same thing with things moved little apart and we will first eliminate this adder and then eliminate that adder and same thing will be done; remove this (Refer Slide Time: 19:35) and just route the interconnections on to this so we remove that adder it was getting two inputs PC and 4. So the input coming from PC is brought in to this point of the ALU so this is a new connection and the second operand 4 will be brought to this multiplexer.

(Refer Slide Time: 20:03)



The output of this adder which was here (Refer Slide Time: 20:12) was eventually going through this multiplexer to be stored back into PC so, that we will have to tag from this point now. So we remove that connection from top and take a connection from here. Now we have still something hanging here; this was PC plus 4 which was being used for offset addition here and generating jump address here. This will now come from PC because we are assuming that you do PC plus 4 in one cycle the result goes back to PC. So the input to this is not taken from here, we take the value; since it is a multi cycle design PC plus 4 would be done in a particular cycle and that value will be put back into PC because remember that we are going to put all values into some storage element in some register at the end of the cycle so that resource which computed that becomes free. Therefore, this input here will come from PC now. I simply make that connection there. We have taken care of removing the first adder.

(Refer Slide Time: 21:46 min)


Eliminate second adder

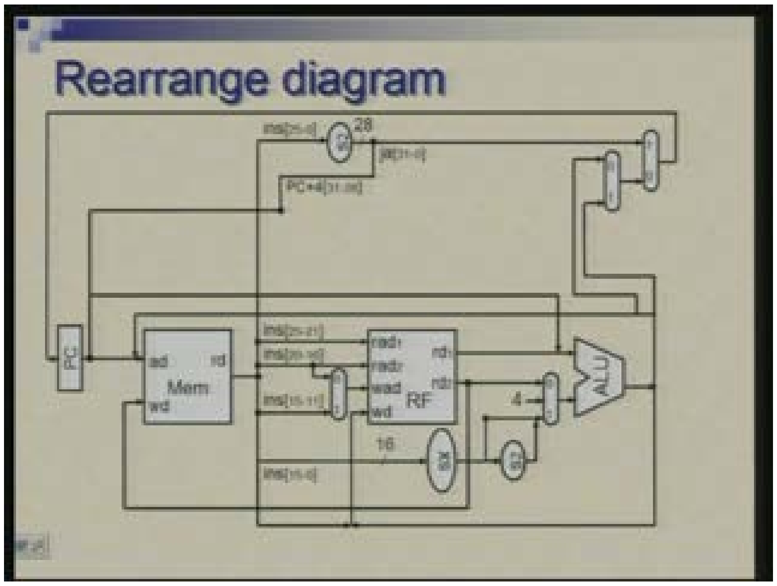
Now we eliminate the second adder; this one will be moved away and its inputs are again brought to the same ALU. So we have this output from PC which is already coming to this so that actually will not bother us that is already there. The second operand is coming from the offset eventually (Refer Slide Time: 22:12) and that would be brought to this multiplexer. The output from here will have to go to this multiplexer. Yeah, so this second output second input to the old adder is removed from there and is connected here. Then we simply get rid of the first input because that is already taken care of and the output I make some space first for that and then connect the output.

One might question actually that sorry…… why is it that we are still having two inputs to this they still seem to be coming from the same point. But actually it may be a little difficult for you to see……. what will actually happen is that although this value (Refer Slide Time: 23:37) which is actually supposed to go to PC this is the PC plus 4 value which is coming on this line it is meant to go to PC immediately but the other thing which is coming here after offset addition is meant to again go to PC but this is after comparison has been done in this ALU of the two registers and this may go or may not go so what will happen is first this value will get immediately stored in another register here before we do it before we take further decision of sending elsewhere.

So, at the moment time I am keeping these two both which look identical right now but one will come immediately from the output of ALU, one will come from a register which is following ALU so that distinction we will show up later but at this point you just take it like this that we are simply retaining.

