Digital VLSI System Design

Dr. S. Ramachandran

Department of Electrical Engineering

Indian Institute of Technology, Madras

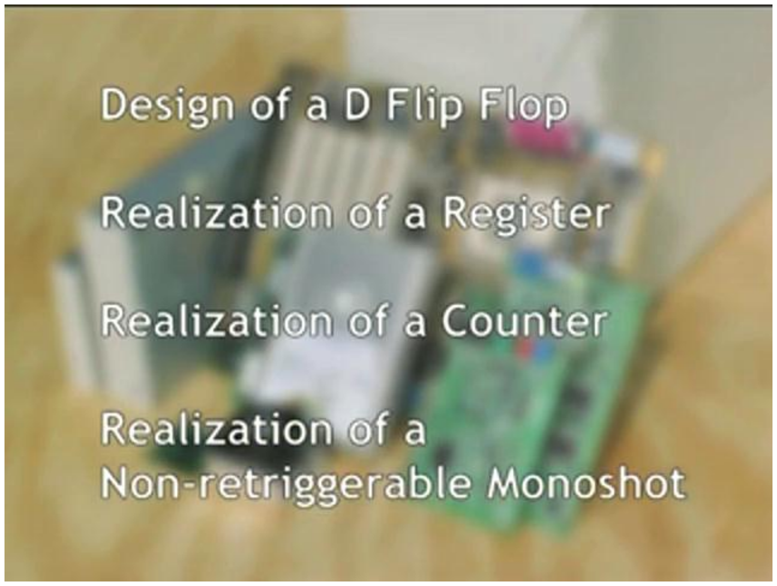
Lecture - 12

Modelling of Verilog Sequential Circuits

Core Statements (Continued)

Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:11)



Design of a D Flip Flop

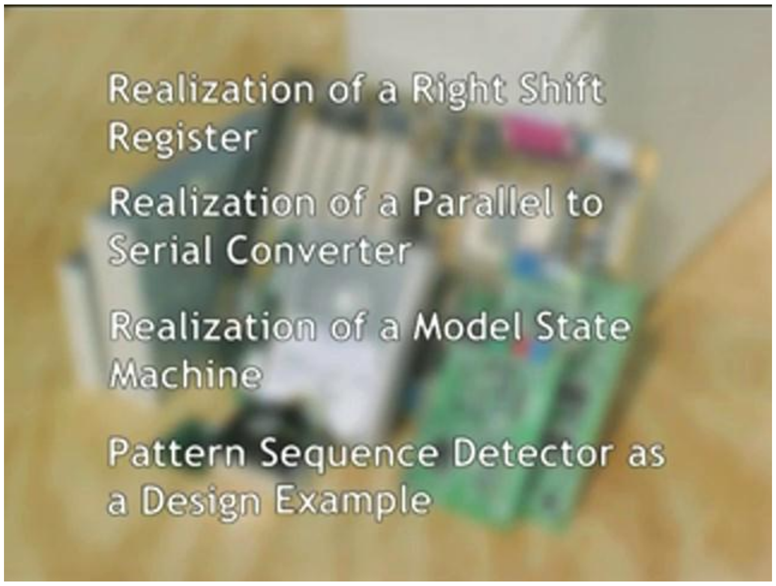
Realization of a Register

Realization of a Counter

Realization of a Non-retriggerable Monoshot

Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:43)


Realization of a Right Shift Register

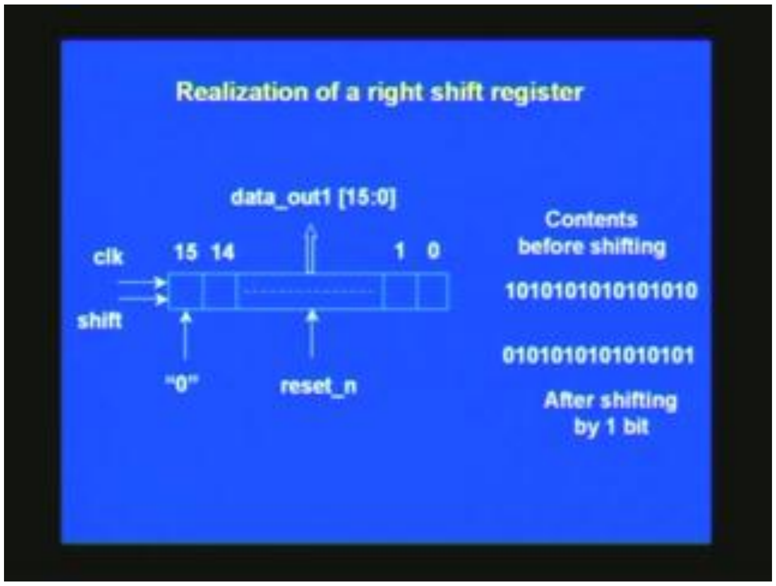
Realization of a Parallel to Serial Converter

Realization of a Model State Machine

Pattern Sequence Detector as a Design Example

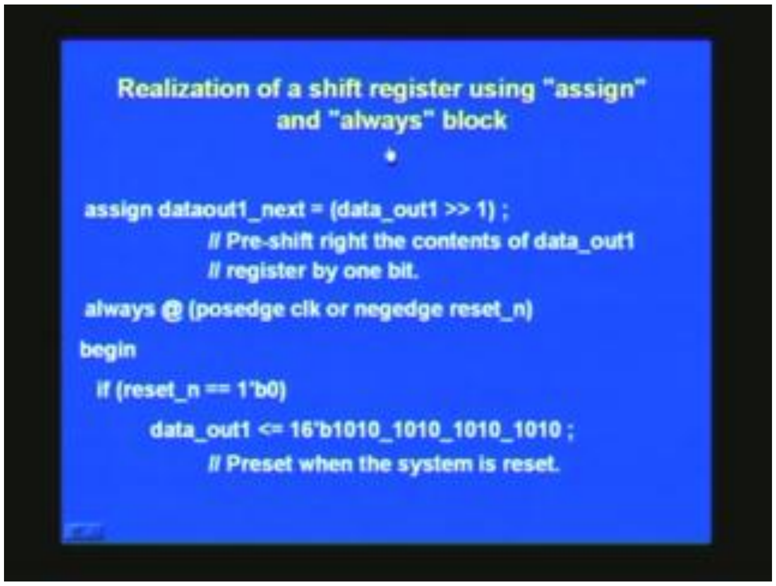
Last time we saw how to model a monoshot multi vibrator. This time we will start with right shift register which we have seen earlier in the combinational circuits.

(Refer Slide Time: 02:11)


Realization of a right shift register

Here we have a 16 bit register and 15 is the weight of the bit. That is MSB and so on, LSB is 0. If you wish to have register contents you can get it out here as a data out 1. Naturally, 15 through 0 indicate the width of the register. It has a clock input and once again we will take only positive edge of the clock. This being a right shift register, shift automatically implies that it is going to be right shift. When shift is asserted only then the shifting will take place at every positive edge of the clock. This when you right shift this 15th bit vacated this will be filled with a 0; this is the case with Verilog instruction. You can reset the whole register once again by a system reset. Let us see what the contents before shift are. Let us say 1 0 1 0 and after the shift by 1 bit naturally 0 will be occupied here. All the contents will be shifted by 1 bit and 0 will get lost in the process. We will see how to implement this.

