Digital VLSI System Design

Prof. S. Srinivasan

Department of Electrical Engineering

Indian Institute of Technology, Madras

Lecture - 22

Microprogrammed Design (Continued…)

Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:09)

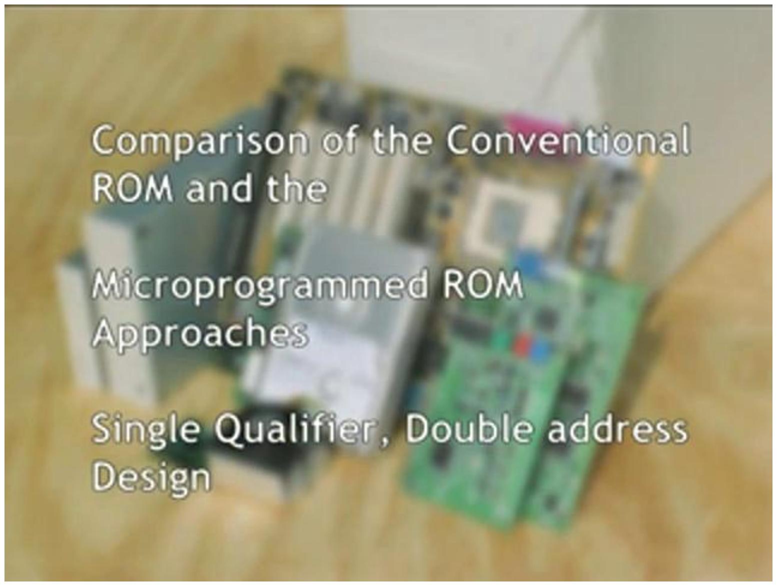
Introduction to Microprogrammed Design of Digital Systems

ASM Chart for a Microprogrammed Design

Microprogrammed ROM Table

Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:28)



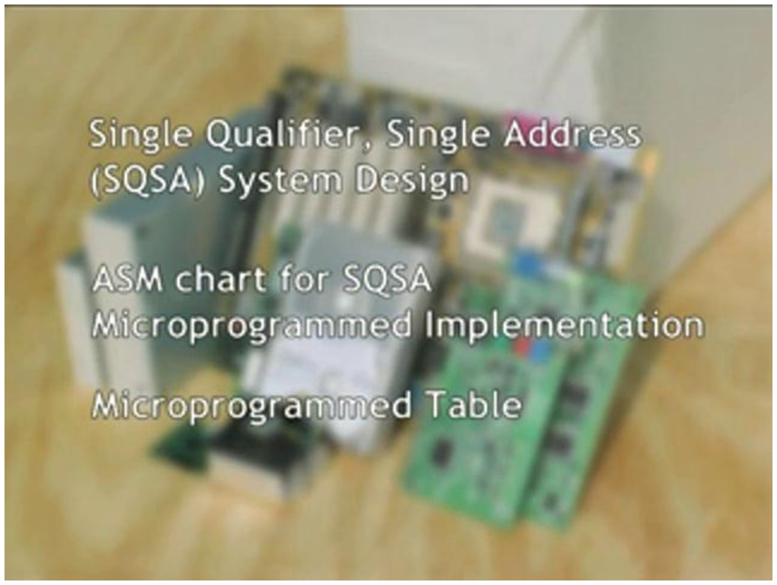
Comparison of the Conventional ROM and the

Microprogrammed ROM Approaches

Single Qualifier, Double address Design

Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:53)



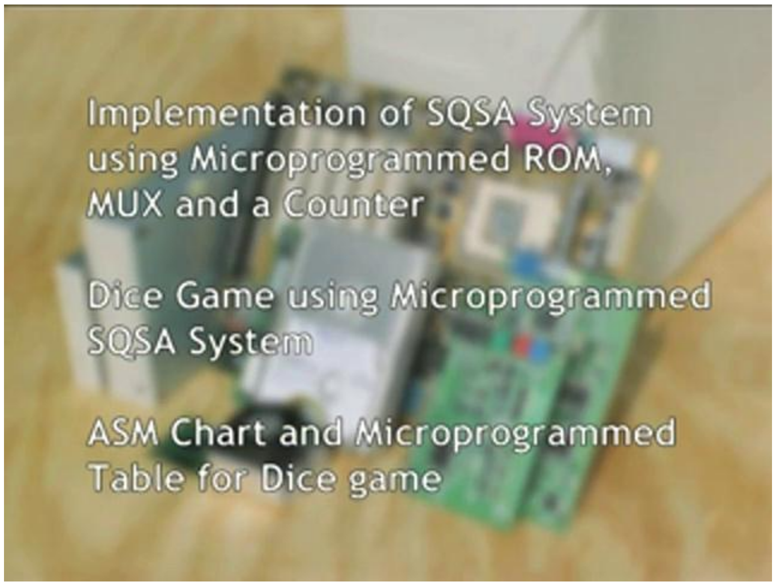
Single Qualifier, Single Address (SQSA) System Design

ASM chart for SQSA Microprogrammed Implementation

Microprogrammed Table

Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 02:13)



Implementation of SQSA System using Microprogrammed ROM, MUX and a Counter

Dice Game using Microprogrammed SQSA System

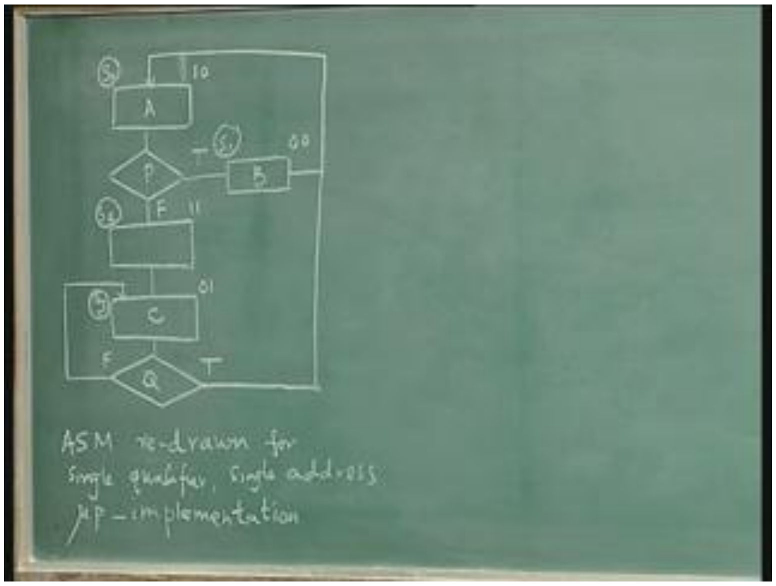
ASM Chart and Microprogrammed Table for Dice game

