Digital VLSI System Design

Prof. S. Srinivasan

Department of Electrical Engineering

Indian Institute of Technology, Madras

Lecture - 21

Microprogrammed Design

Slide – Summary of contents covered in previous lecture.

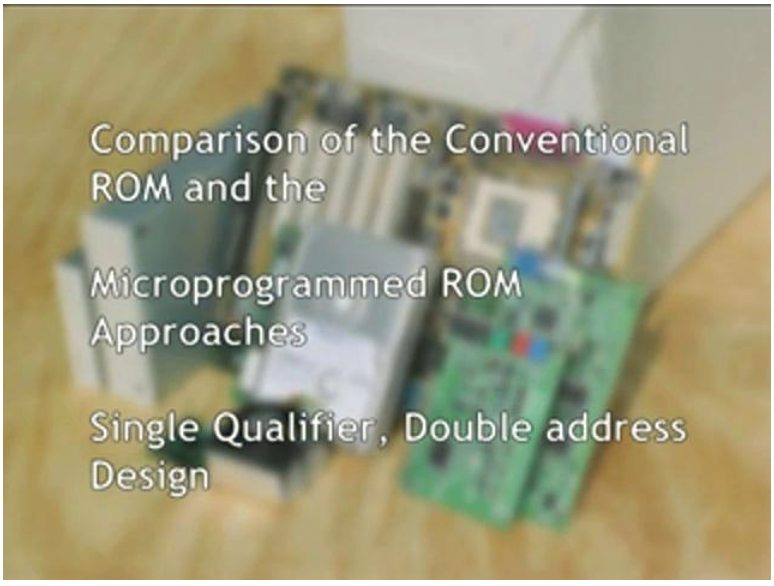
(Refer Slide Time: 01:09)

Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:31)

Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:50)



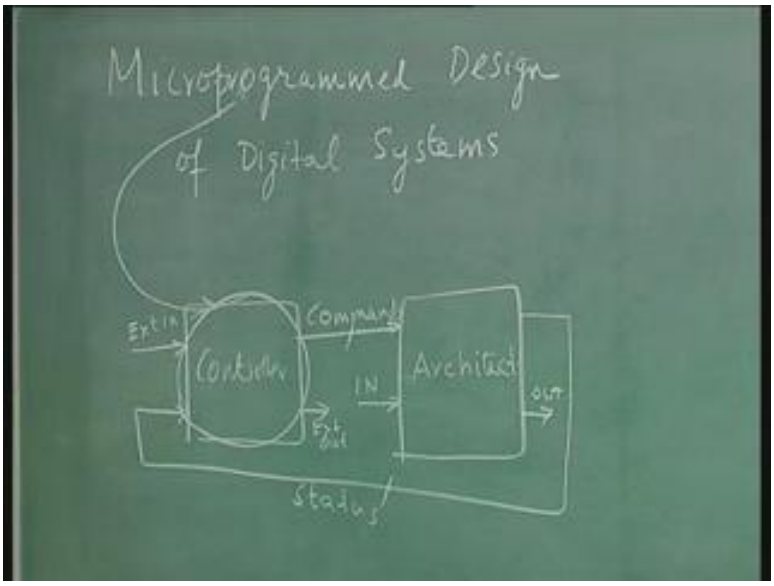
We have seen several methods of design and implementation of digital systems. First, we need to define the system, identify the parts required, then identify the control signals required, partition the system into architecture, and a controller and for the controller you do an ASM and implement the ASM. There are several methods of implementing ASM.

We had talked about ASM using multiplexers and flip-flops. We talked about ASM using ROM or PLA and flip-flops. When you have a digital system whose specifications may change a little bit, it is difficult to change once you design the hardware. You have to scrap the whole hardware and re-do it all over again. Some sort of programmability is desirable. ROM based implementation of controllers for digital systems is very good in that sense, it is programmable. When you say ROM based, really we mean PROM based- Programmable Read Only Memories and PROMs are very good programmable devices.

But, the problem with PROM based design is the table becomes extremely large. It is tedious. I may have let us say 5 or 6 inputs and let us say if about 10, 12 states, it means I need 4 flip-flops and 7 or 8 inputs let us say. For every state I need to know the input corresponding to the status of each of these inputs. So my ROM table becomes extremely large. First of all, it is tedious to make a ROM table involving all the conditions for each of the inputs in each state and secondly the size of the ROM becomes extremely large. ROM based approach suffers from the tediousness of the design and also because of the large size of ROM required for implementation. Of course, the cost of ROM those days used to be very high and that was one of the considerations but today the size of memories is not at all a consideration because memory chips are available very cheap. Cost per bit of memory is very low, but still why do you want to have system whose memory is very large when you can do it with a much smaller sized memory?