(Refer Slide Time: 04:09)

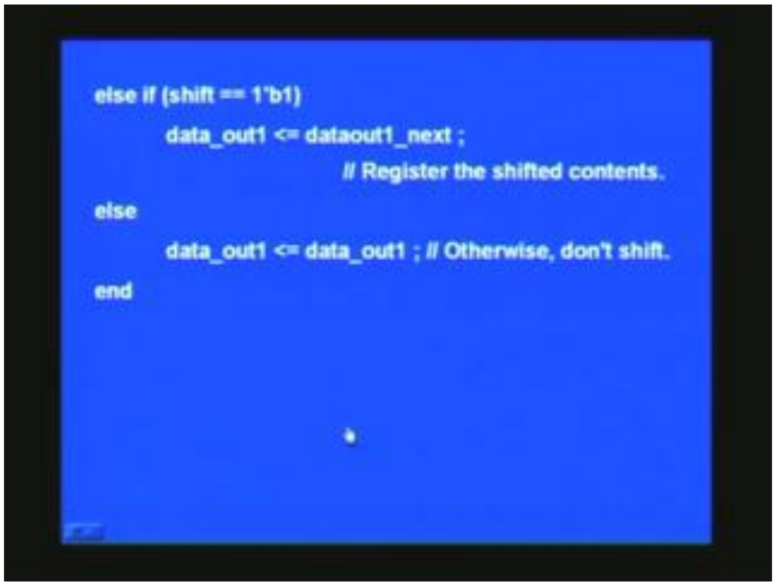


Realization of a shift register using "assign" and "always" block

```
assign dataout1_next = (data_out1 >> 1) ;
                // Pre-shift right the contents of data_out1
                // register by one bit.
always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
        data_out1 <= 16'b1010_1010_1010_1010 ;
                // Preset when the system is reset.
```

Once again this is ‘always’ block. As usual we will use assign. Then we will do the pre-computation; in this case the right shift - this you are already familiar from our combinational circuit. We are going to precisely use the very same thing. Shifting is actually done in the combinational. We also combine a sequential here so that the real system picture will be this combinational. Data out 1 is shifted by 1. This is the symbol for right shift as you had seen earlier. If you want left shift you have the arrows the other way (Refer Slide Time: 04:53). We assign this to what is called next. We have the same name and this implies next value is going to be this and that this value will be used within the ‘always’ block when the clock really makes its appearance that is at the positive edge of the clock. Once again we have an always block at positive edge of clock. We take the action at negative edge of reset. Once again we check the reset condition here. If it is 0 then what we will do is we will preset. It is not resetting we will preset with the desired value that you want. Notice that 16 tells the total width of this data and this is 16 bits. As we have said 16 is there. You need this quote here and b stands for binary. What follows is the binary bit pattern. You notice that 1 underscore is being used. This has no real meaning as far as the system tool is concerned. It simply ignores this underscore. It is only to facilitate ease of reading at the user level; otherwise it has no meaning. Suppose I had put all continuously without this underscore, even then it is valid but readability will be poor when you are dealing with 0s and 1s. 16 bit reading will always be difficult. It is only to facilitate reading at your end. This is the comment for this which is preset when the system is reset (Refer Slide Time: 06:43).

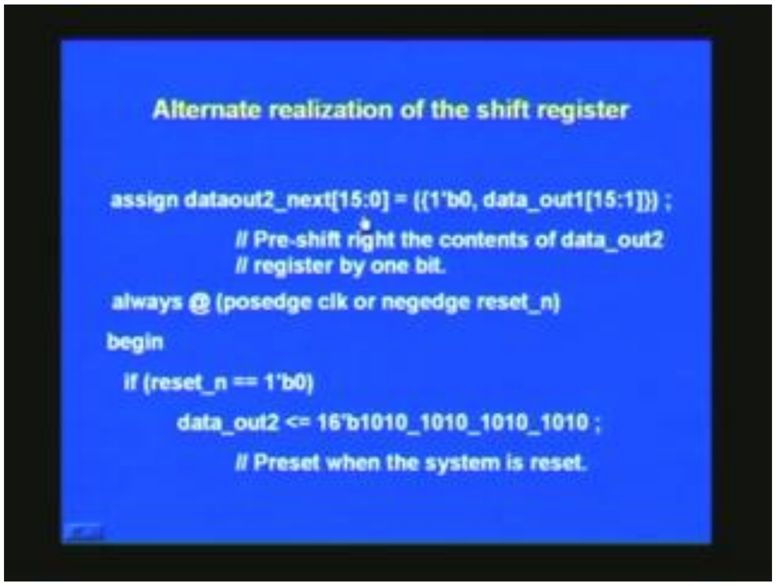
(Refer Slide Time: 06:46)



```
else if (shift == 1'b1)
        data_out1 <= dataout1_next ;
                        // Register the shifted contents.
else
        data_out1 <= data_out1 ; // Otherwise, don't shift.
end
```

We want to shift if that condition is not met. We would like to shift the contents. What we do is we check whether the shift is asserted or not. If it is asserted then what we do is we just assign that data out 1 next. This is containing nothing other than the shifted value here which was pre shifted here. Notice that that is a combinational circuit so that will keep on churning and be ready at all the time so when the clock strikes it is always the correct data that you have. That is why we just assign. This will speed up frequency of operation. That is why always all time consuming computation are always done outside using assign statements. We just transfer the contents when the clock strikes and the comments are clear from this. If a shift is not to be done for example if shift is 0, this block will not processed so what we do is we go to the next and here there is no if statement here because this happens to be the very last of the series. We just wind up with an ‘else’ here and here it only preserves the previous contents. Suppose we do not want to put this we are free to just give a semicolon there.

(Refer Slide Time: 08:21)

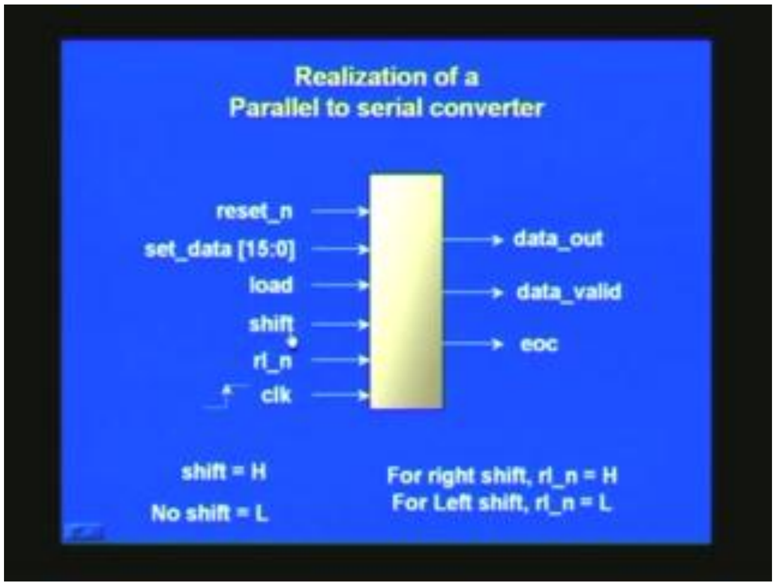


Alternate realization of the shift register

```
assign dataout2_next[15:0] = ({1'b0, data_out1[15:1]}) ;
                // Pre-shift right the contents of data_out2
                // register by one bit.
always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
        data_out2 <= 16'b1010_1010_1010_1010 ;
                // Preset when the system is reset.
```

We can have an alternate way of implementing the same shift register. What we have used is that ‘less than’ symbol twice and you can also use concatenation. For example, let us say the very same example. We just nomenclature it as 2 here instead of 1 because we are going to simulate it later on; it should not have the same name for different outputs although they are basically one. Concatenate here 0 because after right shift what happens to the MSB is we will fill 0 there so we put a 0 here. Then we separate by a comma here (Refer Slide Time: 09:10). It means this whole thing you should regard it as a single 16 bit and the first bit we are forcing it to 0 by putting like this. Then follow it by the actual 15 to 1 because after right shift this 15 to 1 will go to the right most end; in which case 1 bit will be the last LSB here. 0 is not accounted because we do not really care whether is there or not. If you want to preserve it we have to preserve it by another statement which we will be seeing in 1 of the future application and pre-shift right the contents of data out to register by 1 bit. You give a meaningful comment as we have seen earlier and once there is an ‘always’ block here and we take action only on positive edge of clock or negative edge of reset. Once again, there is ‘begin’ and finally, there will be an ‘end’ wherever an ‘always’ block is there now we again inspect for reset being active. If it is what we do is just preset as we had done earlier and if shift if asserted then assign the shifted data here this is precisely the same thing as you have already considered and do not shift here just preserve the contents.

