Computer Organization
Part – I

Prof. S. Raman
Department of Computer Science & Engineering
Indian Institute of Technology

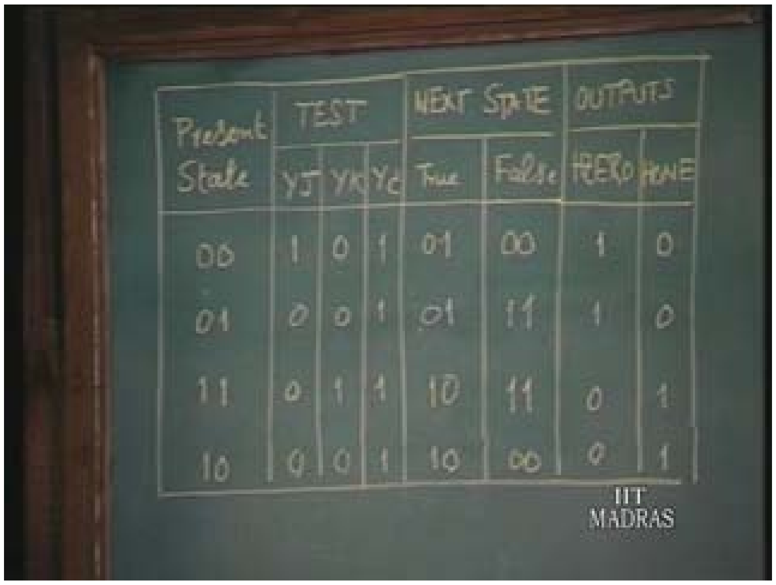
Lecture – 9

Controller Design: Micro programmed and hardwired

We saw that the ASM chart describes the behavior of the entire system in our case; in this particular case it's the master–slave JK flip-flop. We also worked out the possible next states for the given state; the inputs that need to be checked and the output to be generated, and after assigning code for the state, we also worked out the details of this table. Now what we have is for the present state 00, that is for present state a, we now continue this for the next state b, with code 01. In this, the input to be checked is YC, which is 1; the other two are not to be checked. Then the true next state is b itself, which means 01 itself. A false next state is state c, which is 11 and the output being generated in state b is H ZERO as before.

Then we work it out for next state, that is, state c, the code for which is 11. In this, the test inputs to be checked are YK and YC 11. So it is not YJ, and the true next state is this path 10 and a false next state is 11 and the output to be generated is H ONE, not H ZERO. Then we just have one more state, that is, state d, with the code 10. In this H ONE is to be generated; that is the output. YC is the input to be checked and the true next state is 10 and the false next state is 00.

(Refer Slide Time 00:03:43)



Present State	TEST			NEXT STATE		OUTPUTS	
	YJ	YK	YC	True	False	HZERO	HONE
00	1	0	1	01	00	1	0
01	0	0	1	01	11	1	0
11	0	1	1	10	11	0	1
10	0	0	1	10	00	0	1

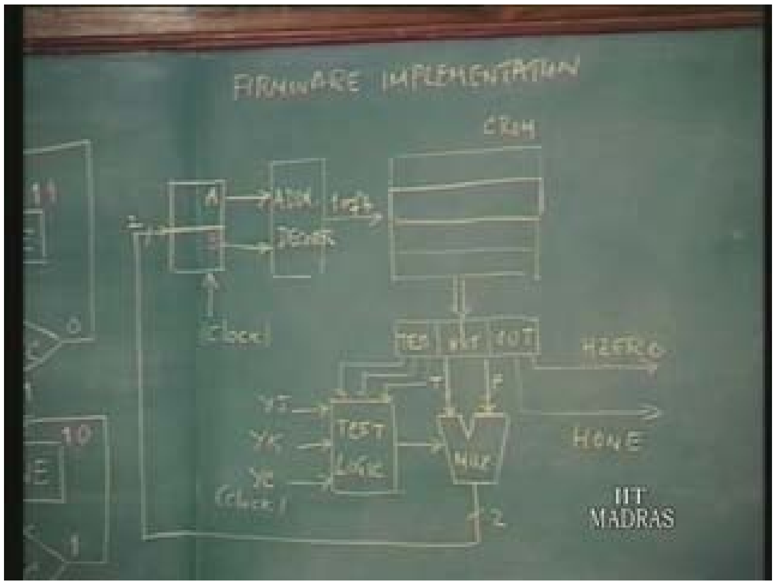
Whatever we have in this ASM chart, we now have the entire thing in this table. Specifically, you can notice that this set of codes that we have here is the one referring to each state separately. So put this code in the memory, that is, in a read only memory.

Put this particular code; so that's our control read only memory if you are going to implement this entire system using firmware technique. There are four codes corresponding to four states. That is whatever we have here in this box is the code, which must go in there, and as before, that is the micro-word register, which will essentially have a test field. We have seen this in detail earlier; the next fields are both true and false fields and then we have the output field. At any given time you would be reading only one word from the memory, which possibly corresponds to state b.

Now here we have the address decoder. So essentially one out of these four will be selected and because we have four states, we have two state variables. So here we will be having just two state variables; let us call one A and another, as I have marked here, B. These are the two things that go there; so a 2-bit code goes and one out of four will be selected. So either this or this or this one out of four will be selected and that will be written here. Here we have two next fields – true and false. So we will have our multiplexer, so as to select this, what do we have here? We have the next address, next state that is false or true. One is a true next state and the other is a false next state; it is just a 2-bit code. One of these is to decode one of these; so that is the 2-bit code, which in fact goes to this, that is, 2-bit code we are talking about.

Here we have the 2-bit present state and we have both this 2-bit true or 2-bit false state, and which of this is true or false should be selected depending on the test logic. Actually it is some selection logic; let us just put it as test logic. We know that in the test field there are three; so essentially from here there will be three things which it will indicate. This will indicate the inputs to be tested. We have YJ as one input; YK is another input; YC is another input. This memory element also will be using the same. This is nothing but our clock; YC is the clock, the same clock will be used here also. There may be some small change, some phase change, we will not worry about it now, and the outputs will be generated. There are only two outputs, one is H ZERO and one is H ONE. This whole system has been implemented using these circuits; now that is our firmware implementation.

(Refer Slide Time 00:09:29)

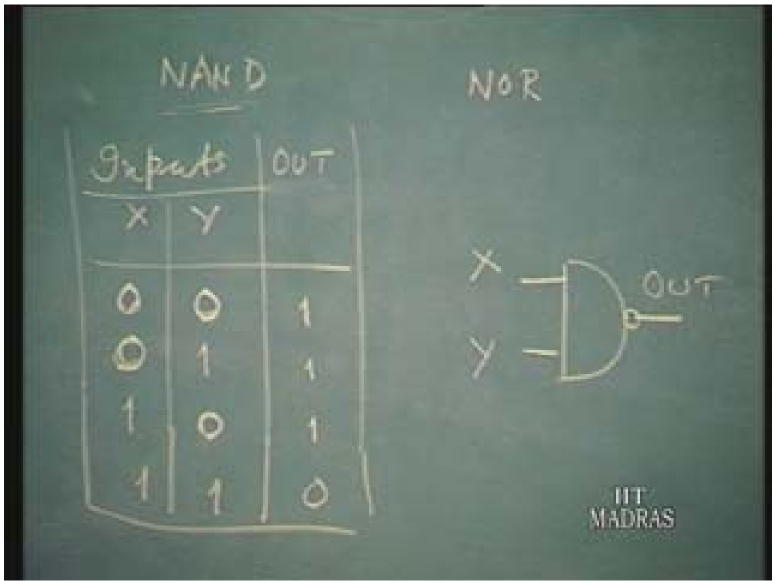


Why? As I was mentioning earlier instead of this code, you pull out this memory and put in another code and the behavior of the machine changes. As long as the code is there, this will be the behavior of the machine. So to start with, assuming if the system is in state a – state a means a will be 0; b will be 0; it will have 00 – assuming it is so when this would have been 00 and 00 will correspond to state a, that is, this particular one word 00. So this word will be selected; this information will be coming here and that tells the code corresponding to state a, that is, specifically this part, so this will be read into this and it would say YJ and YC are the two to be checked in the present state a. Then, depending on whether that combination is 0 or 1, select either true or select the other one false, and pass on that 2-bit code here as the next state. And in state a, it says generate H ZERO; that is, this will be 1 and that will be 0.

Now depending on what this is, the next code will come. If the next code happens to be corresponding to 01, that means the system is moving to the next state, 01. Then 01 will be here, and b will be selected; that is coming here and so on and so forth. This in fact is the firmware implementation. What is the parallel hardware implementation? In fact, mainly to show that particular one, I took up this example because otherwise we cannot discuss this with reference to very complicated systems such as the CPU with many processors. Again the starting point of the hardware implementation is the same thing, the ASM chart, because that is the one which describes the behavior of the system. Similarly, in an ASM chart we have to start in that and what is it we have done? We had worked out the details and we know that there are four states and that with two memory elements; we can implement this entire machine. So you need to have two memory elements – a and b. Now, depending upon those memory elements we have to derive the appropriate equations so that we can design the whole thing. What memory elements shall we make use of? The simplest one is to assume an RS (set–reset) latch. I am assuming on your part that you all know what are an and logic, or logic and not logic. And that these are the basic logic gates with which you can implement any system.

Now sometimes, combining either, and not, or, or and not, we can also have NAND gates or the other one is called nor gates. Now using only NAND gates, you can implement a digital system. Neither similarly, using only nor gates also, you can implement an entire digital system. So I will use NAND gates and then show you how this whole thing can be implemented. The first requirement which I am saying is I need to first rig up a memory element. Now what is the NAND gate? I will just work out the NAND gate assuming that there are two inputs. I will use a b here so I will use x and y. A NAND gate is represented like this: with the inputs x and y. I am assuming a two-input NAND gate, and what is the output? The output of this is like this 00 with any Boolean variables, and with two of these things, we will be having 2^2 or four combinations. What is this output? Basically it says this and this and then invert. Invert is not an inversion; so this and this 0 invert the output will be 1; this and this will be 0 invert and the output will be 1. This and this will be 0 invert; the output will be 1; the output of this and this 1 invert will be 0, which means basically for a NAND logic the same thing can be expanded to more than one input also – as long as any input is 0, the output will be 1; that's the main thing.

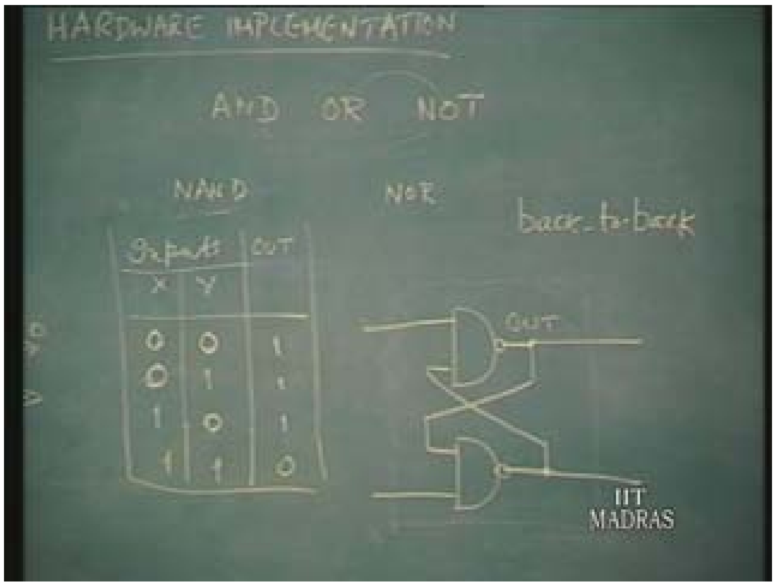
(Refer Slide Time 00:16:16)



Inputs		OUT
X	Y	
0	0	1
0	1	1
1	0	1
1	1	0

Now suppose I take one such NAND gate and then connect it up like this. Suppose I take the output of this, connect to this input, and take the output of this and connect to this input, essentially what I am having is one particular module. With what we call as two NAND gates connected back to back, two NAND gates connected back to back, what's happening? Just work it out and see.

(Refer Slide Time 00:17:14)



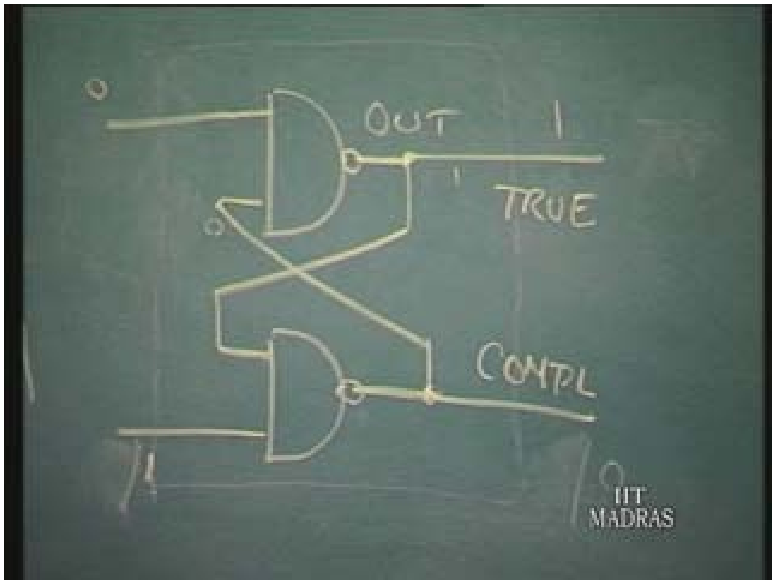
Inputs		OUT
X	Y	
0	0	1
0	1	1
1	0	1
1	1	0

Suppose this input is 0, arbitrarily we will just say the input is 0. We know with one input 0, the output will be 1; so this will be 1. And when this is 1, we cannot decide what it is. Now if it were 0, then this will be 1. On the other hand, if this were 1, this is already 1. This is 1 and so this will be 0. Now see: this is 0. If this were 0, this would have been 1 and if this were 1, this would have been 1, but it does not alter the output, mainly because even if this is 1, the output will be 0. So this is one stable condition. Now on the other hand, if this were 1, the output will be 0 mainly because this we have assumed as 1. This is 1; this is also 1 and so the output will be 0. This would have been 0, because of the same reason. So when one input is 0, the other one can be 0 or 1, both ways the output is stable. Now what is that output? That is, this one is always 1. And what did we see? If this were 0, we said this will be 1.

On the other hand, we take the other one. If this were 1, we said this will be 0. So we will assume this condition then we find that this output is 1 and this output is 0: complement of it. The same thing holds good for the other if instead of this, this were 0 and this were 1. It would have been the other way because it is symmetrical. Now I can use this as a memory element as long as I follow certain restrictions on the input. Why? Remember I said if this were 0 instead of 1? I said this would have been 1 – now what happens? It is not altering the situation here; this part will be 1.

Anyway, both these will be 1, which is not what we want really mainly because in a flip-flop as we had even discussed earlier, when one output is 0, the other output is called complementary output. It must be just a complement of it, that is, 1. If one output is 1, the other must be 0, which is not taking place in this particular one. That is why we say if we follow certain restrictions on the input, this can be used as a flip-flop, in which we can just name it as a true output. This particular one can be termed a true output, and you can use the other one as the complement output. If you want to call this true, call the other one a complement.

(Refer Slide Time 00:21:12)



I will slightly change this. Suppose I make both the inputs 1. What will happen? If we make both the inputs 1, I cannot say anything what will happen here. We saw that earlier if you make both the inputs 0, this also will be 1. It is not following this relation, that is, true complementing – both happen to be 1. Now if I have 00, it is not working. If I have 0 and 1 here, it works nicely. On the other hand, if I have 1 the other one must be 0.

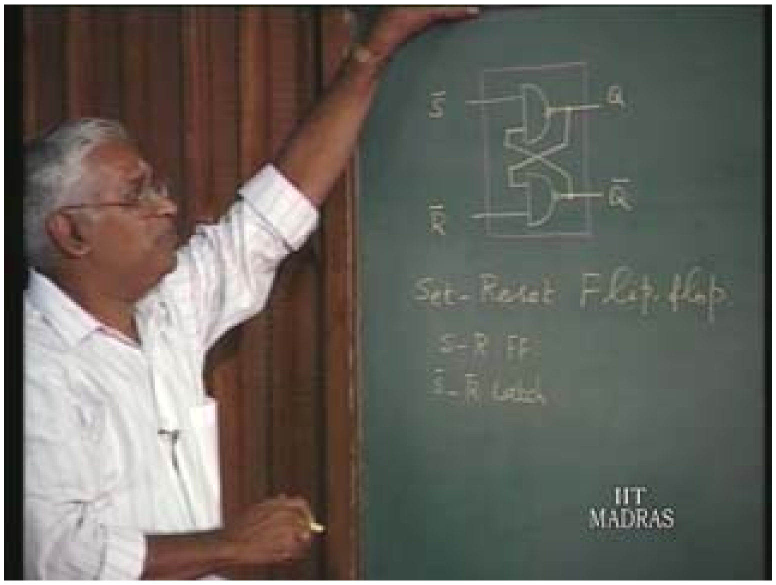
Now what happens when both are 1? If both are 1, I cannot decide; from logic point of view I do not know because we do not know what is happening here. So here, what we say is the physical conditions will come – one of these possibly is faster and the other one is slower. When one comes the physical condition will decide what the logical output will be, and depending on what the logical output is, the other one will be decided. One of these will be always faster because in nature, no two creatures are the same. The same thing holds true in devices too. So no two devices are the same. You may call them as the same type, or the specific IC given or the type number, say, 7400 gate, are the same, but ultimately when it comes to the physical thing, we do not know. One of these will be working faster than the other, and then the corresponding output will be decided. That point is when both are 1. We do not know what exactly is happening here, that is important.

So if this happens to be 0, this will be 1, then there is no problem. This is 11 and this is 0. We will be using this kind of a thing – what did we say earlier? Both should not be 0; this is what we worked out. Both the inputs should not be 0. As long as both the inputs are not 0 we can see to it that one output is true the other is complementary, which behaves very much like what a flip-flop really is. So we make use of this particular one and then develop the necessary logic for that. Specifically this is called a latch, because it can latch on to information and store it as if it is very much like a flip-flop.

You may work out and then see that it is actually called an S bar R bar latch because, remember while discussing the JK I was talking about the set and reset? Similarly there is another one; it is called SR latch. But it so happens that here the input will be an S bar R bar; just work it out yourself and then you will get that. So we will make use of this S bar R bar latch to implement the state variable A, similarly another latch to implement for state variable B, and then work out the relationship. This is the hardware part of it.

I asked you to work out how this particular back to back combination of two NAND gates can be used as a memory element. Essentially the output of this memory element will be a true and a complement; generally we use Q and Q bar output. Then it works as a flip-flop and when I also said that it will be similar to SR, which is set–reset flip-flop. Now it so happens that in the set–reset flip-flop either you can have the set input or the reset input; unlike JK flip-flop the set–reset flip-flop will not accept both being 1. Both being 0 is fine, whatever state it is in; it will stay put. So when the input comes either set must be on or reset must be on; both should not be 1. Now when you work out, you will find here that both should not be 0. In this set–reset both should not be 1. Here you will find out that these two inputs both should not be 0; that is why it gets this name S bar R bar latch. Normally set–reset flip-flop will be called an SR flip-flop; what we have here is an S bar R bar latch, mainly because there is no clock or anything. It just depends on the input.

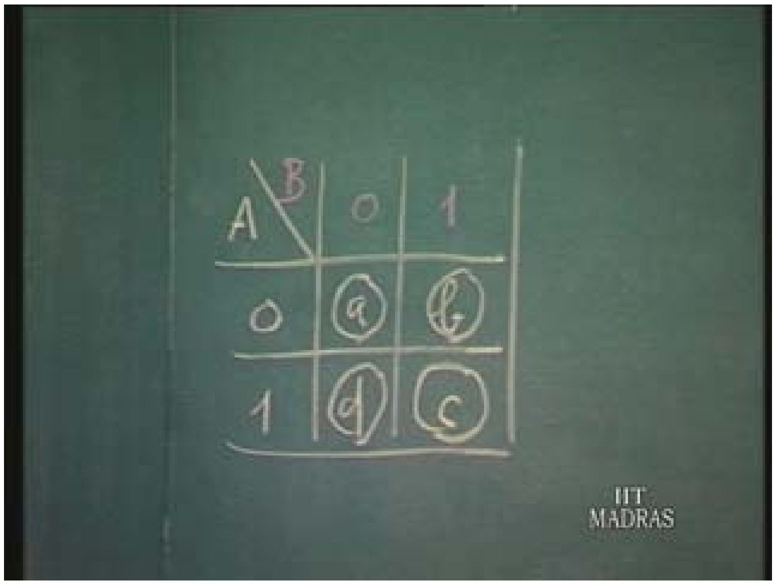
(Refer Slide Time 00:26:57)



Remember I was talking about an asynchronous system? When the inputs change the output will immediately change; this is the easiest or the simplest memory element one can have. So we will have two such memory elements: one is implementing the state variable A and another implementing the state variable B. Now we have to make use of what is known as Karnaugh map. Karnaugh maps are used to derive the equations and you may recall whatever you may have learnt in the switchings theory and digital or may be logic design or switching theory in digital design.

Now I will do the first Karnaugh map for this system itself, which I am calling the state assignment map. What is that? It is the starting point of the whole thing; it is the state assignment map with reference to the ASM chart. What is that? It just says you have two variables, A and B. The two variables may take a value of either 0 or 1 and for the combination of A being 0 and B being 0, the state is a, small a. That is precisely why it is called state assignment map. This map shows what the core of A is and B – that is a variable for the given state. Then when A is 0, B is 1. The state is b, that's small b, and when a is 1 and b is 0, the state is d, and in the other one, both a and b being one state is c.

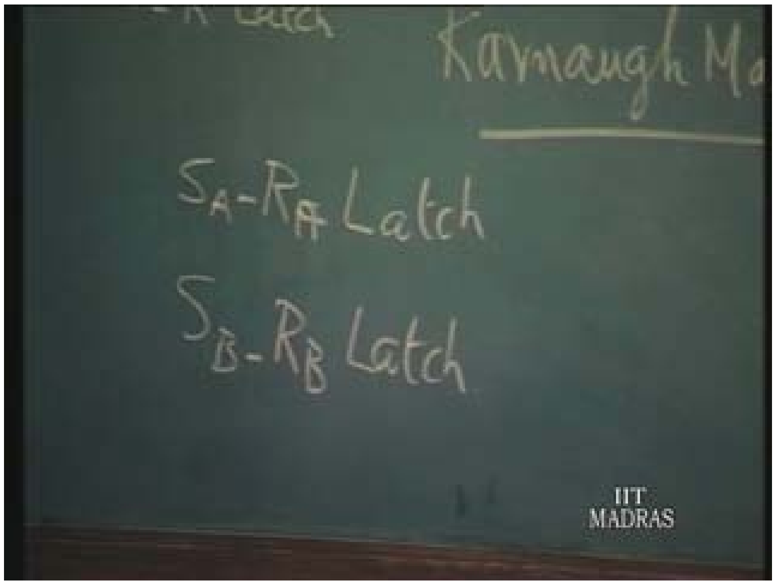
(Refer Slide Time 00:29:35)



A\B	0	1
0	a	b
1	d	c

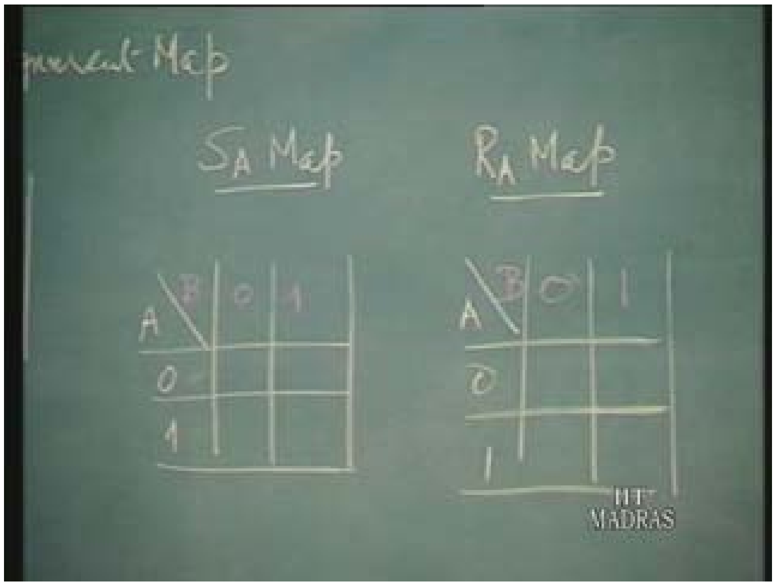
Now we have to basically see the two memory cells or memory elements, which we use to implement a and b, how they behave in each of those states. Once we have that, we have the whole system description. In other words we have to generate the two memory elements; I will call them S_A R_A latch, that is one, and the other one as S_B R_B latch. I have to implement A and B, making use of two elements.

(Refer Slide Time 00:30:37)

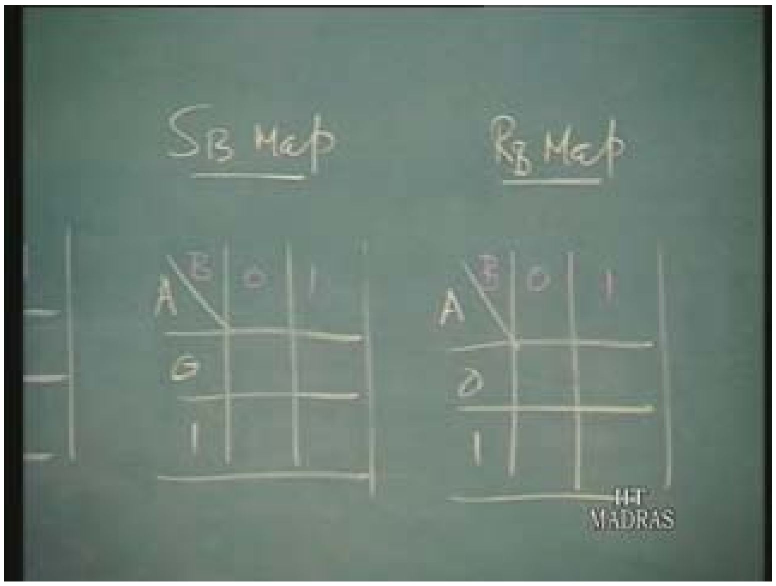


So I will call this one S_A R_A latch; specifically it will be SA bar RA bar latch. We will just draw the bar part of it because we can always complement and get it. So the S_A R_A latch is the one for the variable A and the other one is for variable B. I will just call it S_B R_B latch. Now if I develop the Karnaugh maps for these, which will give me, full information about the behavior of the system. Essentially what it means is that I have to generate the S_A map from which, later on, I will develop the S_A bar function and R_A map. Similarly I will generate for the other two: S_B map and R_B map. We will work it out now. It is the same as before, for S_B and R_B also.

(Refer Slide Time 00:32:10)



(Refer Slide Time 00:32:44)



I hope many of you have noticed we started with a system description; now we are going into the specific state variable behavior; from the system behavior we are going to state variable behavior. And in each state how each of these state variables or the component which implements the state variable behaves – that is what we are going to see.

So we start with state a. That corresponds to this cell; all these correspond to the present state a cell. Now we will first work out for variable a. In the present state a the variable a how does it transit? For this link path, the variable a transition is the present state is 0; the next state is also 0. So we talk about a 0 to 0 transition. What is the other link path? From here to here – it so happens that in the variable a there are only two link paths for state a: either this or this. It so happens in this link path also the variation is only 0 to 0; it means basically that in present state a, the state variable a should never get set. It should never get set. So the set input of this must be 0; it should never get set. Basically what we are writing down here is – what is SA and R – it is an input.

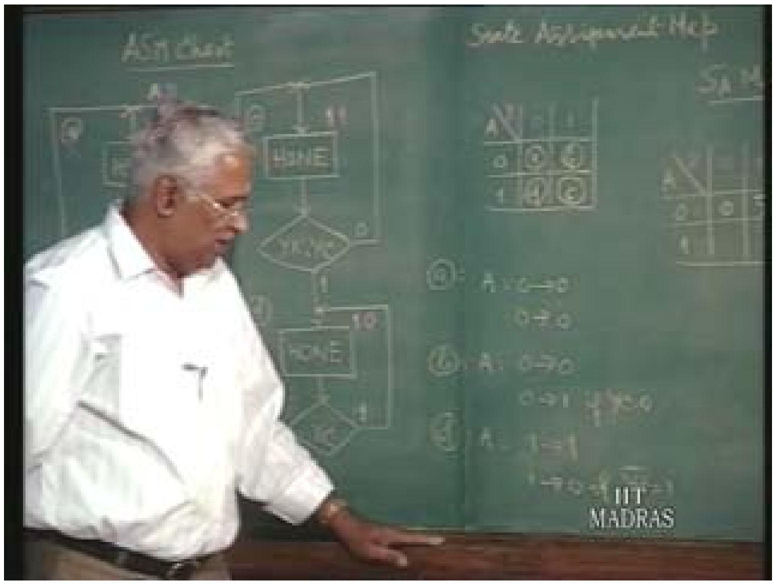
What we say is that if the system is in state a, the element that implements the state variable a should not get set. Can it be reset? Yes, it is already in 0 that the reset input comes; it does not matter. So the reset input is in fact what we call a "don't care". A set input must be 0. Now let us go to the next state; we will complete for variable a and go to the next state b – so this cell corresponds. In state b, there are two link paths: in one it is 0 to 0; in the other one it is 0 to 1. So why don't I write it? In state b, the variable a link path is 0 to 0, the other link path is 0 to 1. It says the next state of this must be 1, when YC is 0; that is the condition. When YC is 0, that is, if YC is 0, then the variable a must set to 1; that is what the input condition is. So we have to see that when the system goes to next state b, the variable a must set only for this condition.

That is, the set input must be there only when YC is 0, which means YC bar is 1 the set input must be given or the set input must be 1 for YC bar. We do not know what the condition of YC will be: 0 or 1; YC will be 0 or 1, and we earlier made restrictions that both should not be 1; set and reset both should not be 1. Now there is a possibility that this can become 1. So if you happen to have a "don't care", which means both can become 1, we have to see that this must be 0. If it is not 0, when it becomes 1, we may be violating that condition, because a "don't care" 0 or 1 means it can be 0 or 1. Now if you do the same thing here, when this becomes 1, we will be violating the original condition. So the entry here must be 0.

Now we go to next state c – one link path 1 to 1; now I do not think I need to work out here; the other one is 1 to 1 again. It says whatever may be the input condition, the variable a must remain set, which means it should not get reset. It must remain set – 1 1 or 1 to 1, whatever be the condition. So in state c, that is next state, it should not get reset so the reset input must be 0. Just the input can be 1 or 0, which in fact is the "don't care" mainly because the next state must be 1.

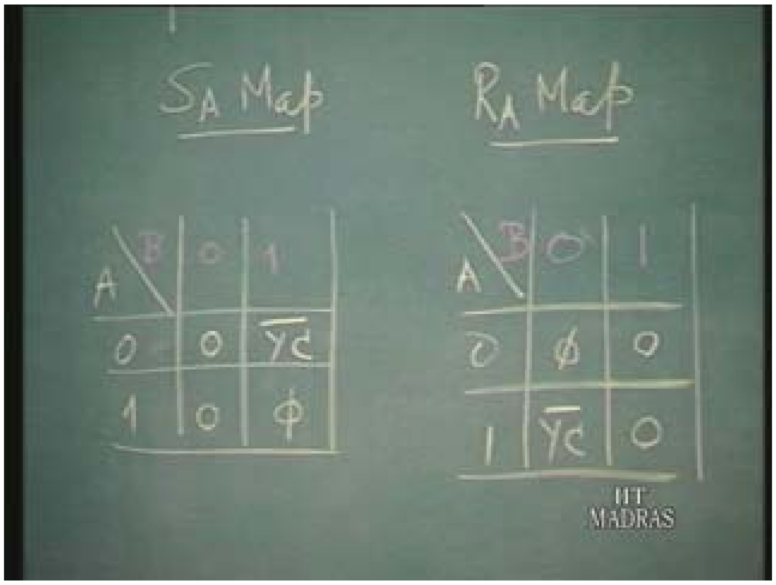
So if set input is 1, the next state will be 1; if it is 0, there is no harm. Anyway it is already in 1; it will continue to be 1 so it is a "don't care". But it should not get reset; that is the point. Now we go to next state d, that is, the last state. For one link path, it is 1 to 1; for another one it is 1 to 0. Now let us write it down because it is a bit crucial – so for one link path the transition is 1 to 1; for the other one it is 1 to 0 for some condition. What is that condition? If YC is 0, I may say if YC bar is 1 it is all the same YC 0 or YC bar is one and the same thing.

(Refer Slide Time 00:41:10)



So in state d, we have to see that the variable a gets reset for this condition; so the reset input must be generated if YC is 0, in other words, if YC bar is 1. Now the corresponding entry here would be 0 because we do not know whether it is 0 or 1, and to ensure that we do not violate our initial rule, we have to ensure that this particular one is 0.

(Refer Slide Time 00:41:45)



S_A Map

A \ B	0	1
0	0	$\overline{YC}$
1	0	ϕ

R_A Map

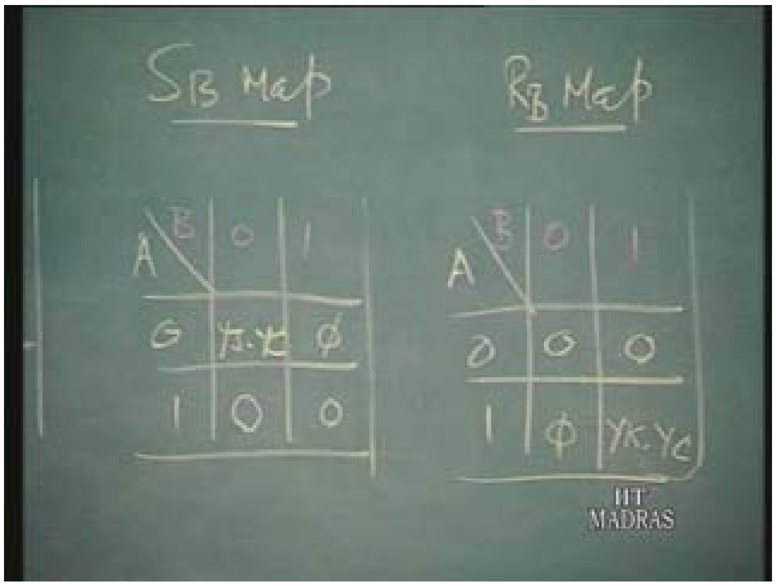
A \ B	0	1
0	ϕ	0
1	$\overline{YC}$	0

Similarly it must be worked out now for variable b. Here we have some Karnaugh maps – may be many of you may not be familiar with these kind of maps. You may be familiar where you have the entries as 0 or 1 or “don’t care”, but here we have some variable also coming in; this calls for a discussion. Now may be we will complete the b part of it also and then learn something so that we can derive the input functions from this.

I will quickly go through to state a; that transition is 0 to 0 or 0 to 1; 0 to 1 means set for condition YJ and YC. So the set input YJ and YC has been there, and the other one must be 0; that is, for state a. For state b, you have 1 to 1, which means it should not be reset. So in state b we have to see that state b, that is, this cell, should not be reset. The reset input must be 0. The S_B can be 0 or 1. Then we go to state c; for variable b, it is 1 to 1; the other is 1 to 0, which means it must reset for this condition. So the reset input must – we are talking about state c, that is, this cell – get reset, so the reset input must be generated, which is for the condition YK and YC. For YK and YC, this must be reset.

Now the last state, that is, d – for d, we have 1 to 1 or 1 to 0; sorry, that is variable a. We are talking about variable b – 0 to 0, 0 to 0. That means it should not get set. In state d, it should not get set; I will come to that. In state d it should not get set, so this is 0. The reset input is “don’t care” and looks like I have not completed this. Since YK and YC is the condition here, this must be 0 based on the logic we have discussed earlier.

(Refer Slide Time 00:45:02)



S_B Map

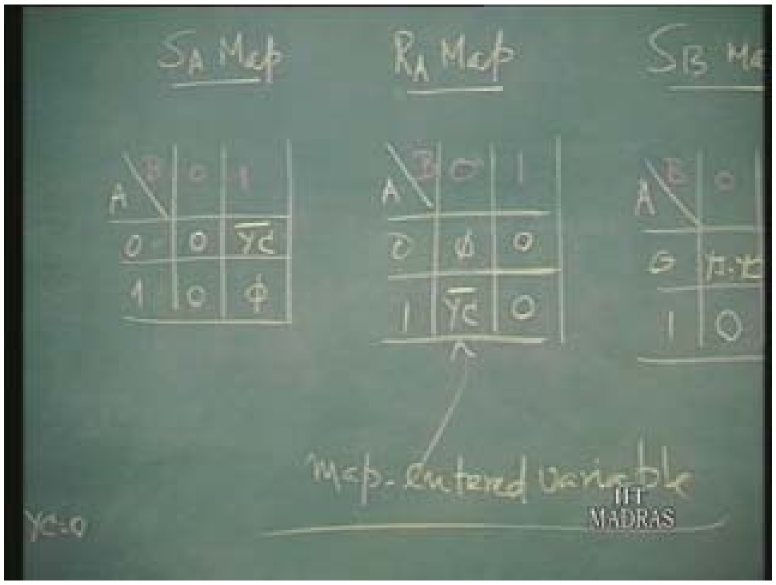
A \ B	0	1
0	$YT \cdot YC$	ϕ
1	0	0

R_B Map

A \ B	0	1
0	0	0
1	ϕ	$YK \cdot YC$

Now how do we arrive at the S_A and R_A function and then subsequently S_B and R_B function? Once we have them, we have the logic circuit, which must go before. So there are two such latches and that function will give us the Boolean expression for S_A, R_A, S_B and R_B or S_A bar, R_A bar, etc. Now can we proceed? There is a problem, mainly because we have what is known as a map entered variable, which many people may not be familiar with.

(Refer Slide Time 00:46:01)



S_A Map

A \ B	0	1
0	0	$\overline{YC}$
1	0	ϕ

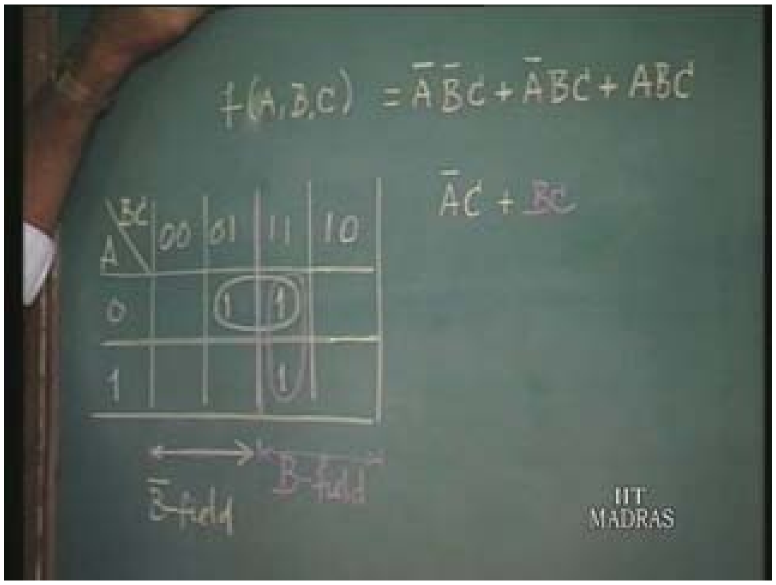
R_A Map

A \ B	0	1
0	ϕ	0
1	$\overline{YC}$	0

Map-entered variable
YC=0
IIT
MADRAS

We have to first study the problem of how to handle a Karnaugh map with map entered variable. To explain the concept of map entered variable, let us take a simple example with which you may be familiar. I have just taken an example about a function of three variables: A, B and C – just some arbitrary function.

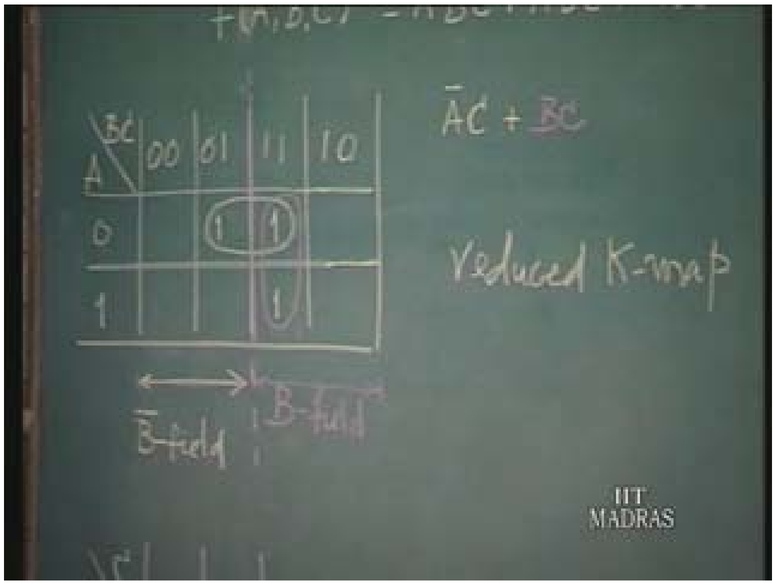
(Refer Slide Time 00:46:27)



Now when you create the Karnaugh map, the conventional Karnaugh map, you would find that essentially you have three entries, that is, min terms; three entries corresponding to the three min terms, and then, when you group them together, finally this is the function that you would be getting. So this encirclement actually means A bar and C, that is, the yellow one, and this red encirclement really corresponds to BC.

Now what is the map entered variable? The map entered variable comes up because we are trying to reduce the size of the map. Take this map. You find that if you just take this as the axis, then, to the right of this axis the variable B is 1, so this entire area is called B field, because left of this, B is 0. So that particular thing correspondingly is called B bar field. Now imagine that about this axis, if I fold the map, then this will merge with this; this will merge with this; this min term will merge with this; and this min term cell will merge with this. What is the effect of that? B merges with B bar – what happens? B vanishes. In other words, what we are doing is by doing this folding around an axis which identifies a variable and its variable bar field, the size of the map gets reduced mainly because we are removing one variable, that particular variable, which the axis identifies.

(Refer Slide Time 00:48:40)



We can also do the same. For instance, if you take this axis and fold it, A will vanish. Now reduced Karnaugh map is what you have here; precisely that is what has been done here. You have not yet carried out the entries; we will do that. So B vanishes mainly because about this axis I am folding the map. Now what is the end effect of this? What will be the entries here? Let us work out.

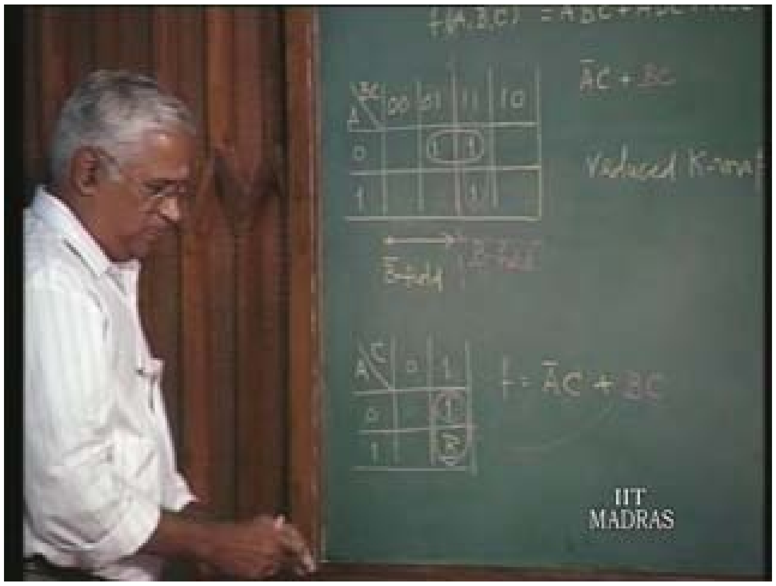
So take this. When you fold this and this; merge this and this; this cell and this cell both have nothing. So correspondingly, there will be nothing here too; nothing actually corresponds to 0. What about this? This will merge with this – what is the entry here and what is the entry here? Both are 1 – actually it means the function does not depend on B. It says the function is 1, whether B is there or not. The function really does not depend on B; it only depends on A and C.

So when these merge, we put an entry 1 to indicate that the function depends only on the value of A and C; it does not depend on B. Then we go to this. Again it is similar to what we had earlier. But coming to this, it says the function depends on the value of B being 1; if the B value is 0, the function goes 0, whereas in the other case, the function was not depending on the value of B. In this case, it depends on the value of B; the value of B matters. So when this folding takes place and this merges with this, it means we have to enter the corresponding cell B to indicate that the function depends on the value of B. Now how do we arrive at the function?

We have a reduced Karnaugh map and this is similar to what we had here. We have some variable entered. Now what the variable is does not matter; let us not worry about that. There is some variable; so to develop a function from this, we must arrive at the same function. I would just state the algorithm, and then proceed. It would say wherever you have 1 – there can be more than 1s – in the event they reduce Karnaugh map, it all depends on wherever you have 1.

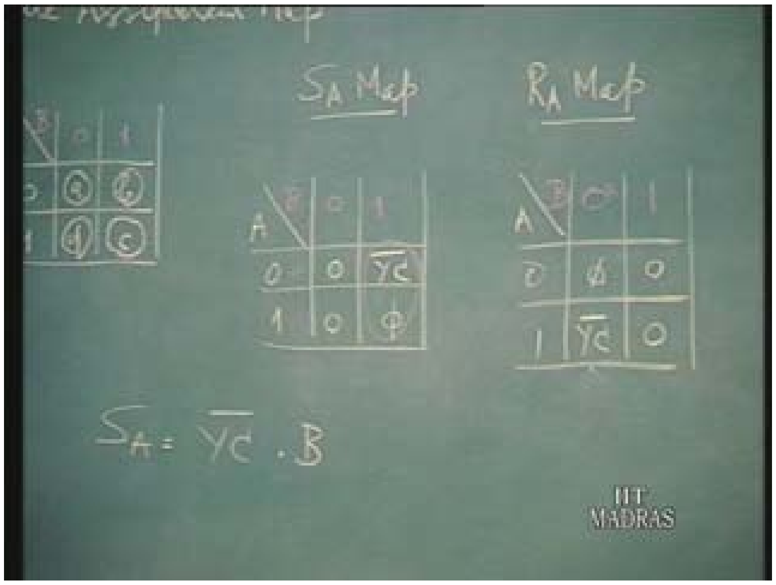
Encircle them separately, and then write out that particular term. What is the term here? It says this is true for A being 0, C being 1, in other words A bar and C. The function should be true when A is 0 and C is 1. That is, A bar C is true. It so happens that the sample or example we have taken is only 1. Now we have another entry of B like this – if you have other entries of B you can combine them together, and what about this entry? This entry actually means it hardly matters. This does not depend on the value of B; a 1 here means it does not depend on the value of B in the reduced Karnaugh map. So we can as well include that. In other words, while encircling, we can as well include this mainly because 1 means it does not depend on the value of B; so it is a "don't care" as far as B is concerned. So then, we find that when you encircle, this encirclement causes the next term or yields a next term, which says C because the whole thing is C; C being 1 and B must be there. A 1 here means B can be 1, B can be 0. Now we are forcing B to be 1, so BC is the other term. So this is how you arrive.