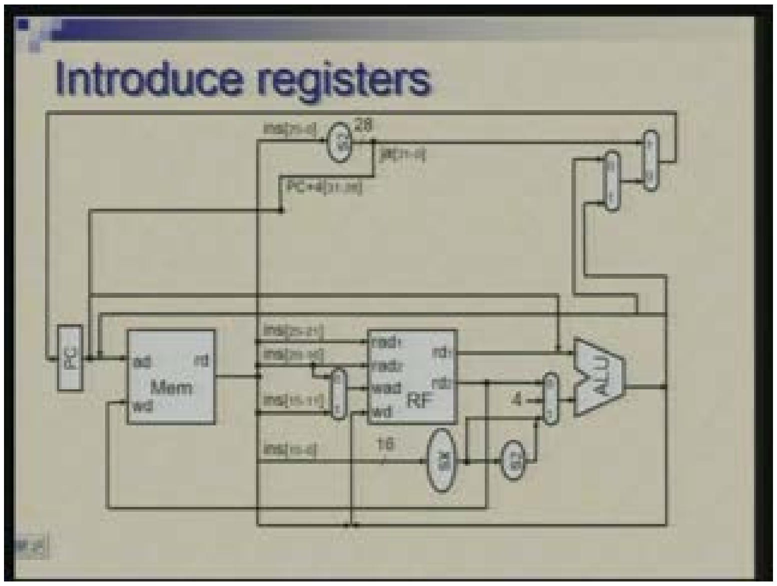
One was following this kind of design process. You might actually…. if you are not able to see that you may eliminate one of these but does not matter. If you need one later again, a separate one then you can bring it back into the picture but I am just following this approach.

(Refer Slide Time: 25:07)



Now we need to introduce registers at various places so that output of every resource is stored in register. I simply make some space for that and gradually introduce registers.

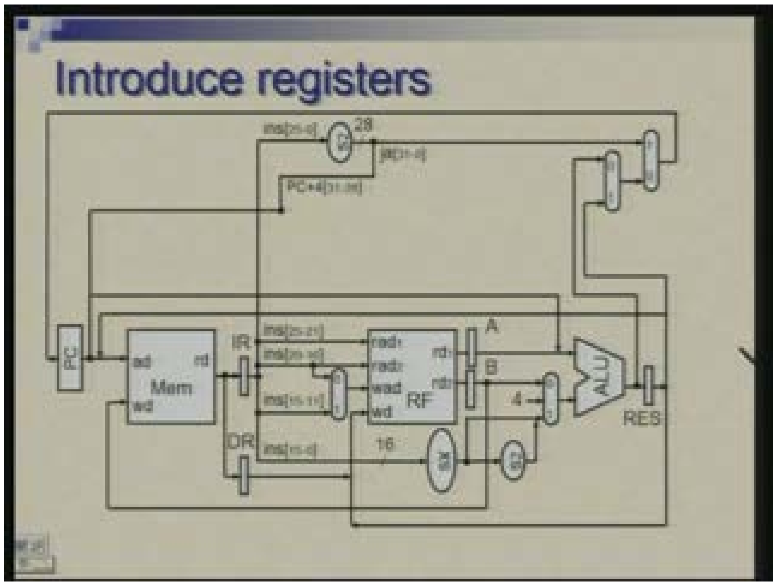
(Refer Slide Time: 25:20)



So, which are the points where we need to introduce registers. what we are reading from the memory needs to be kept in a register; we are reading two things: the instruction and we are also reading the data which needs to go to register file eventually so two registers will be required here where one will store instruction and one will store the data. Then you are reading operands from the register file, they will be kept in small individual registers here, the output of ALU will be kept in another register here. So these are the places where we need to put registers.

