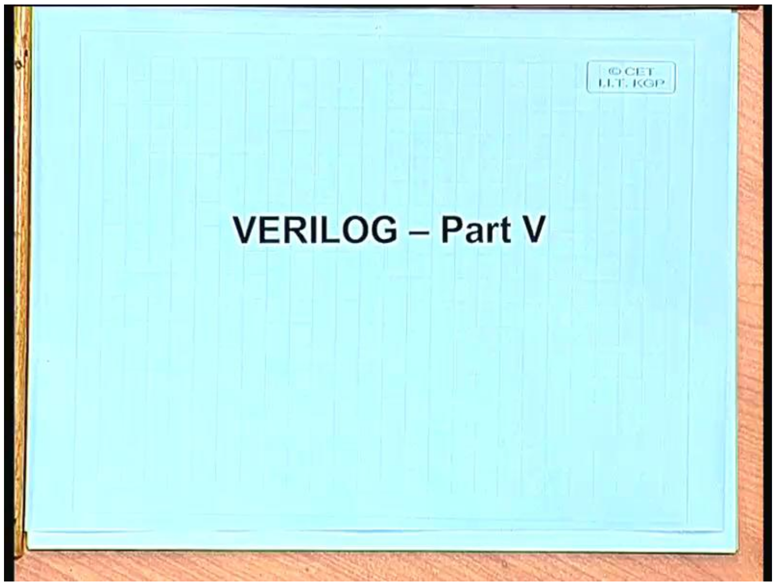
Electronic Design Automation

Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No #6
Verilog: Part V

(Refer Slide Time: 01:06)

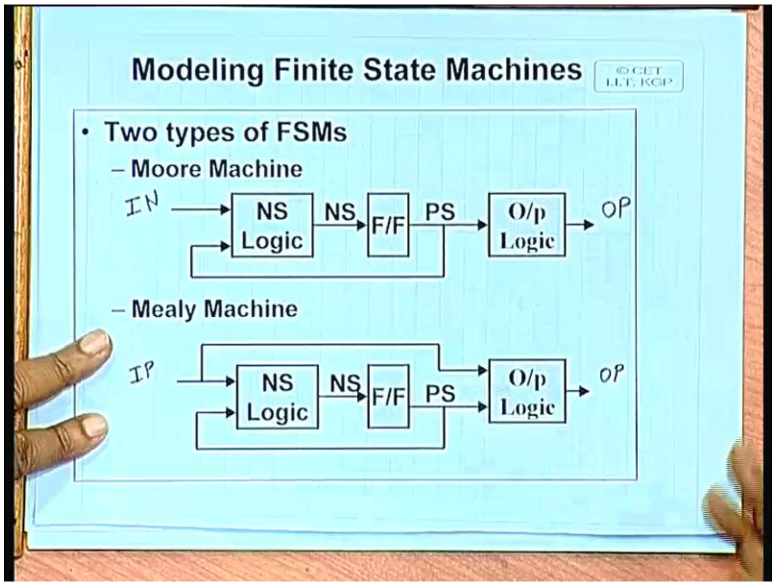


© CET
I.I.T. KGP

VERILOG – Part V

So in this lecture we continue our discussion on the Verilog language. If you recall in our last lecture we were talking about finite state machines.

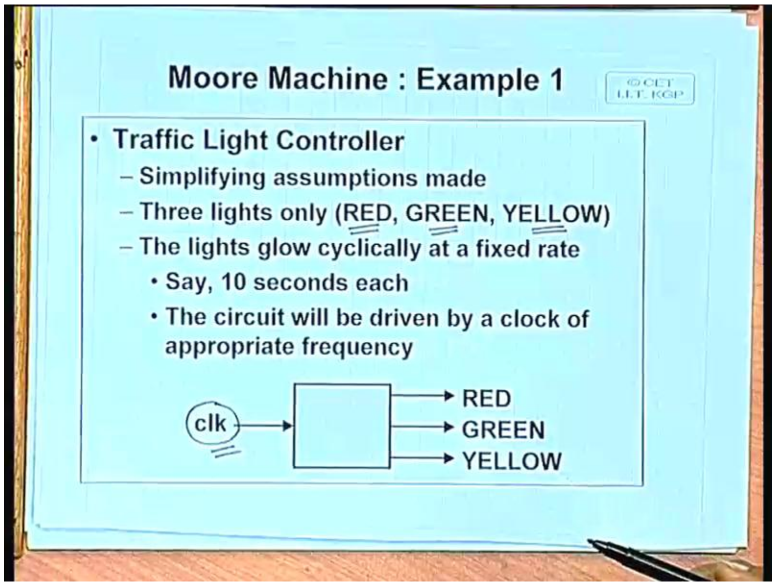
(Refer Slide Time: 01:20)



And we had said that we would be seeing that how we can model such finite state machines using Verilog. Now we had mentioned that there are 2 broad classes of FSM 2 types Moore machine and Mealy machine. The essential difference between these 2 just you recall is that in a Moore machine the next state of the sequential circuit depends on the present state and the presently applied inputs okay. But whereas the output these are the outputs of the circuit the outputs depend only on the present state and is independent of the applied primary inputs. Now in contrast a Mealy machine has both its next state and the output functions depending on the applied inputs and the present state.

Next state depends on these 2 through a logic called the next state logic. Similarly the output logic generates the output as a function of the present state and also the input. So we will see that how we can model both these classes of machines Moore and Mealy with the help of some examples. First let us talk about a Moore machine where the output can be directly derived from the state without any reference or means without using the present value of the primary inputs okay.

(Refer Slide Time: 02:52)



Moore Machine : Example 1

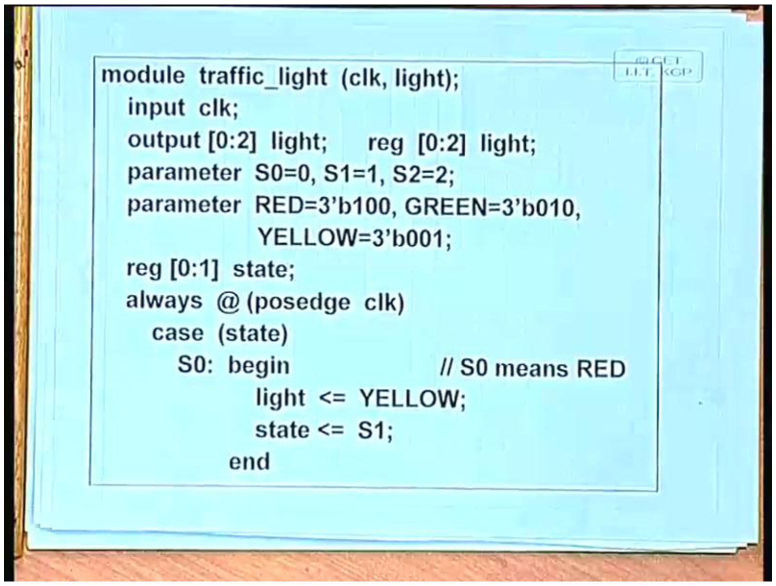
- Traffic Light Controller
  - Simplifying assumptions made
  - Three lights only (RED, GREEN, YELLOW)
  - The lights glow cyclically at a fixed rate
    - Say, 10 seconds each
    - The circuit will be driven by a clock of appropriate frequency

clk
RED
GREEN
YELLOW

So the example we take to illustrate the modeling of a Moore machine is a simplified version of the traffic light controller. Now here what we assume is that we have some thing similar to a traffic light controller. But instead of well going into the complexities just for the sake of illustration we assume that there are only 3 lamps or lights we need to control red, green and yellow. And without loss of generality, we assume that the 3 lights have to be displayed or glowed cyclically at equal intervals of time. Well means in practice the times will not be equal of course. But for the sake of illustration we assume that they will be glowing cyclically at a fixed rate. These are the simplifying assumption we have only 3 lights.

So we consider only 1 road junction only 1 road. So only 3 lights we need to consider red green and yellow and we assume for the purpose of this example is that these 3 lights glow cyclically at a fixed rate. Now this rate can be anything the rate is determined by this applied external clock. This clock can be as slow or as fast as we desire and these lines will be glowing cyclically along with this fine. So now let us see how we can model this using Verilog okay.