So, we go for another approach, a programmable approach using the ROM; because as I said, the alternative programmable is hardware. The hardware approach lacks flexibility and programmable approach using PROMs has the flexibility but PROM is not a very efficient implementation. We want to retain the programmability of the PROM and the cost effectiveness and the compactness of the non-programmable approach.

(Refer Slide Time: 05:52)



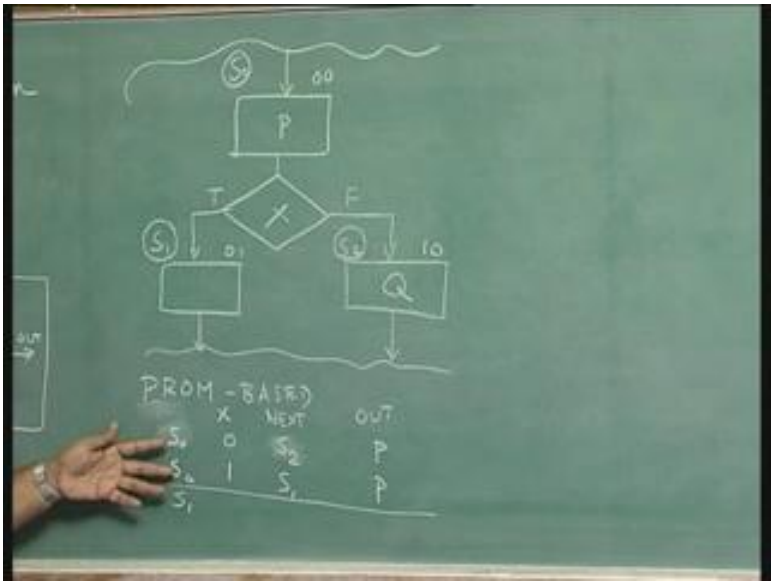
We have an approach called Microprogrammed Design of Digital Systems. I am not going to go to the details of the hardware because, we remember that we had partitioned a system of any individual system into a block called architecture or the functional block and a controller block. We send signal from controller to architecture and got the signal back from architecture in the controller. These are called commands, these are called status signals, these are more than one signal and then they may have external inputs and of course, they also can have external inputs and outputs. When I say digital system design I am not going to go into all these again because we have already seen the examples of all these things (Refer Slide Time: 07:20). When I say Microprogrammed Design of Digital Systems I am going to concentrate only on this block.

Microprogrammed implementation of the controller of a digital system; when you have a controller, you assume there is an ASM already written. A controller starts with a controller design; start with an ASM. ASM stands for Algorithmic State Machine. I am not even going to tell you how to do that because you have seen many of those examples; what is an ASM chart, how do you draw it for a given controller, take into account the input output considerations of the controller. Assuming we have an ASM instead of a multiplexer based approach or a gate based approach or a PLA based approach or a PROM based approach, how are we going to do a microprogrammed based approach?

The discussion of today's lecture is limited to implementation of the ASM of a digital system using a microprogrammed approach. What is this microprogram? Why is it called microprogram? Because it has a programmability built in that is why it is called microprogrammed. Why is it microprogrammed? When you say program, you normally understand a high level computer program like C or even in a microprocessor, you talk of an instruction like assembly language programming, these are instructions when you say add or subtract or multiply or load or store. When a program consists of all these instructions it is called a program. But each of these instructions has to be carried out using several steps; each step of this instruction execution is called a microinstruction.

When you are talking of a digital implementation where the circuit goes from one state to the next state based on an input being available or not available, this intermediate step for a minor step which is required in order to carry out a major task is called a microstep and that is why this approach is called a microprogrammed approach. Otherwise, it is nothing like a programmable approach. So, what does it mean?

(Refer Slide Time: 09:54)



I have an ASM; normally a state, let us call this state S_0 and an input; let us say X. In this state the circuit remains in this state and input X is tested. If X is true it goes to a state we will call S_1. If it is false, let us say it goes to S_2. Each of these states can have outputs in

an ASM state which we write it inside the box. Let us say there is an output of P for this, Q for this, no output for the particular state ASM - you do not write anything in the box - and you write the binary representation of the state variable. Call it 0 0, 0 1, 1 0. This is only a section; P can be reached from some other place, from here it may go somewhere else, we are not interested in all that. We are interested in this slice of an ASM chart.

When I say I have microprogrammed implementation of an ASM chart what do I mean? I will consider what is the address norm, what is the state in which the circuit is, what is the input that is being considered in the state, what happens with the control if the input is true, what happens to the control when the input is false and in each of these states what is the output? This is all the information we need in order to execute the ASM chart properly. To fulfill the controller properly, we need all this information.

