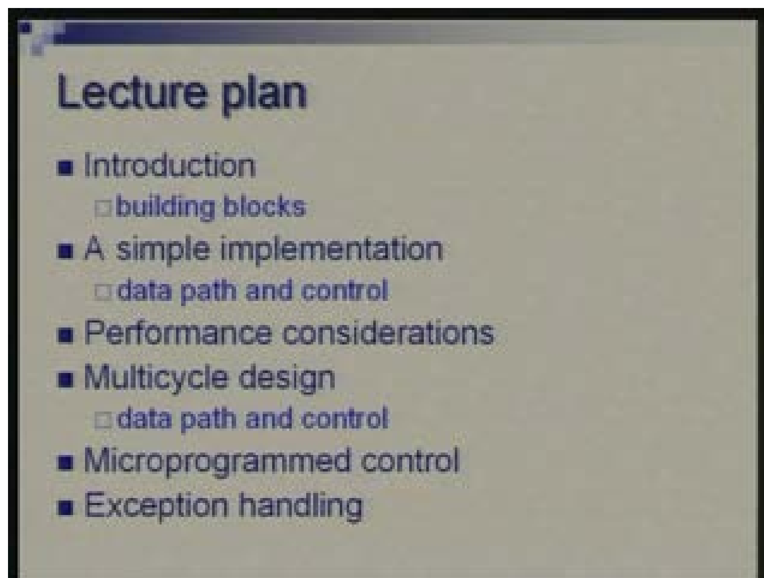
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 19
Processor Design - Simple Design (Contd…)

We will continue with the simple design of a processor which we began in the last lecture. What we had done primarily was that we have taken a small subset of instructions and try to design the data path. What we will do today is that we will look at that design from a slightly different angle and come up to the same point; we will then go in to details of control part design. So data path is the one which is doing the computation and control is the one which guides or directs these computations so that the right action takes place at the right time.

(Refer Slide Time: 01:36 min)

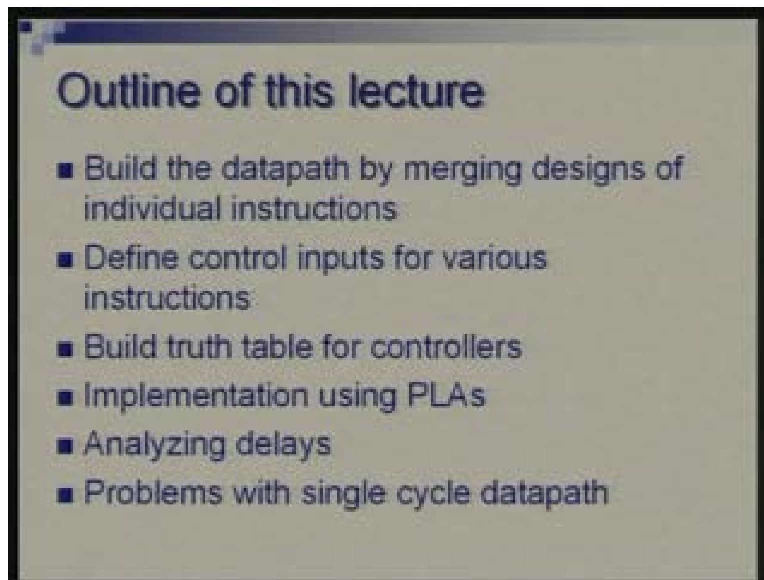


This is part of the overall plan. We are looking at a simple implementation where the entire instruction gets executed in a single cycle. After we finish with this we will look at the performance considerations and probably go for a more involved design which is the multicycle design.

So today we will arrive at the same data path design by looking at it from a different angle. What we will do is we will look at each instruction or group of instructions separately and then try to merge the results or merge the outcome of each separate consideration and then we will look at what control inputs are required to drive various components in the data path and based on that we will build the specification of the controller and then go in to details of the controller design.

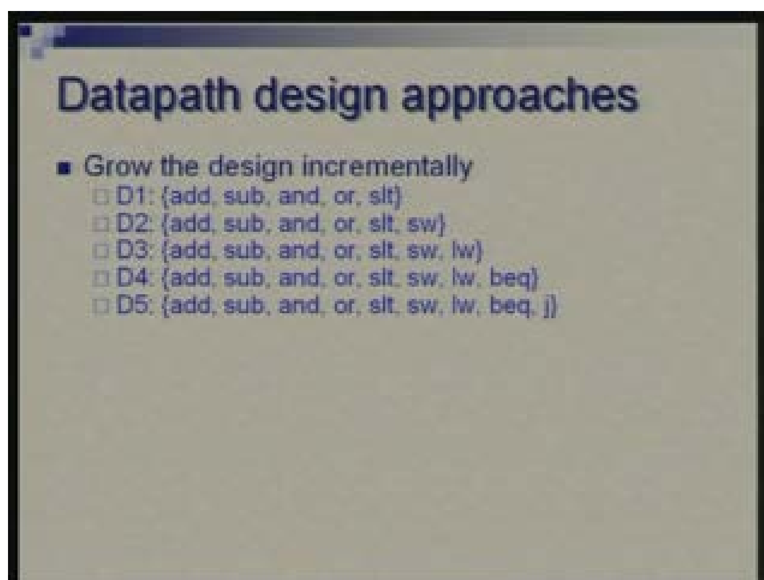
We will see one possible implementation using what is called PLA Programmable Logic Array and if time permits I will talk of performance consideration as how do you analyze performance or analyze the delay of such a design. At the end we will notice that this particular design approach has some limitations and once you understand those limitations we will able to talk of an improved design.

(Refer Slide Time: 03:03 min)



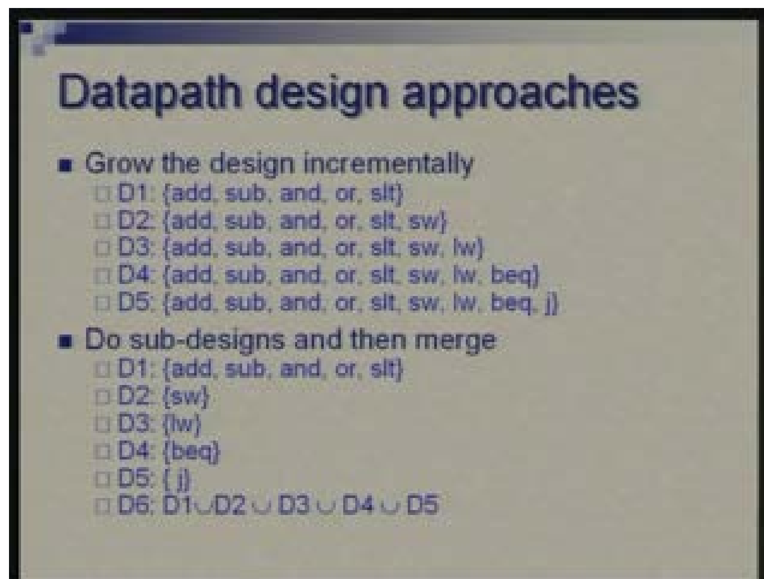
So let us compare the two approaches which I am talking of: One approach we followed in the last lecture and another one we will follow now.

(Refer Slide Time: 3:15)



What we did yesterday was that we began with one design let us call the D1 where we looked at five instructions which belong to one particular group namely arithmetic logical group. We augmented or enhanced the design by including store instruction. We improved it further we expanded it further by including the load instruction then we added on beq instruction and then finally we threw in place j instructions. So, that is how starting with a very simple design we gradually built up a more and more involved design and this process actually can be continued so you you can you can look at the entire instruction set look at more instructions and by following a similar approach you can keep on adding or enhancing the design that is one approach you follow.

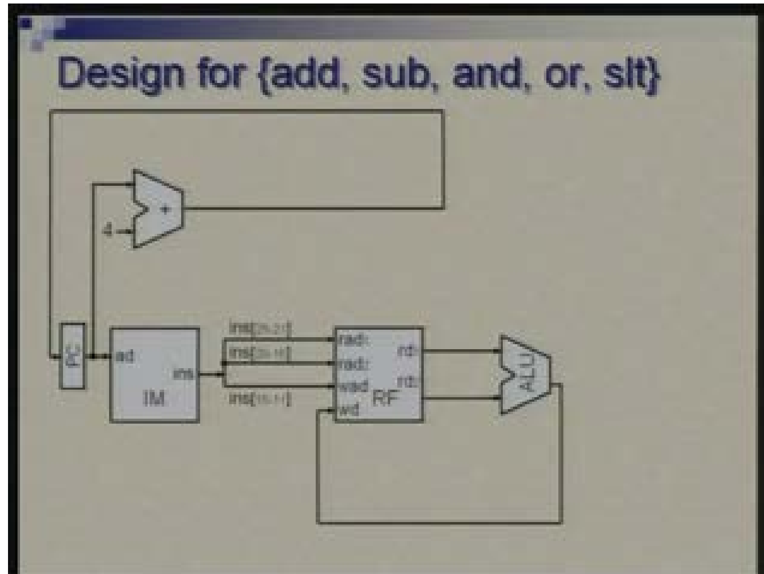
(Refer Slide Time: 4:06)



What we will it do today is that we will look at separate designs as if there were only limited number of instructions. So first we will have a design which looks at these five arithmetic or logic instructions; of course the answer will be no different from what we had earlier. But then we will leave that apart and look at just store word instruction and see what are the requirements for this particular instruction.

Similarly, we will do it for lw separately, then beq in isolation then j in isolation and then we will try to merge these five different designs which may have something in common so that common part will remain common but we will kind of take a union of whatever the requirements are of different solutions and put them together. Since our approach is same that we are trying to do the whole thing in a single cycle and the given broad components may be identified we will approach the same design point at the end.