(Refer Slide Time: 10:41)



Realization of a Parallel to serial converter

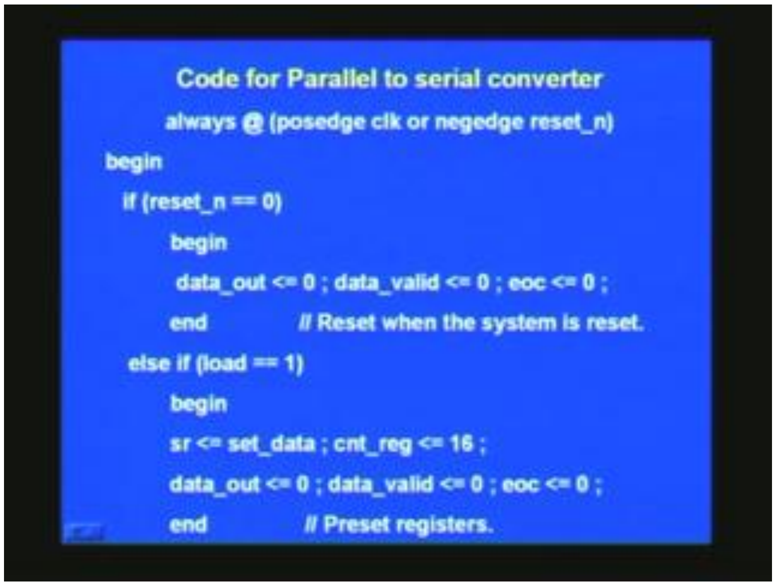
shift = H
No shift = L

For right shift, rl_n = H
For Left shift, rl_n = L

Next example we will be considering is, suppose we have parallel data we want to serialize this data. What we do is we can use a combination of shift register and a counter. This can be very easily coded in Verilog. Let us consider this block diagram (Refer Slide Time: 11:10). We are not explicitly showing a shift register or counter here. We can readily incorporate it in the coding see let us say once again. We want to once again reset the system with a power on reset or system reset and you want to load a particular value at any point of time. You can assert this load and prior to that you can set the data using this bus, 15 through 0. 16 bits is what we stated earlier as far as the width is concerned.

The output will manifest here; this will be a serialized output, single bit and when this is valid that also will be indicated by another signal which is a data valid if required. We can use this if it is merely asynchronous you are not really bothered about these 2 signals. When the conversion is over you have to have an ‘end of conversion’ sort of thing so we give 1 more signal here as an end of conversion. Coming to this, it loads the set data into the shift register which is 16 bit width and when shift is asserted and only then the shift will take place. The shift can be either right or left, for that also you need some more information and that is this signal. R stands for the high actually indicated here and this is left, l stands for left. Left I have put underscore n as usual the same notation we have followed earlier for coding it.

(Refer Slide Time: 13:05)



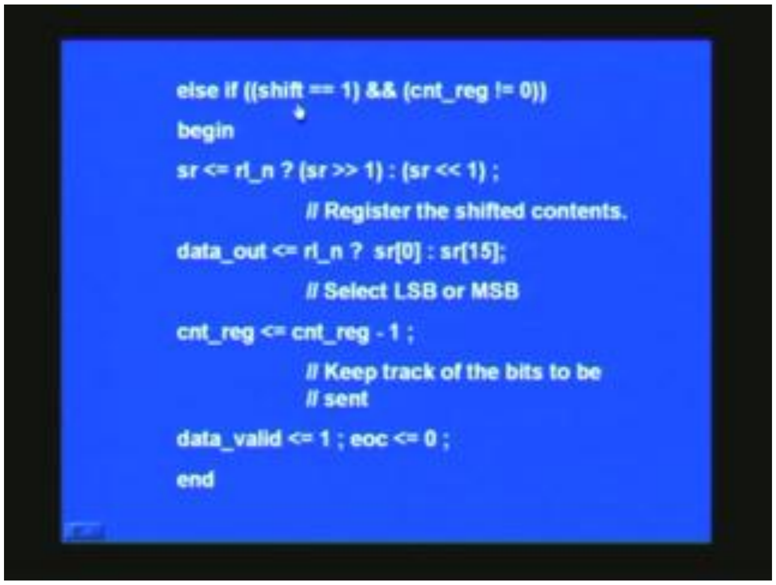
Code for Parallel to serial converter

```
        always @ (posedge clk or negedge reset_n)
begin
 if (reset_n == 0)
        begin
         data_out <= 0 ; data_valid <= 0 ; eoc <= 0 ;
        end           // Reset when the system is reset.
  else if (load == 1)
        begin
        sr <= set_data ; cnt_reg <= 16 ;
        data_out <= 0 ; data_valid <= 0 ; eoc <= 0 ;
        end           // Preset registers.
```

Once again, we use an ‘always’ statement because all sequential circuits are presented by ‘always’ with a positive edge of clock. Once again if reset is asserted we initialize all the outputs we have a data out and data valid signal and end of connection all of them must be reasserted. That needs no explanation. If load is asserted then what we have to do is we have a shift register inside which is 16 bit in width. We do not have to put the bit here although it is a good practice to put here and what we need to do is we need to take the set input set data input. What we have seen earlier here, this will have to be read and put into the shift register that is what this statement is doing (Refer Slide Time: 14:05).

One statement is complete only with a semicolon and you can put any number of statements here in the same line or you can fold it just to save space to show in a single slide. I had done it in this fashion. Once again, data out we do not have to assert right now because data is not going to out at this stage. What we are doing is, merely loading the pre set value that is the set data. We have a counter to keep track of how many bits we have sent and since we want 16 bits to be sent in total we will just initiate this 1 counter this is 5 bits; 16 is loaded into it; it is a preset value. Once again it deals at all the outputs. Next appears if shift is asserted and also the counter is not 0 which implies that it has been processing, sending the information. If it is not asserted this is 1 second and symbol and this is a total logic expression. Therefore you see 2 ‘equal to’ if you put 1 ‘equal to’, tool will complain. Here the reason is also if it is ‘not equal to’, you can use in this fashion. We have already seen here and mind you, you should not use that tilda sign like symbol. You should not use that here; in logical expression only this exclamatory mark is for use.