In a PROM-based approach what we will do is, we will say the present state, let us say S_0 and say the input being tested X is 0, goes to S_2 and the output is P. If X is 1, it goes to S_1 and next state is again P and so on. Now as I have only 1 input so X is 0 and 1 then I can go to S_1 and start writing the table. But in practice, in the real system there are several inputs.

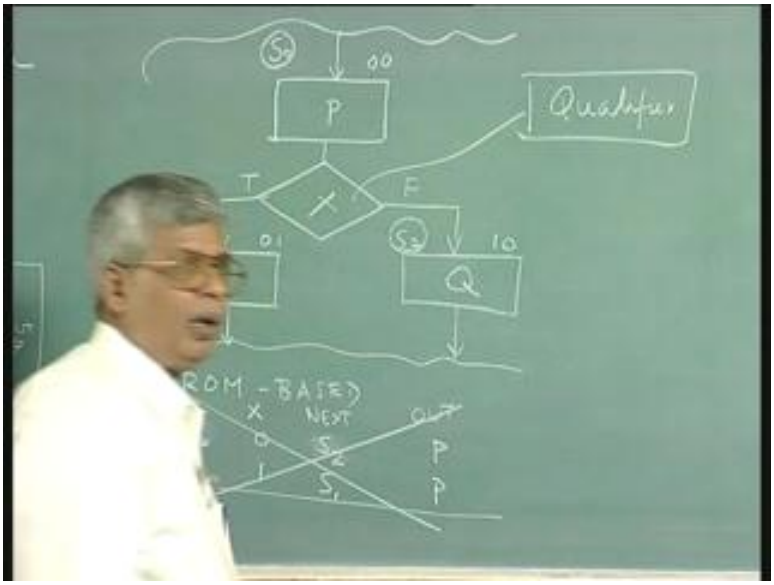
In a large number of inputs, all these inputs have to be given here. Supposing instead of 1 input X, it has a set of 7 or 8 inputs, all the 8 inputs have to be given here X, Y, Z, P, Q, R, S, T, whatever it is. If each of this input is 0 or 1, what is the next state? For 1 input, I have to take 2 rows of the ROM table. 2 rows of the ROM entry are required for 1 variable. If there are 2 inputs there will be 4 rows required. The number of rows in the ROM will depend on the number of inputs in the system, so that if there are 8 inputs for example, each input will have 2 to the power 8 that is 256. Like that for each state, suppose there are 12 states, the number of rows in the ROM table will be 12 times two to the power 8 that is 256 and it becomes an enormous table.

But most of the times in a 0 state the only input of significance is X. There may be Y, Z and so many other inputs; but, these inputs are not considered instead of S_0. But because you have to give the complete table of what happens in state S_0 when input X is equal to 0, Y is equal to 0, Z is equal to 0 and all other inputs are 0, then what happens to the state

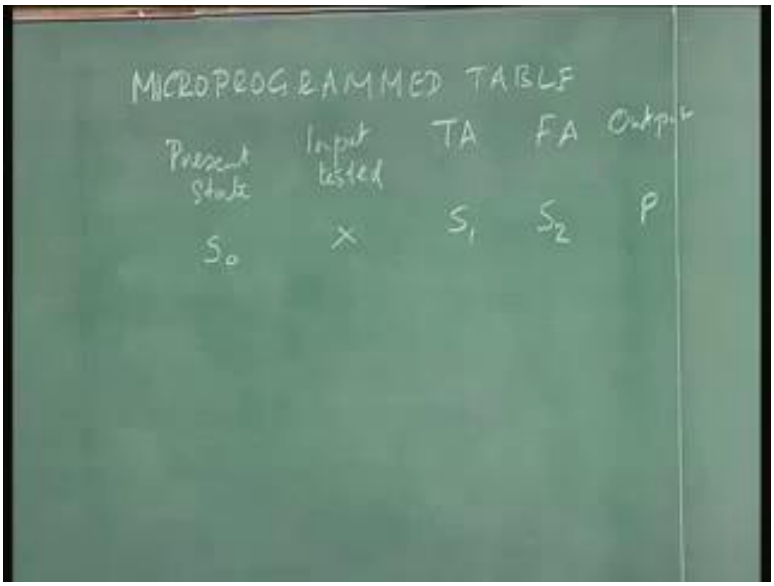
S_0 when x is equal to 1 and all other states are 0 and then when x is equal to 0 and y is equal to 0 and so on.

Keep on writing it even though we do not have any significance of them. The only variable of interest in state S_0 is X which is kept very easily written in 2 lines but because I have more than 1 input I need to complete a table. ROM is a truth table; remember that. ROM table is nothing but a truth table of a system. In a truth table I should give all possible input combinations whether they exist or not; whether they are practical or not. If the system has 8 inputs, all the 8 inputs in all the conditions should be available for each of the states. That makes the table very large and the ROM size very large -to make the table is cumbersome.

(Refer Slide Time: 16:05)



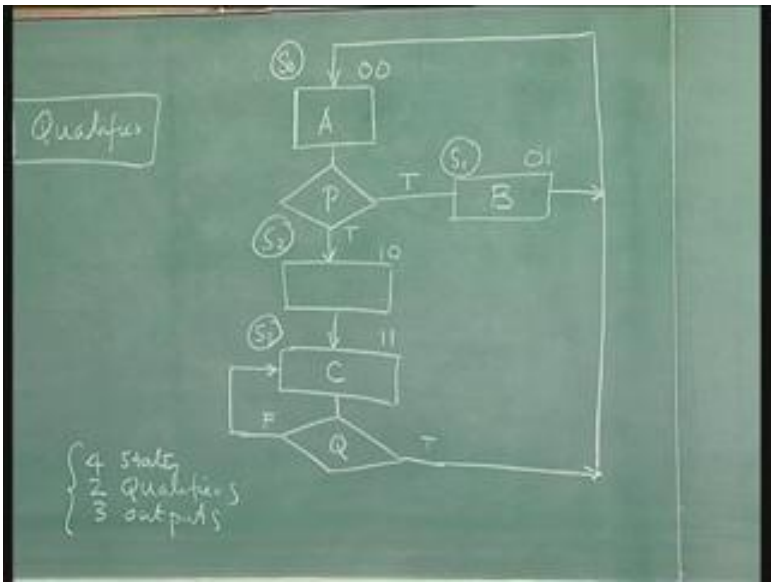
(Refer Slide Time: 16:12)



So in a microprogrammed approach what I am going to do is, I will replace this ROM table by what is known as a microprogrammed table. When I say the present state, input to be tested and if the input is true what happens and if input is false what happens? Present state is S_0 and input tested is X; if X is true it goes to S_1, if X is false it goes to S_2 and what is the output in this state? It is P. I am going to replace for each condition to be tested this condition to be tested is called a qualifier; it is a name for a qualifier.

Each qualifier to be tested will make only 1 row in the microprogrammed table. That means my memory will be much smaller now because I have to implement this table; earlier it was huge -so to make a microprogrammed table is very simple. More than making it, to implement the microprogrammed table is also very simple. So I am going to have a smaller size of ROM, smaller size of memory in a microprogrammed approach. Let me explain this problem with a simple example: we will consider an ASM chart like this.

(Refer Slide Time: 18:16)

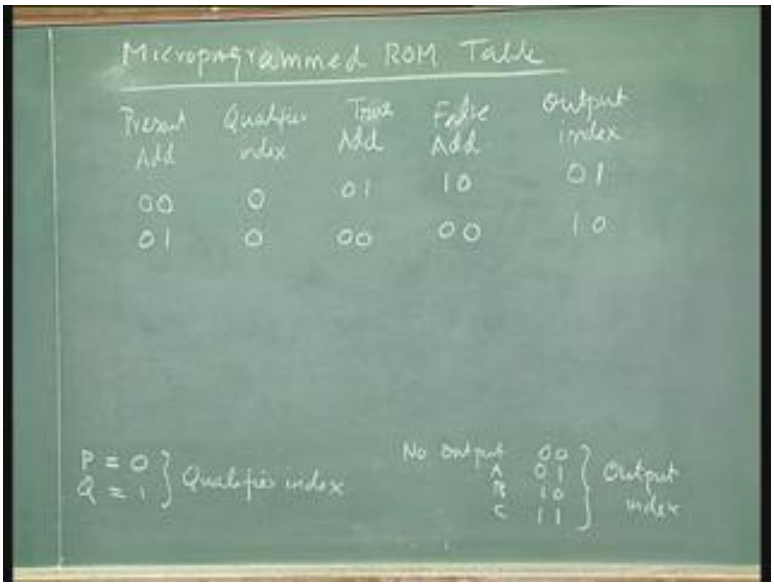


I have an ASM chart with 4 states. To make it very simple at the same time make it very clear. I have chosen an example with only 4 states. Let us call the states S_0 S_1 S_2 S_3 as usual. Let us say there are only 2 inputs in this: P and Q are the qualifiers. If P is true it goes to S_1 and the cycle is from state S_0 straight to state S_1. If P is false it goes to state S_2 from S_2 to S_3, there is an unconditional transition. In S_3 qualifier Q is tested. If Q is false it remains in S_3; if Q is true it goes to S_0 (Refer Slide Time: 19:48). It is a very simple ASM to illustrate the method of microprogramming an ASM chart for implementation.

I have 4 states S_0, S_1, S_2, S_3, we will assign binary values 0 0, 0 1, 1 0 and 1 1 in a natural binary sequence. There are 2 qualifiers P and Q, Let us call the 3 outputs A B and C: Output of S_0 let us call A; output of S_1 as B; output of S_3 as C and S_2 has no output. That means I have here a table with 4 states, 2 qualifiers and 3 outputs. How to do a microprogrammed table out of this? Once you have a microprogrammed table, that table is implemented with the ROM, so it becomes a microprogrammed ROM.

So instead of making a generalised or straight forward ROM table, we will do a microprogrammed ROM table. That is all.

(Refer Slide Time: 21:36)



Here I am going to have the following things for Microprogrammed ROM Table: The circuit is going to be in one of the present states, so I will call this present address. There are only 4 present addresses; binary value 0 0, 0 1, 1 0, 1 1 or if you want to write in decimal 0, 1, 2 or 3. Next, I am going to have the next qualifiers to be tested which are P and Q, so I need an index. The qualifier to be tested has to be given a binary value. What do you mean by qualifier index? This qualifier index will tell you which of the qualifiers is being tested in the current state we are considering. Considering state S_0, qualifier tester is P; in state S_1 no qualifier is tested; in S_2 no qualifier is tested; in S_3 qualifier Q is tested. We should have an index for that.

We will say P is equal to 0, Q is equal to 1, this is called qualifier index. If the qualifier is true what happens and if the qualifier is false what happens? If the qualifier is true it goes for 1 state; if the qualifier is false it goes for another state. I should have a true address and false address. What are the outputs? Outputs can be again coded.

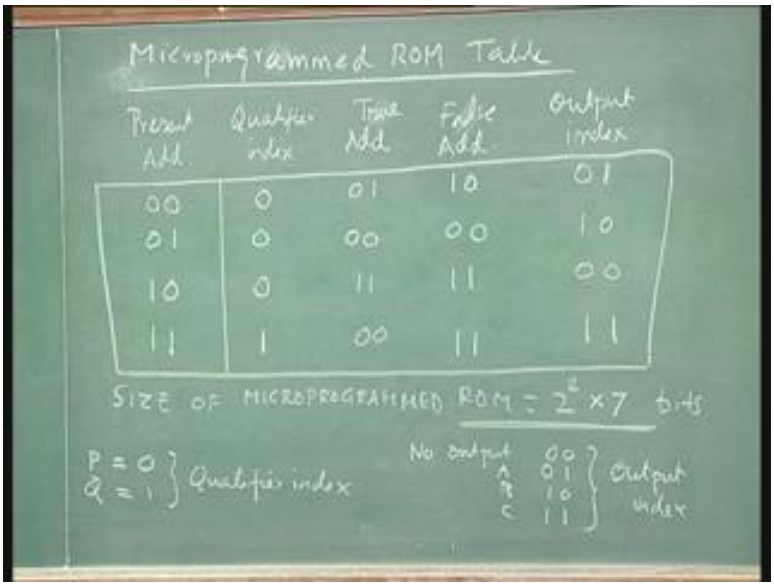
There are only 3 outputs A, B, C. I can have an output index; no output is 0 0, A is 0 1, B is 1 0, C is 1 1. This is called output index. My job is very simple; I am at present address 0 0, qualifier being tested instead A is P which is serving an index as 0. If it is true it goes

to 0 1 which is the true address. If it is false it goes to 1 0 and output index is A which is 0 1.

That is all, this state is considered. Earlier if it had been a ROM based approach I would have written with A is equal to 0 0, if P is 0, Q is 0 what happens, if P is 0 Q is 1 what happens, if P is 1 Q is 0 what happens, if P is 1 Q is 1 what happens. I should have written four rows. Now instead of 4 rows I am writing only 1 row. That is the simplification I am talking about, both in the design as well as in the hardware. Then going to the present address 0 1.