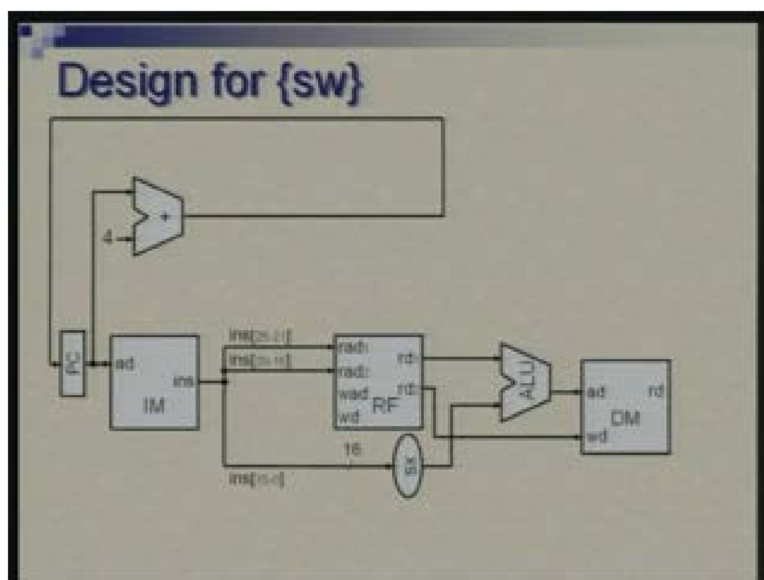
(Refer Slide Time: 05:06 min)



This is what we had; the starting point even yesterday, that, for doing add subtract operation you need PC to supply the instruction address, from the instruction you will look at the register address fields, fetch two operands from register file, apply them to ALU, result of ALU goes back to register file and mean while we also make sure that PC gets incremented to PC plus 4. So these components can be put together to carry out the first group of five instructions.

Then now let us look at sw alone in isolation.

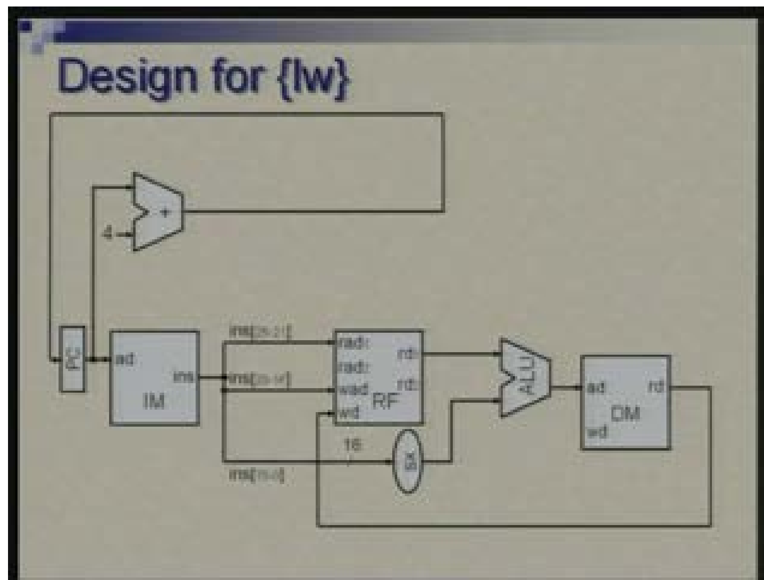
(Refer Slide Time: 05:45 min)



Here ALU does something different; ALU does the address calculation and we bring in the data memory component so ALU feeds the address for data memory and the data input I do not know what happened to the curser I am not able to get curser here…. suppose leave that so the address is found by adding one register content and 16-bit as an offset coming from the instruction.

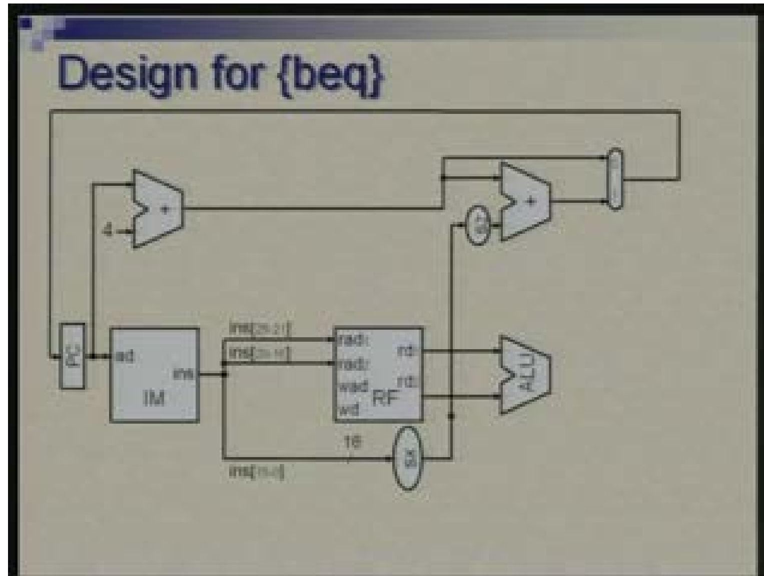
Next we have data path portion of data path which will take care of lw if that was the only instruction.

(Refer Slide Time: 06:45 min)



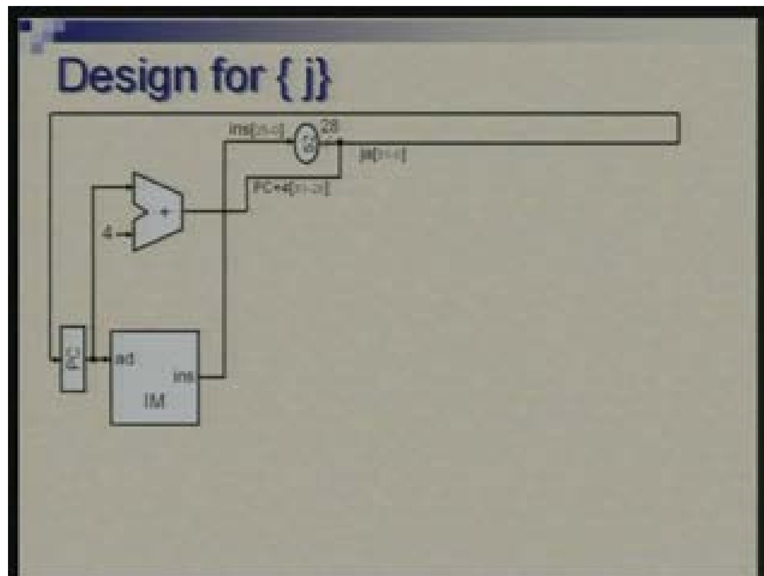
So notice that now we are still looking at two addresses from the instruction but while in case of stored both were being used to read, now one is being used to read and the other is for writing; it is a load instruction so one register takes care of address generation and the other address is where there is destination of data coming from memories.

(Refer Slide Time: 07:20 min)



This is for beq. So again I have omitted data memory. ALU is being used for comparison. we are looking at two operands from the register file which is being fed to ALU for comparison purpose and the target address in case the condition is true we need to go to some particular target address; that is being calculated by adding an offset to PC plus 4 and note that we have sign extension and we have shift by 2 and all these things here.

(Refer Slide Time: 7:57)



Finally for jump instruction all that we need is putting the right bits together to form the destination address. We do not need register file. This is instruction does not look at

register file, does not require ALU, does not require memory and all that you require is some little circuitry to wire the bits forming the address appropriately.

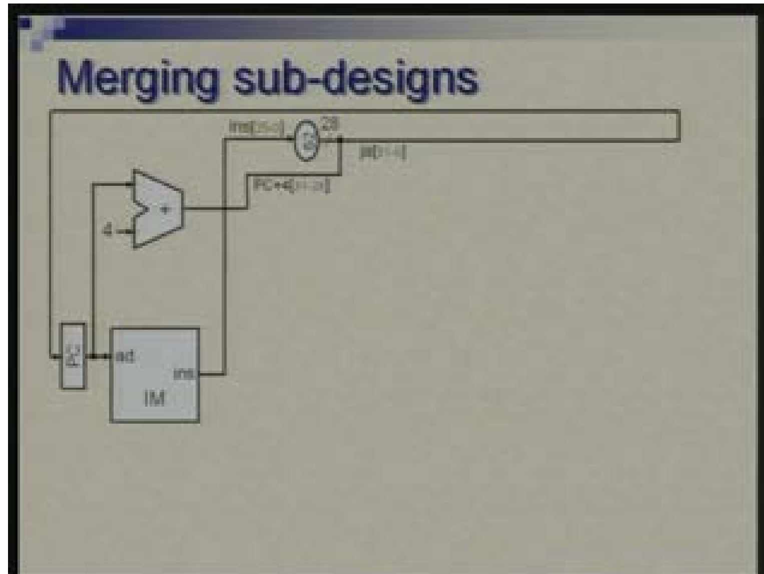
Now we have we have come up with these five partial designs; we need to put them together to form the final design. So these I am calling these as sub-designs we are going to merge these so this was one. Let us quickly cycle through these; that is the second one; this is for store instruction, then this is for load instruction………

(Refer Slide Time: 08:50 min)

(Refer Slide Time: 08:53 min)

This is for beq instruction.