In the last lecture, we introduced the concept of microprogrammed design of controller for ASM. Any digital system can be partitioned into architecture or a functional part and the controller part. We had seen how to draw an ASM - algorithmic state machine chart for the controller. In many earlier lectures, we had seen how to implement the ASM using several techniques like gate based design, multiplexer based design, PROM based design and PLA based design - Programmer Logic Array basic design. In the last lecture, we introduced the concept of microprogrammed implementation with 2 advantages: one is a little flexibility in change, hardware versus programmability. In a hardware design, once you make a design if you make changes you may have to redo the design. A programmable approach is better. So, that way a ROM based approach is very good but the problem with ROM based approach is that the size of the ROM becomes very large when the system becomes large. If the number of states increases and the number of inputs increases, the size of the ROM increases exponentially.

On the other hand, we saw in the last lecture the microprogrammed approach which is also similar to the ROM based design. We do not give the ROM table or the microprogrammed table as we did in the conventional sense, where for each state we give the input conditions for all the available inputs. We only select such inputs which are going to affect the transition from the given state to the next state. That means you are not going to give the program table involving all the inputs. We only select those inputs which are going to affect the immediate transition and we saw that as the size of design increases, the increase in the hardware size of the ROM is only linear not exponential. It makes a lot of difference in terms of hardware. Not only the hardware cost which is not very much today, ROM cost is not very much but the tediousness with which you have to generate the ROM table will become extremely cumbersome if the number of states are very large and the number of inputs are very large.

We took an example of a simple ASM and saw how to implement it using a microprogrammed based approach where we saw that each state, we will test one of the inputs and sometimes no input is tested and decide if the input tested is true, what happens to the controller? Which state does it go to? If the input is false to which state does it go? We decide and make a table. In that approach we saw that for each state we test one qualifier sometimes 0 or no qualifier also has to be given. We have to give true addresses in a false address: that is, the next state when the qualifier is true; and the next state when the qualifier is false. We said we could do one more improvement in this design by not indicating both the false address and true address for a qualifier test. We will only give 1 of the 2 states and the other state will be an increment of the existing state. So at any given state, let us say 000, we will test a qualifier and if the qualifier is true one of the conditions is true or false we will make the next state as 001 and for other test the result of the qualifier remaining different then instead of going to 001 it will go to some other state. This means, I do not have to keep writing the next state which is an increment of the present state. I can drop that from my table and to that extent I can shrink the size of the ROM. This is where we stopped in the last lecture.

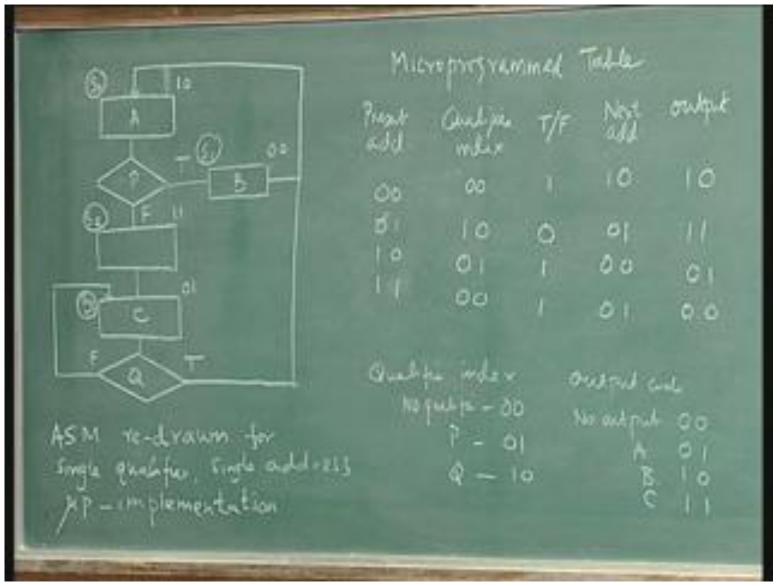
(Refer Slide Time 07:29)



We will take the same example of the ASM that we took in the last lecture where there are only 4 states $S_{0,}$ $S_{1,}$ $S_{2,}$ $S_{3,}$ 2 qualifiers P and Q and 3 outputs A, B, C. The only change I have made, in fact we did it in the last lecture itself is the assignment of the states. Earlier, I had taken arbitrary assignment; I had called 0, 1, 2, 3 corresponding to $S_{0,}$ $S_{1,}$ $S_{2,}$ S_3. Now I have made a slight change in this. Today I am calling this 0 0 state, this 0 1 state, this 1 0 state and this is 1 1 state. Why did I do it? I did it because whenever the qualifier is tested one of the next states should be an increment of the present state. For example, if the circuit is in the state $0_,$ P is tested, either S_1 or S_2 to which it goes, one of them should be next state to 1 0. If this is 1 0, this is 1 1. S_2 happens to be next binary state to S_0 in a binary sequence. Likewise, the other qualifier tested here is Q in condition S_3; it can be false or true. If it is false it goes back to 0 1. There is no way I can make this next state of 0 1 because this has to go back to this state. That means the next state when the qualifier is true from S_3 to S_0; this has to be an increment of this. If this is 0 1, this has to be necessarily 1 0 because 0 1 cannot go back to 0 1. There is no question of this being increment of this state whereas, in the other path this state can be increment of this state.