(Refer Slide Time: 15:20)



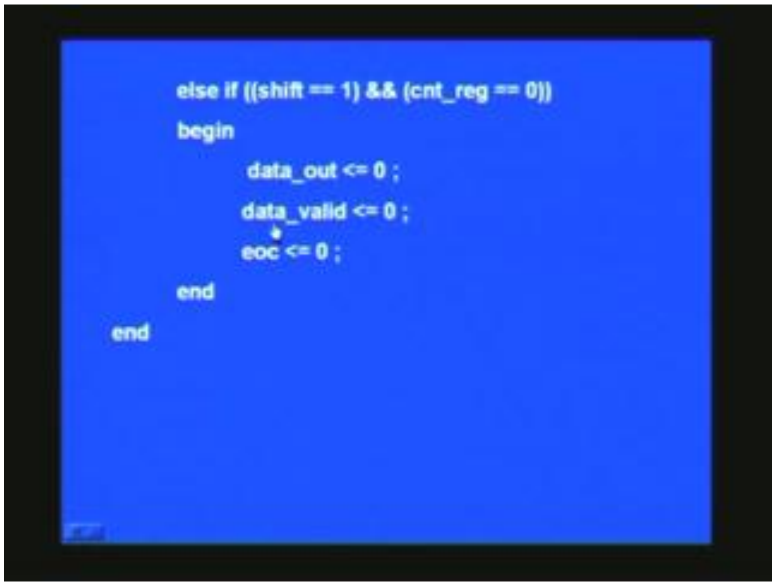
```
else if ((shift == 1) && (cnt_reg != 0))
begin
sr <= rl_n ? (sr >> 1) : (sr << 1) ;
                 // Register the shifted contents.
data_out <= rl_n ?  sr[0] : sr[15];
                 // Select LSB or MSB
cnt_reg <= cnt_reg - 1 ;
                 // Keep track of the bits to be
                 // sent
data_valid <= 1 ; eoc <= 0 ;
end
```

If both are satisfied then what happens this thing will return 1 and this will return 1 if the conditions are satisfied 1 and 1 will be 1 (Refer Slide Time: 16:00). Finally, the total AND operation result only will be returned here. It will be sensed accordingly and if that happens the condition is satisfied what we have to do is do the shifting actually. While shifting we have to take care of right shift and left shift. We have already seen the MUX implementation with a question mark here earlier. That is precisely what we are using now. We had to take action if the select pin is rl underscore n. If it is high, it means right shift so that is what we are doing here. Remember that for MUX, 1 input is here and 0 input is here with a colon. Of course, termination of this statement and this 1 if this is not satisfied that is if it is left shift it has got to be left shift by 1 bit. That is the symbol here and this is either this thing is assigned to s r or this 1 assigned to s r here is right shifted content here it is left shifted content so that is what goes to s r. s r once again is 16 bits. Therefore, this result is also 16 bits each of this result; whereas this is only 1 bit. The comments are the appropriate things here, register the shifted contents.

Next what we have to do in the same block is same ‘else if’ block, data out we have to take care this is the data which is going out right now. Once again you use the same rln symbol here and if it is right shift what we need to do is we had to take the shift register LSB content which is s r 0. Output that just assign it to just transfer these contents into the data out if it is left shift naturally this will be taking force and here you assign the MSB here 15 is the MSB, please note that and s r within brackets you just specify only that particular bit; that means, this is a single bit (Refer Slide Time: 18:20). Select either MSB or LSB depending on whether rln is 1 or 0.

Having done this, we should not forget to decrement the counter because we started with 16 earlier at the reset time. 16 when it is decremented will be 15; but, we will not check the status right now; we will take action later on. Suppose, had the counter been 1 instead of 16, 1 minus 1 is 0; 1 bit is already processed here and it naturally exits the entire always block in the case where counter reg was initialized to 1. This is to keep track of the bits to be sent. How many bits are remaining to be sent will be known if you just inspect the counter reg. One more thing we have to do is the data we are going to send right here that means, we should also send data valid signal here. That is what is being sent here and since end of conversion is not really over because there may be more number of bits to be sent.

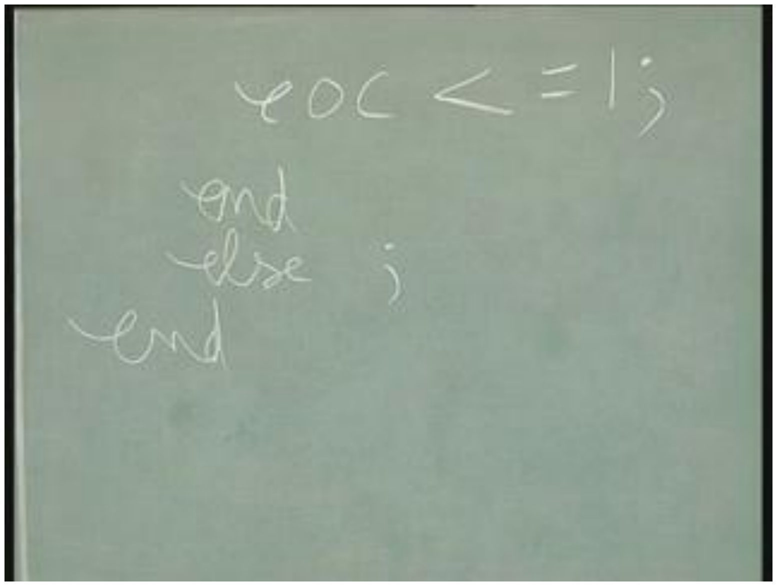
(Refer Slide Time: 19:48)



```
        else if ((shift == 1) && (cnt_reg == 0))
        begin
                data_out <= 0 ;
                data_valid <= 0 ;
                eoc <= 0 ;
        end
    end
```

Once again if shift is still present and the counter value has touched 0 which implies that all the 16 bits have been already sent in the earlier block; we have seen that. This will take effect only at the next clock pulse. Earlier clock pulse all the 16 bits are already dispatched. What we have to do is please correct that. End of conversion must be 1 here. That is what is implied here and in fact I have forgotten one more thing. One ‘else’ statement also I have forgotten. You just take it as after the end you have one more ‘else’ statement. You do not have to take any action so you just put a semicolon wind it up. We have end of conversion.

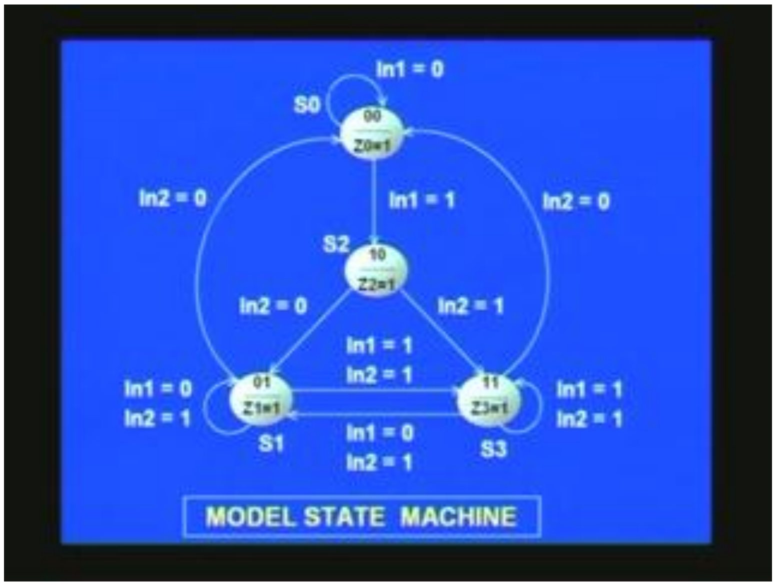
(Refer Slide Time: 20:41)



```
      eoc <= 1;
    end
    else ;
  end
```

It is written there 0 it must be corrected as this and then what follows is an ‘end’ (Refer Slide Time: 20:53). This is corresponding to the end of the ‘begin’ there. In the second line and then you have one more ‘end’ because we started with one more ‘begin’ right at the top after the ‘always’ statement. Corresponding to it is this ‘end’ so we have also to add one ‘else’ because we have not accounted for that. That ‘else’ is put here and since we are not interested in output anything you just put a semicolon implying it is a null statement. This you will have to incorporate.