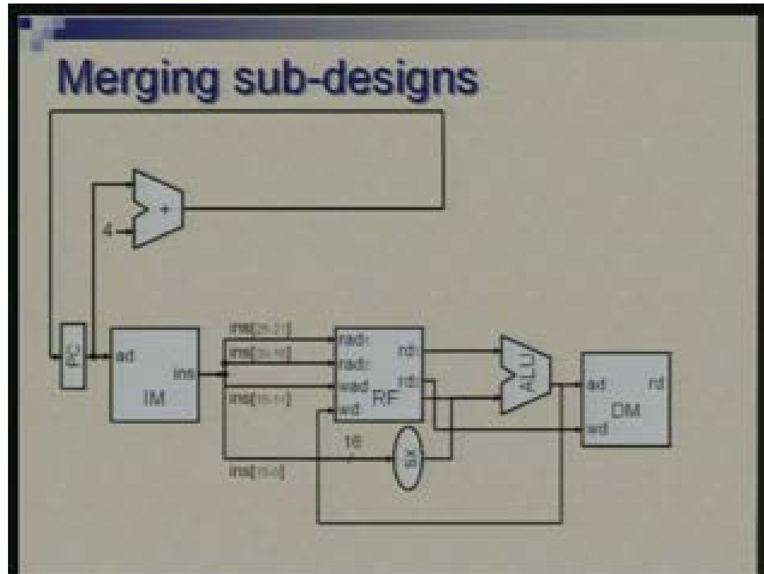
(Refer Slide Time: 08:59 min)



This is for jump instruction. So you would need you would notice that these are few things in common; each instruction may not utilize all the resources but if you take union of all these we can reconstruct the entire designs and that is what we will do now.

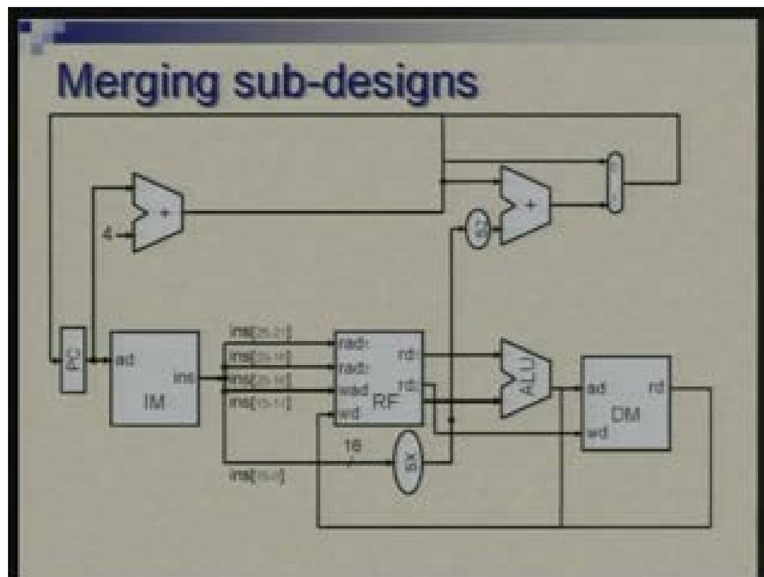
We will now….. now we will we will place we will take these five as if there were five sheets of paper and just superimpose one over another and naturally the things which are common will fall in place and which are which are not common we will kind of take union of those but there will be another thing that there will be somewhere a conflict; in one sub design we want one piece of data go to a particular destination, in another design we want something else to go to that destination so we will identify those and take corrective measures.

(Refer Slide Time: 10:22 min)

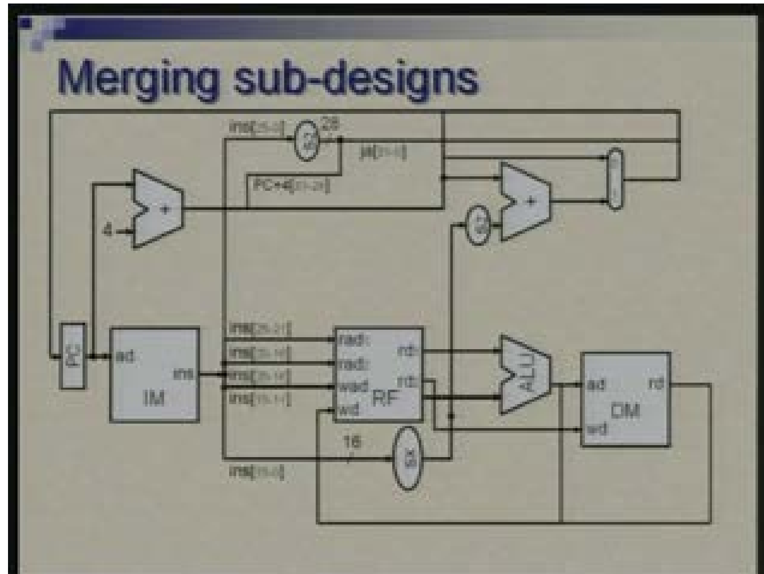


So the second one is superimposed over the same thing; third one is again put over that so we are not erasing anything and somewhere the lines are getting doubled up somewhere two different things will go to the same point.

(Refer Slide Time: 10:42 min)



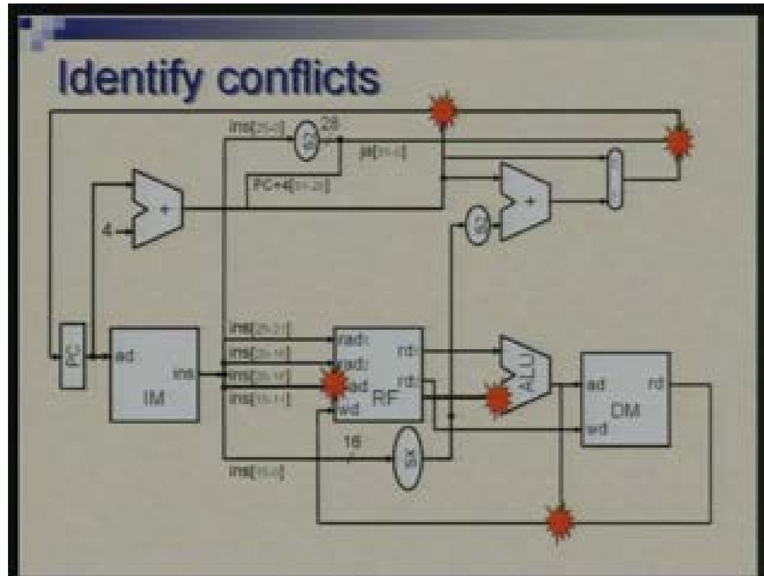
(Refer Slide Time: 10:45 min)



This is for beq and finally this is for jump. So now we have everything just superimposed and what remains now is to identify the points of conflict where we are trying to send two different things to the same point in two different situations. So we now try to identify conflicts. This is one point of conflict (Refer Slide Time: 1:08) where add subtract instructions send one thing here and the load store instruction tries to send something else here.

In one case we are adding two operands coming from register file; in other case we have one from register file, one is a constant coming from instruction so there is a this is a point of conflict. This is another point of conflict (Refer Slide Time: 11:34) in terms of what goes back to the register file. So, in arithmetic instructions output of ALU goes back, in load instruction output of data memory goes back. And associated with that is this point of conflict (Refer Slide Time: 11:53) where there are different addresses used to address the right port of the register file. It is either INS 11 - 15 or INS 16 – 20; different fields in the instruction are addressing the same right port. This is also a point of conflict wherein instruction other than branch and jump we simply want PC plus 4 to be sent back to PC. But in case of branch we have something else coming out of address calculation and that needs to be going to the PC and finally this point is where jump has one way of calculating address, branch has another way of calculating address and ultimately they need to go to the same destination.

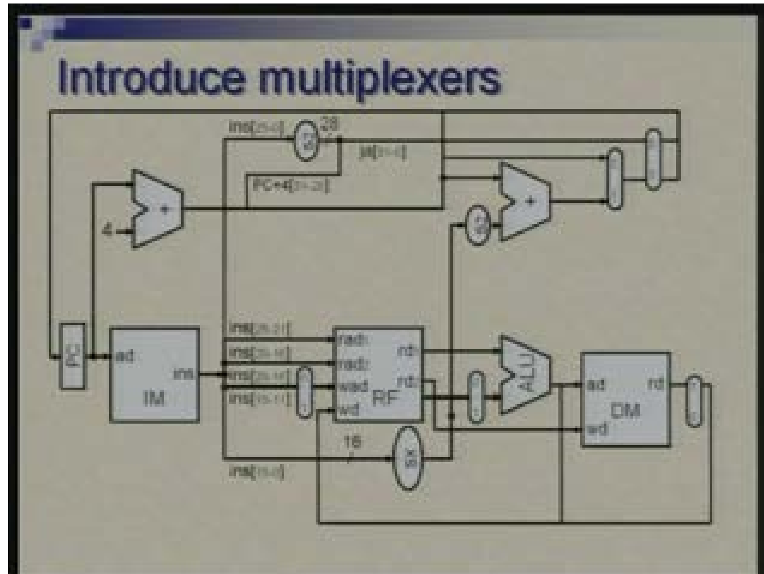
(Refer Slide Time: 12:38 min)



What do we do for these?

From the other design we can actually get an idea that we need multiplexers and multiplexer will be driven by suitable control signals. So we introduce multiplexers; basically we have to introduce multiplexer and also connect the signal appropriately. So you would notice that except for one place I have put in the multiplexer; the fifth one the point of conflict between non-branch and jump instruction and other instruction would get resolved automatically because the multiplexer we have put for branch itself will take care of that because it is a conditional branch it has a provision of passing PC plus 4 as the next address; so you are either having PC plus 4 plus offset or PC plus 4. So the provision which we have PC plus 4 will suffice for non-branch and jump instruction so we do not really need a multiplexer there.