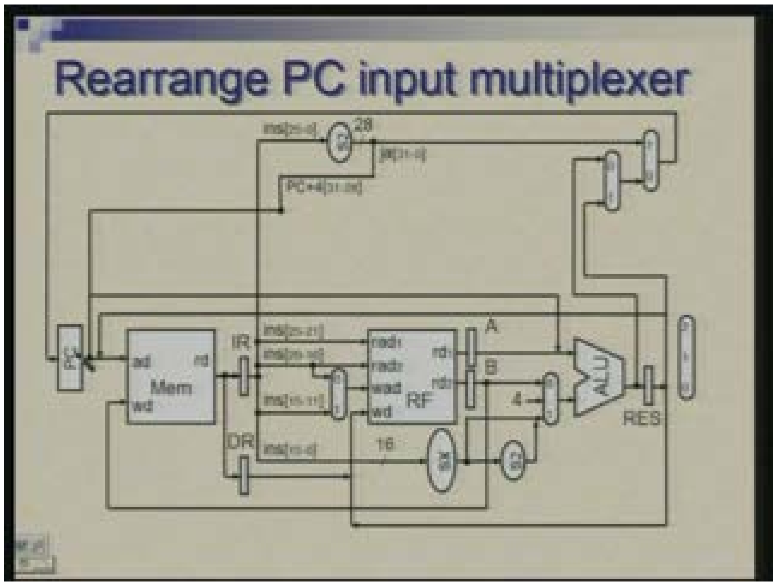
So we place one register here we call it IR or the instruction register, one is to be placed here we call it DR or data register so both are carrying information brought from the memory. Then A register holds the first operand and B register holds the second operand which come out of the register file. Then there is a register which we call RES or the results which comes out of the ALU and here I have made the distinction that PC plus 4 value is immediately going to the PC from ALU output; it is it is getting stored but getting stored in this PC whereas the other things what else ALU is doing; ALU is doing normal addition, subtraction, AND OR, slt operation so that that data will get stored in this register before it goes to register file and also the memory address which is being calculated for load store instruction will be resting in this here before it goes to the memory and also for branch instruction the next instruction address which I calculate by adding offset will sit in this register before I make a choice here whether to take PC plus 4 or PC plus 4 plus offset, so, that is the purpose this register will serve.

(Refer Slide Time: 26:09 min)



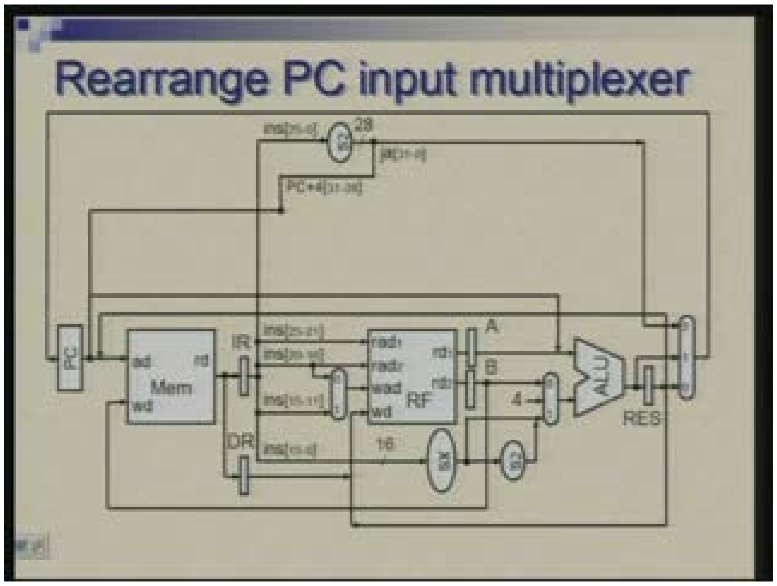
Now the last thing which remains is to introduce the multiplexers. We will At some place we will require fresh multiplexers, somewhere we will simply do a rearrangement may be enhance the size or restructure the multiplexer inputs. So we have multiplexer already there which are feeding the program counter. What I will do I will just collapse them into a three input multiplexer three input one output multiplexer which appears simpler and physically just bring it out here (Refer Slide Time: 28:20) so introduce a new multiplexer here but I will eliminate the old ones; this has three inputs; one input comes from RES register, one comes from ALU directly and one comes from this jump address. Those three are brought in here the output of this is going to feed the PC.