So, whenever there is a qualifier tested from a given state, make one of the next states as a binary increment of the present state. The other state can be anything. If there is no qualifier tested this condition does not apply. To that extent I need to make some changes in the ASM; which I have done.

(Refer Slide Time: 10:00)



Micropragrammed Table

Present add	Qualifier index	T/F	Next add	output
00	00	1	10	10
01	10	0	01	11
10	01	1	00	01
11	00	1	01	00

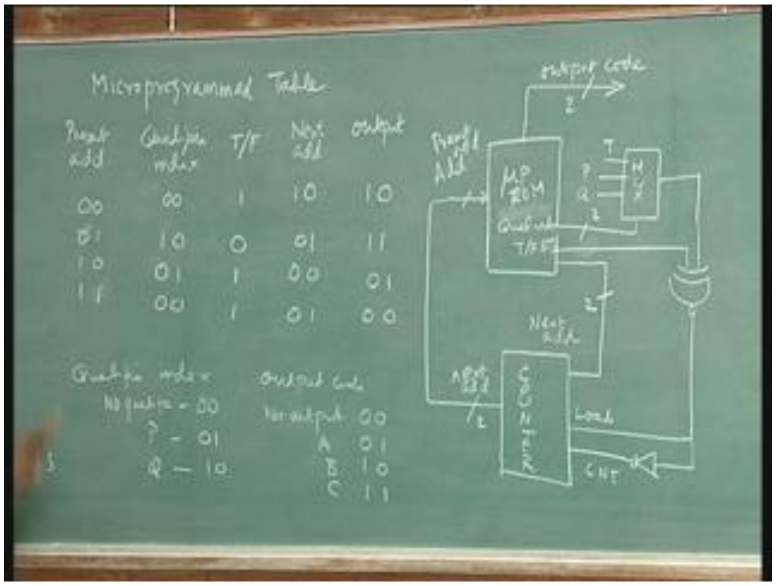
Qualifier index
No qualifier - 00
P - 01
Q - 10

Output code
No output 00
A 01
B 10
C 11

I can now rewrite the microprogrammed table with the present address, qualifier index and next state is only 1 now; the other next state is automatically increment. So I do not have to write 2 columns or I need not write 2 sets of next states, and then the output. Making use of the same type of definitions as in the last lecture, the qualifier index has only two qualifiers P and Q; in last lecture, we called P as 0 and Q as 1. Whenever there is no qualifier, we use one of them. But it is not possible because here I need an extra qualifier saying no qualifier. So I need to have one more index. This means I am going to make 0 0 no qualifier, 0 1 is P and 1 0 is Q. This no-qualifier condition was not required in previous implementations because I can always make one of those qualifiers as qualifier tested, even if there was no qualifier, to be tested. I made both the true address and false address same. I did not have to worry about the no qualifier case but here I have to worry. I have to put the qualifier index as 0, 1 or 2. Output code, we have already decided: no output is 0 0, output A is 0 1, output B is 1 0 and output C is 1 1. This is same as what we discussed in the last lecture. I purposely left a blank here (Refer Slide Time: 12:09) if you had observed that. I would like to know when a qualifier test done the next state is given here, whether that state will go if the qualifier is true or that state will go if the qualifier is false. I need to know whether the next status given by this table is a true address next state or a false address next state. That means I will have to put true or false bit. Now I will put the present states 0 1, 0 0, 1 0, 1 1. In 0 0 state no qualifier is tested; when no qualifier has to be tested the next address has to be 1 0; output in this state is B 1 0. I will come to this bit in a minute. This is 0 1, qualifier tested is Q and now we can see one more than 0 1 is 1 0 but I do not have to write it. What I have to write is what is not the incremental value which is 0 1 again; output in this state is C which is 1 1. Next, 1 0, state A or state S_0 qualifier tested is P which is 0 1. 1 0 goes to 1 1 you do not have to write it; 1 0 goes to 0 0, I have to write it because this is a non-incremental value, so 1 0 to 0 0.

Finally, F 1 1, qualifier tested is no qualifier tested and next state is 0 1. Output is nothing. Question is what do you write here? If it is true does it go to this state? If it is false does it go to this state? That is what I have to write. It goes to this state whether it is true or false. For example, take the qualifier to be tested 0 1, this state P; this state goes to 0 1, if this goes to 0 0, then 0 1 goes to 1 0 if the qualifier is true; you do not have to write it; it goes back to 0 1 if the qualifier is false. So we have to write here false. Likewise, when I am testing this condition in 1 0 state, qualifier tested is P which is 0 1. If it is true it goes to 0 0 which is a non-incremental value of this. If it is false it goes to 1 1 which is an incremental value, I do not have to write this but I have to write the non-incremental value, which is true, which is 1. What do you write in these 2 places? It is the true value we will write here because this is anyway a non-incremental value. This is what I have to implement, so my logic is going to be very simple.

(Refer Slide Time: 16:25)



Microprogrammed Table

Present add	Qualifier index	T/F	Next add	output
00	00	1	10	10
01	10	0	01	11
10	01	1	00	01
11	00	1	01	00

Qualifier index	
No qualifier	00
P	01
Q	10

Output code	
No output	00
A	01
B	10
C	11

I am going to have a microprogrammed ROM whose address comes from the present address. Of course, the output code will be there, 2 bits for output code. Earlier, I had a true address coming and a false address coming and one of them was selected using multiplexer. Now I do not have that, I will have only the qualifier index. The qualifier index, which is n plus1, where n is the number of inputs. Actually, it is the logarithmic value of the total number of inputs. There are 2 bits although there are 3 qualifiers there are only 2 bits; this goes to a multiplexer whose very first value is always true, that is, no qualifier because whenever we go to the no qualifier we have put a true value; that is the new value to be added is the true value. I need a true for the first one; rest of them will be the actual qualifiers namely P and Q. This is a MUX (Refer Slide Time: 18:04).