(Refer Slide Time: 13:40 min)



Let us now remove the conflicting line and put appropriate connections in place of those and we reach the same design which we had arrive at yesterday. So it is just a the idea is to give you a different view different way of looking at it so whatever is convenient that approach could be followed and there is nothing secret about one another.

Is there any question at this point of time? Alright, if this is okay let us move towards working on the design of controller. So you would recall that we have identified control points; since it is the same design we have same points. Primarily we are controlling the multiplexers, we are controlling read write operation of memory, write operation of register file and we had a gate controlling the PC source so that the controller design is simplified.

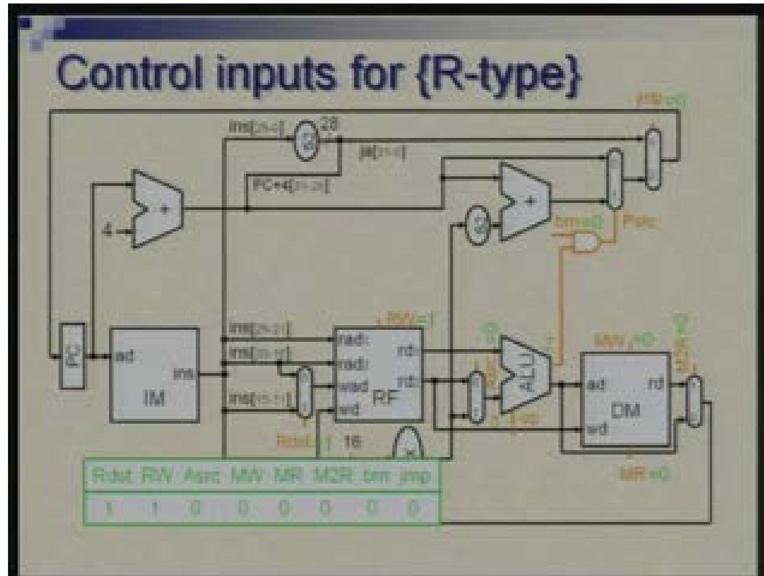
(Refer Slide Time: 14:57)

Now what we need to do is a look at each instruction or group of instructions and see what value needs to be given to these signals; what input needs to be given. So, for example, let us start with R-type instruction or basically arithmetic logic instruction. RDST needs to be made 1 because the write port has to be addressed by bits 11 to 15 of the instruction so we need to give…. this control signal is 1. RW tells whether the register file has to take in a new value or not; this also needs to be made 1 because you are writing something in to the register file. ALU source is 0 we want to take the top input, MW is 0 we are not writing into memory, MR is 0 we are not reading from memory and M2R is also 0 because we are taking the bottom input of the multiplexer.

Going further it is not a branch instruction; we keep the signal 0 it is not the jump instruction we keep this signal jmp also 0. Now let me point out that there is a change which I have made in the diagram in relation to the previous one; it is that the label 1, 0 I have reversed it as jump control multiplexer. that is just for convenience that we have the meaning of the jump signal the polarity is 1 when it is a jump instruction and there is nothing very particular about this but just to call it jump otherwise I would need to call it jump bar that is the usual convention.

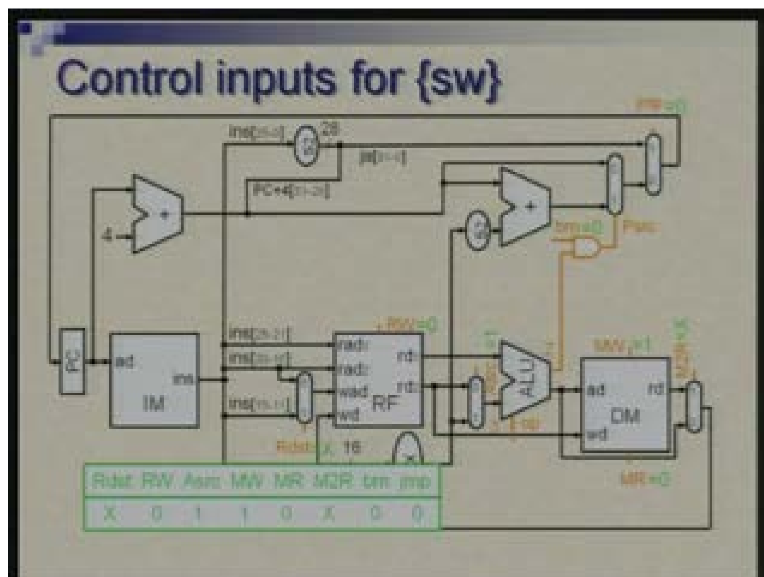
So now this is the set of values this is the set of inputs you need to give at control points and if that is given the instruction done by this data path will be the R-type instruction. We still have not specified what input we give to opi I will take it separately but let us look at the others for each instruction. So all these put together is a table of values we need to give at different control points.

(Refer Slide Time: 17:00 min)



Let us go to the next one sw instruction and repeat this exercise. Rdst is don't care because we are not writing into register file and that will be indicated by making RW as 0. So basically when RW is 0 value of Rdst could be anything it does not matter. If ALU source is 1 because we need to take the bottom input, memory write is 1 it is a store instruction, memory read is 0 and M2R is also x this will also go with Rdst. So, when RW is 0 Rdst will be x and M2R will be x these will be don't cares (Refer Slide Time: 17:54) it does not matter which one you select.

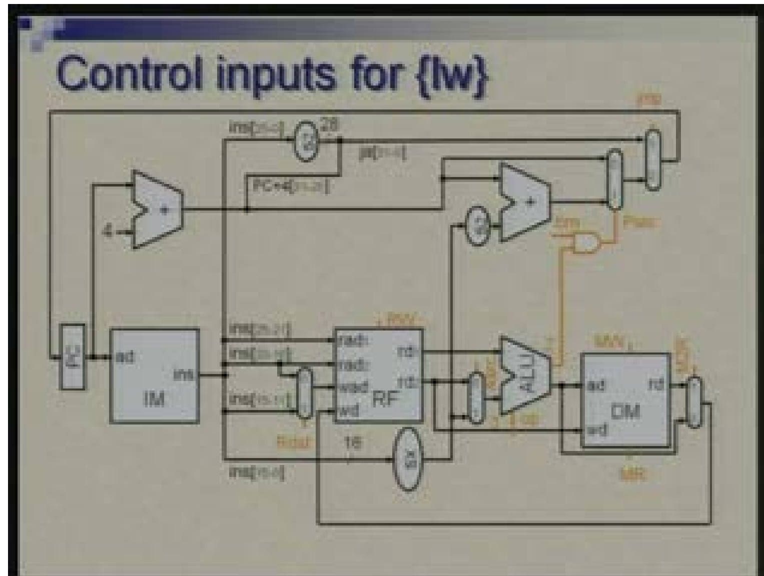
(Refer Slide Time: 17:58)



So as you would be familiar with a logic design that if there are don't cares in the specification these could be exploited to simplify your circuit so from that point of view we will try to put x wherever we can and further brn is 0, jmp is 0. Putting all these together we tabulate all the control signals which are required for this particular instruction.

Next is the load instruction; similar exercise.

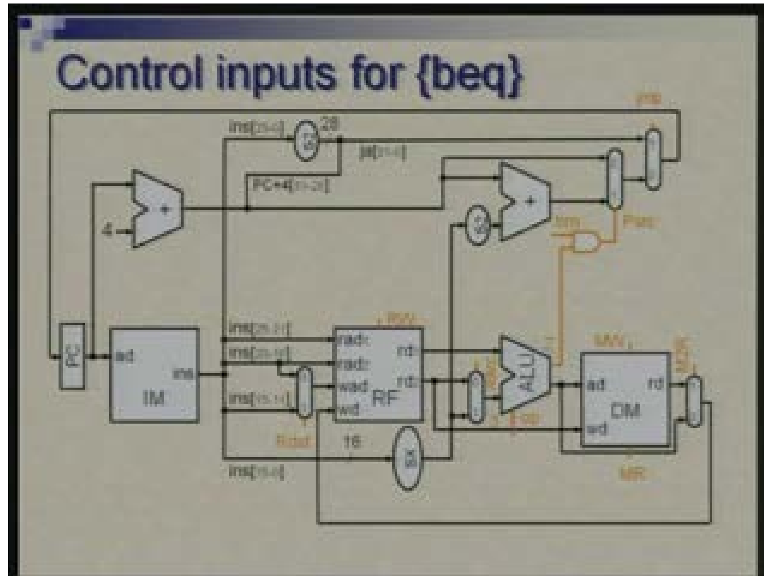
(Refer Slide Time: 18:28)



Here we are putting things back into register files so Rdst is 0 we are taking the top input, RW is 1, ALU source this is same as what we had for store and now we are not writing memory but we are reading so MW is 0, MR is 1 and M2R is also 1 because output of memory is being sent to the register file; brn and jmp continue to be both 0s. So these signals can be put together again in the table and these green table which I am forming I will have to collide these together to form an overall truth table.