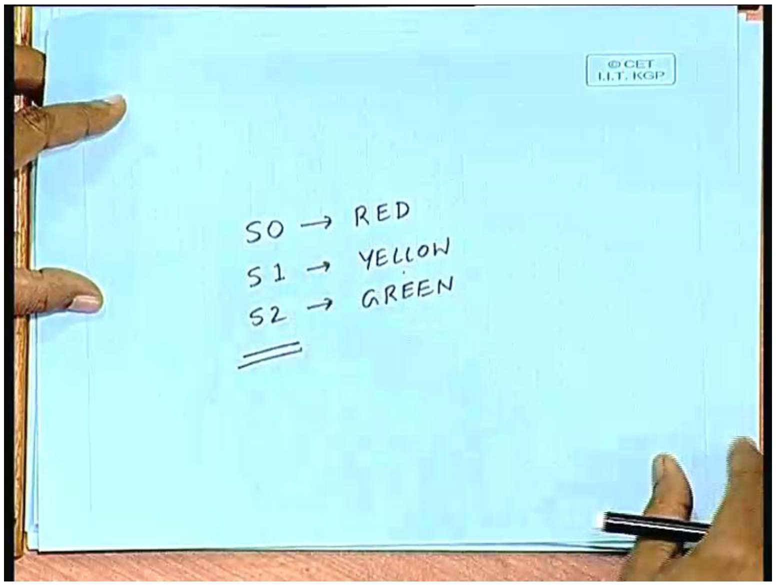
(Refer Slide Time: 04:34)



```
module traffic_light (clk, light);
  input clk;
  output [0:2] light;    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: begin                // S0 means RED
            light <= YELLOW;
            state <= S1;
          end
```

This is the first part of the code for the module. So we have the traffic light module which takes an input which is the clock CLK is the input and the output is light actually light is a vector. You can see it is a vector 3 bits actually. There are 3 lights this is your light. Now depending on which light is glowing presently, this circuit clearly can have 3 different states. Now in this program we are classifying these 3 different states as 0, 1 and 2 in terms of the internal state variables and we call them s0, s1, s2 by these parameter constant definitions. So the 3 states will be defined as s0, s1 and s 2. Now means our interpretation is like this.

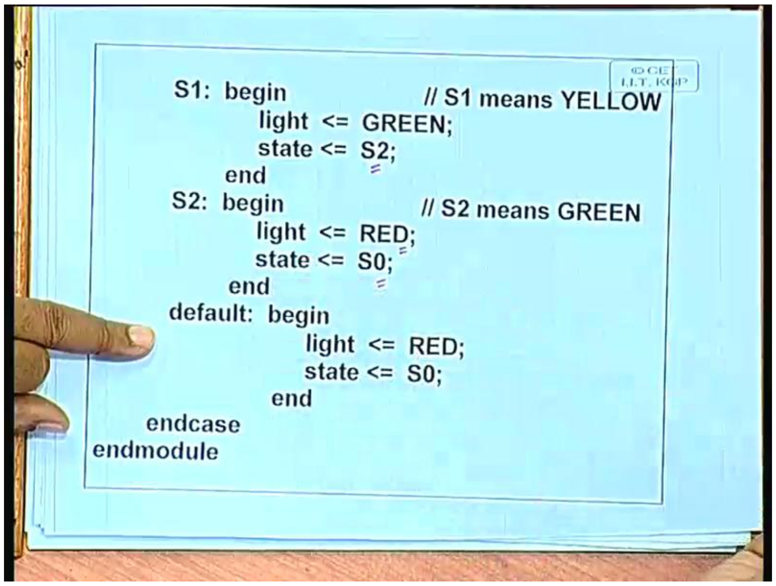
(Refer Slide Time: 05:38)


© CET I.I.T. KGP
S0 → RED
S1 → YELLOW
S2 → GREEN

This s0 in state 0 we will assume that the red light is glowing in the state; s1 we will assume that the yellow light is glowing and in s2 it will be green so these are the interpretations of the different states okay. It is not easy to code. Now the 3 lamps which are glowing, this we assume that the first 1 is red, the middle 1 is green, this 1 is yellow. So again this 3 bit pattern of the light 10 00 10 or 00 1, well 1 means the lamp is glowing 0 means it is not glowing. So these 3 we again define as a constant parameter red we call 10 0 as red 0 1 0 as green and 0 1 as yellow this names I have given just for the sake of convenience for nothing else and since the machine the FSM can be in 1 of 3 different state we need minimum 2 state variables or 2 flip flops to store the state. That is why the internal state you are representing as a 2 bit vector it is of type reg of 2 bits.

Now here the states are changing always in synchronism with the clock we are assuming that they will change at the positive edges of the clock. So the code goes like this. Always at the positive edge of the clock, we do a switch based on the value of state. Suppose we are currently in state 0 state 0 means the red light was on. So now if a clock comes we have to light the next 1 yellow. So if it is state 0, then we will have to go to the next state. The next state the way we go is that we first store this yellow pattern into light so that the yellow light glows and s1 will be the new state now okay. So to continue this similarly if you are in state s1, this is yellow.

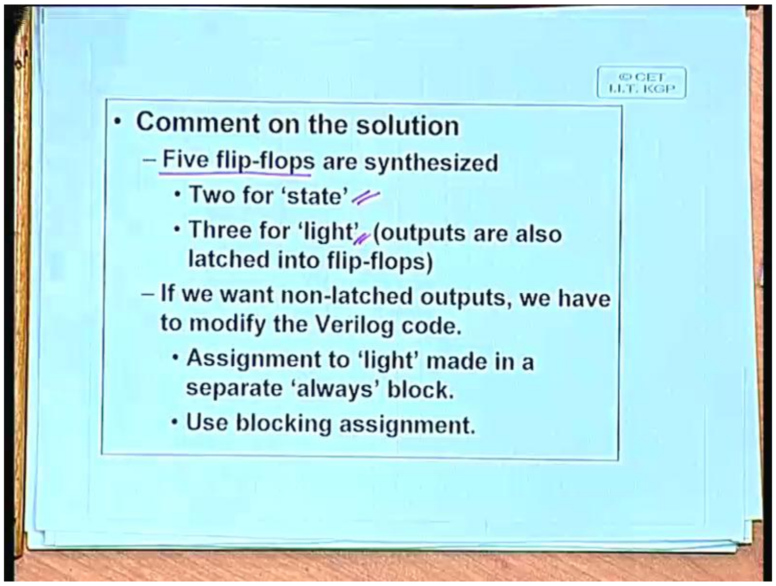
(Refer Slide Time: 08:04)



```
S1:  begin                   // S1 means YELLOW
          light  <=  GREEN;
          state <=  S2;
     end
S2:  begin                   // S2 means GREEN
          light  <=  RED;
          state <=  S0;
     end
default:  begin
               light  <=  RED;
               state <=  S0;
          end
   endcase
endmodule
```

So if a clock comes next it will be green which is s2. Similarly if you are in s2, if a clock comes you will go to red; red means s0 and if you, if means at the initial state when you start the machine if the state variables becomes by mistake 1 1. So in order to set it to red we set the default clock also if it is default the light will be red and state will be s0. So by default you always start with red okay. Now we look at this code once more you see the way we have written this code of course we have given a always block with a clock triggered event. So this will be modeled as a sequential circuit obviously and these assignment statements are non-blocking. This will imply that the light and the state these variables where you are storing the values these will be synthesized as registers that is flip flops okay. So this solution will be having the following characteristics.

(Refer Slide Time: 09:21)

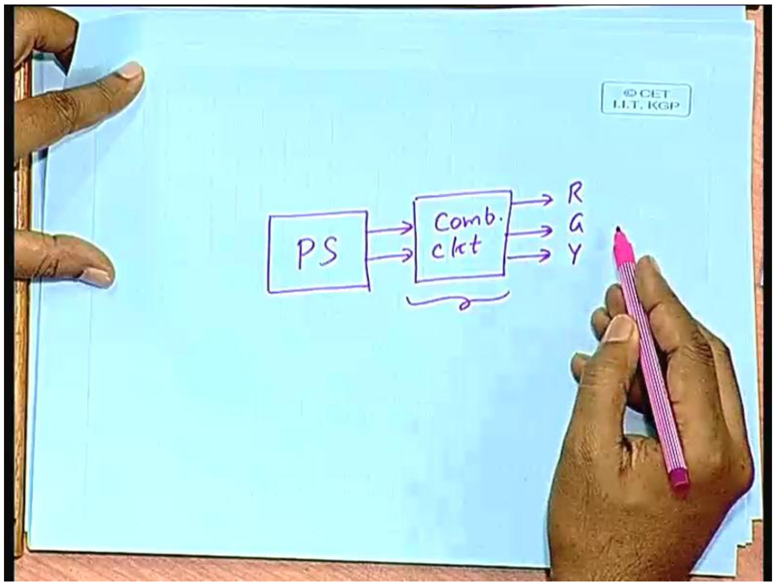


© CET I.I.T. KGP

- Comment on the solution
  - Five flip-flops are synthesized
    - Two for 'state'
    - Three for 'light' (outputs are also latched into flip-flops)
  - If we want non-latched outputs, we have to modify the Verilog code.
    - Assignment to 'light' made in a separate 'always' block.
    - Use blocking assignment.

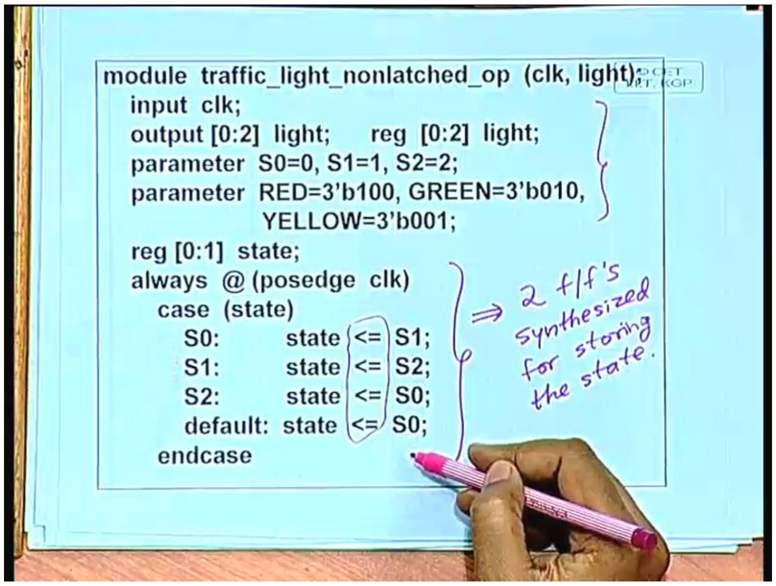
There will be 5 flip flops which will get synthesized 2 for the state variables and 3 for the lights right. But 1 thing you understand the lights which are being generated these are stored in some latches and from they are generated. Let us again come back to this code this light we are storing this red pattern into this light variable and s0 into the state variable. Now this light variable this by definition the way you have written the code. This is defined as a register and this pattern will get stored into a register. So this is how the synthesis will take place. But suppose we want to carry out the synthesis in a way where we get or means we specify that the lamps will be driven like this.