I have this counter here which will give you the next address which is 2 bits. This counter will produce either a new value or an incremented value. If the next state is incremented value of the present state variable it has to add 1 or if it is a new value it has to load a new value. So, the counter will have a feature called load and count. What is loaded is the value given by the next address. What is incremented you do not need; the present address plus 1 is the next address, so I do not have to load it and I do not have to count it. Loading will be this for the next address; this case again 2 bits; next address is loaded or it is counted. How you will decide that? That will be decided based on the bit being true or false. That means I need to have a comparator; I need to have an exclusive OR gate wherein I take this value from the multiplexer and compare it with the value which is the true false bit. In each case, we have to decide whether false is going to give the loading or true is going to give the loading and accordingly do this. This true false bit along with this will be compared with an exclusive NOR gate and that will be given to load and inverter will be given to count (Refer Slide Time: 20:54).

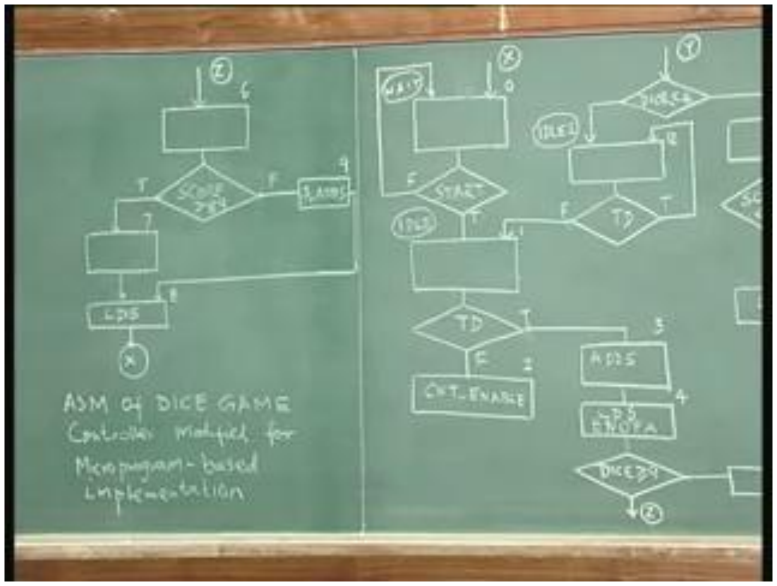
For example, (Refer Slide Time: 21:03) take the first row no qualifier, true false bit is 1. No qualifier index is 0 0. So when 0 0 index is presented here, so 1 comes here; true false bit is 1 from this table, so 1 1 is compared output is 1 when compared with exclusive NOR gate. So, 1 will load the next address which is this. On the other hand, let us take this. I take 0 1 the qualifier index is 1 0 which is Q and when Q is false true false bit is 0 and I need to load this value. If true false bit is 0 I do not need to load it; if true false bit is 0, this is a 0 and this is a 0 which are equal and I will load it. On the other hand, if the true false bit is 1, this is 0 always. If Q is 1 and this is 0, this will increment and will count. It will increment by 1 or it will load a new value.

This is as architecture we drew, in the last lecture, for the single qualifier double address; true address and false address. We have now reduced in this case; of course, we do not see any reduction very much because we have taken a trivial example of 4 states. So true address - 1 address was here another address was there. In the last implementation (Refer Slide Time: 23:00) we had this, this and this. This is same as this and output code is same. Next add 2 bits. We have 2 of this, 1 column is completely removed and instead we have an extra column of true false bit and the qualifier index, unfortunately, there are only 2 qualifiers 0 and 1, 2 qualifiers in double address case we need a qualifier address 0 or 1. I have increased 1 bit for the qualifier index and one bit for this so I have no saving. But imagine if the number of states is very large. Supposing, we had 5 qualifiers instead of 6 qualifiers. Earlier there were 5 qualifiers; they would have required 3 bits. Extra qualifier index being 0 0 0 will not add any more bit, so the number of bits increase is not there. One extra bit will be there for true false bit but if the next address is several bits long I will increase 1 bit for true false bit but save on one address. It looks like a trivial example because I do not see any appreciable change in the size of ROM. The reason is there are only 4 states and 2 address lines; 2 address bits are required to represent 4 states. Out of which one has to go for true false bit and this particular example happens to be one more bit has been added to the qualifier index because of this. Suppose there is no bit added to the qualifier index supposing the qualifier index was 6 earlier now it is 7 there is no problem. Suppose the number of bits for the next address is several bits then we have saved by adding 1 extra bit for true false. We will knock off entire address. So to that extent I can shrink the size of the programming ROM.

Basically, we have now given a tool by which an elegant implementation can be done. ASM is a must in any controller; any digital system requires an architectural controller which is best represented by ASM and ASM can be implemented anyway you like. You can use gates, you can use multiplexers etc. These are hardware examples in which no change is possible, especially gate based; if it is multiplexer based we can make some slight changes; if it is PROM based there is a sort of programmability but it is a cumbersome procedure because the size of the PROM becomes very large, the system is very large and writing the ROM table becomes a tedious affair.

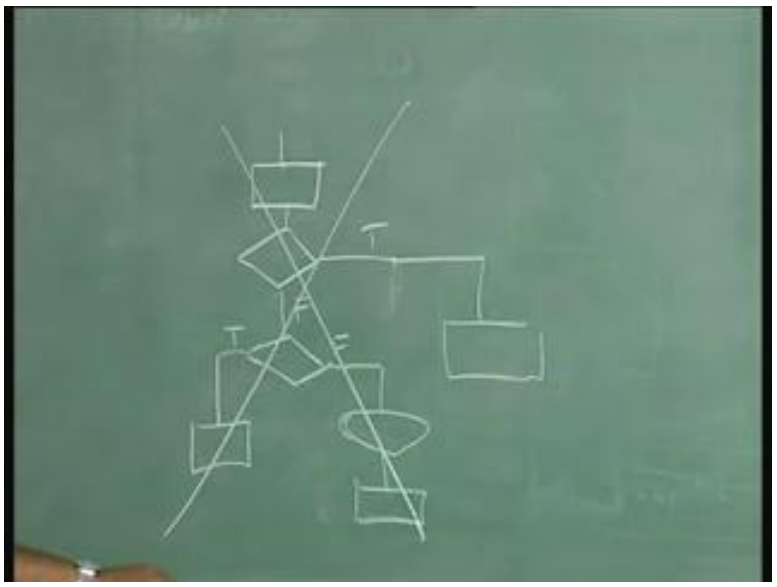
This microprogrammed approach takes the essence of the PROM based approach and simplifies the hardware and gives an elegant method. There are 2 types in which each of these we will draw ASM with 1 qualifier to be tested. The 0 qualifier is also considered as one qualifier because it is no qualifier, only 1 qualifier is tested in each; two addresses will be given namely true address and false address or we can give only 1 address and assume the next to be the increment of the previous address. These are the 2 options: one is called single qualifier double address that is what we did in the last lecture; what we see today is called single address single qualifier.