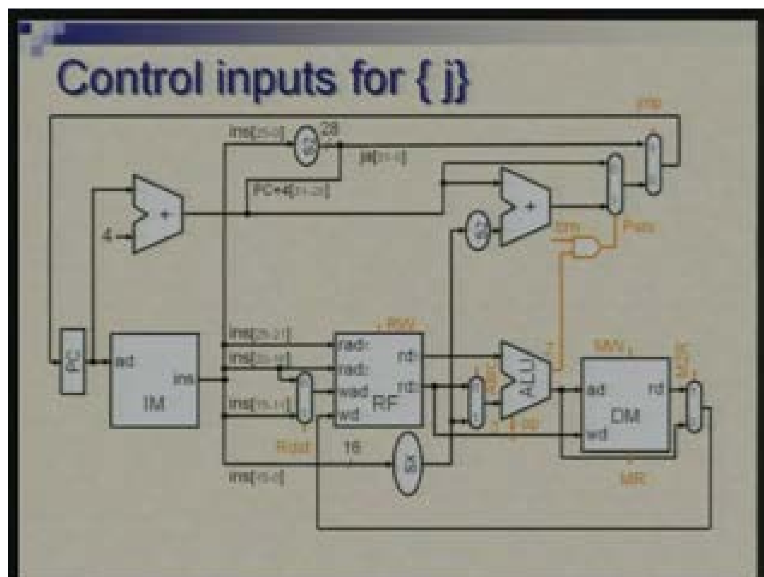
Next we move on to beq instruction.

(Refer Slide Time: 19:17)



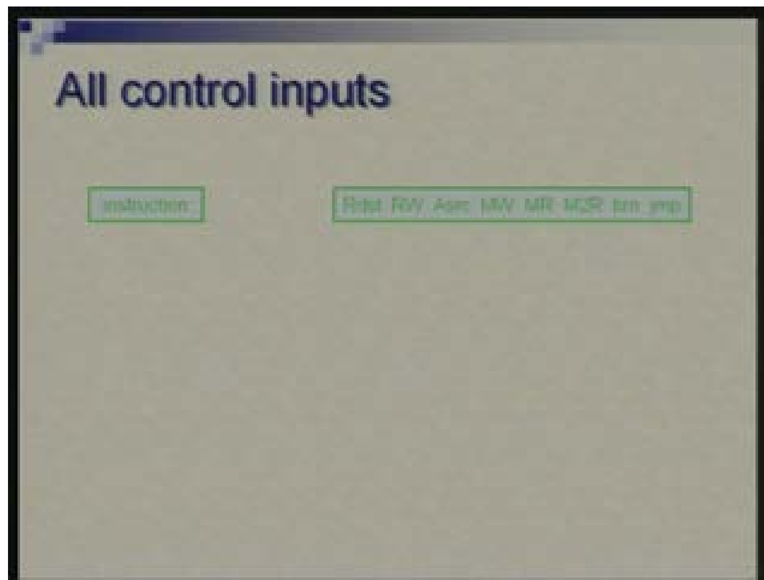
In beq instruction neither we write in to RF nor we read or write from data memory so those corresponding signals will be 0, Rdst is accordingly x, RW is 0, ALU source is 0 we need to take two operands from register file and compare them; MW MR both are 0, M2R is x, brn is 1 it is a branch instruction now. So now the z output of ALU will be brought into consideration and that together with brn will control PC source; jmp continues to be 0 and all values put together are in this green table (Refer Slide Time: 20:02).

(Refer Slide Time: 20:06 min)



Finally we have this jump instruction. So, jump instruction again does not require anything RF, ALU, DM all these are inactive so Rdst x, RW 0, ALU source is also x now. Since we are not looking at we are not choosing ALU it does not matter what you give as input, MW is 0, MR is 0, M2R is x, brn is also x. Now why it is x is because once this last multiplexer this control by jmp where we have made 1 you are looking at the upper inputs so it does not matter how the bottom input is being computed; you may select any of these it does not really matter. So this is the set of control inputs for jump instruction. Let us now put these together, well this is just a sub-table for jump.

(Refer Slide Time: 21:07 min)



We will tabulate the instruction and against them we will have all the control signals. So, if we put the opcodes of these instructions from those opcodes we need to derive these control signals and these two together will form a kind of compact truth table for controller circuit. And we know how to once the truth table is given we know how to design a combinational circuit. There are many ways to do it but I am sure you at least know one. Therefore, let us a put in the opcodes for these instructions.

(Refer Slide Time: 21:48)

All control inputs

instruction	opcode	Rdst	RW	Asrc	MW	MR	M2R	brn	jmp
R-type	000000	1	1	0	0	0	0	0	0
sw	101011	X	0	1	1	0	X	0	0
lw	100011	0	1	1	0	1	1	0	0
beq	000100	X	0	0	0	0	X	1	0
j	000010	X	0	X	0	0	X	X	1

All R-type instructions have 0 as opcode; these arithmetic logic instructions; this part is common and that is why we are treating them together (Refer Slide Time: 21:54). So there are also opcodes I have put for sw, lw, beq and j. So what are the opcodes are? We are just taking from whatever definition we have so there is nothing we are specifying here; this is what is given.

so now here is a…. So, in the overall design there were two control boxes; this is the main controller and after this now we can look at the second controller which controls the ALU. So as far as this is concerned we have completed the design up to this point, we have come to the truth table.

Now we first defined what is how we are encoding opc. opc if you recall, is a signal coming from the main controller and going to the second controller which controls the ALU. We said that we will have all instructions falling in three categories: R instruction is one category, load store is another category and all the rest are yet another category. And basically I should say branch is another category, for jump it really does not matter. So we need 2 bits to encode those three groups and what we will do is we will put in another column here which will define this.

(Refer Slide Time: 23:29)

Encoding opc

instr	opcode	opc	Rdst	RW	Asrc	MW	MR	M2R	brn	jmp
R-type	000000	10	1	1	0	0	0	0	0	0
sw	101011	00	X	0	1	1	0	X	0	0
lw	100011	00	0	1	1	0	1	1	0	0
beq	000100	01	X	0	0	0	0	X	1	0
j	000010	XX	X	0	X	0	0	X	X	1

So, arbitrarily let us choose some code; R-type is a 10, sw is 00, so is lw, beq is 01 and jump it is don't care. Now the truth table is actually complete. the main controller now as you can see has 6 bits as input; the 6 opcode bits are input for this and you have 2 bits of opc plus 8 bits in the other table altogether and these 10 outputs have to be produced so it is a 6 inputs 10 outputs circuit which we now need to design. But before we do that let us see what do with opc. So we need to look at opc and the function field in case of R-type instruction and decide 3-bit input control for ALU.

(Refer Slide Time: 24:30 min)

ALU control input

type	opc
R-type	10
R-type	10
R-type	10
R-type	10
R-type	10
sw	00
lw	00
beq	01
j	XX

Hence, now the group of R-type instruction needs to be expanded; we need to consider requirement of each individual instruction. So we repeat this R-type entry for five times because we are considering five instructions: add, subtract, AND, OR and slt. So these will differ in terms of their function field; opcode filed is same for these. The third column shows individual instructions and the last column shows the function field. the function field is not to be seen for instructions other than R-type instruction so I just put don't care there because in those particular bits we have something else; we have address offset or some part of address and they are not specifying function bits.

Now, for different instructions we need to figure out what input we need to give to ALU. For the second control circuit, for the ALU control circuit there are 8bits of inputs: 2 bits of opc and 6 bits of function and there are 3 output bits which we will decide now by first noticing what is the function ALU is supposed to perform in different instructions. So, in the R-type instruction the action to be performed is directly given by the instruction as add operation for add, subtract for subtraction and so on; for store and load instructions we also need to perform add operation and for beq instruction we need to perform subtract operation
and for jump we need nothing.

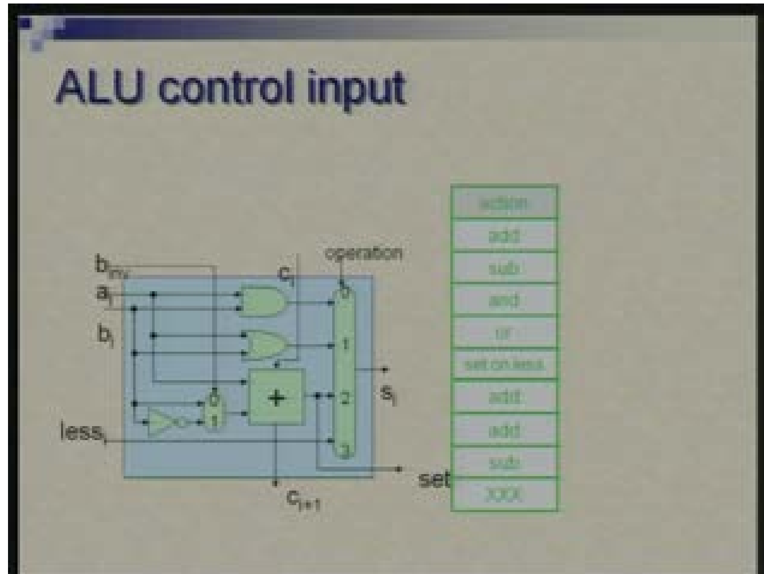
(Refer Slide Time: 26:13 min)

ALU control input

type	opc	instr	function	action
R-type	10	add	100000	add
R-type	10	sub	100010	sub
R-type	10	and	100100	and
R-type	10	or	100101	or
R-type	10	slt	101010	set on less
sw	00	sw	xxxxxx	add
lw	00	lw	xxxxxx	add
beq	01	beq	xxxxxx	sub
j	XX	j	xxxxxx	XXX

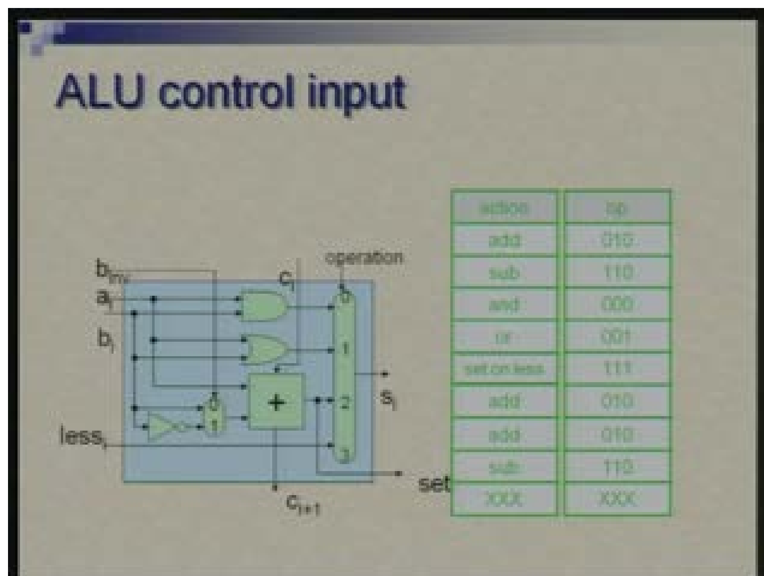
So we have enumerated the action required by ALU for these different instructions and we also we we are now in a position to define what control input is required for ALU. I am recollecting the ALU design, 1-bit of ALU is being shown here and you would notice that we need 2 bits to control the output multiplexer; it has to select from one of the four possibilities and 1-bit is actually used to invert b if necessary. Whenever you are doing a subtraction operation then b needs to be inverted. So we need to form 1's compliment and give initial carry so that negative of b comes so 3 bits are required.