(Refer Slide Time: 28:47)


Rearrange PC input multiplexer

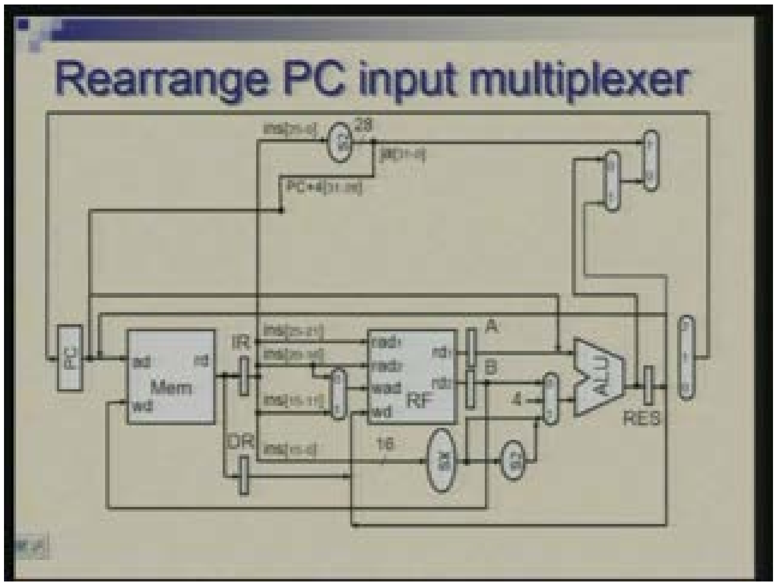
So, first I switch the output then one input comes from register result, one input comes from ALU and the third input comes from jump address and I have gotten rid of the earlier multiplexer.

(Refer Slide Time: 29:15)


Rearrange PC input multiplexer

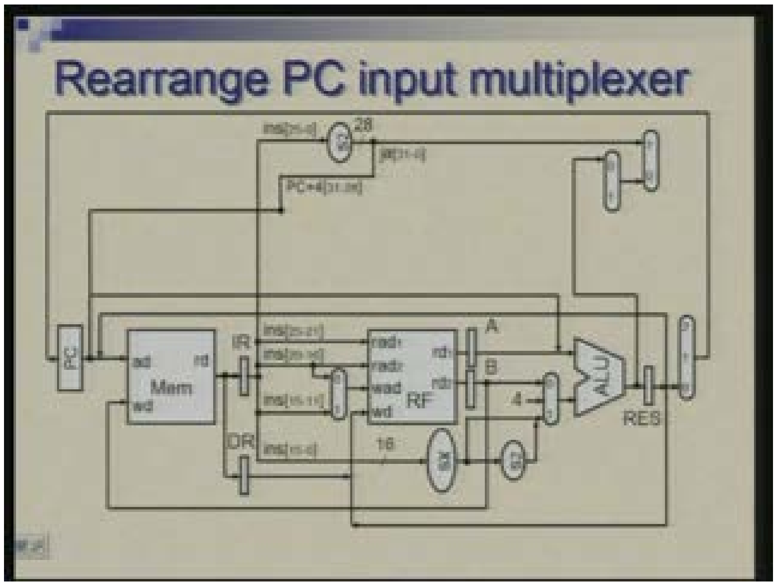
You can see it again. So this is one connection.

(Refer Slide Time: 28:55)



Rearrange PC input multiplexer

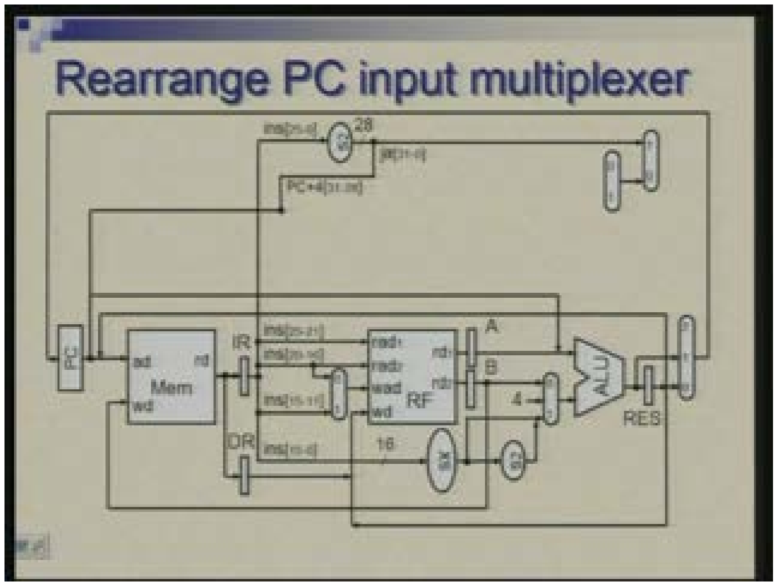
(Refer Slide Time: 29:28 min)



Rearrange PC input multiplexer

This is second connection.