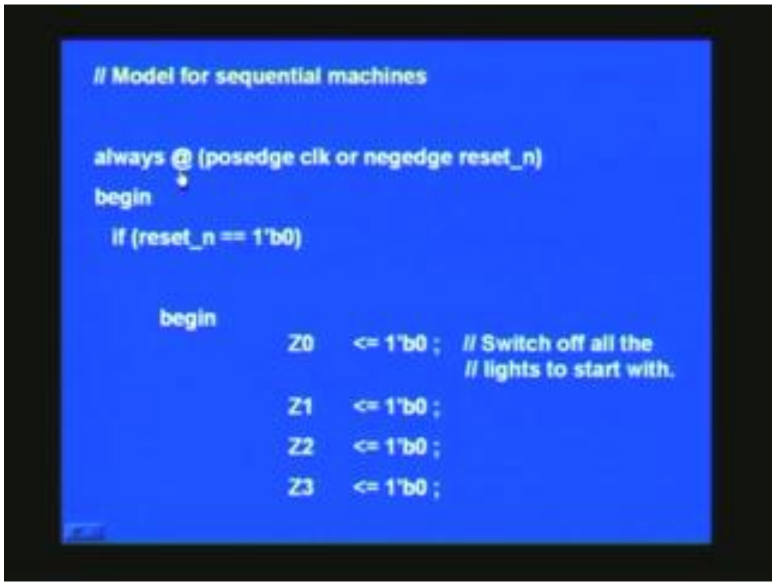
(Refer Slide Time: 19:48)



Next we will consider how to model a state machine. Let us say we have 4 state machines and in each of these states you wish to light up an LED or a lamp. Zee stands for one lamp; zee 0 stands for the s not condition state indicator. If you are in s not state, then this lamp will glow and likewise in all other states and this s not is the state and in binary notation. Let us say it is 0 0.

After this it goes to second state and after this depending upon the inputs it will go either to s1 state or s3 state. Coming back to the first state if we have 2 inputs and based on these inputs all this transition will take place. Suppose if it is it is 0, input 1 is 0, so what you do is you continue to remain in the same state. This will be lit and when In1 is 1 it will go to state s2 and thereafter it will go to either s1 or s3 depending upon this In2 input. If it is 0, it goes to s1 otherwise to s3. Once you land up in s1 again it depends upon In1 and In2, if it is 0 1 note that this 0 1 is same as this state. It remains in the same state and if it is 1 1 this happens to be 1 1 so it takes you to this (Refer Slide Time: 23:20) this state. Once again from here, to and fro we can shuttle if you wish and depending upon this control and this control 0 1; once again is same as this 0 1 condition. In either of the cases you can return back to the first initialize initial condition state by depending upon In2 values. Let us see how to implement this.

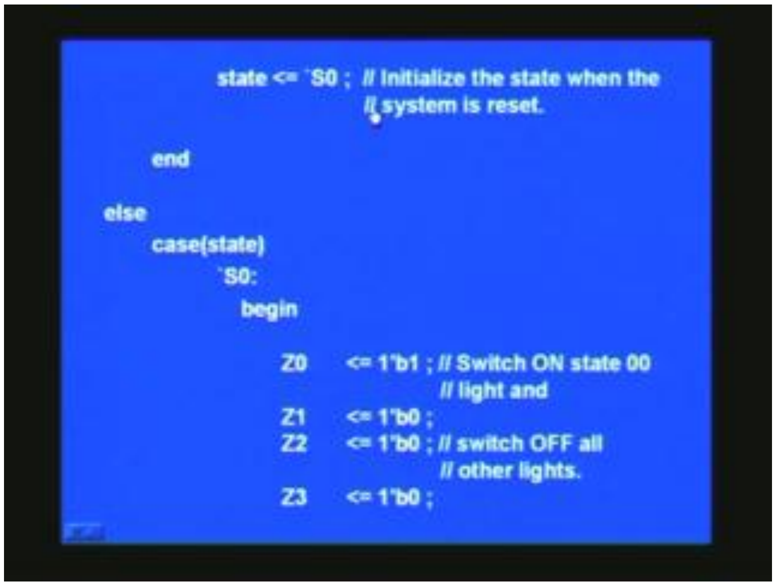
(Refer Slide Time: 23:49)



```
// Model for sequential machines

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)

      begin
                Z0     <= 1'b0 ;   // Switch off all the
                                   // lights to start with.
                Z1     <= 1'b0 ;
                Z2     <= 1'b0 ;
                Z3     <= 1'b0 ;
```

Once again we use the ‘always’ block and positive edge of clock negative edge of reset. As usual the reset here in this case there are 4 lamps. We will have to light them up and all of them are initialized to 0. This is deliberately done for the time being because in s not state this will be lit. The moment you reset it will go to the s not state; in s not state we have appropriately done that. That will take care of itself. We have another variable called the state which is same as s not earlier mentioned earlier or 0 0. That is what we have already seen here.

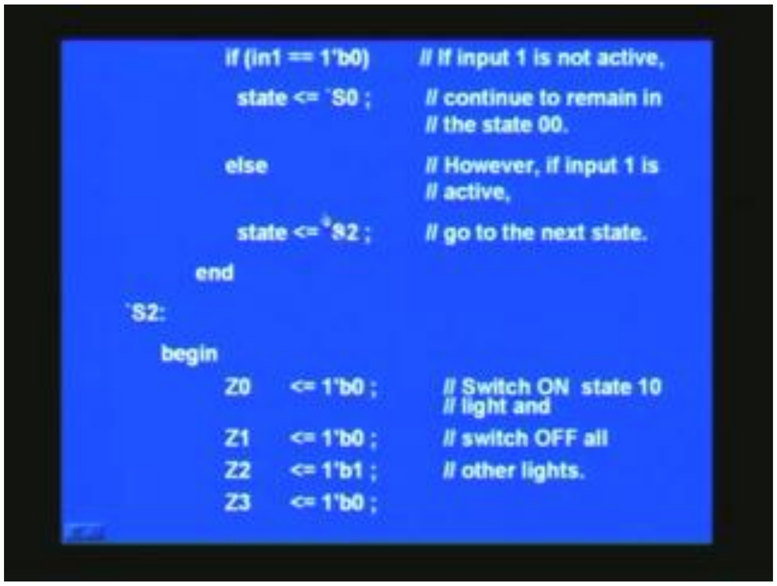
(Refer Slide Time: 24:32)



```
            state <= `S0 ;  // Initialize the state when the
                            // system is reset.

      end

 else
      case(state)
            `S0:
              begin

                Z0     <= 1'b1 ; // Switch ON state 00
                                 // light and
                Z1     <= 1'b0 ;
                Z2     <= 1'b0 ; // switch OFF all
                                 // other lights.
                Z3     <= 1'b0 ;
```

That is what we are initializing here. In this initialization once again note that these are non-blocking statements. Within the 'always' block you have to use this; it is 'equal to' only in assign statement. You have to put ordinary 'equal to' and note this one. This is reverse apostrophe used here to indicate that the string you can consider this as a string variable s not is the normal nomenclature in a valid name that you give (Refer Slide Time: 25:18). It can even be a state 0 or whatever you want to give you can give here. Every line is terminated by a semicolon so we initialize when the system is reset. In earlier statements, so many 'else if' were used. We have only one if and then else and here starts a case statement; it is similar to what we have already seen in the combinational circuit. That depends upon the actual state which is a variable here. Indicating which state we are in and if the state happens to be this s not then only this block is executed. Similarly you have for different blocks for different states. Once again a group of statements will have to have 'begin' and 'end'. We set the outputs 0 through 0 3 depending upon which state it is in. For s not we want to turn on zee not LED.