(Refer Slide Time: 27:08)



And we can I am assuming that the left most bit out of the 2 bits is b invert and the other 2 bits control the multiplexer. So 00 if you want is the top input, 01 if you want the next input, 10 and 11. Therefore, for AND operation b invert is 0 but actually one could put it as don't care; if you are not using the adder part of it b invert becomes don't care but I am just putting it as 0. So for AND we are selecting the first or the zeroth input in the multiplexer so the code is 000; for R I am keeping b invert as 0 and the multiplexer is controlled by 01 and so on. So based on this the way we have designed ALU we can actually define what is the control input required for this case and with this our picture is complete.

(Refer Slide Time: 28:17 min)



(Refer Slide Time: 28:18)

ALU control input

type	opc	instr	function	action	op
R-type	10	add	100000	add	010
R-type	10	sub	100010	sub	110
R-type	10	and	100100	and	000
R-type	10	or	100101	or	001
R-type	10	slt	101010	set on less	111
sw	00	sw	xxxxxx	add	010
lw	00	lw	xxxxxx	add	010
beq	01	beq	xxxxxx	sub	110
j	XX	j	xxxxxx	XXX	XXX

Now, here we have the complete table. This is the truth table for the second controller or the ALU controller where we have 8 inputs 8-bit input and a 3-bit output. So one stage of design actually finishes here. Now it is a very straightforward, almost a mechanical process that once you are given a truth table how do you put down a circuit which will implement that truth table.

given a truth table you can actually write sum of product expression for it and put necessary AND gate, OR gates; you can also follow other realizations as product of sum, or NAND NAND or NOR NOR and so on. But one approach which is commonly used in a case like this when you have multiple outputs and for control design typically that is the situation. We follow a PLA based design where the design is actually derived in a very straightforward manner from the truth table itself.

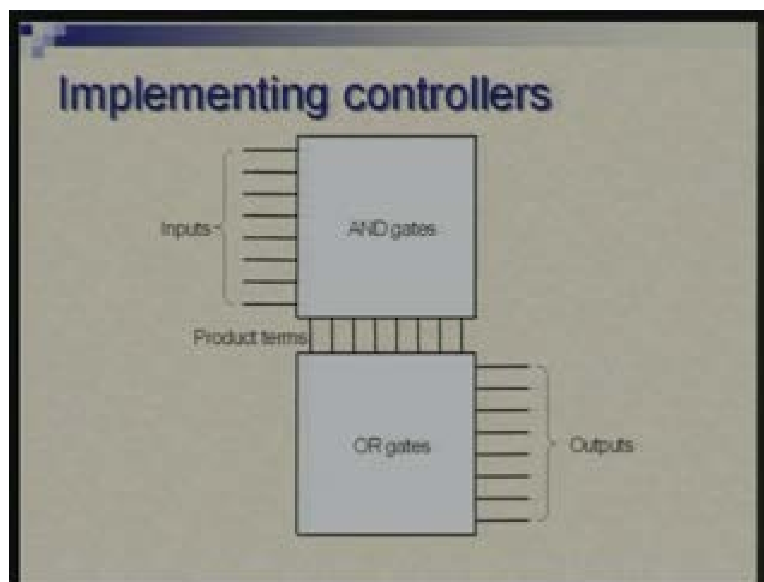
PLA basically stands for Programmable Logical Array. Logic array means that you have an you have a array of gates which are…… and this structure is somewhat universal, you can program it to implement any desired function. The PLA which will implement this would be if very large. I will not describe the PLA which will implement this but I will try to illustrate the basic principles, what is the basic idea behind this approachable design.

(Refer Slide Time: 28:19 min)

ALU control input

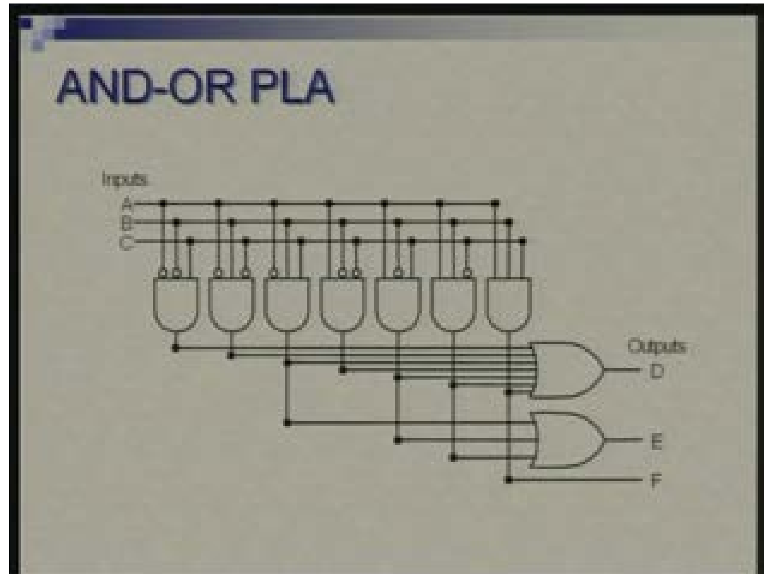
type	opc	instr	function	action	op
R-type	10	add	100000	add	010
R-type	10	sub	100010	sub	110
R-type	10	and	100100	and	000
R-type	10	or	100101	or	001
R-type	10	slt	101010	set on less	111
sw	00	sw	xxxxxx	add	010
lw	00	lw	xxxxxx	add	010
beq	01	beq	xxxxxx	sub	110
j	XX	j	xxxxxx	XXX	XXX

(Refer Slide Time: 30:08)



PLA has two parts: one is called AND plane and the other is called OR plane. which corresponds to this This is a basically with AND OR type of implementation or sum of product implementation. So the AND plane has a row of AND gates which implement all the product terms and OR plane has a set of OR gates which implement the sum terms. So the input goes to AND plane, the output of AND plane is a set of signals representing the products and the OR plane actually sums those products to form the sum of products and gives the final output. So the structure This is the overall outline and what goes inside the plane is a direct correspondence of what you have put down in the truth table.

(Refer Slide Time: 31:18)



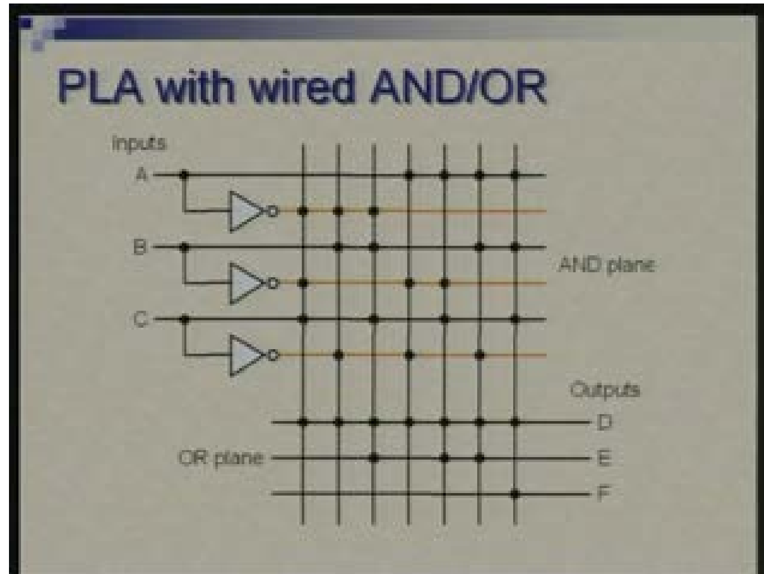
Let me illustrate; so, for a simple situation let us say you have 3 inputs called ABC and 3 outputs DEF; you have a set of AND gates which would implement the products of which are required in that implementation. So each gate here is capable of taking every input either in true form or in complimentary form. So now the connection of each gate to the inputs is what is programmable. So, if you are talking of M input and N output PLA then each AND gate will be an N input AND gate. that means it can you can form an arbitrary product of size up to at most M that means each literal for example literal means A or A bar B or B bar and C or C bar so you can form…. what you see here is that the left most AND gate is forming the product A bar B bar C; the next one is A bar B C bar and so on.

In general you can form power terms which are of less than M inputs also so the sum of these may not be connected and therefore a PLA just needs to form enough number of terms so that all your outputs can be realized. And then you have a row of OR gates which will pick up necessary required subsets of these product terms and sum them.