(Refer Slide Time 00:53:22)



A bar C or BC – we have got the function from both, we have to do the same thing here too. That is, what is the value of SA? See there are no 1s here, so we do not have to bother. What we find here is just one thing and then there is a "don't care"; 0s cannot be included. That is like the empty ones there. So this is an encirclement; take this encirclement and then develop the function as a YC bar and B. Similarly, R_A: here we have YC bar and B bar – we can do this for the others also.

(Refer Slide Time 00:54:08)



S_A Map

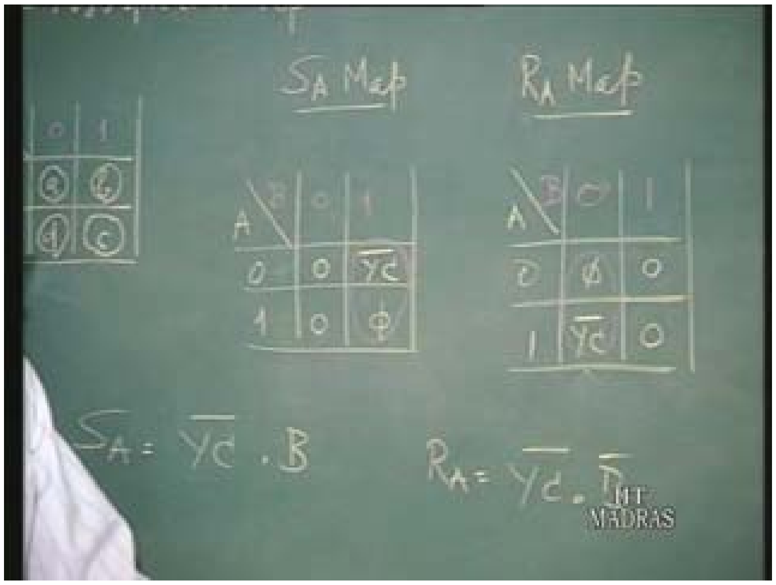
A \ B	0	1
0	0	$\overline{YC}$
1	0	ϕ

R_A Map

A \ B	0	1
0	ϕ	0
1	$\overline{YC}$	0

$S_A = \overline{YC} \cdot B$

(Refer Slide Time 00:54:28)



S_A Map

A \ B	0	1
0	0	$\overline{YC}$
1	0	ϕ

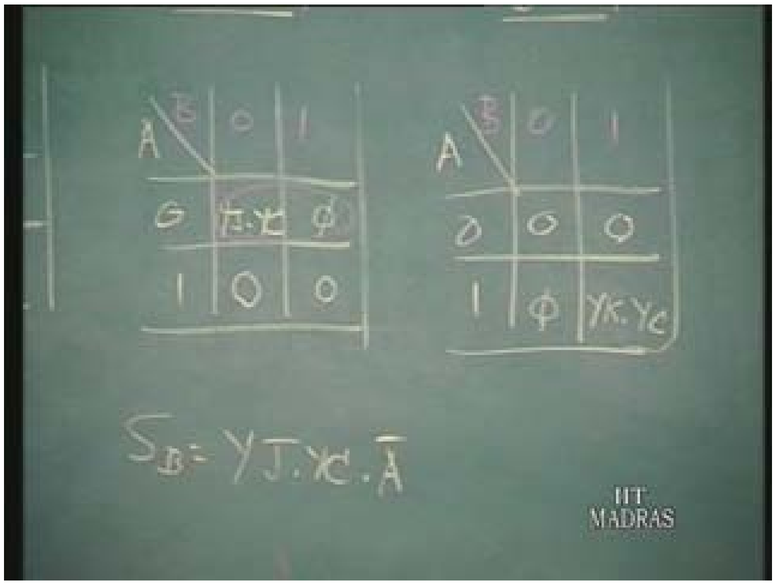
R_A Map

A \ B	0	1
0	ϕ	0
1	$\overline{YC}$	0

$S_A = \overline{YC} \cdot B$

$R_A = \overline{YC} \cdot \bar{B}$

(Refer Slide Time 00:54:45)

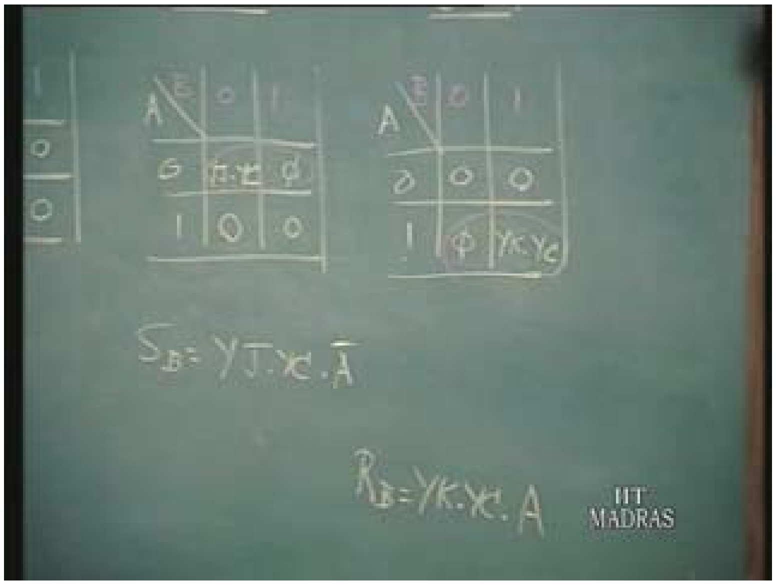


A \ B	0	1
0	YJ.YC	φ
1	0	0

A \ B	0	1
0	0	0
1	φ	YK.YC

$S_B = YJ.YC.\bar{A}$

(Refer Slide Time 00:55:00)



A \ B	0	1
0	YJ.YC	φ
1	0	0

A \ B	0	1
0	0	0
1	φ	YK.YC

$S_B = YJ.YC.\bar{A}$

$R_B = YK.YC.A$

What is S_B? We have this encirclement YJ and YC and A bar; and R_B is YK and YC and A. So we have arrived at the input functions for the memory elements, which implement the state variables A and B. We started with S_A bar R_A bar latch or S_A R_A SR latch, and so we are developing the equations for that input.