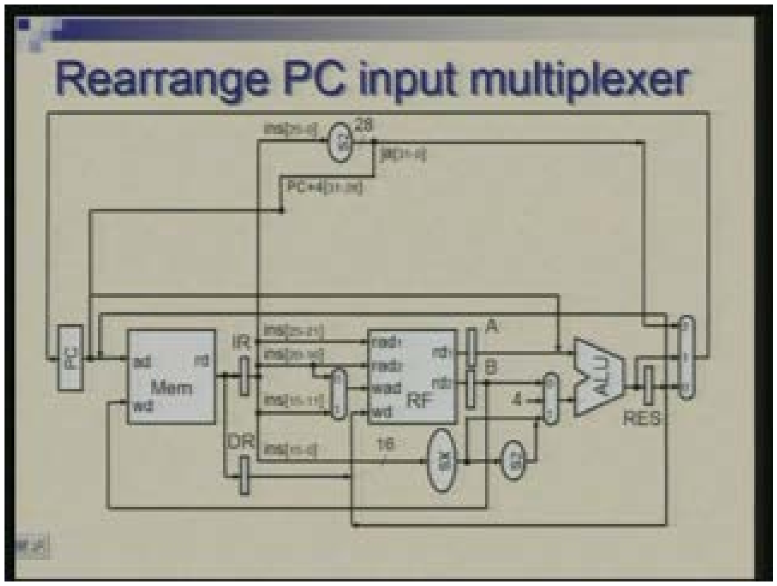
(Refer Slide Time: 29:32 min)



Rearrange PC input multiplexer

This is third.

(Refer Slide Time: 29:35 min)

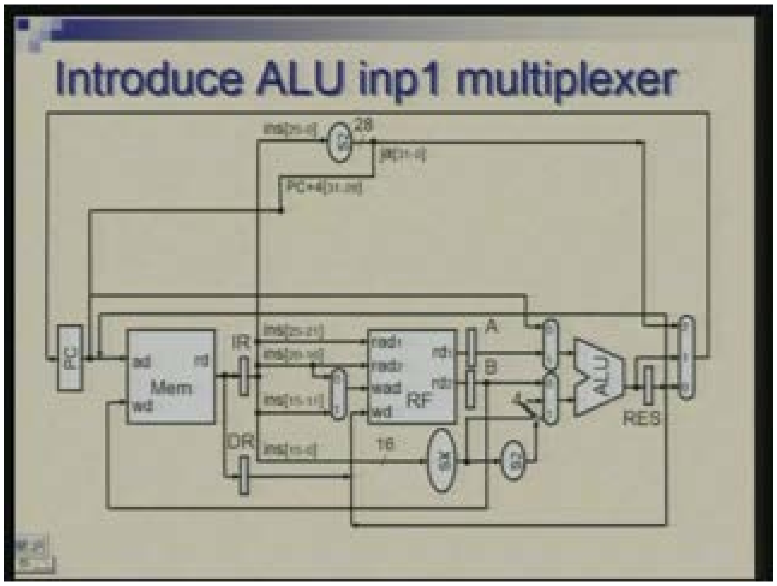


Rearrange PC input multiplexer

This is the fourth. So one output and three inputs they are all rearranged.

Next we look at the multiplexer which is taking care of the first input of ALU. ALU has two inputs now. The first input comes from either A register A or it comes from PC from here (Refer Slide Time: 30:03). So simply I need to introduce a multiplexer here and connect these two inputs to that multiplexer.