(Refer Slide Time: 27:55)



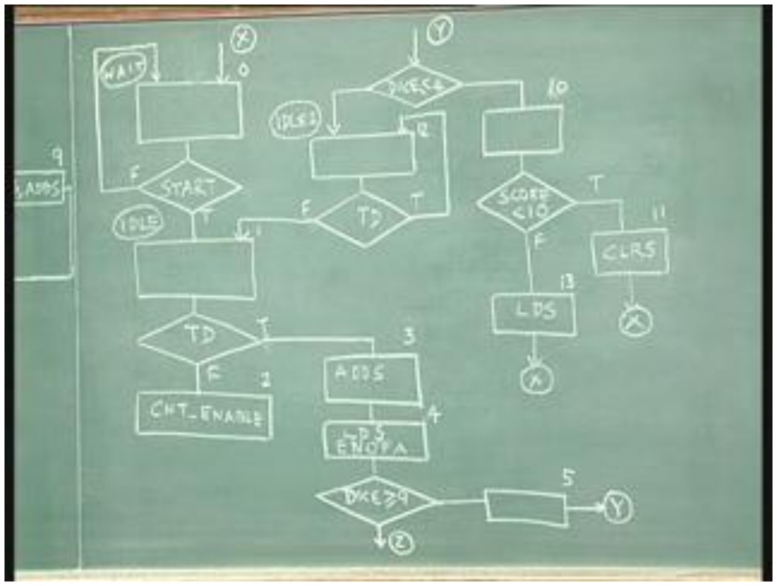
With this background, let us revisit an example. We have done several examples in our previous lectures and one of the very interesting and somewhat extensive example was a dice controller if you remember, we had a dice game. Let me show you the ASM of the dice game. I want you to revisit the dice game controller design using the ASM approach. Using PAL we have done, programmable array logic, in one of our earlier lectures. Instead of spending a lot of time trying to explain the problem I request you to go back and look at the problem definition, the specifications, the architectural requirements and the ASM of that. The same ASM has been redrawn here (Refer Slide Time: 27:52) for implementing as a microcontroller microprogrammed based approach implementation. Redrawing is necessary for the following reason: we can only test one qualifier at a time. In one given state I can test only one qualifier or 1 input; input is same as the qualifier.

(Refer Slide Time 28:20)



Earlier, in many examples, we have seen something like this as an accepted ASM. We know how to draw the next state table for this or the ASM table as we can call it. We can draw the ASM table for it and implement it using multiplexers or using ROM. But this is not allowed for a microprogrammed approach.

(Refer Slide Time: 29:30)



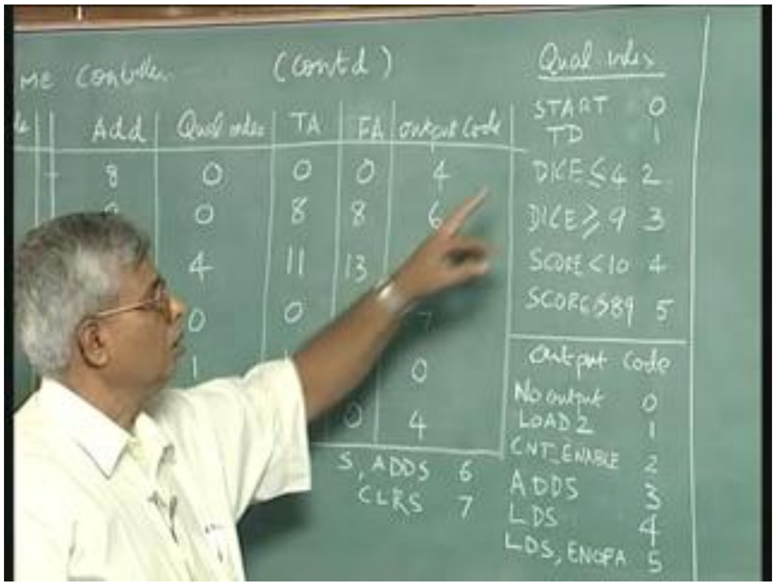
In a microprogrammed approach, I need to have only 1 qualifier for example I cannot have 2 qualifiers tested in this state. This is okay; but when it is false it should go to another state only then the next address can be true address or false address. That is what I mean by redrawing the ASM of the controller or the controller of the dice game keeping in mind that I can only test 1 qualifier at a time in a given state. The second change I have made is, I have eliminated all conditional outputs. In this state, if this is false and this is false, this is the output and then this goes to this state. So this can be state 1, this can be state 2, this can be state 3, this can be state 4. In state 1, if this is false and this is false this output is generated and then it goes to state 3. That conditional output has no place in a microprogrammed approach. So, all conditional output boxes have to be replaced by output boxes. In effect, we are increasing the number of states for 2 reasons: one is to introduce states between qualifiers. If a series of qualifiers to be tested in a given state we will have to introduce an extra state in between and whenever there is a conditional output I need to introduce an extra state. So I may have to introduce a few additional states. But remember, number of states increases is not going to increase the number of variables; variable increase is going to be logarithm to base 2 of the number of states.