(Refer Slide Time: 10:29)



Suppose we have the state, so this represents the present state with 2 flip flops. So whatever is the present state well I can always design some combinational circuit out here and from this combinational circuit I can generate those 3 bit pattern red, green, yellow. So really we do not need a separate latch out here to store the value and from there to drive the lights this part of the circuit can be a combinational circuit alone. So now let us see how we can specify that since actually what we want is that we want non-latched outputs. The states have to be latched the states will have to be stored in flip flops at the outputs which are getting generated we do not want any latches out there. So if in modeling a Moore machine we do it in the way we have just shown then the outputs will be latched by default. But if you want non latched outputs we have to change the Verilog code in a slightly different way. The thing we do is that we use 2 separate always blocks 1 for the state and 1 for outputting the light combinations and 1 of them will be non-blocking. 1 of them will be blocking. Let us see and how so we make some changes to the Verilog code we do something like this.

(Refer Slide Time: 12:08)

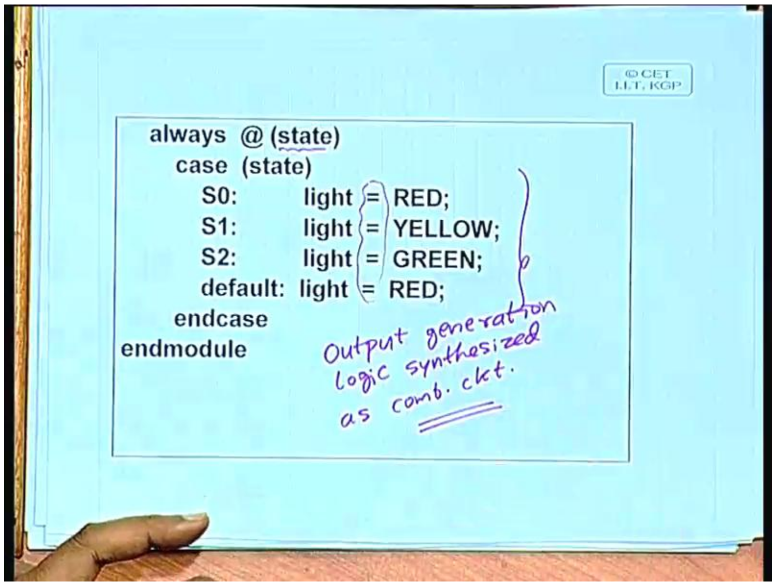


```
module traffic_light_nonlatched_op (clk, light);
  input clk;
  output [0:2] light;    reg [0:2] light;
  parameter S0=0, S1=1, S2=2;
  parameter RED=3'b100, GREEN=3'b010,
            YELLOW=3'b001;
  reg [0:1] state;
  always @ (posedge clk)
    case (state)
      S0:      state <= S1;
      S1: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

⇒ 2 f/f's synthesized for storing the state.

The first part is the same there is no change here. But after that whenever the clock comes we are dividing this into 2 parts. That means in the earlier code both the state variable and the lights were assigned in the same always block. But here in the first always block we are only using this first state changes if there is a positive edge of the clock if the state is s 0 then go to s 1. If it is s 1 can go to s 2, s 2 then go to s 0, default go to s 0. So here we are only modeling the state change behavior and since this is non blocking assignment. So this will imply that there will be 2 flip flops synthesized for we can say storing the state. But we have not mentioned anything about output here. Now output we will be specifying in a separate always block and that always block need not be triggered by the clock.

(Refer Slide Time: 13:27)

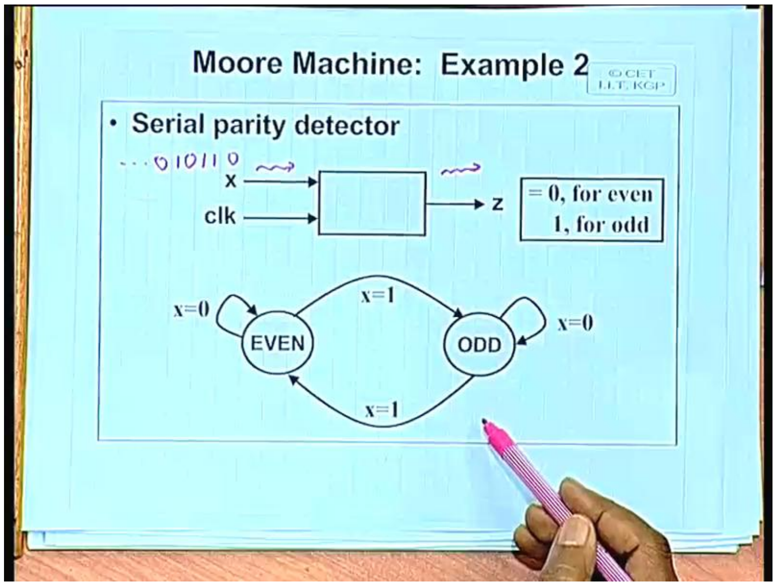

© CET I.I.T. KGP

```
always @ (state)
  case (state)
    S0:      light = RED;
    S1:      light = YELLOW;
    S2:      light = GREEN;
    default: light = RED;
  endcase
endmodule
```

Output generation logic synthesized as comb. ckt.

What we do is that the second always block we trigger it by changing the state because change in the state will be taking place anyway in synchronism with the clock. So here indirectly we are specifying that whenever there is a change in state we execute this block and you recall this is a non blocking assignment sorry blocking assignment equal to. So blocking assignment normally if it is not triggered by a clock this is synthesized into combination logic. So if it is s 0 this red pattern is sodium light, if it is s 1 yellow is stored, if it is s 2 green is stored, if it is default red is light. So if you specify in this way then what will happen is that the output generation logic synthesized as combinational circuit. So if we really want this normally in a Moore machine we want this Moore machine. There is no point in storing the output in a latch. Because anyway the output can be directly derived from the state it is better to have a combinational logic to do that okay. So for Moore machine this is a good idea or this may be the better way to specify the next function okay. So let us take another example of a Moore machine before moving to an example of Mealy machine okay. So the example we take is a serial parity detector.

(Refer Slide Time: 15:15)



Moore Machine: Example 2

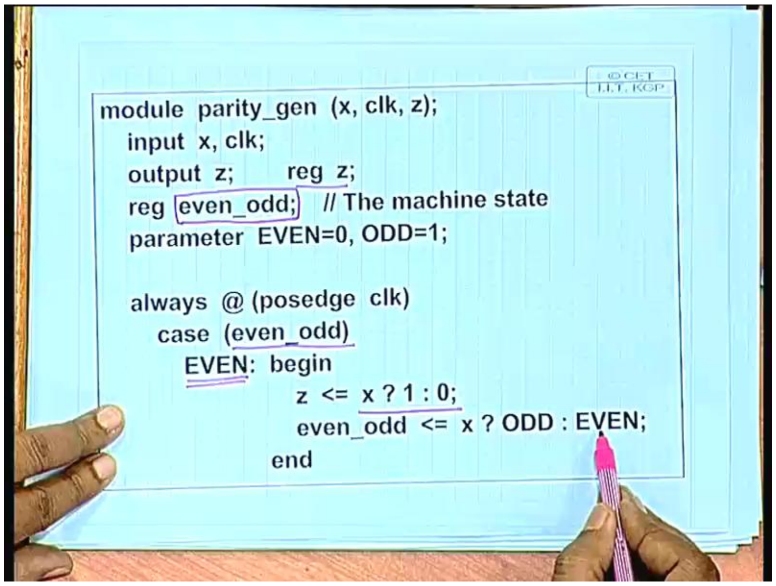
- Serial parity detector

© CET I.I.T. KGP
...010110
x
clk
z
= 0, for even
1, for odd
x=0
x=1
EVEN
ODD
x=0
x=1

Well here we assume that we want to design a circuit block like this which will be having a bit stream x; x is a stream of bits serially it is coming it may be 0 1 1 0 0 1 0 anything is coming stream of bits is coming. And they have been fed to the circuit in synchronism with the clock the output z will be continuously monitoring the parity of the bit stream which is which has been so far 0 will be for even parity 1 will be for odd parity. So as the bit streams are coming the streams of the parity will be coming out continuously first for 0 it will be 0 0 1 it will be 1 0 1 1 it will be 0 0 1 1 0 it will be 0 and so on. So it will continuously compute the carry of the bit stream. That is, that has been seen so far and the output in the next time instant will be the corresponding parity value.

This can be very easily modeled by a state diagram like this. So here we have 2 states we are remembering the state whether the bit pattern we have seen. So far has a even parity or a odd parity these are the 2 states. So if it is even parity and an input 0 comes it remains even if it is odd and an input 0 comes it remains odd well only change takes place if the input next input is 1. Then even becomes odd and odd becomes even. So this example illustrates that I have an FSM model that a state transition diagram like this or a state transition table from there I want to write the Verilog code. (( )) (17:06) No input is from this order 1 by 1.

(Refer Slide Time: 17:19)

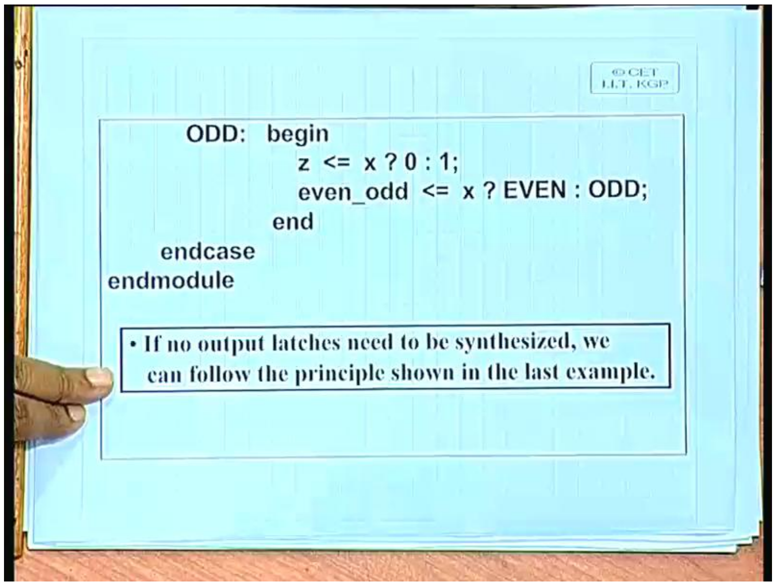


```
module parity_gen (x, clk, z);
  input x, clk;
  output z;      reg z;
  reg even_odd;   // The machine state
  parameter EVEN=0, ODD=1;

  always @ (posedge clk)
    case (even_odd)
      EVEN: begin
               z <= x ? 1 : 0;
               even_odd <= x ? ODD : EVEN;
             end
```

So the way we write the code is like this. Parity generator is as I mentioned x and clock are the input z is the output. Output I am declaring as reg and this is the state I am calling the state even odd. Since there will be 2 states 1 variable is sufficient and even or odd I am calling 0 as even and 1 as odd and using parameter I am giving the names of these constants. Now let us see how we do? Always whenever there is a clock we are checking the state and doing accordingly case even, odd. If the state is even then you again come back to that diagram previous diagram. If the state is even then if the input 0 comes, you will remain in the same state and also the output will be 0. Because this even means output is 0 okay. So you see in this code if it is even we are first thing we are given a conditional if x is true then 1 else 0 exactly what is specified in the diagram. Similarly for the next state if x is 1 then go to odd else even okay. So here the style of modeling is the last 1 the means for the first 1 we have mentioned using non blocking assignments where both the output logic and the state will be synthesized as flip flops. Similarly this state is odd we do a similar thing.

(Refer Slide Time: 19:09)

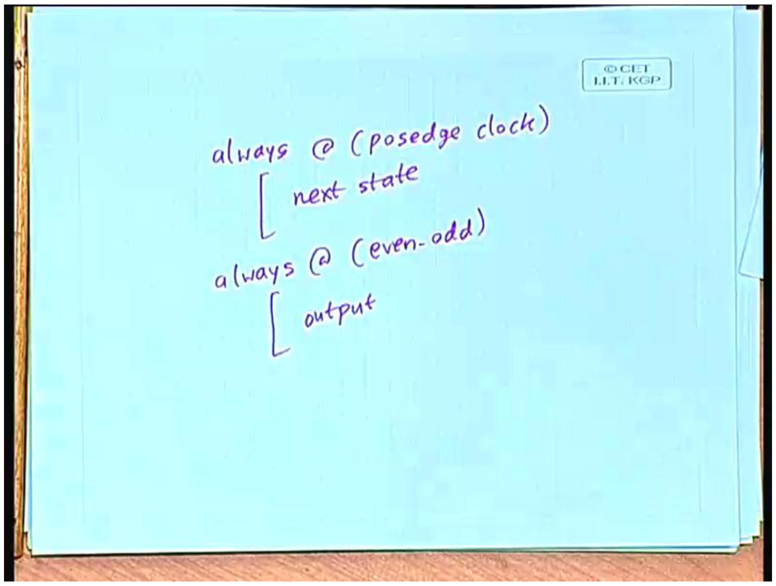