(Refer Slide Time: 30:16)

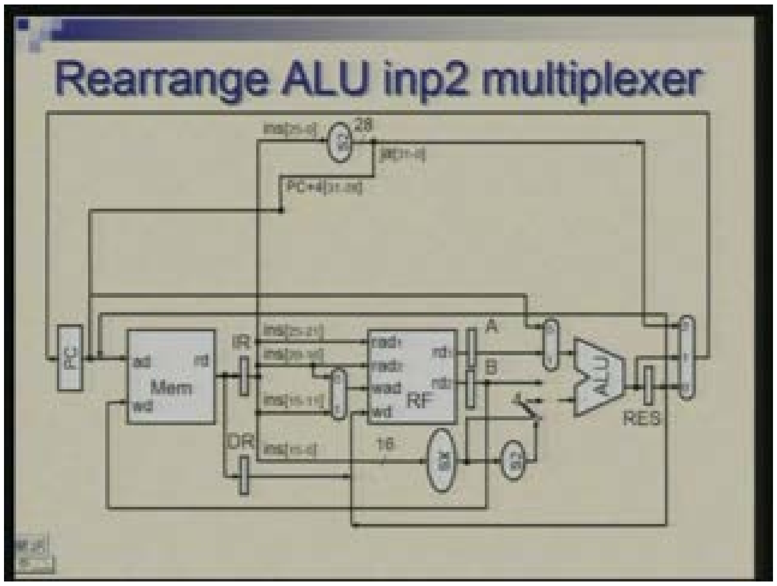


Introduce ALU inp1 multiplexer

So this is a two input multiplexer: One input comes from PC, another is already there shown from register A. Then this multiplexer which is feeding second input of the ALU now has four inputs B, B register, the constant 4, this offset which is used for load store instruction for calculating the address; it is the offset in the instruction with signed extension.

For branch instruction we need to do a shift also we need to sign extension followed by a shift so these are the two separate values and there are total of four possibilities. We need to have a bigger multiplexer here now.

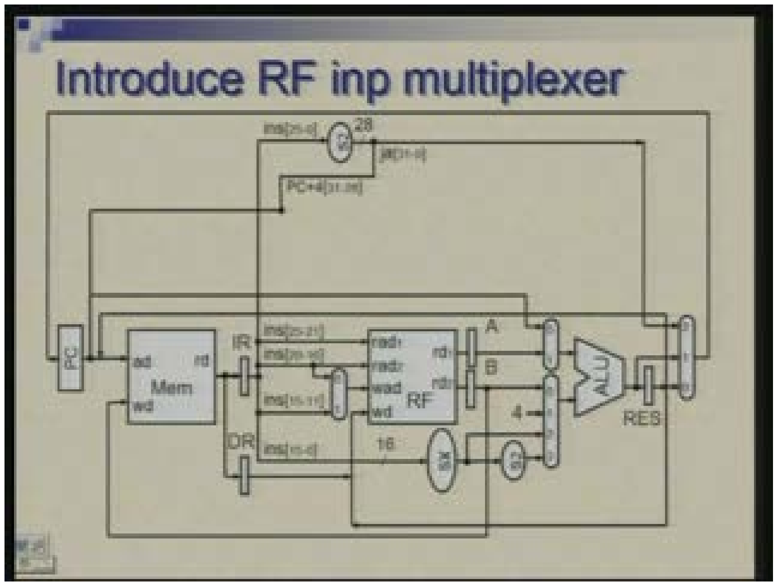
(Refer Slide Time: 31:12)



Rearrange ALU inp2 multiplexer

So place a bigger multiplexer and simply connect these signals properly. So we would need to see we will need to worry about the control of these multiplexers; what control value control input is required for different instructions and that you will analyze later in the usual manner. Then we move our attention to this area where we have data coming from DR or from RES that needs to be multiplex before we feed into the register file. So just move that wire at the bottom; this one (Refer Slide Time: 32:06) to make some space here where I am placing this multiplexer so this is one input from DR the output is properly connected and this signal coming from result is actually extended brought to this.