In this case D is summing the output of all the product terms, E is summing three of them, F is summing one of them and this pattern of what gets connected to the AND gates, which input gets connected to which AND gates in what pattern could be worked out in a very convenient form from the truth table. And similarly, which product term goes to which OR gate is also very easily derivable.

Now, in actual practice the gates do not look like this. these gates are actually what is called wired AND gates or wired OR gates and in real practice they may not be actually AND gates and OR gates you would either have NAND NAND organization or NOR NOR organization but for conceptuality let us continue to think in terms of AND and OR and the structure may look more like this.

(Refer Slide Time: 34:01)



So the top part is the AND plane and the bottom part is OR plane where the number of vertical lines or each vertical line represents one product term. And you can see that each vertical line is crossing every input and its compliment. so here you have three inputs and therefore the vertical line which correspond to AND term or the product term are crossing all possible input lines and their compliments so you you can wherever there is a dot put here it means that the connection is formed.

So, given certain number of inputs outputs in product term the overall structure is same; to go from one functional specification to another functional specification all you need to is place your dots differently. You move these dots around different power terms get formed and different product terms get summed so it is a kind of universal or programmable logic where by placing these dots you can form the required connection and get the required function out.

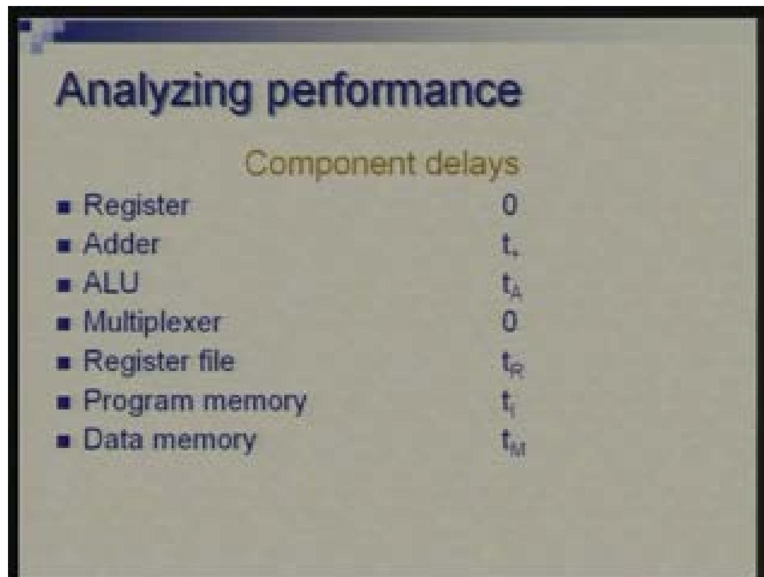
So, in actual practice the AND gates are formed actually vertically so they are distributed….. I am not showing that exact circuit; at each cross point there will be a transistor which will connect to the crosswire or you will not connect. So I suppose we leave those details to course on Digital Hardware Design which we might do later but just in overall functionality you understand that it is a programmable structure which can be automatically generated from the given specification.

Having done that now let us a look at what good work we have done. We have got a design now how does it fair in terms of performance. So, for doing so we need to calculate that delay and their impact on the overall clock period. So, first of all, let us look at the component delay so that we will build it bottom up.

We have register one register in this case PC for which we will assume that delay is 0. Actually just to simplify the analysis we will look at the delay of some prominent

components take them as nonzero others we will take as 0. So register delay we will assume as 0, adder delay, now I am referring to the adder which does PC plus 4 and the one which does PC plus 4 plus offset. So let us call let us denote that by t plus the delay of ALU we will denote by t A, delay of multiplexer we will assume again 0, one does not have to assume this; I am just doing this to make the expressions or analysis little simpler. but of course the assumption is a rational assumption in the sense that the delay delays which are actually small compared to the others are being assumed to be 0; register file t R, program memory t i, data memory t M and then finally the bit manipulation components has again there are only different wirings we take it as 0

(Refer Slide Time: 37:38 min)



Strictly speaking, even the interconnecting wires will have some delay and if your gates or logic is too fast then wire delay becomes comparable or significant. But again for simplicity we will ignore that.

So now, in terms of these parameters can we determine the clock period? Yes, if we analyze the paths we can…… what we will do is, you recall that I mentioned, at one edge of clock you have a new address in PC from where the cycle begins, the data flows through various components and ultimately you write something in the register file or write something in memory and additionally write something in the PC at the next clock.
So the clock period has to be sufficient to allow all these information to propogate to the destinations.

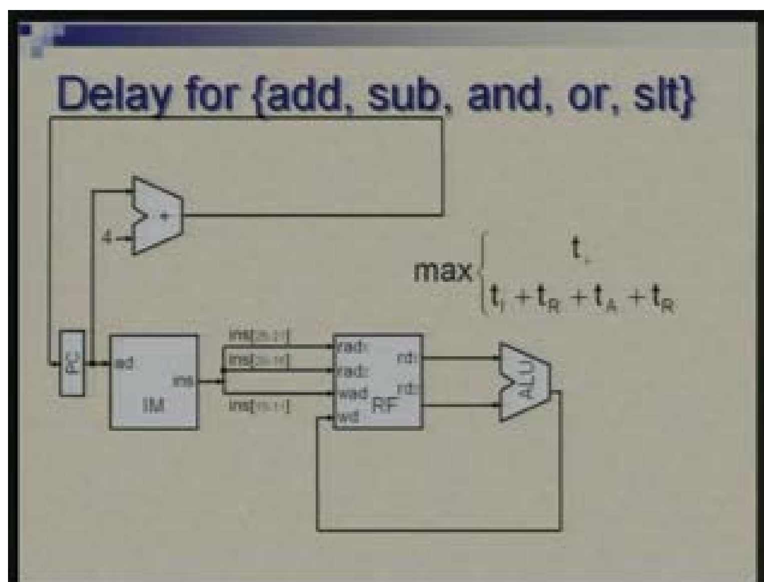
Once again… yes, [Conversation between Student and Professor……(38:37)…..] Sir, what is the difference between register delay and register file delay?
Well, register file is an array is a collection of registers and register delay I am talking of single register like program counter. Now, register file has additional circuitry that you are giving an address that address is getting decoded one of the register is getting selected

and there is some multiplexing at the output end. So depending upon the size of register file how many register it is it will have some delay. Of course it is not difficult to see that the delay of memory would be comparatively much more than delay of register file. And, in fact it could be larger by one or two orders of magnitude. But we will assume that, for the moment assume that the memory is reasonably fast and it is not……. if that was the case then if you take normal memory the bulk memory which you have in the processor and take the access time of register file we can probably set everything to zero except for the register file except for the memory.

In actual practice what would be influencing the clock period of a processor is not the main memory but cache memory. So we will worry about those details later; at the moment let us assume that the way we have been designing; we have put memory as part of the processor design which is an over simplification. But let us assume that it is the same technology which is being used to build other components of the processor and the memory and all the delays are comparable. So the way I am looking at now in the sceneries is that all the parameters which I have put as nonzero or somewhat comparable.

Therefore, we will again to simplify the problem look at each instruction or group of instructions separately, see what is the demand on time placed by those instructions and then put the results together. So if you would consider add, subtract, AND, OR these instructions then there are two paths we need to worry about: a path which is computing the sum or difference of AND and OR which goes through instruction memory, register file, ALU and back to register file so there are t i plus t R plus t A plus another time t R these will all these will get cascaded so all these gets summed; the other path is going through the adder to compute PC plus 4.

(Refer Slide Time: 41:28)

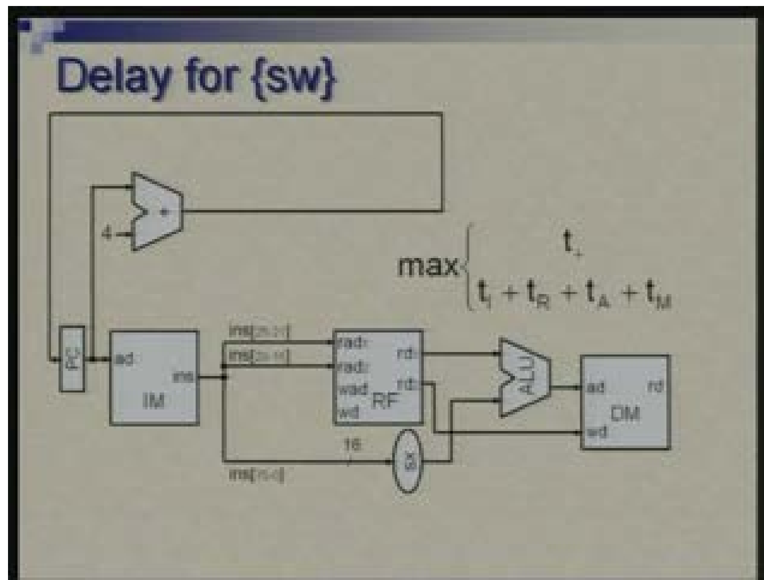


So, now if I am saying that all are comparable then the first one is actually meaningless, you know, I put it straightaway and neglect that and the second expression will dominate.

But let me be a little open here and say that the clock period is max of these two (Refer Slide Time: 41:49). The reason I want to keep t plus in picture is that as far as ALU is concerned, since ALU timing is getting summed with other things I will have to make an attempt to make this ALU very fast otherwise things will be back.

So it is typically in the ALU I will put carry look at logic and stuff like that. Whereas the other adder has much more room much more let us say cushion so I can afford to have a slow adder here because that is going to cost to me less. Therefore t plus could be; I can afford to have t plus larger than much larger than t A. you would recall that the delay in carry look at addition and delay in carry propagate addition they could differ quite vastly; One is proportional to 1 and the other is propositional to log N so the difference between them could be large.