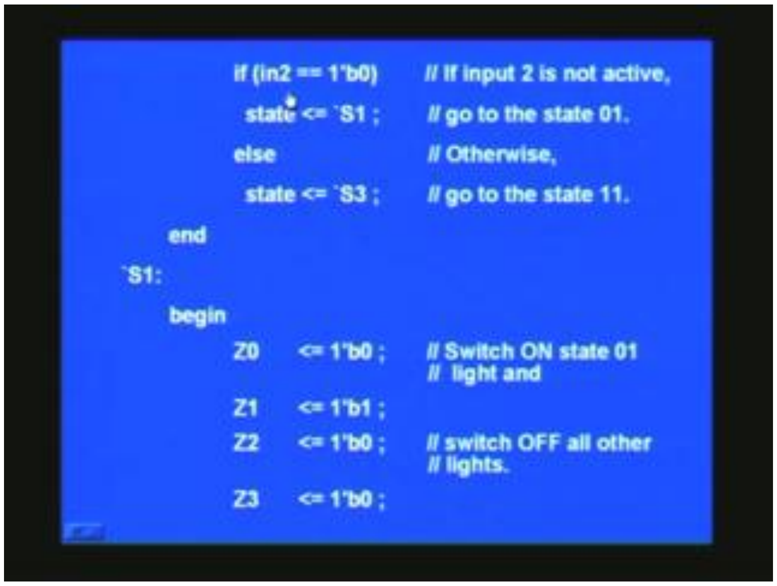
(Refer Slide Time: 26:24)



```
                if (in1 == 1'b0)        // If input 1 is not active,
                  state <= `S0 ;        // continue to remain in
                                        // the state 00.
                else                    // However, if input 1 is
                                        // active,
                  state <= `S2 ;        // go to the next state.
            end
      `S2:
         begin
                Z0     <= 1'b0 ;         // Switch ON  state 10
                                         // light and
                Z1     <= 1'b0 ;         // switch OFF all
                Z2     <= 1'b1 ;         // other lights.
                Z3     <= 1'b0 ;
```

In this case, suppose In1 is 0, that is what the first condition is if it is 0 we have to remain in the same state s not. If this is not satisfied that is In1 is 1, then only you go to the second state s2. No where In2 is in the picture; it is a really do not care situation. It depends only upon the In1 and not upon the In2 because we have not put that. Suppose after this this condition is satisfied and if this condition is satisfied, we continue to remain every time the clock strikes you come to the same route. They are always revolving around the same state and only if In1 changes to 1, this will not be processed and it will go to next state which is s2 here and so in s2 state it is precisely same except that z2 is lit here.

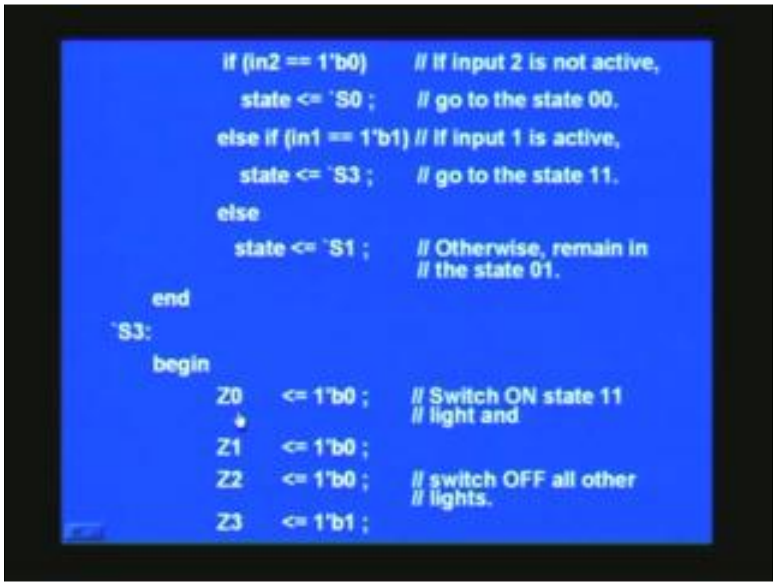
(Refer Slide Time: 27:28)



```
            if (in2 == 1'b0)     // If input 2 is not active,
             state <= `S1 ;      // go to the state 01.
            else                 // Otherwise,
             state <= `S3 ;      // go to the state 11.
        end
    `S1:
        begin
            Z0    <= 1'b0 ;      // Switch ON state 01
                                 //  light and
            Z1    <= 1'b1 ;
            Z2    <= 1'b0 ;      // switch OFF all other
                                 // lights.
            Z3    <= 1'b0 ;
```

In this case we need to inspect In2 and based upon whether it is if it is 0 we go to the next state which is s1. Otherwise we go to another state s3. In s1 state it is once again similar to this and 1 is set for zee 1.

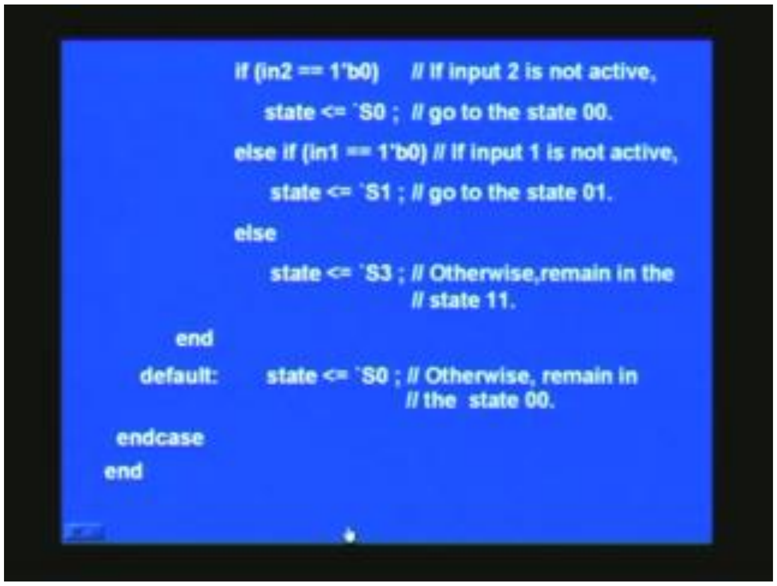
(Refer Slide Time: 28:02)



```
                if (in2 == 1'b0)        // If input 2 is not active,
                  state <= `S0 ;        // go to the state 00.
                else if (in1 == 1'b1) // If input 1 is active,
                  state <= `S3 ;        // go to the state 11.
                else
                  state <= `S1 ;        // Otherwise, remain in
                                        // the state 01.
          end
      `S3:
          begin
                Z0    <= 1'b0 ;         // Switch ON state 11
                                        // light and
                Z1    <= 1'b0 ;
                Z2    <= 1'b0 ;         // switch OFF all other
                                        // lights.
                Z3    <= 1'b1 ;
```

Once again we inspect In2. If it is 0 we take back to s not condition and if In1 is 1, we go to s3. At this point when it enters it implies that In2 is 1. If In2 is satisfied this will be satisfied and it will exit this block. That is why I said, ‘if’, ‘else if’, they have this veto power; that is, a priority can be assigned in this way. If the top priority is met it nearly exits the ‘always’ statement. That implies that In2 is 1 here; In2 is 1 and In1 is 1. If this condition is satisfied it will go to the s3 state. It will not go to s3 state immediately. It will go only in the next clock edge it only prepares the ground to go for the next clock here right here (Refer Slide Time: 29:00). This will become very clear to you if you see the timing diagram while we simulate the whole thing. Finally, we use 1 else and because all the conditions that we want to meet have already been met here. Whatever does not meet is actually... In1 is 1 here, In2 is 1 here and In1 is 0 here, only then it will come to this state automatically (Refer Slide Time: 29:25) because, the other state is taken care here. Only state that remains is In1 is 0 which will naturally take it to s1. Suppose from here we have landed up in this state s3 state, so similar things for us here. We once again turn on all LED s off except for z3.