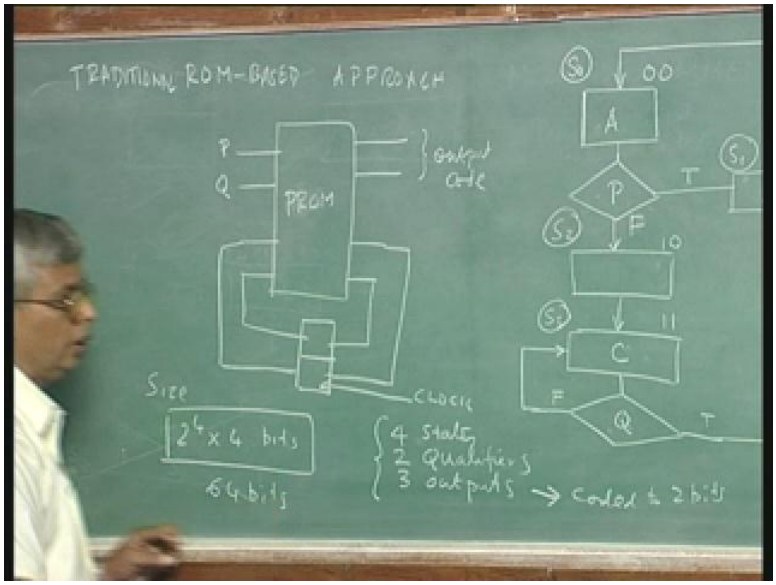
I will write it in the natural sequence, what is the qualifier being tested? In state S_0 no qualifier is tested because if circuit is in state S_1 it has to go to S_0 no matter what. There is no qualifier to be tested. When there is no qualifier to be tested I do not have any index for that. But what I can do is I can use any index; I can either test P or Q, so I will put P is equal to 0 as the index to be tested. Make sure both for true address and false address you write 0 0. What I mean by this is: in circuit when it is in S_1 we may have a qualifier whether it is true or false the address is going to be the 0, the next state. From S_1 it goes to S_0 that means from 0 1 it goes to 0 0, irrespective of whether the qualifier index is present or not, then output is B which is 1 0.

(Refer Slide Time: 28:56)



Now the next state is 1 0 that is present state S_2. Again no qualifier to be tested and both cases the addresses to go to 1 1 and there is no output. Finally S_3 which is 1 1; qualifier to be tested is Q whose index is 1. If it is false it goes to 1, if it is true it goes to 0 0. Output of state S_3 is C which is 1 1. This completes my microprogrammed ROM table. I need to implement this in a ROM. I give the present address and as soon as I give the present address it has to give the qualifier index is 0; it has to give this true address value, it has to give this false address value and it has to give the output index. So what will be the size of my ROM? Size of microprogrammed ROM …this is my address, these are the outputs; so 4, 2 power 2, 2 address lines, 2 times the width of the ROM is 1, 2, 3, 4, 5, 6, 7; is equal to 2 to the power of 2 times 7 which are the number of bits, the width of the ROM.

(Refer Slide Time: 30:39)

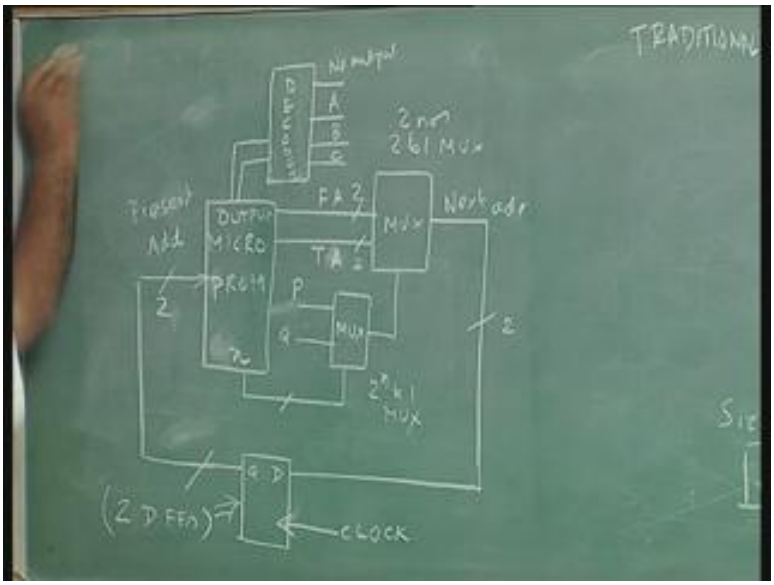


In a traditional ROM approach, these 4 states would have given 2 qualifiers and 3 outputs, if I implemented this using a ROM to try to give you a traditional ROM based approach: 4 states will require 2 variables, 2 inputs, 2 qualifiers that is my P and Q. It will give me the output in the next state and outputs can again be coded (Refer Slide Time: 31:49). If I want I can code it, the output code, into coded to 2 bits.

Size is 2 to the power of 4 times 4 bits which is 16 times 4 is 64 bits. Here it is 2 to the power of 2 times 7 which is 28 bits. We can see the difference; 2 power 2 is 4, 4 times 7 is 28 bits in size; 2 power 2 is 4, which is 16 times 4 is 64 bits. This is only for 2 inputs. Remember, as the number of qualifiers increases the address lines increases; the address lines come in exponential. Size of the ROM is determined by the exponential address line. On the other hand, when I have more qualifiers, only the qualifier index is going to increase. Even with the 2 qualifier example, I am sure I have shown you the size of the ROM is considerably small, but as the number of inputs increases there is an exponential increase in the ROM size in the traditional approach; whereas in the microprogrammed approach it is only linear increase.

There is a lot of difference between the linear increase and an exponential increase and an exponential increase will very soon reach the very large size which is unmanageable both in terms of desire and cost. As I said not only it is the cost and the size, it is also the programming inconvenience. To write a ROM table with so many inputs becomes a nuisance and tedious. Now, what is the hardware, now of course this is the hardware for ROM based approach. What is the type of hardware for the microprogrammed approach?

(Refer Slide Time: 34:30)



Hardware for the micro programmed controller: I need to have a ROM of course which is where we are going to store the microprogrammes called the microprogrammed ROM. I will call it MICRO PROM. Input address is given; in this case there are 2 address lines in this particular example. The size will vary from problem to problem depending on the number of states and number of input qualifiers. Then we have the qualifier index given to a multiplexer; a qualifier index we will call n. The qualifier index chooses the qualifier; the qualifiers are in this case P and Q. The qualifier index is given to a multiplexer, which chooses between the 2 qualifiers P and Q and the value of P and Q is given to another multiplexer which is given the true address TA and the false address FA. If this is false it will select this address, if it is true it will select this and this becomes the address of this ROM for the next state.

Given 2D flip-flops; present address, next address and clock. The output index goes through a decoder into no output, A output, B output, C output. These are 2 addresses, so this has 2 lines, P and Q are qualifiers with value true or false that is selected by this multiplexer and this multiplexer gives the new address, next address is also 2 bits and that is stored by these 2D flip-flops here and given as next address. So there is an extra logic. This is the PROM? Here we have the flip-flops and the PROM; 2 bits of 2 bits no problem and the output code has to be here also in output code to be DECODER. So that part is also common. What is extra? DECODER, no output A, B, C .what is extra in this? 1 multiplexer with n inputs and n control inputs and 2 to the power of n inputs and another multiplexer with 2 inputs but it has be series of multiplexers because there will be a 2 address so they will be a 2 of this, this is 2 into 2 to 1 multiplexer 2 numbers, this is 2 to the power of 1 multiplexer, 2 to 1 multiplexers - 2 numbers.

These are the two extra things. On the other hand, I say the size of the ROM even for a 2 bit as I said, even for 2 qualifiers, the size is less than half and if it is going to be more qualifiers the size is going to be enormously small at the cost of some extra multiplexers. But more important than that is the versatility, programmability. I can do it very elegantly as it is an elegant design.

This is the final circuit of the Hardware for Microprogrammed Controller. And as I said n is the qualifier index, TA for true address, FA for false address. Elegant design, fine -we will have to see a practical example, we will see next time. Before that I want to tell you one more thing: This true address and false address; 2 addresses I have to give for each line. If the circuit is in present state and if qualifier is true it goes here, if the qualifier is false it goes here.- 2 addresses Even when there is no qualifier to be tested I have to give a true address and a false address, 2 fields both the same. For example, this does not have a qualifier in S_0 0 1, we do not have a qualifier but still we put an artificial qualifier as 0 and give the true address as 0 and false address also 0. Giving this an extra field of the false address can be removed if I can make some minor changes in my ASM.

To make a minor change in my ASM, I can make 1 address as an increment of the previous address, that is: 1 added to the previous address and the other address is a different address. I can eliminate 1 true address field completely. Of course, the size of the ROM is going to be same interval where the address is 2 to the power of n. But the other 1 into 7 bits, the number of output bits can be reduced by removing 1 address from the original design. This approach is called single qualifier double address implementation in which there is a true address and a false address. I can remove 1 of the address and change it a single qualifier, single address microprogrammed design which means I can eliminate 1 of these 2 columns and reduce and shrink the size of the ROM. Further, I need to make a minor change in the ASM; I am going to remove a binary addresses. That is all. I am not going to do anything else because, I cannot change my ASM. ASM is what I have to implement.

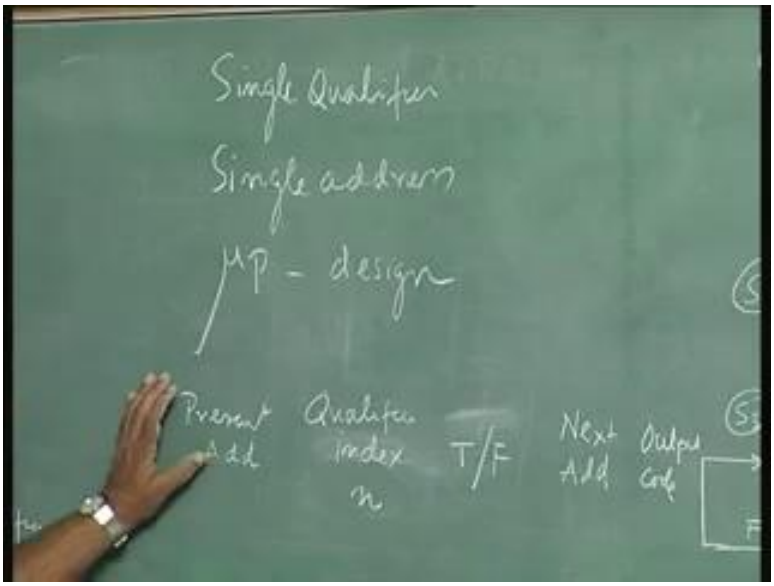
What I am going to do though, is to change the addresses as we know. I am going to call them differently, I am going to call this address 0 0, S_1 I am going to call 0 0, this one I am going to call 0 1 and this one I am going to call 1 0. Size of the ROM has been reduced because the number of address lines of a microprogrammed ROM is much smaller compared to a conventional traditional approach. Having done that I want to reduce the size of the ROM in the sense,the width of the ROM, the word size for each address how many bits are there? In this case it is 7; I want to see if I can reduce it. That I can do by reducing one of the addresses; instead of storing 2 addresses for each of the

present addresses, I will make it only 1 address. That is what single qualifier single address. But I cannot change the ASM -ASM is what I need to implement.

We are going to make minor changes in the hardware in the addressing scheme. I am going to get 0 0, 0 1, 1 0, 1 1. There is logic here, this is not arbitrary. I will tell you how to do it next time. I am going to tell you how it is going to reduce the hardware and in the next lecture I am going to tell you the logic behind this and also the implementation. When there are no qualifiers we need only one address, we are writing two addresses unnecessarily. When there is no qualifier 0 1 or 1 0 we are unnecessarily reducing two addresses which are both same. I can eliminate one of them. There is no problem about states where there is no qualifier. For states for which there is a qualifier I want to make one of them one more than the present address. That is all I need to do. This state has a qualifier, A state S_0 as a qualifier 1 0 is next to 1 1.One of the addresses has to be incremented or 1 more to the present address only if there is a qualifier. If there is no qualifier that rule need not be followed. From B it goes to A, 0 0 to 1 0 no problem because there is no qualifier. Similarly, from S_2 to S_3 there is no qualifier so it goes from 1 1 to 0 1.

But in state S_3 if the qualifier Q one of the addresses has to be one more than 0 1.one more than 0 1 is 1 0. If the qualifier is true I go back here, so 0 1 becomes 1 0 if it is true and 0 1 becomes 0 1 when it is false. The rule is simple; whenever there is a qualifier make one of the next addresses one more than the present address. When there is no qualifier there is no such rule (Refer Slide Time: 48:50). If you make that rule very simple, my table will change.

(Refer Slide Time: 49:24)



I need to know the what is the present address, what is the qualifier index and what is the next address. I need to write true address or a false address so the next address is true address or false address and the output code.

So I am going to eliminate 1 address; next address is now only one, not next address true and next address false. I am going to eliminate that. I am going to increase 1 extra bit here so I may have to increase 1 bit but then the address is more than 1 bit. Generally, in this simple example itself it has 2 bits but in practical examples the addresses are large number of bits so there is no problem about that. But with 1 extra bit I am able to knock off the next address. For the same table, same ASM, how to write the revised microprogrammed ROM table for a single qualifier single address mode? We will see that and what is the hardware change we make. In the next lecture what we will see is how do you write the single qualifier single address table for this example and how do you implement it. What change in the hardware are you going to make? And after seeing that we will take one practical example of the one of the problems we have seen several problems earlier and do it again.

Thank you.

Summary of Lecture 21

(Refer Slide Time: 51:44)

(Refer Slide Time: 52:03)

(Refer Slide Time: 52:28)

(Refer Slide Time: 52:48)