Supposing we are having a 10 state example, I need 4 state variables. I can add another six states without changing the size of my ROM. Even if I go to 17 or 18 states all I need to this increase an extra state, that is, only 1 extra variable. I am not changing the number of qualifiers; number of qualifiers is what is giving you the enormous size of the ROM. Why are we getting a normal size of the ROM in a ROM implementation? It is because of the exponential power of the number of qualifiers. The size of the ROM is generally determined by 2 to the power the number of state variables plus the number of qualifiers. Here, the number of qualifiers is completely dropped from the ROM size; it only comes as a width as a qualifier index. By adding a few states extra you are not going to increase the size enormously. With that in mind, I am not going to go through these examples since we have already done this enough. Consider in great detail all aspects of this game. I can show you a couple of places where the state has been increased now. For example, in these state variables (Refer Slide Time: 32:25), I have given the binary decimal numbers to make life simple; from state number 4 to state number 5 there is a qualifier tested, dice is greater than or equal to 9, we will have to see whether true or false. This goes to Y where I have to again test whether dice is less than or equal to 4. This dice greater than or equal to 9 or less than or equal to 4, that means the dice value between 4 and 9. I could have put 1 qualifiers, one after the other. I cannot do that now. I introduce an extra state 5 because the 5 has no value and it is a dummy state. The state has been introduced just to separate this qualifier from this qualifier. This is one example of what we do to add extra state.

Likewise, to introduce extra state, in the case of the earlier figure we had, we had a clear code register, a conditional output. I have to make it as a regular output state. The conditional output box has been taken as a state in which the output is clear code register signal is generated. I have not changed the names of the qualifiers signals; ASM in principle is the same; changes have been made. I request you to go over this state ASM along with the previous ASM and see what the changes that we have made are. Functionally there is no change, no variable has been introduced, no variable has been deleted either as a qualifier or as a output; number of states have been increased because of 2 reasons; conditional outputs has been removed and extra state has been introduced whenever 2 qualifiers are tested in series. We are not increasing the number of states as exact I told you. So, as per the previous ASM had six states now I have, as you can see here starting from 0 and the highest number of state is 13 that means total of 14 states; 14 states require 4 state variables. Earlier ASM had 6 states which required 3 variables, which means, I am increasing 1 state variable. This is not going to be an enormous increase in size; whereas, the number of qualifiers which is coming in the exponential power of the size of the ROM is now going to reduce the size of the ROM that we require in this case.

With this values of state addresses starting from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, a total of 14 states, the qualifiers are START, TD- throw dice, dice greater than or equal to 9, dice less than or equal to 9, score less than 10, score less than 89. These are the qualifiers. Outputs are counter enable or count enable, add score, load score, enabled number of attempts counter, clear score, load score, add score. S is the value which you have to give to either add or subtract a particular value to the score based on the game algorithm. These are all the same; no change has been made either in the symbol or the description of any signal. With this, we will write the microprogrammed table for this implementation and compare the sizes. There are 6 qualifiers starting from 0 to 5, START is 0, TD is 1 etc.

(Refer Slide Time 37:21)



ME Controller (Contd)

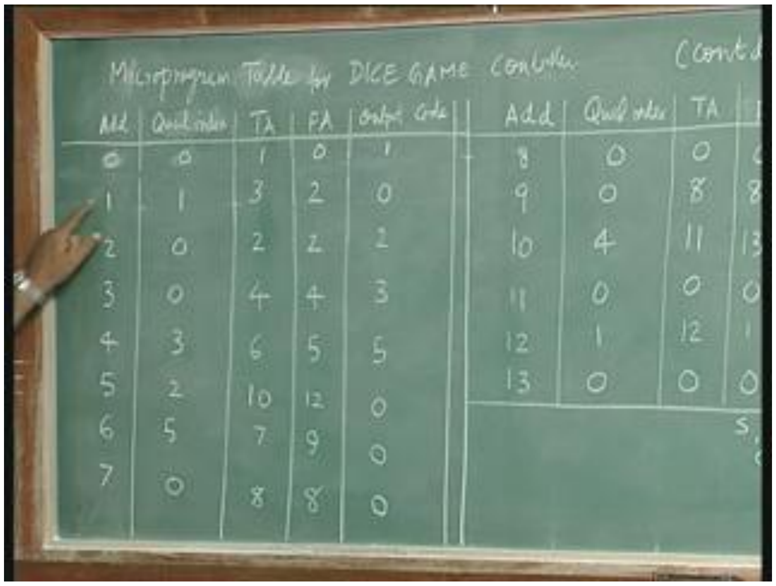
Add	Qual index	TA	FA	output Code
8	0	0	0	4
[illegible]	0	8	8	6
[illegible]	4	11	13	[illegible]
[illegible]	0	0	[illegible]	7
[illegible]	1	[illegible]	[illegible]	0
[illegible]	[illegible]	[illegible]	0	4

Qual index	
START	0
TD	1
DICE ≤ 4	2
DICE ≥ 9	3
SCORE < 10	4
SCORE ≤ 89	5

Output	Code
No output	0
LOAD 2	1
CNT_ENABLE	2
ADDS	3
LDS	4
LDS, ENOFA	5
S, ADDS	6
CLRS	7

I have given the qualifier index 0 to 5. There are 8 outputs including no output case, so I have given the codes for each one of them; no output is 0 code, LOAD 2 is 1, counter enable is 2, add score is 3 etc. You have 2 codes in which I have given 2 outputs. For example, LDS and enabled number of attempts as a single output. The reason is LDS separately comes as a code 4 but then when I put the code 4 here I should not get these 2 together. In only one particular instance should I get these 2 together; the rest of the time only LDS should come. That is why I have given a separate code for this. Likewise, we have an S and ADD S, a separate code even though we have a code for ADD S. This is the microprogrammed table just take one example.