(Refer Slide Time: 29:51)

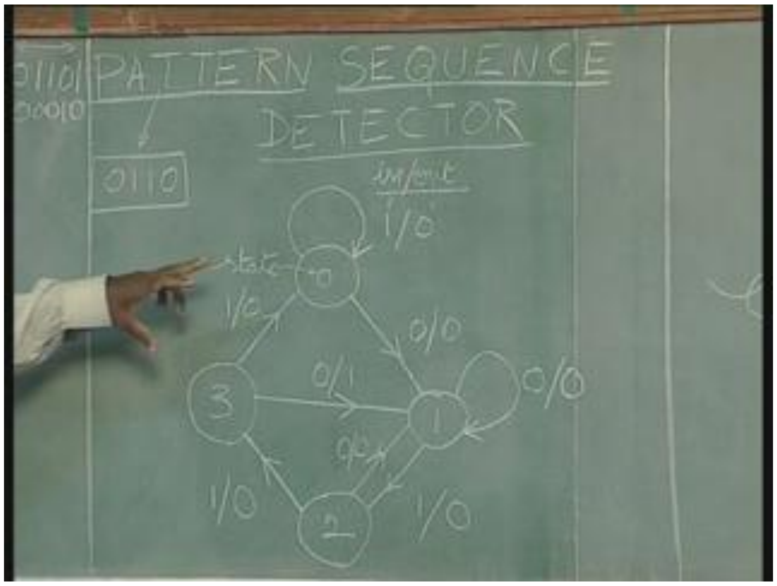


```
                if (in2 == 1'b0)     // If input 2 is not active,
                   state <= `S0 ;  // go to the state 00.
                else if (in1 == 1'b0) // If input 1 is not active,
                   state <= `S1 ; // go to the state 01.
                else
                   state <= `S3 ; // Otherwise,remain in the
                                  // state 11.
            end
        default:     state <= `S0 ; // Otherwise, remain in
                                    // the  state 00.
    endcase
  end
```

You turn on here; if you forget to turn off the other things it will be considered as a register and remains in the previous state. You should not forget to do that. If In2 is 0 in this case we are in s3 state it will take it to s not. Otherwise if that is In2 is 1, then In1 is 0, it will take you to s1 and if this is also not satisfied that is In1 is 1 it will take you to s3. We have also put a default although it is not necessary here because we have 2 bits of state. We have only four possibilities as 0 through 3 here. Then why default? It is not just customary, probably it may happen so that tri state zee or do not care situation arises. It might not arise in sequential circuits, but it is safer to put here. Another point is, instead of 2 bits you might have 3 bits in which case you have used only s not to s3. S4 to s7 are not utilized. What will happen in such cases? You should take that into account.

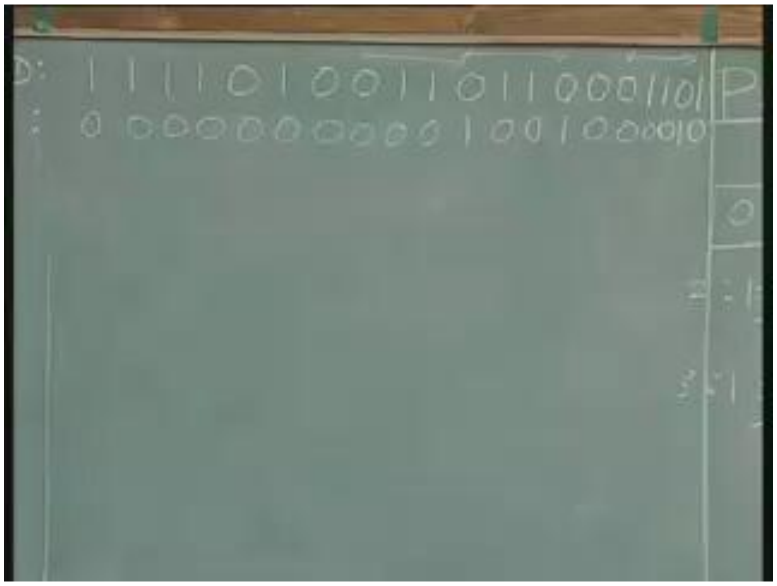
It will be trouble free and then we started with case so there must be an 'end' case here and we also started 'begin' after always so this 'end' accounts for that. For example, let us say In2 is 1 here and In 1 is also 1 both are 1 so it should naturally take you to s3. In s3 state, In2 is 0 here; When you come here it implies In2 is 1 otherwise it cannot enter this point. So, this is the condition to be met. If you have to come here, you have to (Refer Slide Time: 32:05), In1 equal to 1; that is same when this is executed. So In2 and In1 should be 1; let us verify that. We were in s3 state here when In2 is 0, it returns to 1; we found In2 1 and In1 one it should continue to remain is s3 state. If In2 is 0 it returns to s not state this you can verify once again. You can see this and we are here if In2 is 0. That is the top priority. We have given so it does not really care for In1. If it is 0 straight away take to s not. I will work out another example which you have already seen that is pattern sequence detector. We have the pattern sequence detector here we wish to find the pattern 0 1 1 0 and this is exactly same what you have already seen (Refer Slide Time: 33:24).

(Refer Slide Time: 33:11)



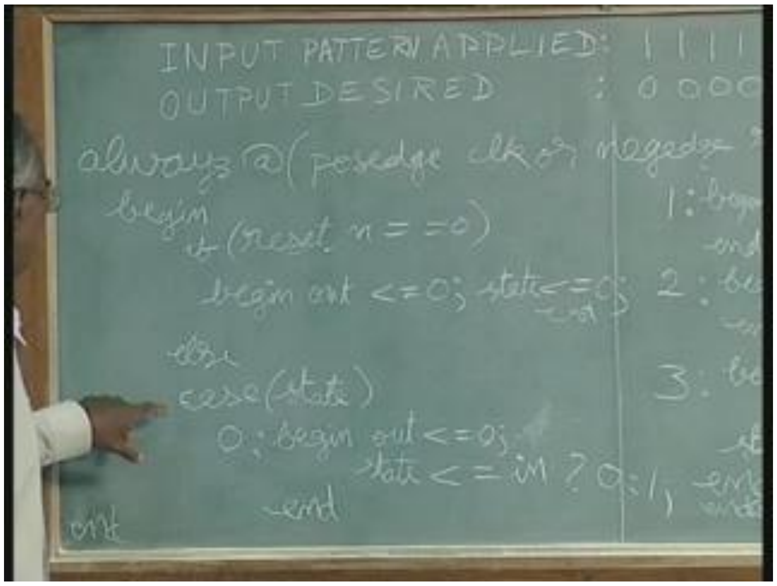
I do not have to go into this. I think I will start writing the program. The only difference is earlier you had used the notation s not s1 and being a binary bit here, 2 bits here. I am dispensing with binary and even state s not, s1 which we have seen in earlier example of sequential machine. We straight away put a decimal number 0 1 2 3 and the arrows indicate and this is the input here. First 1 is the input and output is this (Refer Slide Time: 34:10). That is all you have to remember and another variable is this state. You have as far as the output is concerned you have just 2. That is 1 is out which is single bit. State is actually 2 bits because it goes through 0 through 3; 3 means in binary it is 1 1. You need to keep track of that while writing a full fledge code. You have to remember all that; get it reflected on the code that you have. Let us start writing here and before we write. Let us take the input pattern applied as this.