```
        ODD:  begin
                  z <= x ? 0 : 1;
                  even_odd <= x ? EVEN : ODD;
              end
     endcase
endmodule
```

• If no output latches need to be synthesized, we can follow the principle shown in the last example.

If x is 1 then we go to the other 1 0 else 1, this will be the output and state if x is 1 then go to even otherwise you go to odd. So this style models the first 1 where both outputs and states are synthesized as flip flops. But if you do not want that then you can again break it up into 2 always blocks. As I mentioned earlier you can do the same thing. So if you do not need the output latches, then we can follow the principle in the last example whereby you can use 2 different always blocks.

(Refer Slide Time: 19:50)

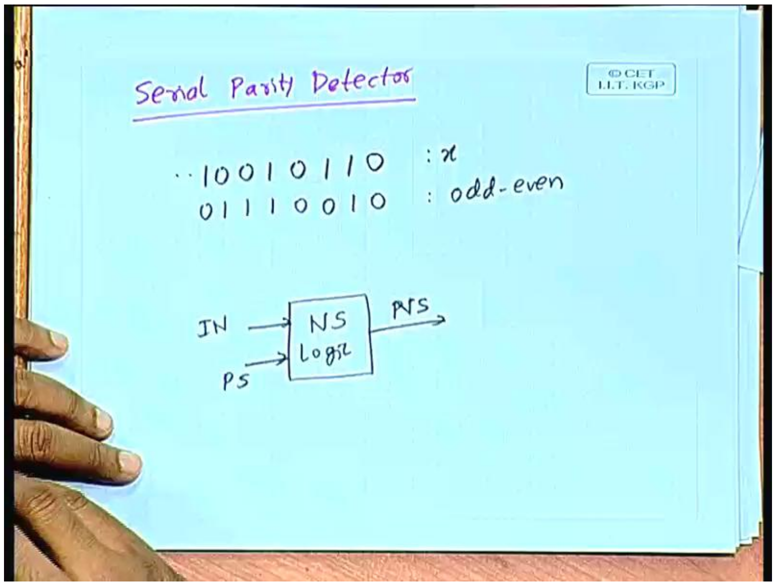


©CET
I.I.T. KGP

always @ (posedge clock)
    [next state
always @ (even-odd)
    [output

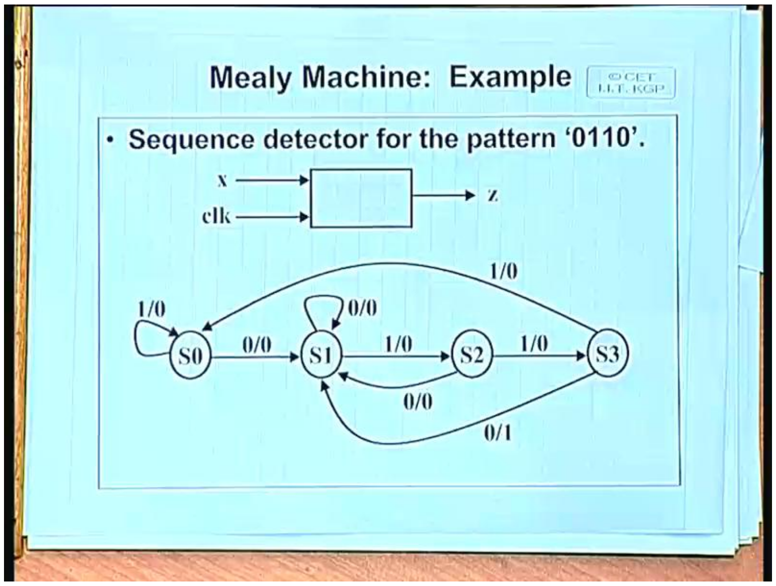
Well means in 1 always block we can trigger by the posedge of the clock. So here we will only be changing the state there will be another always block. So here we will be changing that only when that even or changes state. So here so next state computation will be here and the output computation will be here. So you can do the same thing. (( )) (20:28) Yes. (()) (20:32). In the first example the output was dependent on the input. But here there whatever is the state that same thing you can take as the output the combinational circuit is just a wire. Nothing else is needed. But in general you can need some transformation. (()) (20:50) Mealy machine means the output depends also on the primary inputs. But in that example was it depending on the primary input. (21:03) (()) No, no not really let me let me try to correct it out.

(Refer Slide Time: 21:20)



See here what we are actually designing is a serial parity detector right. Serial parity detector. Let us take an example and illustrate, suppose this is okay. This is the first bit which is coming I am showing from this side onwards. Suppose the bits are coming in this way this is what I am calling as x and state odd even I am trying to write odd even depending on whatever we have seen so far the state will be set accordingly. So I have seen 0 the state is even 0 0 1 state is odd, 0 1 1 state is back to even, even, odd, odd, odd, even, so these are the states. Now you look at the model of the Moore machine again there is a next state logic. Next state logic generates a next state in response to the present state and the present input. But in this example whatever is the present state. (()) (22:54) So, from state you can directly derive the output for the output you do not need x. This, what I mean. This you can categorize as a Moore machine this is not really a Mealy machine. (()) (23:11.490) The way we wrote the code it look like Mealy machine we try to mix them up but actually the way it is working it is independent of the input whatever state you are in from the state you can directly derive that what should be the output at the current point okay fine.

(Refer Slide Time: 23:36)

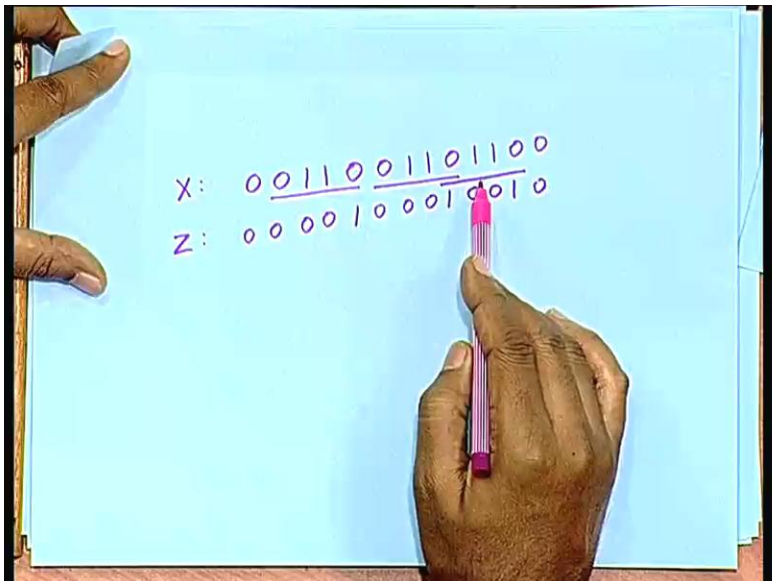


Mealy Machine: Example

- Sequence detector for the pattern '0110'.

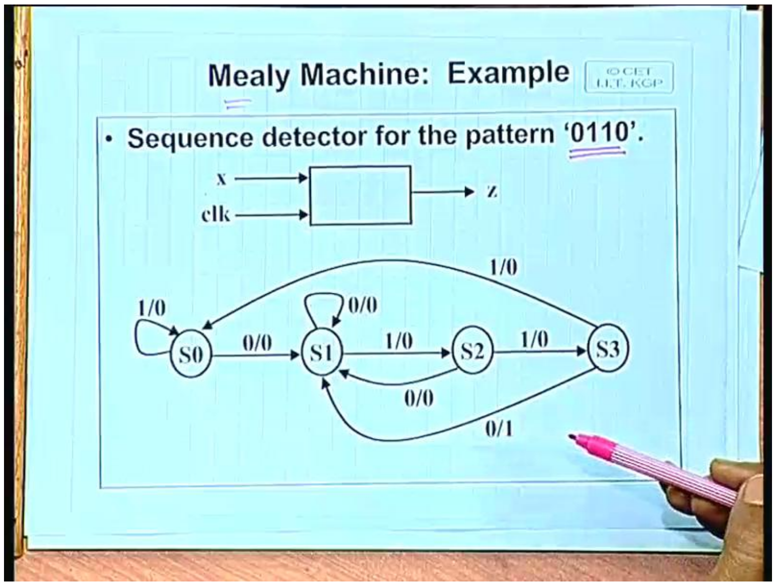
So now let us take an example of a regular Mealy machine. So here we take a slightly more complex example a sequence detector. So a sequence detector this is the state transition diagram we want to detect the sequence 0 1 1 0. In this input pattern x this x is again a serial bit pattern coming in synchronism with the clock and z is the output. Now here what I mean is that what I really want is that.

(Refer Slide Time: 24:10)



Suppose well I am writing from left to right for continuous suppose I have a well I want to detect 0 1 1 0 okay. Suppose I have a pattern 0 0 1 1 0 0 1 1 0 1 1 0 0 suppose. So the output z will be detecting the first 0 1 1 0 pattern here. So the output will become 1 here. It will detect this again 0 1 1 0 pattern here. So second time the output will become 1 here and we are also the way we will be designing we will also be allowing overlapping patterns. So the third 0 1 1 0 pattern will be this one; overlapping with the previous. So the third time the pattern will be detected here this is how we are going to design the machine okay. So here we will be allowing this overlapping pattern also 0 1 1 0 the last 0 of the previous pattern can be the start of the next pattern right.

(Refer Slide Time: 25:19)



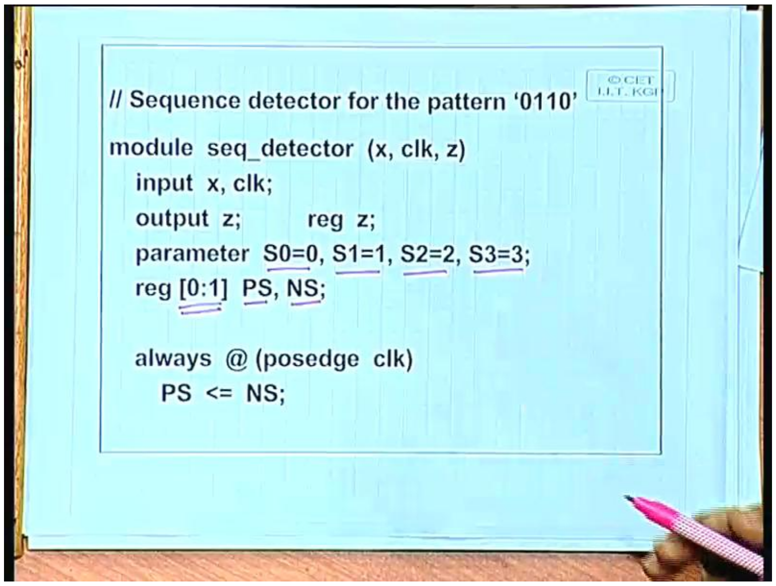
Mealy Machine: Example

- Sequence detector for the pattern '0110'.

Let us see just we look at the state diagram we will understand. See s 0 is the initial state we do not know anything well we get s 0 we move to s 1 we get a 1 we move to s 2. This is the input slash this is the output, so we go to s 0 we given s 0 to s 1. If the input is 0 and the output is 0 because we have not seen the entire pattern as yet. Similarly the next 1 leads us to s 2 the third bit leads us to s 3. Now when we get the final 0 well we have detected the pattern. But instead of going back to s 0 we go back to s 1 because this 0 is possibly the beginning of the next pattern we do not know. That is why we move back to s 1 not s 0 and here you see that the output is 1. But if you have otherwise say s, while in s 1 means I have seen the first 0.

Now if stream of zeros comes I will remain in s 1 because I have seen the 0 s 2 if a 0 comes I again come back to s 1 because I again start with pulse 0 from s 3. If there is another 1 comes I will again have to begin from s 0 s 0 if a 1 comes I remain here. So this is the state transition diagram I want to model. Now you see in this example the output which is being generated that does not only depend on the state it also depend on what input we are getting at that point in time for example while in s 3 if the input is 0 the output is 1 if the input is 1 the output is 0. This is what Mealy machine is all about so the outputs are also dependent on the primary inputs okay. So the way we can model this is as follows.

(Refer Slide Time: 27:31)

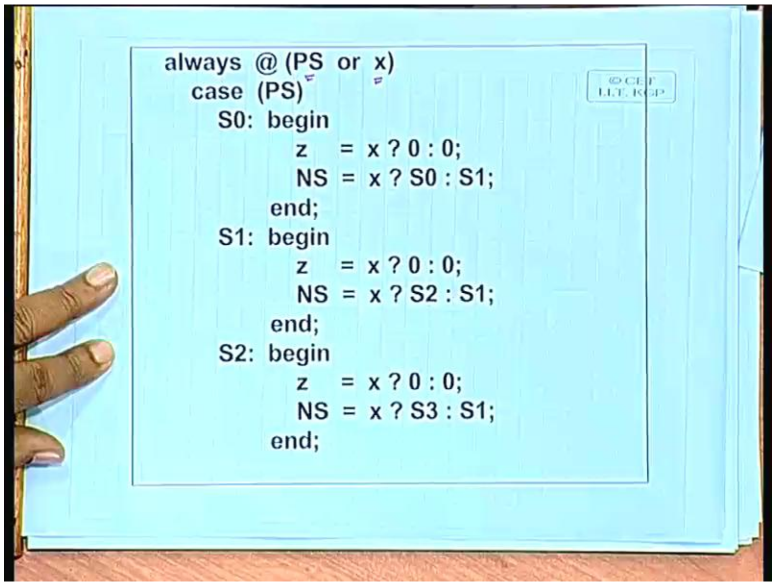


```
// Sequence detector for the pattern '0110'
module seq_detector (x, clk, z)
  input x, clk;
  output z;        reg z;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [0:1] PS, NS;

  always @ (posedge clk)
    PS <= NS;
```

First part is similar we have 4 states this you denote as 0, 1, 2 and 3 we call them s 0, s 1, s 2, s 3, we have 2 we have the present state. And next state both they will have to represent by 2 bits because there are 4 steps and we use 2 different sets of always blocks. This is the style to be followed for Mealy machine you just note in the first 1 we simply give an always block where whenever there is a clock the next state gets transferred to the present state nothing else the next state becomes a present state.

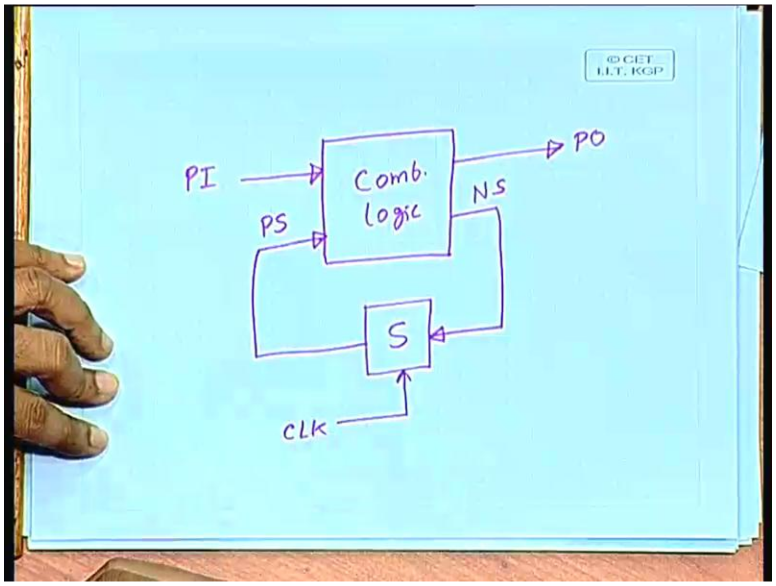
(Refer Slide Time: 28:20)



```
always @ (PS or x)
  case (PS)
    S0: begin
          z  = x ? 0 : 0;
          NS = x ? S0 : S1;
        end;
    S1: begin
          z  = x ? 0 : 0;
          NS = x ? S2 : S1;
        end;
    S2: begin
          z  = x ? 0 : 0;
          NS = x ? S3 : S1;
        end;
```

Here is another always block where we are triggering by any change of the state or any change of the primary input depending on which state we are in, we are computing the output we are computing the next state. Let us try to understand what you are doing in the model of a sequential circuit.

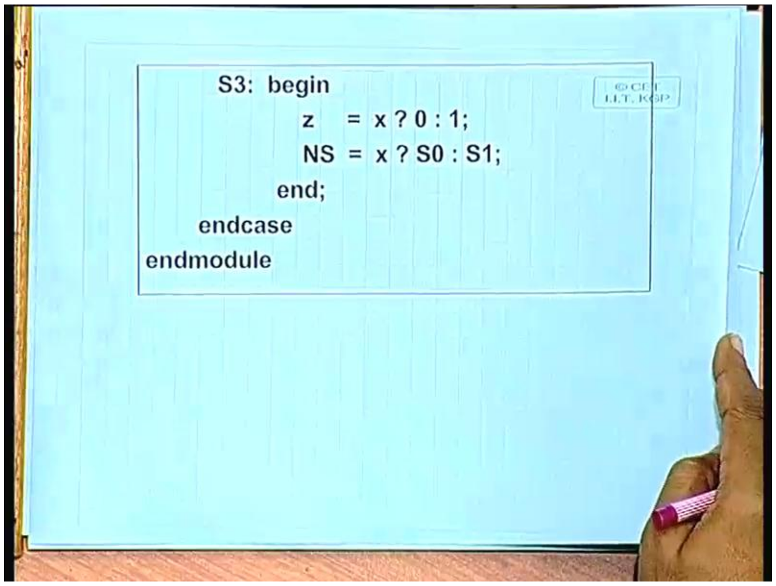
(Refer Slide Time: 28:47)



Well I am drawing it in a slightly different way we have a combinational part. This is the combinational logic, this is your primary input, this is the primary output and you have some state variables here okay. Let us call it is so the output of the combinational circuit is fed here and this is fed here so this is your present state and this is your next state. So both the primary output and the next state will be some function of primary input and present state. Now if you look at the code in the first always block we are simply assigning a next state to present state. Whenever there is a clock, so what it means with respect to this diagram is that clock is only coming here whenever there is a clock next state gets transferred to present state. No other computation is clock driven computation of primary output computation of next state that is purely through a combinational logic that you can compute continuously.

But whenever there is a clock pulse this NS value will come to the output of this okay. That is why for computing the primary output and the next state we have used another always block okay. Now in this always block depending on the state we are computing the primary output we are also computing the next state. Now these we have directly taken from the state transition diagram that if you are in state s0, then if the input is x then the output is x otherwise also output is, output is 0, output is 0, otherwise also. Next state if the input is 1 you remaining s 0 if the input is 0 you move to s 1 just from the state diagram you can directly code this. So in a similar way you can code the output transition and next state transition for all the different combinations.

(Refer Slide Time: 31:22)

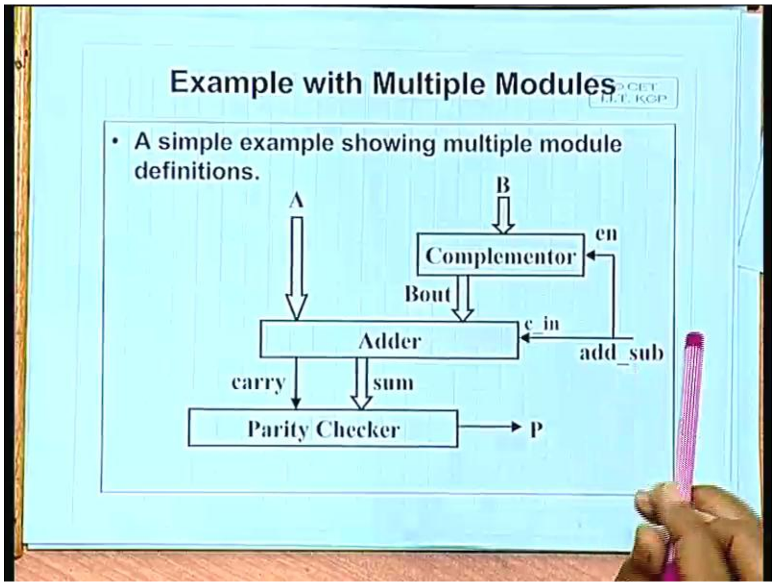


```
        S3: begin
                z    = x ? 0 : 1;
                NS = x ? S0 : S1;
              end;
      endcase
endmodule
```

So this is s 0, s 1, s 2 and s 3. So actually what we get finally will be a circuit like this where only the state will be coded as flip flops and these 1 these seems to be have coded as blocking assignments. They will be mapped into combination logic not sequential circuits. So the generation of primary output and next state will be through combination logic only. So if you follow that particular design style you will be getting a circuit where you will be. The model will be like this. So the state transition will be through flip flops the remaining will be through combination logic okay fine. So now that you know that how you can model a Moore machine and a Mealy machine. Suppose I give you a state transition diagram you can write the code if I give you a state transition table which is another way of representing the diagram state transition diagram. You can also do it or if I specify the FSM is any other way through a set of if then else statement. For example then also you can directly transfer it into a Verilog code.

So any FSM if we have in whatever way you can code it in FSM; in the language Verilog. But it but it depends whether you want the output to be latched or not latched. So the kind of assignment statements will be giving in the always block will depend on that blocking or non-blocking okay. So the one last example, we would be discussing during this lecture. That is an example where we would be showing you some kind of modular design or some kind of hierarchical design. So means any big system design. Suppose we want to design a system which is relatively big. So a natural way of designing will be to break it up into smaller pieces or smaller sub modules design. The sub modules individually and then try to combine them up. So I will try to illustrate with a simple example that how this is done.

((Refer Slide Time: 33:48)

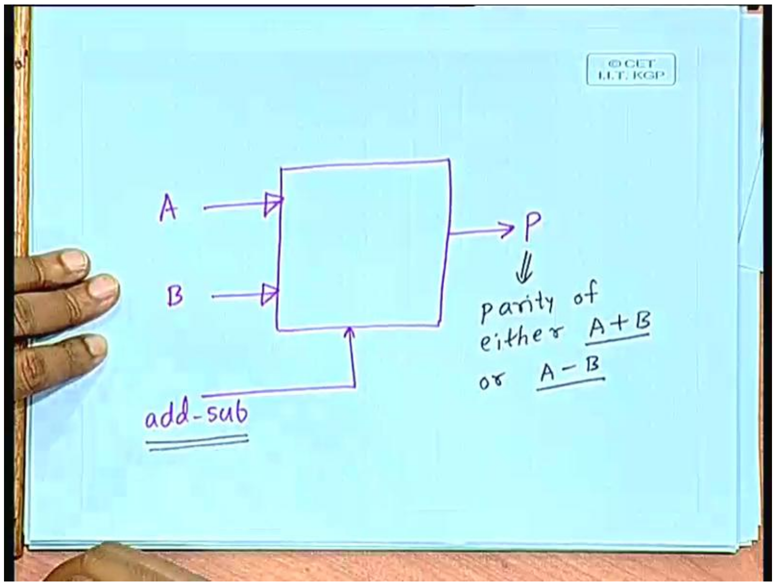


Example with Multiple Modules

- A simple example showing multiple module definitions.

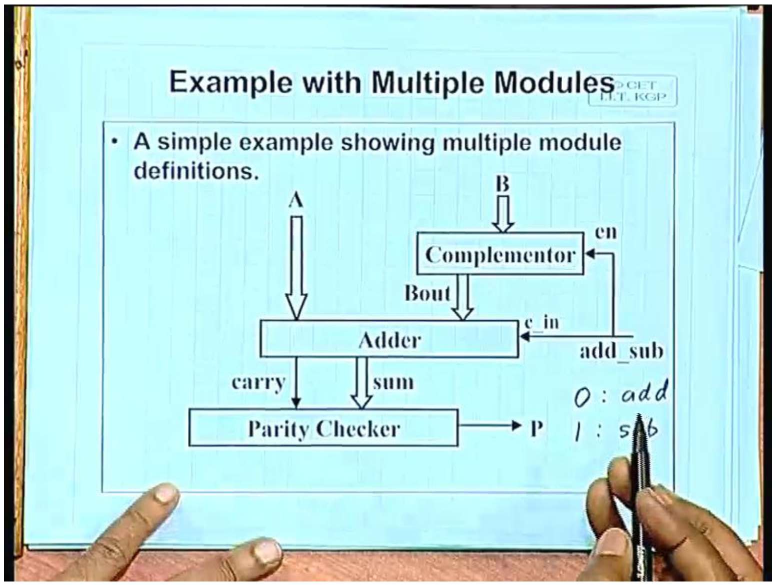
Now this is a hypothetical example I have taken this for the sake of illustration. This is one example where as you can see you have 3 different identifiable modules. Now actually what we want to do is that we have a adder subtractor and you want to check the parity of the result. So actually what we want is something like this.

(Refer Slide Time: 34:18)



We want to design a block which will accept 2 input vectors a and b there will be a control signal add sub and there will be an output called p. So the specification of these blocks is that this p will be this p is the parity of either a plus b or a minus b. Now whether you are adding or you are subtracting that is determined by the control signal. So internally either these 2 numbers are added or subtracted and then the parity of the result is computed just a hypothetical example I told you in practice you may not be needing such a thing to compute the parity of the result.

(Refer Slide Time: 35:38)

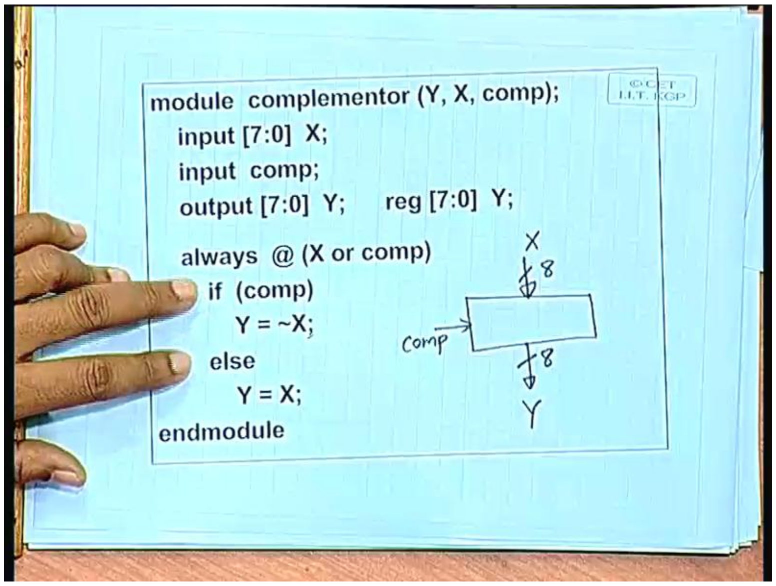


Example with Multiple Modules

- A simple example showing multiple module definitions.

So naturally we can partition the design into 3 parts. We need both addition and subtraction. So we will be using an adder and we will be using a complementor in case we need to do the subtraction. Now adder will be taking a as 1 of the inputs and it will be taking either b or the 2 complements of b as the other input. Well not really 2s complement it is actually 1s complement and this add sub control signal if it is 0, this means add if this is 0. 0 means add 1 means subtract. So if it is 0 then the complementor will straight away pass the input to the output and also the carry into the adder through the same signal that is also 0. So simply we are calculating or computing a plus b. But if the signal is 1 then this complementor will be complementing or it will take the ones complement of b, b out will be the ones complement of b this bit is 1 so another extra 1 is getting added. So this one's complement plus an additional 1 which means 2s complement is being added. So a plus the 2s complement of b. So the output will be the subtraction a minus b. And finally there will be a parity checker module which will be computing or calculating the parity of the result and will be sending out to result okay. Now let us try to see the different blocks out here to start with the complementor you say the complementor takes input and output as vectors and takes 1 control signal okay, whether to complement or not complement.

(Refer Slide Time: 37:49)

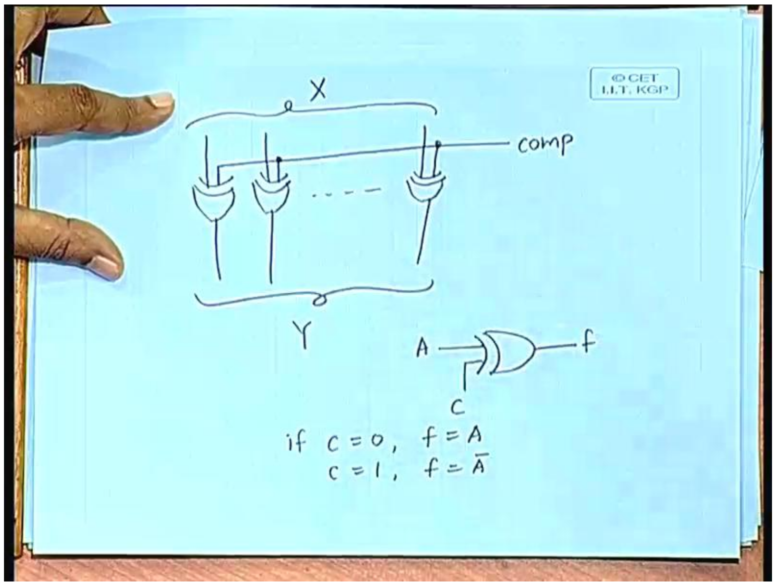


```
module  complementor (Y, X, comp);
   input [7:0]  X;
   input  comp;
   output [7:0]  Y;      reg [7:0]  Y;
   always  @ (X or comp)
      if  (comp)
         Y = ~X;
      else
         Y = X;
endmodule
```

©CET
I.I.T. KGP
X
8
Comp
8
Y

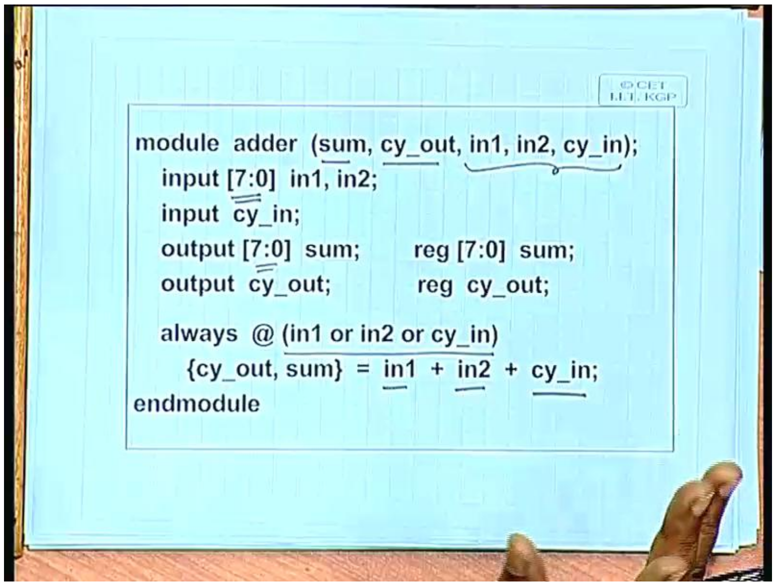
Now in this design we are assuming that all the operators are 8 bit quantities. So the complementor as is being specified in this module at the input x is an 8 bit quantity vector 7 to 0 output y is also an 8 bit quantity and there is a control signal called comp here we are giving an always block that whenever x changes or the control signal changes. If the control signal is 1 then you complement the output y is equal to not of x not means bit by bit inversion. This x is a vector else x equal to y. So this is a condition where you are specifying all possible values of the comp means what to assign. So this will be synthesized as a combination logic. Now the logic that will get synthesized out here will be something like this.

(Refer Slide Time: 38:54)



There will be an array of exclusive or gates which will be synthesized. These are the x inputs, these will be the y outputs and the second input of the x or gate that will be the control. This is comp. So we know that in x or gate if 1 input is a say the output is f and this is say c. So if c is 0 then f equal to a, and if c is 1 f equal to a bar. So we are following the same principle here well. So actually when the synthesizer will be analyzing this code and will be trying to synthesize, it will be synthesizing like this because the way the logic functions will be generated after analyzing, this will be the x or function. They will be all be the x or functions. So the synthesizer will be intelligent enough to synthesize it as an array of x or gates typically okay. So after the complementor comes, the adder the adder is very simple it takes 2 inputs and a carry in and it generates a sum and a carry out.

(Refer Slide Time: 40:38)



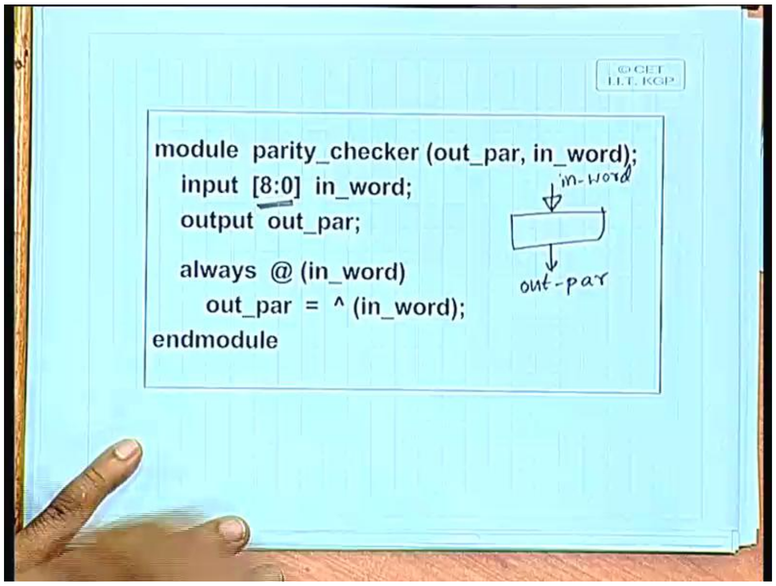
```
module adder (sum, cy_out, in1, in2, cy_in);
  input [7:0] in1, in2;
  input cy_in;
  output [7:0] sum;       reg [7:0] sum;
  output cy_out;          reg cy_out;

  always @ (in1 or in2 or cy_in)
    {cy_out, sum} = in1 + in2 + cy_in;
endmodule
```

So sum and carry out are the outputs and these 3 are the inputs. Now inputs are 8 bit quantities sum is also an 8 bit quantity carry in and carry out also sum and output are declared as registers. Now here whenever 1 of the inputs changes we are computing this. So we are this is a behavioral level description I am giving at a higher level in 1 plus in 2 plus carry in say at least for addition behavioral level description is acceptable. Because well as a designer it is good if we have some idea as regards to what kinds of modules are available in your library because you know that simple modules like adders, subtractors and multipliers are already available. So addition you need not specify in a structural form in the form of gate level interconnections and so on.

You can simply specify the behavior and the synthesizer will be picking up the best adder it finds out from the library and put it in combination logic only. (()) (41:51) Yes since it is combination logic you can do it with assign also yes you can do it with assign also it does not make a difference because this always block, this always block here out here makes sense only for the simulator but for synthesis always block or you can give assign also they are the same. So these 3 values are getting added up and we will be getting a 9 bit result sum and possibly a carry out these are concatenation. This is how you specify okay and finally you have the parity checker right. So in the parity checker well you can simply specify it in this way.

(Refer Slide Time: 42:48)



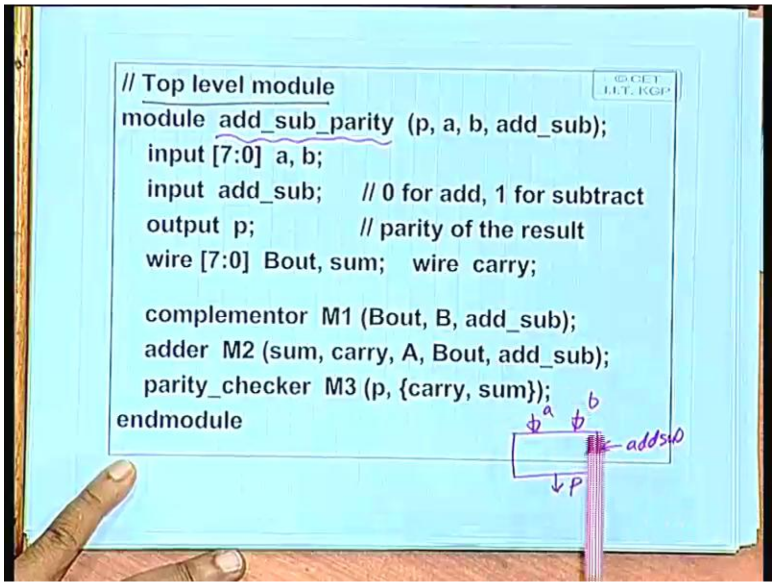
```
module parity_checker (out_par, in_word);
   input [8:0] in_word;
   output out_par;

   always @ (in_word)
      out_par = ^ (in_word);
endmodule
```

©CET I.I.T. KGP
in-word
out-par

Module parity checker. This is the output so the input will be a vector and the output will be a single bit and input in that circuit it is a the concatenation of the sum and the carry it will be the nine bit quantity. So here this will be a 9 bit quantity and output will be out par. This is the output, so this is inward and this is out par and whenever the input changes you are giving a reduction operator. Reduction operator I had said an operator before an operand means you take a bit by bit operation of that and you generate a single bit out of it. This means it is bit by bit X OR. You take x or of all the bits that will be your parity okay. So you get this okay. So now with respect to this design we have specified all this 3 blocks complementor adder and parity checker. But what we have not specified as of now is that how this 3 blocks are interconnected together. That is what we need to specify. So in any design where you have some kind of a hierarchy you will be designing the low level modules fine. But you will also have to design a top level module where the low level modules are picked and placed and interconnected.

(Refer Slide Time: 44:25)



```
// Top level module
module add_sub_parity (p, a, b, add_sub);
   input [7:0]  a, b;
   input  add_sub;      // 0 for add, 1 for subtract
   output  p;           // parity of the result
   wire [7:0]  Bout, sum;    wire  carry;

   complementor  M1 (Bout, B, add_sub);
   adder  M2 (sum, carry, A, Bout, add_sub);
   parity_checker  M3 (p, {carry, sum});
endmodule
```

©CET I.I.T. KGP
a
b
addsub
p

So this is the example top level module for the circuit we are discussing presently. So the name of the module we are giving as add sub parity. This is the name of the top level module and in the highest level this module takes 3 inputs a b then add sub and it generates an output p. This is at the highest level. So these are the interface specifications say internally whatever you do, suppose overall you are trying to design a block like this where a b add sub and p are the input output points. So at the high level the whole thing we will be treating as a black box and you will be seeing what are the interface pins or signals and what are their sizes so in the top level module. You have to specify all these here these are the signals or the wires or the pins whatever we call that will be coming in or going out of the module which you treat it as a black box. Now if you go into slightly more detail of this firstly we will have to specify the size of each of these. So a b will be 8 bit quantities add sub is a single bit. Similarly p is a single bit. Now internally, now since you have this structural diagram already with you know that there are some intermediate signals you need like b out sum carry.

These signals are not available to the outside world. But used internally b out and sum are 8 bit quantities carry is 1 bit. So here we declare b out as sum as wires 8 bits carry as a signal bit wire. Now we start instantiating the complementor adder and parity checker see the complementor as we have earlier designed there you had specified the parameters in a particular order. First 1 is the output second 1 is the input vector third 1 is the control. So here also when you are instantiating the parameter ordering is kept the same b out is the output, b is the input, add sub is the control. So this you can again refer to this diagram b b out is the output b is the input and add sub this signal is coming right. Now comes the adder see the with respect to the adder the outputs will be this intermediate sum and carry and the inputs will be a this b out and this add sub right. So you see this adder this adder you will have sum and carry as the outputs and a b out and add sub as the inputs.

Similarly the parity checker parity checker will have 1 output p and the input will be sum and carry. See the way we have defined or declared the parity checker we have accepted only 1 single input 9 bit input. So the input will be the concatenation of these 2. So this we have exactly we have specified in that way parity checker takes 2 parameters; one is p other is the concatenation of carry and sum. So once we instantiate this, the interconnection between these modules get automatically defined right. So this example shows how we can create a design hierarchically starting with the low level modules. We can get a design at high level we can go up step by step. Now given a problem you can structure or you can divide up the problem into smaller sub problems. You can divide the modules into sub modules. In that way developing a bigger design hierarchically is a good idea you should design it that way. But once you have designed you will have to check whether the design is correct or not. So typically the first step you go about before actually trying to synthesize is to simulate and find out whether the code you have written is functionally correct or not.

Because this code was translated from some kind of description; behavioral description or some kind of block diagrams something. So in order to check whether your coding is correct you will have to go for simulation. Now for going for simulation you will have to write the so called test bench that I had mentioned earlier. So you will have to write a test bench which will be applying the inputs to the module you have written and it will also accept the output and either compare or store the output in a file or display it on a screen something. So in our next lecture we will be we will be primarily looking at how we can write the test bench and what are the different features available there. Because writing test bench you will be required not only for behavioral simulation also after you have tried out the synthesis you have the gate level netlist possible, there also you will have to simulate and you will have to possibly compare the behavior with the netlist. So you will have to run the simulator twice and then you will have to compare and find out right. So all these things we will be discussing in the next class. Thank you.