(Refer Slide Time 38:28)



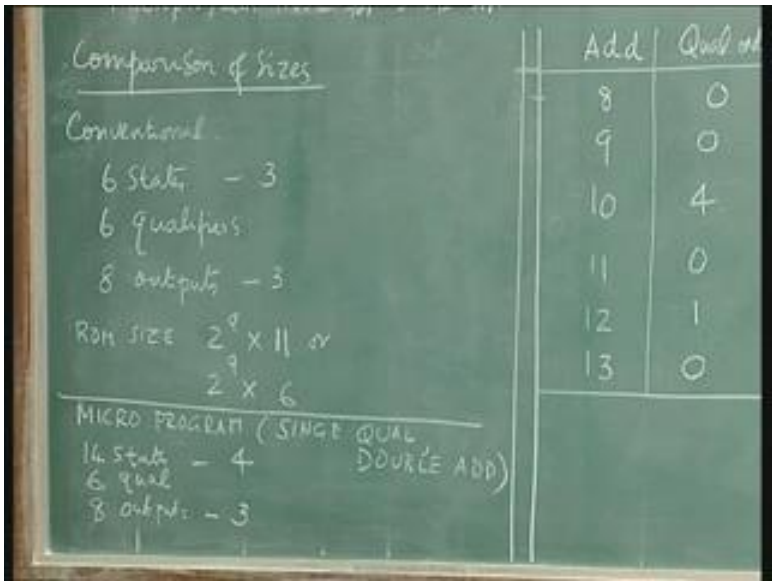
Microprogram Table for DICE GAME Controller (Contd

Add	Qual index	TA	FA	Output Code
0	0	1	0	1
1	1	3	2	0
2	0	2	2	2
3	0	4	4	3
4	3	6	5	5
5	2	10	12	0
6	5	7	9	0
7	0	8	8	0

Add	Qual index	TA
8	0	0
9	0	8
10	4	11
11	0	0
12	1	12
13	0	0

The present address 1, qualifier tested is TD which is 1, true address is 3, false address is 2 and there is no output here that is why no output is 0. So, present address is 1, qualifier address is 1, true address is 3, false address is 2, output code is 0. That is it. We can get the whole table, then, go step by step, state by state. What is the qualifier to be tested? Look at the qualifier index value, put that value, look at the true value of the address, and put the false value of the address and then, what is the output to be generated at that time? If there is no output if it is 0 otherwise give the appropriate code and you have got the table. All you need is a ROM to program the whole thing. I have already given the architecture of the microprogrammed ROM implementation in the last lecture using the simple example of a 4 state ASM. So, for this case, let us compare the sizes of the ROM required for the conventional approach and the microprogrammed approach.

(Refer Slide Time: 40:00)



Comparison of Sizes

Conventional

6 States – 3

6 qualifiers

8 outputs – 3

ROM SIZE $2^9 \times 11$ or

$2^9 \times 6$

MICRO PROGRAM (SINGLE QUAL DOUBLE ADD)

14 states – 4

6 qual

8 outputs – 3

Add	Qual
8	0
9	0
10	4
11	0
12	1
13	0

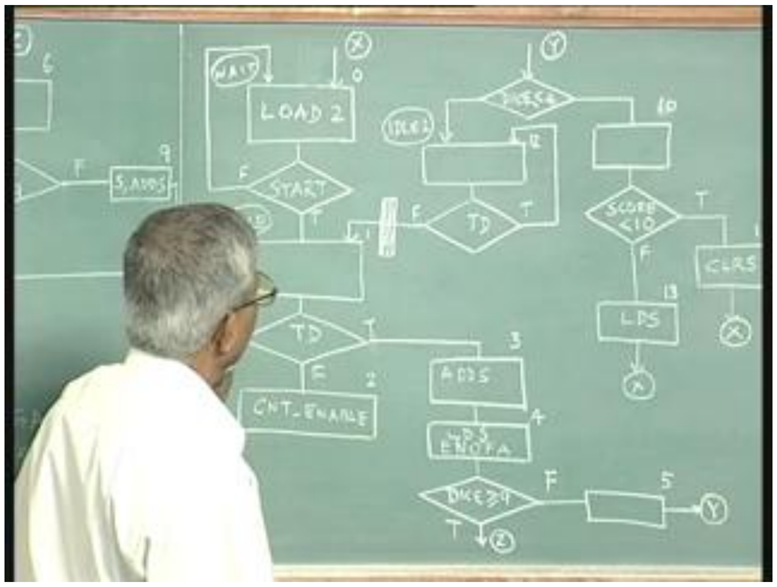
In the conventional approach, (Refer Slide Time: 40:15) there were 6 states as I said; you have to verify the earlier notes on implementation of ASM for dice game; 6 qualifiers, 8 outputs, these are all same. 6 states will require 3 state variables. 6 plus 3, 9; ROM size is equal to 2 to the power of 9 times the number of states 3 plus output 8 is equal to 11 or if you want to code this output into 3 bits again, 2 to the power of 9 times 6, that is, each 3 state variables and 6 qualifiers are the input address so we have 9 address lines, 2 to the power of 9 is the size. The number of output lines is the state variables and the output state variables of the 3 outputs are 8, 11or if you want to code the outputs into 3 bits it is 6.

In the microprogrammed approach, single qualifier double address. How many states are there? 14 states are there starting from 0 to 13. 14 states require 4 state variables, the number of qualifiers is same and number of outputs is same. This can be coded into 3 bits. The size is now the state variable alone, the address alone; address is this number of states alone, the number of bits in the state. There is no need to give the qualifier as the address. The size should be 2 to the power 4 times the qualifier index how many bits? It has 3 bits plus true address 4 plus false address 4, so 4 plus 4 plus 3 is 11. I can have 8 of this or 3 of this depending on the code of this or not code. If I do not code, it is 19; if I code, it is 2 to the power 4 times14. If I code it becomes 3 plus, 4 plus, 4 plus, 8 plus 3, 11 plus 3 is equal to 14. So, either 2 to the power 4 times 14 or 2 to the power 4 times 19 bits. In this case, 2 to the power 9; 2 to the power 9 is what? 512.

Let us go with the coded output. In the case of conventional ROM, you see it is 512 times by 9, 6 bits and for the microprogrammed it is 16 by 14. You decide which is bigger and by how much. And that is only the size; I am interested in the elegance of design. It is easy to do it. One minor modification in the end and that will be to make it into a single qualifier single address design. The only requirement being one of the addresses should be an increment of the present address; other address can be anything and that is also called as or jump address.