(Refer Slide Time: 34:52)



All 1 1 and it may not be the very same example we had taken earlier. Any pattern is good enough because this is a machine which should cater for all sets of bit stream. Now if you inspect here - this is the output desired (Refer Slide Time: 35:04). What we want is all of them 0s. As long as the desired pattern is not got, the very first encounter with this. 0 1 1 0 pattern is here. It is all underlined here I mean rather lined over the top and 0 1 1 0 is encountered here. At that point of time we want to put it 1 here and once again this 0 1 1 0 will produce 1 here and 0 1 1 0. Once again will produce 1 here and now let us write the code for this; once again we use ‘always’ block.

(Refer Slide Time: 34:41)



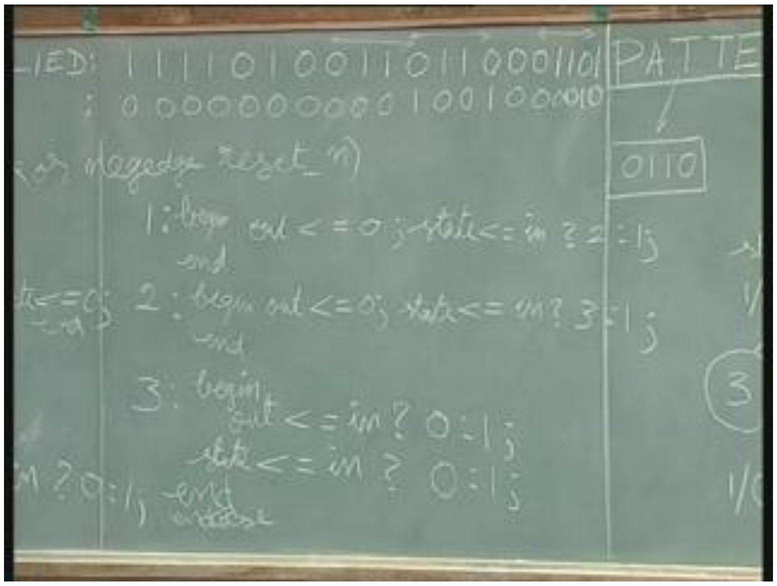
```
INPUT PATTERN APPLIED: 1 1 1 1
OUTPUT DESIRED       : 0 0 0 0

always @(posedge clk or negedge
 begin
  if (reset n = = 0)
    begin out <= 0; state <= 0;
                             end
  else
   case(state)
     0 : begin out <= 0;
              state <= in ? 0 : 1;
         end
end
```

Positive edge of clock or negative edge, notice that there is no semicolon you should remember that. Once again we will have ‘begin’ and finally ‘end’ somewhere. I will put it here. What we have seen is we have to account for 4 states 0 through 3. What we had to do is there is reset signal is not shown there. I have incorporated in the program. The first thing we have to do is if reset is encountered there are 2 outputs mind you. In this state and state itself must be 0 once again. We will put the state here; ‘begin’ and ‘end’ here. We have covered the reset portion so it not that is else. What we should do is keep on accounting for each of these states there are just four states. We will use a case statement here (Refer Slide Time: 37:45) and case depends upon the actual state here the state is only decimal number for ease of writing.

This is not a capital c but ordinary c because Verilog is case sensitive. You have to take care of the case so whereas this is not case sensitive. That is the difference. If state is 0, in this case we are putting one after another. If state is 0, then we take this action. Now once again, we say ‘begin’ and what do you want here? Output must continue to be in 0 only. What should happen to the state? (Refer Slide Time: 39:00) Let us see it all depends upon the input condition. This is actually when you enter this, we are in state 0. What you are going to write here is the next state; from this state where do you go? That state is what is implied here and that will take effect only at the arrival of the next clock. With the present clock arrival, we are here with the next arrival this will be taken into account. State will have depend upon In; suppose In is 1, we need to stay right at state 0.

Otherwise we need to go to state 1; that is what is implied here. If it is 1 you remain here if it is 0 you go to state 1. This is once again a case for using a MUX.

(Refer Slide Time: 39:53)



```
IED: 1111010011101100001101
   : 0 00000000001001000010
negedge reset_n)
    1: begin out <= 0; state <= in ? 2 : 1;
       end
te<=0;  2: begin out <= 0; state <= in ? 3 : 1;
  end      end
        3: begin
             out <= in ? 0 : 1;
           state <= in ? 0 : 1;
in ? 0 : 1;  end
           endcase
```

PATTE
0110
3

What is the selector pin here? It is nothing but In. We will put In, then question mark. It is a very simple thing it has to go either to... if it is 1, should remain in the same state. We just put 0 here, then colon then 1. Let us try in same fashion here. We can write here. For case 1, ‘begin’ and finally ‘end’. For this case, out should be 0 (Refer Slide Time: 40:46). If input is 0, it continues to remain there.

So, In, then question mark then what we have is this (Refer Slide Time: 41:14). If it is 1 what should happen here? If it is 0 it should remain there. If it is 1 it should go to 2 so we start with 2 and then colon 1. End is already there. Similarly, for 2 once again ‘begin’, ‘end’ then out here also it is 0 then state here is, we are in state 2. That is this state so input is 1. This is the output so if it is 1 you go to state 3. It is 3 here and otherwise 2 to 1 you go. This is the input and this corresponding output is the same linked to that state. We need to worry about only this input if input is 0, we go to state 1. The final thing we can write on similar fashion.

Let us take care; out will change now. We are in state 3; depending upon the input let us say the input is 0 then you have to go to state 1 as well as set the output 1. On the other hand, if input is 1 output is 0, and then it has to go back to this. Let us take this condition first. We will take the output; if it is 1, we have 2 conditional outputs; we have to use the MUX again. Select pin is In; if In is 1, output is 0.

If first 1 is corresponding to 1 so, we are dealing with output. We are just assigning 0 here and if it is 1 what is the output for 1 it is 0, for 0 this is the condition 3. We have in 1 0 that is this. If it is 0 the output is 1; so it has to be 1 here. so what remains to be done is only the state (Refer Slide Time: 44:25). Once again based on in you take action. You see from 3 you can if In is 1 you can go to 0 state. We are in 3 from 3 if it is input is 1 you can go to 0 state; if it input is 0, you go to 1 state. Finally, we say an 'end' here and there was a case here so what we have to do is write 1 'end' case also. Each of these blocks, there must be a 'begin' and 'end' and finally there must be a case. It is a counterpart 'end' case. Then coming back here this 'begin' and 'end' is already covered here. There is one more 'begin' here we said we will put one more 'end' that is this here or you can just say right here.

You can actually take this and go through this we can take an input pattern (Refer Slide Time: 45:55) and go through this and convince yourself whether the code really does what is intended to do. This is clear I suppose. Either 0 is assigned to output or 1 is assigned to output. So is the case for here state is assigned either 0 or 1 in case where it is 3, let us say you need naturally 2 bits here so state is 2 bits in width whereas out is just single bit. We will see 1 of the patterns at least for 0 1 1 0. Shall we consider all of them are 1s, when you reset, it takes you to state 0. On the arrival of the first clock pulse, because it was state 0 it initiates this; once it does its role, it has no more role to play here because it is a priority encoder. It takes you out straight away out of this 'always' block. