

**Hardware Modeling using Verilog**  
**Prof. Indranil Sengupta**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

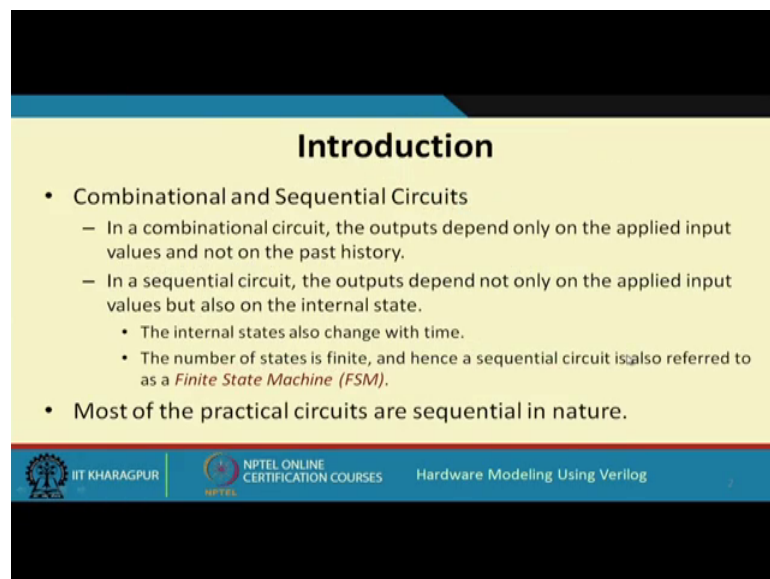
**Lecture - 23**  
**Modeling Finite State Machines**

So, we have so far looked at the design of both combinational and sequential circuits using Verilog. The combinational circuits that were looked at, were modeled using the continuous data flow kind assign statements, or even using the various kinds of procedural blocks, typically using blocking assignment statements. And the examples of sequential circuits that we saw, were all modeled using procedural assignments both using blocking and non blocking assignments.

Now in this lecture, here we shall be looking again at the modeling of sequential circuits, but not directly from the behaviour as we had seen earlier, but in a more formal way from the so called state table or the state transition diagram representation of the sequential circuit.



So, the topic of our discussion today is modeling of finite state machines ok.

(Refer Slide Time: 01:28)



**Introduction**

- Combinational and Sequential Circuits
  - In a combinational circuit, the outputs depend only on the applied input values and not on the past history.
  - In a sequential circuit, the outputs depend not only on the applied input values but also on the internal state.
    - The internal states also change with time.
    - The number of states is finite, and hence a sequential circuit is also referred to as a *Finite State Machine (FSM)*.
- Most of the practical circuits are sequential in nature.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, as I said that you can have two different kinds of circuits; combinational and sequential. So, in a combinational circuit, the idea is that, the output or the outputs will

depend only on the applied input at that point in time. It will not depend on the previous or the past history.

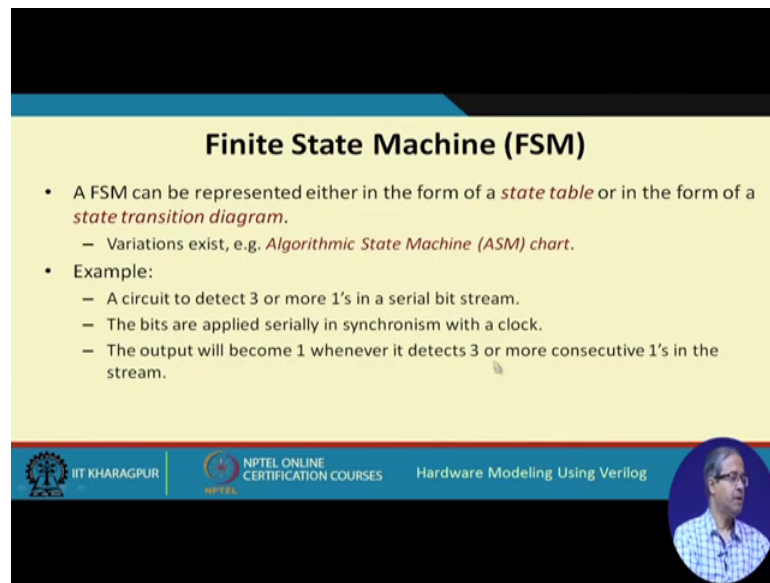
So, as an example you can think about a full adder, where whatever inputs a b c were applying right at this point in time, the output sum and the output carry would be generated based on those only. So, it will not depend on the previous history means, what are the a b c values, that we had applied in the past, this is what is made by a combinational circuit, the output depends only on the applied inputs. So, in contrast for a sequential circuit the outputs that are generated, will depend of course, on the applied inputs, but also on some of the past history which is represented by the internal state. So, we shall be looking at a number of examples in this regard, to indicate what a state means, like I can give a very simple example, you think of a binary counter say a 4 bit binary counter that counts from 0 1 2 3 4 up to 15 then again back to 0.

Now, suppose when the value of the count is 5 let us say, and we apply a clock. So, the output that we expect is 6, that depends not only on the clock you are applying, but also on the previous count value that 5, that previous count value that can be regarded as the state of the machine. So, the next output that you get, will depend on some previous information; that is the state and on the applied input. So, just as in the example of the counter its stated, the internal states change with time.

And for any practical hardware implementation of such a sequential circuits, the number of states has to be finite. It cannot be infinite clearly, because the states ultimately you will be representing or storing in flip-flops. Suppose it is a 4 bit counter, will be storing the 4 bit of information in 4 flip-flops. So, if I say there are infinite number of states, then it is not practical to build such a machine, because you will be requiring an infinite number of flip-flops fine. So, because of that the practical sequential circuits are also referred to as a finite state machine.

So, there is a concept of a state, number of state is finite and there is a formal definition of a machine that will see, and most of the practical circuits that we see is around us are sequential in nature, very fewer combinational.

(Refer Slide Time: 04:39)



**Finite State Machine (FSM)**

- A FSM can be represented either in the form of a *state table* or in the form of a *state transition diagram*.
  - Variations exist, e.g. *Algorithmic State Machine (ASM) chart*.
- Example:
  - A circuit to detect 3 or more 1's in a serial bit stream.
  - The bits are applied serially in synchronism with a clock.
  - The output will become 1 whenever it detects 3 or more consecutive 1's in the stream.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

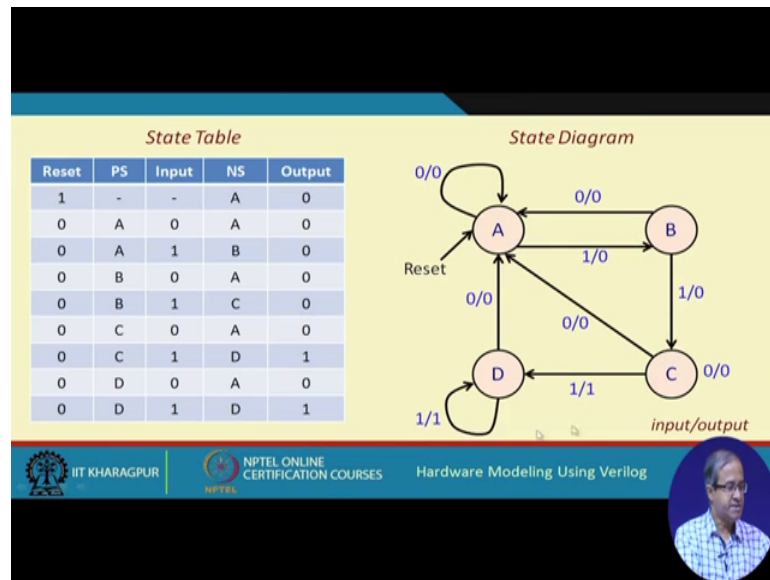
Now talking about a finite state machine; so a finite state machine or FSM in short, can be specified or represented in the form of a state table or in the form of a state transition diagram. Now in the examples that we shall be showing we shall be using the state transition diagram rotation, but you recall earlier when we talked about the user defined primitives or u d ps, there the sequential functions we represented in the form her table that was an example of a state table. We showed how to create the state table of a d flip-flop, d latch, j k flip-flop, then s r flip-flop and so on.

So, those are examples of state table that same information we can also represent in a diagrammatic form that is easier to visualize sometimes. We shall be showing such diagrammatic representations here; that is called state transition diagram. And of course, there are some variations which many people prefer to use. Say means one such structure that is used is called algorithmic state machine or ASM. So, I mean well just take some examples of ASM later during the course. So, when we talk about designing more complex systems.

So, some examples of FSM, let us say we have a circuit that detects 3 or more ones in a serial bit stream. Means I am assuming that there is one input where a sequence of bits are coming one by one; 1 0 0 1 1 1 0. They are coming continuously in synchronism with the clock let us say. So, here I want to find out whether there are 3 consecutive ones in the bit stream or not. So, every time 3 consecutive ones are detected, the output again a

single bit output should go to one otherwise the output should remain at 0. So, the output will become 1 whenever it detects 3, at least 3 or more consecutive ones in the stream. This is what our example here is. So, this is the.

(Refer Slide Time: 06:59)



State transition diagram, this is how it depicted of this example the circuit to detect 3 or more ones. This is the state table description; this is the equivalent state diagram. Let us explain first this state diagram it is easier to understand, then we shall see the state table. Here the circles they indicate states of the machine. Now how many states will be required for a particular description it will depend on the problem specification. For this case, because we are trying to detect 3 consecutive ones, we need at least 4 states. What does the 4 states will signify. This a will indicate the initial state, b will mean that we have seen one, a single one. So, far c will mean we have seen two consecutive ones, and d will mean we have seen 3 consecutive ones.

So, for this case we need at least 4 states. So, let us see the notation used are; this 0 slash 0 mean the first one refers to the input, after slash it is the output. So, the circuit has a single input and a single output. So, suppose I am in the initial state. So, when you reset the circuit it starts with state a now if I get a 0. So, it is not a one. So, I remain in state 0. So, I am waiting for the next one. So, whenever I get a 1 I go to state b 1 slash 0. So, still I have not got 3 ones, a single one, I go to state b. b remembers the information that I have seen a single one. So, so in b if I get another one I go to c. So, again the output is 0,

c remembers that I have seen two ones, and in c if I get another one I go to d with the output being made to one, because 3 ones have been detected, and while in d if I receive more number of ones I remain in state d and the output is also made one, but while in b if the next bit is 0 I have to go back to a, because again I will have to start with the next one. Similarly in c if I get a 0 I have to go back to state a, and similarly from d if I get a 0 I have to go back to state a this 0 0 does not mean anything. So, this is the state transition diagram.

Now this same information can be depicted in the form of a state table. So, what does the state table indicate. It indicates the inputs. So, here my inputs are; reset and this, this input value first bit. I am calling it as input and I have a single output. Now in addition to it there is the notion of the state. I call it present state as p s next state as n s. So, the idea is that if my reset is high, irrespective of the present state and the input applied my next state will be a. So, reset it goes to a and the output will be 0. So, if the reset is 0, then my operation starts, if my present state is a and the input is 0, then I go to next state a with an output 0. This is indicated by this arc.

If my present state is one input a and input is one, then my next state is b with the output 0 it is this edge input is 1 output is 0 next state is b. So, in this way for every edge in the state transition diagram, I can have one row in this state table. So, these are equivalent representations, but the diagram is more easily, means understandable visually, because this table is a little more complicated to interpret fine.

(Refer Slide Time: 11:07)

**Mealy and Moore FSM Types**

- A deterministic FSM can be mathematically defined as a 5-tuple  $(\Sigma, \Gamma, S, s_0, \delta, \omega)$  where  $\Sigma$  is the set of input combinations,  $\Gamma$  is the set of output combinations,  $S$  is a finite set of states,  $s_0 \in S$  is the initial state,  $\delta$  is the state-transition function, and  $\omega$  is the output function.
- Here,  $\delta : S \times \Sigma \rightarrow S$ 
  - Present state (PS) and present input determines the next state (NS).
- For Mealy machine,  $\omega : S \times \Sigma \rightarrow \Gamma$  (output depends on state + inputs)
- For Moore machine,  $\omega : S \rightarrow \Gamma$  (output depends only on the state)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Now, let us look at slightly more formal definitions of FSMs. There are two kinds of finite state machines that we distinguish; one is called a Mealy machine other is called a Moore machine. Let us see what these are. Now any finite state machine, well deterministic finite state machine means whenever we apply an input. Whenever I know what is the state.

So, the next state and my output is absolutely deterministic. So, all practical circuits are deterministic, but for modeling, theoretically there is another kind of FSM which is also defined which is called non deterministic, but from the practical design point of view, we are only considering deterministic of essence. So, mathematically it can be represented  $\Sigma, \Gamma, S, s_0, \delta$  and  $\omega$  where  $\Sigma$  denotes the set of input combinations. Suppose the circuit has two inputs; a and b, there will be 4 input combinations 0 0 0 1 1 0 and 1 1,  $\Sigma$  indicates the set of all the inputs. Similarly  $\Gamma$  indicates the set of all outputs. So, if there are 3 outputs, the set of all outputs can be 0 0 0 0 0 1 up to 1 1 1, set of all outputs is denoted by  $\Gamma$ .

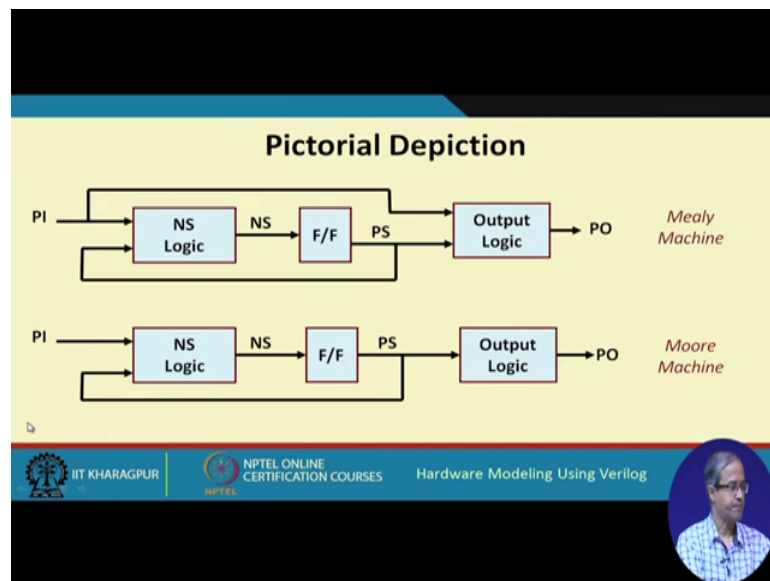
$S$  denotes the set of states, and  $s_0$  which is a member of  $S$ , is the initial state. So, we start with  $s_0$ .  $\delta$  is defined as the state transition function and  $\omega$  is the output function. So, how are they defined?  $\delta$  is defined as follows,  $\delta$  colon means this is the definition it means  $S \times \Sigma$  means this is a Cartesian product. So, some state,

given a state, given a input it will determine the next state. So, given some value from capital s given some input from capital sigma, it will determine some state in capital s.

So, this will actually tell what the next state will be, present state and present input will determine the next state. And for the output function, here we distinguish between Mealy and Moore machines. For a Mealy machine the output sigma will be determined by this state as well as the present input. So, the output depends on state plus the inputs, but for a Moore machine, the output does not depend on the inputs, it depends only on the present state, output depends only on the state.

So, in this way you can distinguish between Mealy and Moore machines. We shall see some examples.

(Refer Slide Time: 14:09)



Pictorially the Mealy and Moore machines can be shown like this. You see for both Mealy and Moore machines, there are two functions; one is the next state logic other is the output logic. The next state logic is actually the delta function, it takes a primary input, it takes my present state which has stored in flip-flops, and it generates my next state this is a combinational circuit n s logic, and the output logic is again similar it takes for Mealy machine it takes a primary input, and the present state it generates the primary output. So, for a Mealy machine the output is dependent on the primary inputs as well as the state, but for a Moore machine the output is dependent only on this state not on the inputs, this is the difference.

(Refer Slide Time: 15:05)

**Example 1**

- There are three lamps, **RED**, **GREEN** and **YELLOW**, that should glow cyclically with a fixed time interval (say, 1 second).
- Some observations:
  - The FSM will have three states, corresponding to the glowing state of the lamps.
  - The input set is null; state transition will occur whenever clock signal comes.
  - This is a *Moore Machine*, since the lamp that will glow only depends on the state and not on the inputs (here null).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Fine let us take an example, this is an example of a Moore machine. So, means our example is very simple. let us assume that there are 3 lamps red green and yellow. So, there is a circuit, there are 3 output lines the 3 lamps are connected to the 3 output lines, and there is a clock signal which is coming. There are no separate inputs there is only a clock, and these 3 lamps are supposed to glow cyclically with a fixed time interval, let us say 1 second interval.


Now clearly here the finite state machine will be having 3 states, because you have to remember that which red which lamp had blown in the last state, because it was red, if it was red then next time the clock comes it should be green, if it is green next time clock comes, it should be yellow and from yellow it should be red. So, this can be captured by a state transition diagram like this red to green, green to yellow, yellow to red.

So, this is very simple there are no separate inputs, a very simple finite state machine. This is a Moore machine, because the output depends only on which state it is in, it does not depend on inputs, because there are no inputs in this example, the input set is null fine.




(Refer Slide Time: 16:33)

```
module cyclic_lamp (clock, light);
  input clk;
  output reg [0:2] light;
  parameter S0=0, S1=1, S2=2;
  parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
  reg [0:1] state;
  always @(posedge clock)
    case (state)
      S0: begin // S0 means RED
            light <= GREEN; state <= S1;
          end
      S1: begin // S1 means GREEN
            light <= YELLOW; state <= S2;
          end
      S2: begin // S2 means YELLOW
            light <= RED; state <= S0;
          end
      default: begin
            light <= RED;
            state <= S0;
          end
    endcase
endmodule
```




IIT KHARAGPUR



NPTEL ONLINE  
CERTIFICATION COURSES

Hardware Modeling Using Verilog




So, let us see how we can code this example in verilog red green yellow. so the first thing is that let us make some assumptions. Let us say I define 3 states s 0

(Refer Slide Time: 16:46)

00 S<sub>0</sub> → RED  
01 S<sub>1</sub> → GREEN  
10 S<sub>2</sub> → YELLOW  
11 X  
2 bits

© CET  
I.I.T. KGP



S 1 and s 2; so I define is 0 as red S 1 as green and S 2 as yellow. So, talking about the finite state machine, you can have it like this; say this is s 0, this is s one, this is s 2. So, when you are in s 0, if there is a clock there are no inputs. So, you can write a dash, your this S 1 should glow, S 1 means green red green yellow or means if you put it in that

order red first green then yellow. So, if it is green. So, there should be red green yellow 0 1 0 should be the output

And when in S 1 if a clock comes, again there are no inputs S 2 means yellow, red green yellow. And in S 2 if a clock comes you go to S 1 it is red. So, first one will be 0. So, this will be my state transition diagram, there are no inputs, but there are 3 outputs and there are 3 states. Now the point to know is note that is that, here we are talking about hardware implementation. So, there are 3 states. So, we need two bits to represent the states. let us say S 0 we can indicate by this state 0 0, this S 1 we can indicate by a state 0 1, and S 2 by 1 0. Where is 1 1 is an invalid state, 1 1 will be an invalid state fine. So, let us see how in verilog we can code this state transition diagram.

So, we have declared a module, where there is only clock as the input, clock and light is the output, light is a vector, there are 3 lights red green and yellow, clock is an input, and we are declaring this state as a 2 bit variable 0 1 2 bits, and for convenience for ease of understanding, we are declaring a parameter constants S 0 we are calling it as 0 S 1 as 1 S 2 S 2. You see in binary 0 0 means 0 0 1 means 1 1 0 means 2. So, S 0 S 1 S 2 we are denoting by the decimal number 0 1 2. Similarly the 3 colors of the lamps were again defining as a parameter for convenience; red we are declaring as a 3 bit number 1 0 0, green as a 3 bit number 0 1 0, yellow as a 3 bit number 0 0 1, this will indicate which lamp we are glowing; first one is red, second one is green, last one is yellow.

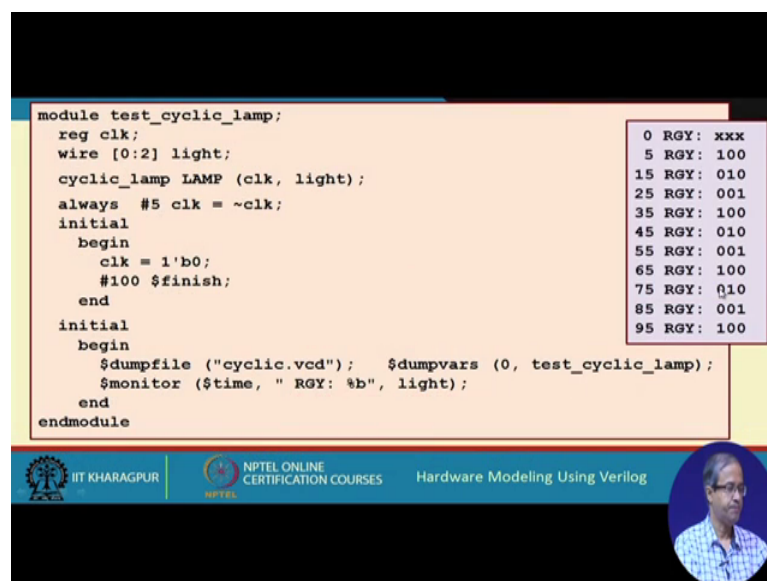
Now, our description is very simple here we are executing this procedural block triggered by the positive edge of the clock. So, whenever there is a positive edge of the clock, we are seeing in which state we are. Suppose we are in state S 0 well and a clock has come. So, my next state will green. So, my state will be changing to S 1. This S 1 will be indicating the green state you recall, S 0 is red, S 1 is green, S 2 is yellow, and my current output will become green.

Now if I am state S 1 which means green and if a clock comes my next will be yellow. So, state will go to S 2 and my light will be yellow and if it is in S 2 in yellow state will again go back to s 0, and it will glow the red light. Now when the circuit starts up suppose the states the flip-flops sorry they start with the 1 1 state something else. So, you will have to also give a default state. So, you are not initializing the state. So, if it is

something else other than S 0 S 1 or z 2, these are the 3 values then you start with red as the initial state, state S 0 radius assigned to light.

Now, you see here we have used only non blocking assignment statements. So, if this is given to a synthesis tool, the synthesis tool will be generating a circuit, where for this state it will generate two flip-flops, because it has to remember it, and because of the non blocking assignments we have used for light also, it will be using 3 flip-flops, and for every positive edge of the clock this assignments will take place right.


(Refer Slide Time: 21:41)



```
module test_cyclic_lamp;
  reg clk;
  wire [0:2] light;
  cyclic_lamp LAMP (clk, light);
  always #5 clk = ~clk;
  initial
  begin
    clk = 1'b0;
    #100 $finish;
  end
  initial
  begin
    $dumpfile ("cyclic.vcd");    $dumpvars (0, test_cyclic_lamp);
    $monitor ($time, " RGY: %b", light);
  end
endmodule
```

0	RGY:	xxx
5	RGY:	100
15	RGY:	010
25	RGY:	001
35	RGY:	100
45	RGY:	010
55	RGY:	001
65	RGY:	100
75	RGY:	010
85	RGY:	001
95	RGY:	100

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



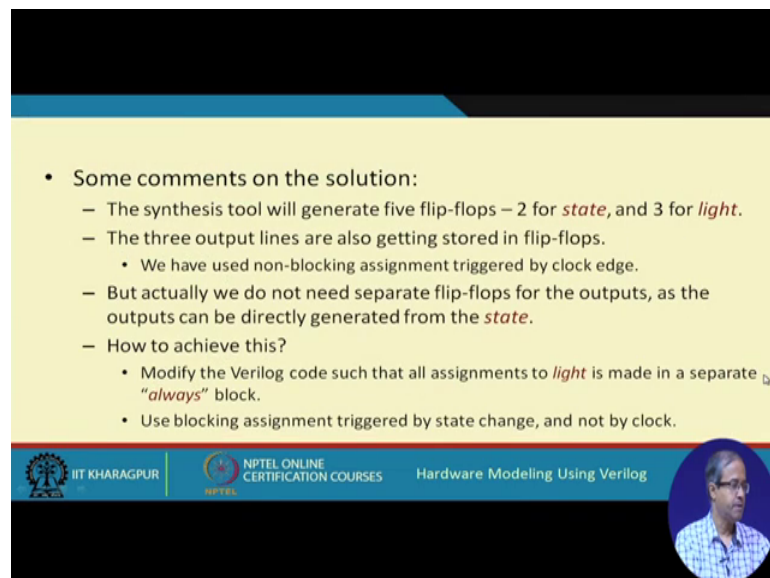
So, I have also shown a sample test bench for this, for this circuit to show how we can write a test bench. You see here we have defined this clock as reg, and the output of this circuit light as wire 3 bit. We have instantiated it called it lamp. In this initial block we have started by initializing clock with 0 at time t equal to 0, and in this always block after delay of 5 we are writing clock equal to not clock which means I am generating a clock signal with a time period of 10 units. It starts with 0 after 5 it goes to one, after 5 it goes to 0 again and so on, it repeats.

Well and here I am just applying clocks, and here I am saying, I initialize clock to 0 then this always block will take over and at type hundred I finish, because here I have no separate inputs to apply in this example, the clock is the only input. So, whenever a clock comes the lamp will change from one to the next. And in the just initial thing there is another block where I have specified a dumb file, where to dump the values value

change dump and 0 test cyclic lamp means, that here all the variables I want to dump, and also I have given a monitor where I am asking that you display the time and r g y equal to b the value of light.

So, if you just run the simulation we get an output like this. So, you see that that initially at time t equal to 0, this r g y are not initialized, because clock has not yet come. So, the values are x x x say at time 5 first positive edge of the clock comes. So, so it gets initialized to red, because it is the default case, initially it was x x x. So, default it starts with x and with a clock period of 10 15 25 35 45 like this goes and you see red then green then yellow, again red green yellow in this way it continues, and it will continue till 95, because at hundred you are finishing. So, it stops here right.


(Refer Slide Time: 24:04)



• Some comments on the solution:

- The synthesis tool will generate five flip-flops – 2 for *state*, and 3 for *light*.
- The three output lines are also getting stored in flip-flops.
  - We have used non-blocking assignment triggered by clock edge.
- But actually we do not need separate flip-flops for the outputs, as the outputs can be directly generated from the *state*.
- How to achieve this?
  - Modify the Verilog code such that all assignments to *light* is made in a separate “*always*” block.
  - Use blocking assignment triggered by state change, and not by clock.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



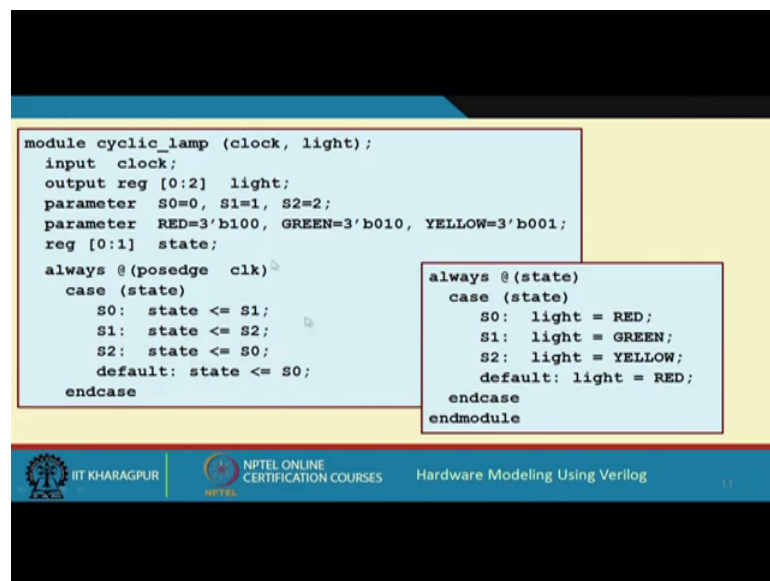
So, as I had said that in this solution 5 flip-flops will be synthesized; two for the state variables and 3 for the light. This is because we have used non blocking assignment triggered by clock for both state and light, this as I said, but you see for this design do you really need this. Do you really need to store the lamp output values in a flip-flop, because if I know this state then I will know that which lamp to glow. So, if I know that I am in state s 0, it means red should be activated, if I know I am in state S 1 I know green is to be activated. So, I really do not require a flip-flop to store the value of the outputs, storing the state is sufficient.

But because of the way I have specified the design the synthesis tool is generating flip-flops also for the outputs. So, let us see how we can overcome this. So, we make a modification. So, we split they always block in to two always blocks. First thing is that all assignments to light, will be made in a separate always blocks, where we use blocking assignments and not by clock, because whenever you specify an always block triggered by a clock the synthesizer will be made to believe that you are trying to do something in synchronism with the clock.

And hence you have to synthesize them as flip-flops or registers, but if you are using a blocking statement then the synthesizer will analyze, that well do you really need to store the values or is it a pure combinational circuit, it will try to decide; so the modified verilog code.

(Refer Slide Time: 26:07)

```
module cyclic_lamp (clock, light);
  input clock;
  output reg [0:2] light;
  parameter S0=0, S1=1, S2=2;
  parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
  reg [0:1] state;
  always @(posedge clk)
  case (state)
    S0: state <= S1;
    S1: state <= S2;
    S2: state <= S0;
    default: state <= S0;
  endcase
  always @(state)
  case (state)
    S0: light = RED;
    S1: light = GREEN;
    S2: light = YELLOW;
    default: light = RED;
  endcase
endmodule
```



Will look like this. So, you see, here we have split the always block in to two. So, in the first always block. The first part is the same no change. So, in the first always block, we are triggering it by the positive edge of the clock, and what is happening here only the state changes nothing else. We are doing a case on state. So, if it is in state S 0 my state will become s one, if it is in state S 1 it will become S 2 if it is in s 2, it will become a 0 and default it will start with s 0. Now in another always block, here I am using you see blocking assignment statements, and I am triggering it not by clock, but by state. So, whenever state changes you do this. What you do? If this state is S 0 you blow red. If the

state is S 1 you go green, if it is S 2 then yellow and default, since we are starting with a 0 it will be red ok.

So, if you do this if you make a specification like this what the synthesizer will see. Synthesizer will find that. Well, I see that I have specified the output value for all combinations of state for 0 0, it will be red 0 1, it will be green 1 0, it will be yellow and for 1 1 it will be red. So, it can actually find out that what will be my output just from the state. So, it will be a combinational circuit.

(Refer Slide Time: 27:44)

• Comment on the solution:

- The synthesis tool will be generating only 2 flip-flops corresponding to the first clock-triggered “always” block.
- The second “always” block will be generating a combinational circuit that takes *state* as input and produces *light* as outputs.

state ( $s_1, s_0$ )	Light (RGY)
S0: 00	1 0 0
S1: 01	0 1 0
S2: 10	0 0 1
11	x x x

Logic expressions after minimization:

$$R = s_0' s_1'$$

$$G = s_0$$

$$Y = s_1$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, just about this solution, this synthesis tool will be generating only the flip-flops for the states, but the second always block where you are initializing the light, this 1. For this no flip-flops or latches will be generated why because I can create a truth table where depending on the states I can uniquely identify the values of the light. So, I know if it is 0 0 1 or 1 0 which light to glow, but if it is 1 1 it is an invalid state. So, the output will be x x x. So, if you do a (Refer time: 28:21) map minimization of these 3 functions this is a two variable function let us say S 1 and S 0 are the two state variables,.

Then this r g and y outputs can be obtained as r will be just S 0 bar S 1 bar, g will be just S 0 and y will be just s 1. So, whenever S 1 is 1 then you glow yellow. So, whenever g, this S 0 is 1 here, you glow yellow green and so on. Red will be both are 0 0, both are 0 0 red. You see this is a pure combinational circuit description, and synthesis tool will be

looking at this it will find out that well this is actually a combinational circuit description, and it will be generating a combinational circuit.

So, with this we come to the end of this lecture. So, we had talked about finite state machines, distinguished between Moore and Mealy machines, and worked out one example, that how we can represent the FSM, and how we can code it in to Verilog, and we looked at two alternate kinds of design; one in which unnecessary flip-flops are synthesized, and the other, and improved one, where only the essential flip-flops are generated.

Thank you.