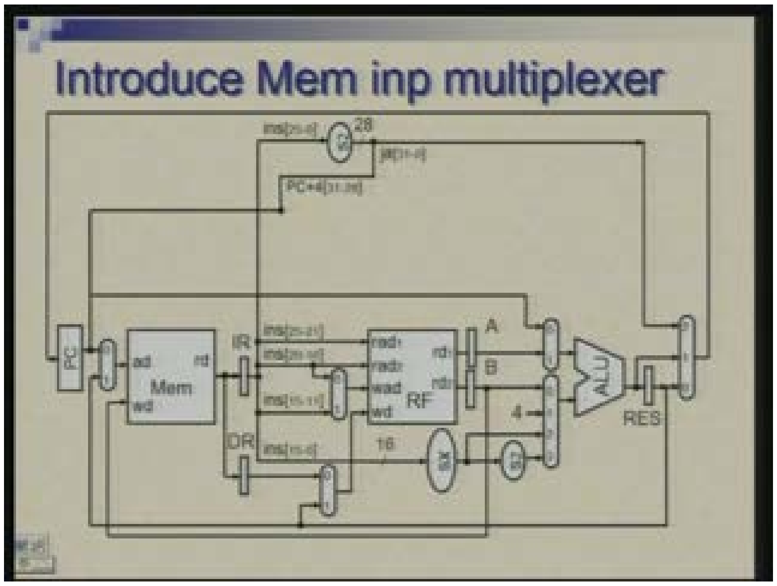
(Refer Slide Time: 32:24 min)



Introduce RF inp multiplexer

Now this is taken care of. Finally I need to put a multiplexer at that point. So here we have address; there are two address resources PC for fetching in the instruction and this RES where load store address would be calculated and kept, this is another address (Refer Slide Time: 33:06) so it will be simpler to just extend this line here and I will remove this line which is coming from top; same thing, I just pull it more neatly from the bottom.

(Refer Slide Time: 33:31)

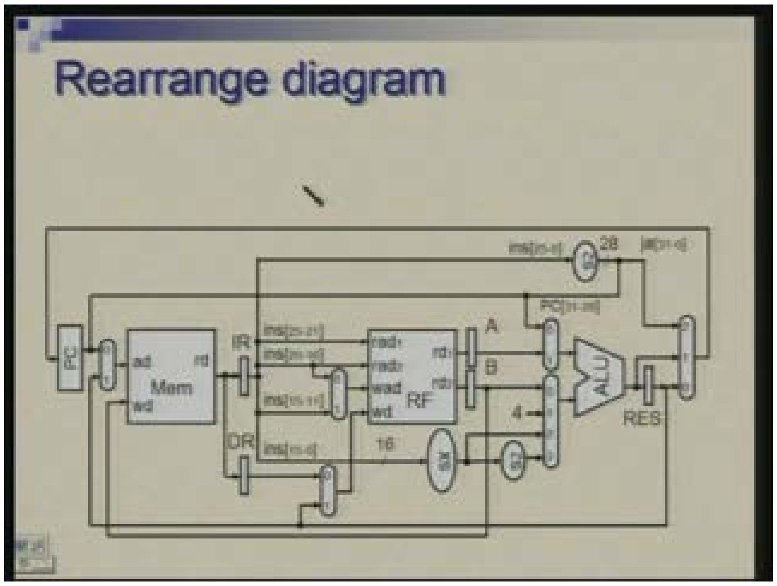


Introduce Mem inp multiplexer

Now this is a complete design of the data path for multi cycle. Basically more effort was there to ensure that resources are properly shared. For that we need to collapse multiple components on to the same thing and with the help of multiplexer we are able to feed different inputs to those resources at different times. And then registers were introduced to break the time interval. So time interval gets broken in to multiple clock cycles and now the paths which need to be considered for delay analysis are a much shorter path.

For example, you can take paths going from one storage element to another storage element. When you are fetching instruction it is this path PC to IR or there are paths from IR to A or IR to B or from DR to register file. And similarly, path from A and B through ALU to result register and so on. So, main sources of delay are these three components: memory, register file and ALU and if these three are roughly balanced when this will work very nicely. So, that is complete design. We will look at the control part in the next lecture. I will just rearrange this diagram to make redraw this little neater.