Even though you call it true address or false address because here we have incremented by one or jumped to a new location; some people or books call it jump address. We need to give only 1 address called the jump address which is a non-incremental value of the address. To that extent I have to make modification in the ASM. I am going to make it and I am going to leave the rest to you. I am going to tell you where the modification needs to be done.

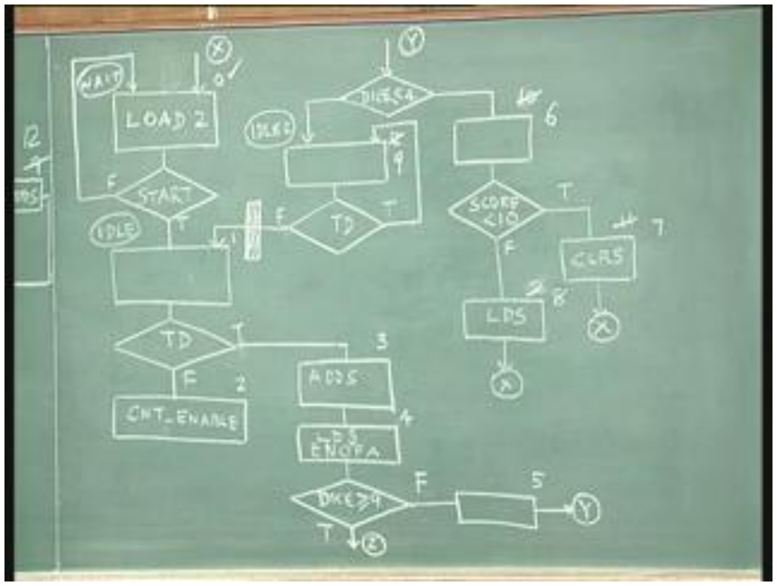
(Refer Slide Time 46:24)



You need an increase; there is a need for an extra state here, I will tell you why an extra state is required here; because this is 0, this is 1 plus 1, this is 12, this is going back to 12 and this is 1, I cannot have this. One of them has to be the next state of this as well as next state of this, it is impossible. It is not possible for this state to be next state of this as well as next state of this.

If it is the next state of this it cannot be the next state of this. If it is the next state I can modify it and make this 12 and this 13. In that case, this will not be next state of this. I need to have an extra state so this extra state is the only extra state required in this case. Then I need to make changes in the addressing.

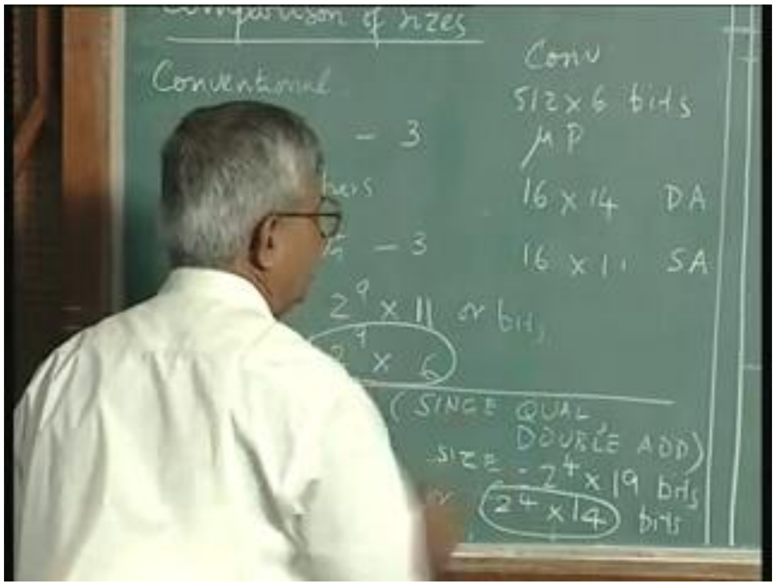
(Refer Slide Time 47:25)



Address will change like this: 0 will remain 0, this will become 1, this 2, this will be 3, this will be 4, this will be 5, this will become 6, this will become now 11, this will become 12, this will become 13, this will become 14, this will become 9, this will become 6, this will become 7, this will become 8. I need to make 2 changes; one change we introduce an extra state for which the address will be 10. I need to introduce an extra state with the address 10 and I needed to make some changes in the addressing scheme.

The new addressing scheme will be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 (Refer Slide Time: 48:35). That means 1 extra state and some changes in the address. I am not going to do the rest of work; you do the table, present address, qualifier index, true or false bit, true bit to be 0 or 1, the jump address, the next address, the output code.

(Refer Slide Time: 49:25)



I can only give you the size comparison; size of single qualifier single address microprogrammed ROM would be 2 to the power of 4 times 16; if you do not want to code it the outputs are 2 to the power 4 times 11 bit. Since we are comparing the coded outputs, 2 to the power 4 by 11, so it is 16 by 11 is single address. Conventional is 512 by 6, double address microprogrammed is 16 by 14, single address microprogrammed is 16 into 11. We can also see the change in size. In the earlier example of a simple example of 4 states I could not show you the size improvement. Now I can show you the size improvement. There is a 64 bit saving in the size and not much change in the state diagram, only one extra state and redo the addressing. This is a very nice thing because in major digital designs, in a computer for example, a controller has so many functions, so many qualifiers, and so many address outputs that it is very difficult to keep track of a ROM where for each state we will write the inputs true or false.

We are concerned with the local state and this neighbourhood. What happens at this state? What are the qualifiers tested? Where does it go? What is the output? That is the approach we are going to take and the design will be very efficient. The microprogrammed approach has a very good potential in major digital design; lot of people use it. Microprogrammed controllers are very common in microprocessors and so many other digital systems; except if any program takes more time than the hardware.

Hardware means already having everything connected so that as soon as the system is turned on its signal passes and we get the output. When it is microprogrammed, it takes time for you to go from state after state. Except for the change in this speed, other than in ultra speed designs, microprogrammed approach is a very practical and elegant option. We will stop here. Thank you very much.