(Refer Slide Time: 42:45 min)



For sw what comes into picture is t i t R t A and then t M. Now, incidentally here I am assuming the same time for reading and writing both in case of memory and in case of register file. In general they need not be same but again just to keep things going out of proportion I am trying to I am just using the same value for each. t plus again remains another path; for lw the chain seems to be longest t i t R t A t M and again t R.

(Refer Slide Time: 43:20 min)

Delay for {lw}

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

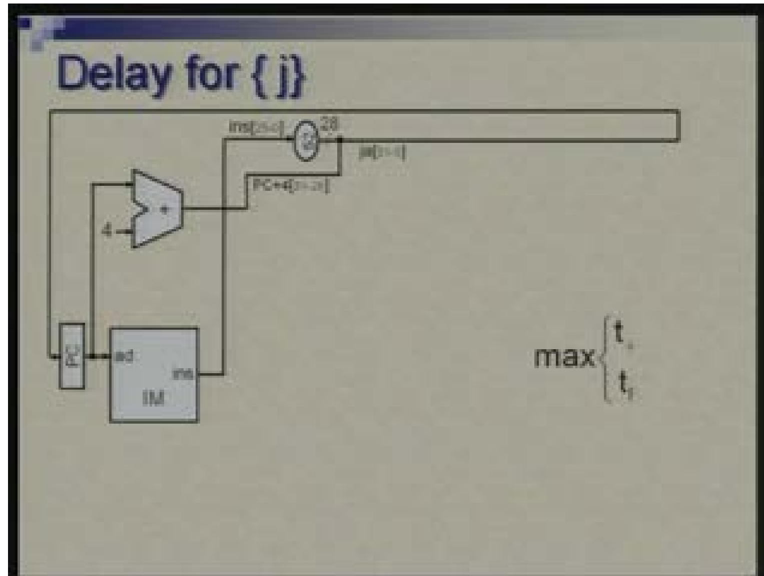
(Refer Slide Time: 43:32 min)

Delay for {beq}

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

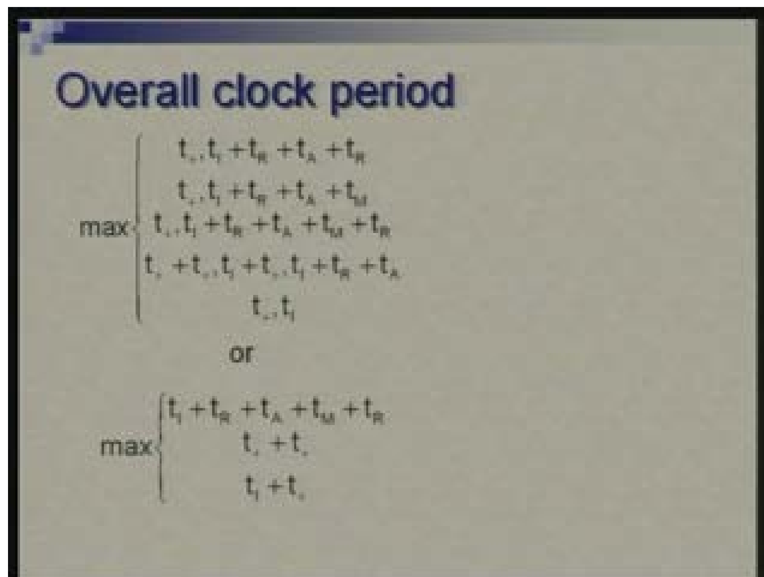
For branch there are three paths I need to consider. one goes through both the adders so t plus and t plus; another path goes through t i the instruction memory from where we are picking the offset and then adding it to PC plus 4 so it is t i plus t plus and for the third one the way the comparison being done is t i t R and then t A so whichever is the largest of these will dominate.

(Refer Slide Time: 44:05 min)



For jump it is t plus or t i. Now let us put…. basically now what we will say is that the clock period has to be large enough to accommodate all these possibilities. So it is max overall this max which I can just put as max of all these terms.

(Refer Slide Time: 44:26)



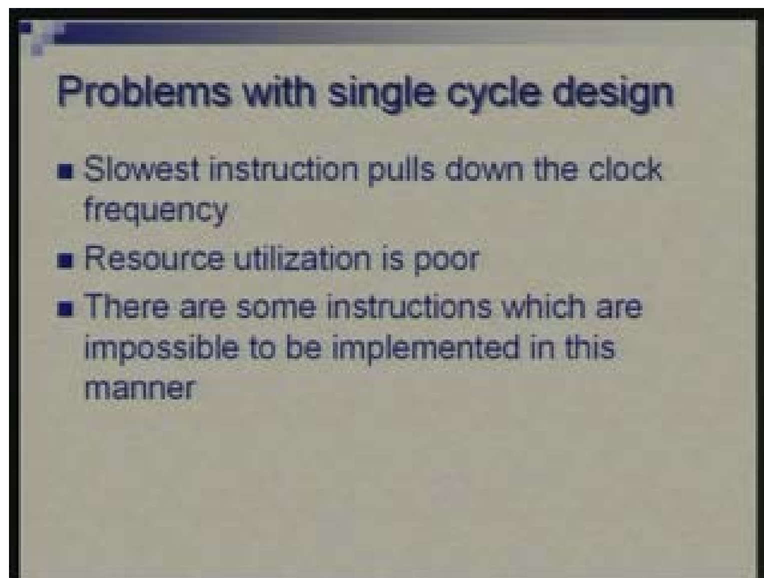
So each row in the upper group of expression each row corresponds to one instruction or its group. So t plus t i plus t R plus t A plus t R these are coming from R - class instruction, second line is for store, third line for load, fourth line for beq and fifth line for jump.

Now, looking at all these together now something can be thrown off because they are the other things which are clearly dominating those. So t plus alone can be thrown out because there is a t plus plus t plus. The long expression we have for load word lw dominates the big expression we had for add subtract and also one we had for store word. So this reduces without any assumption to the bottom expression where we are saying it is max of three of these expressions. The first one is coming from load word and the next word is coming from beq.

if you If all these terms each individual factor each individual t were comparable then basically let us say each of these were 1 nanosecond just for the sake of arguments then your clock period would be 5 nanoseconds which means 200 MHz clock is what you can learn with. So this is how the things stand at the moment. we will We will see what we can do to improve things here. But let me just summarize now.

What we observe over this is that the slowest instruction is pulling down the clock frequency or clock performance which would most likely would be the load word instruction. The other thing which are not directly obvious and I am not pointed them so far is that resource utilization is poor. We have ALU sitting at some place but we have put two adders for doing an operation which ALU could have done. So question is a design possible where we do not need those extra adders; can ALU do everything without loss of performance. So we would need to answer that question. Then the third question is it possible to do any instruction in this particular manner; given any instruction it may be complex, we have taken very few simple instructions is it possible to do all instructions in a single cycle the answer is again no. There are some instruction which may necessarily require one to take multiple steps and go through multiple clock cycles.

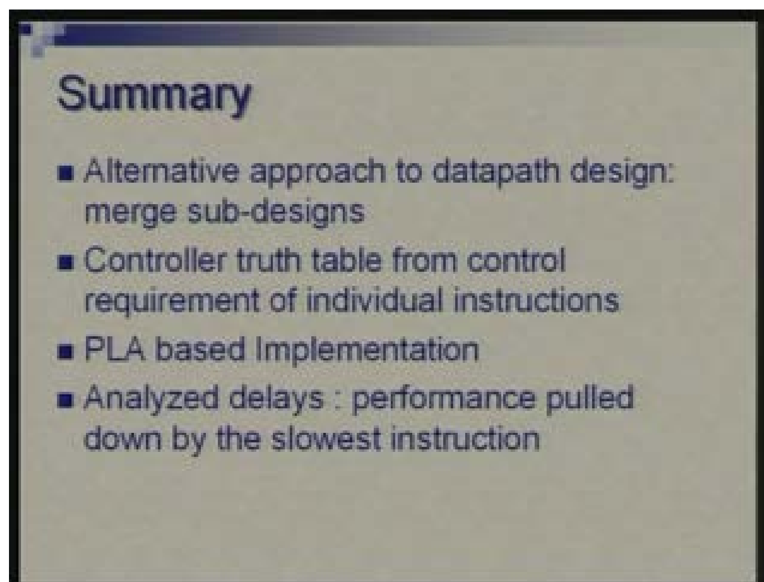
(Refer Slide Time: 47:16)



For example, if there was some instruction which reads from memory and also has to write in to memory with the given constraint of data memory with single port where there is a single address we can either do read or write or in best case you can do read write but from the same address. On the other hand, if you have an instruction which requires reading from one address, doing something and writing in to another address that certainly cannot be done in a single cycle. Or complex instructions which require moving a block of data from one area in the memory to another area would require several reads several writes or instructions which require in similar manner multiple operations which have to be necessarily sequenced. There is a fundamental limitation. So, because of various reasons we would need to go for different designs and try to address all these questions.

So I will close with summarizing what we have discussed today.

(Refer Slide Time: 48:28)



We first of all started with a different approach to arrive at the same data path design and approach was basically to have several simpler design and merge them together; resolve conflicts by putting multiplexers wherever was necessary. Then we examined the control requirement of various instructions, identified the value of control signals which need to be applied, tabulated them and came up with the control parts specification as truth tables. We looked at a possible way of realizing which is one that is one possible way of doing it PLA based and finally we analyzed the design from performance point of view, looked that how different paths are getting formed through which data has to flow what are their implications what is its implication on the performance and we notice that in this particular design approach the slowest instruction pulls on the performance of everything and in the subsequent lecture we are going to see how to get rid of that, thank you.