(Refer Slide Time: 35:25)



Rearrange diagram

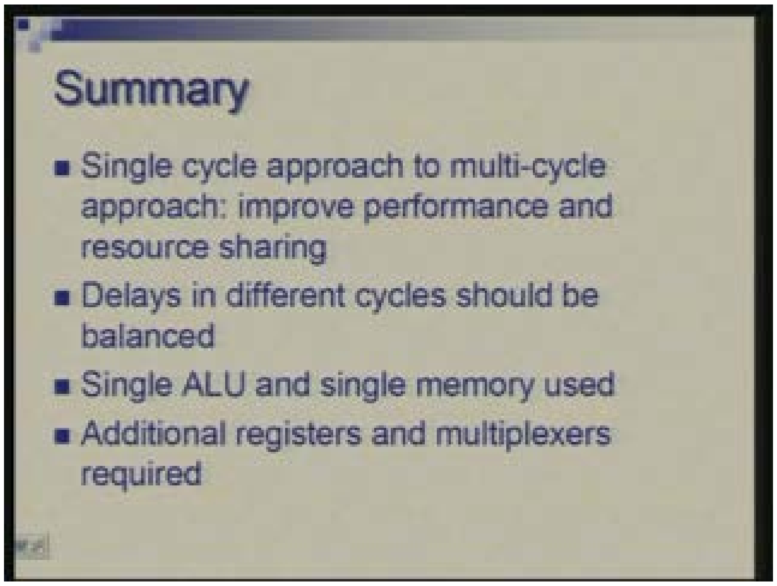
We will go into details of control signals. Basically now we have lots of multiplexers. We have 1 2 3 4 5 6 six multiplexers are there which require control and multiplexers like this one will require 2 bits to control them. Apart from that we have the usual control requirements for memory, register file and ALU. We also need to now look at these registers in which cycle we load these registers and which cycles we do not.

So, for example, if you take let us say add instruction, after the first cycle IR gets loaded, after the second cycle A and B gets loaded, after the third cycle RES gets loaded and then in the fourth cycle this RF writes RF gets the value written into it. Also, we will typically load PC with the new value PC plus 4 at the end of first cycle itself. If it is a branch instruction and branch has to be taken that we will overwrite that PC plus 4 value in the PC by a different value and similarly in jump we will write something else in that. So, even PC will require a control.

In every cycle we need to determine whether a new value goes into this or does not go because there is no register which is now loading a new value in every cycle so this needs to be taken care of and the number of control signals therefore is quite large. We have these six multiplexers out of which two require 2-bit input so a total of eight control signals we require just for multiplexers.

We have two for memory, 1 bit for RF and 3 bits for ALU so this is another 6 bits and we have 1 2 3 4 5 and 6 registers so one control signal for each of these registers. So there is a large number of control signals which are required. And also, the control would be different in the sense that in every cycle there is a different set of control. So the control will no longer be a simply combinational box; it will be a sequential machine which goes through a set of cycles four cycles, five cycles or three cycles depending upon what the instruction is and in each cycle it tries to control things differently. So the design is a more involved in that sense and that is the reason the single cycle design was considered the first being the most simple design.

(Refer Slide Time: 38:37)



Summary

- Single cycle approach to multi-cycle approach: improve performance and resource sharing
- Delays in different cycles should be balanced
- Single ALU and single memory used
- Additional registers and multiplexers required

So, in a summary we have moved over from single cycle design to multi cycle design. we have compared their performance. Basically it gives you better performance and by resource sharing it tends to reduce the cost. But now we must notice that there is a trade off although we are trying to gain both in time and in cost somewhere there are losses also and we have to be careful that losses do not overshadow the gains. So for example, in performance improvement if the quantization has to coarse and if there is an imbalance on the values of timings for different individual actions then on one hand we might gain but on other hand we might lose all the gain simply because there is an imbalance.

Similarly, while resource sharing we have eliminated those adders and we have removed we have eliminated one memory but we have incurred extra cost in terms of registers and multiplexers. So you are sharing but there is an overhead of sharing and we have to be a little more careful in our calculation and ensure that the overhead does not overshadow the gain. Finally just to close we had design where data path conceptually is simple in terms of key resources with just one ALU, one memory and one register file but there are registers and multiplexers which glue these all together. So, next we will look at the issue of control design for such a data path, thank you.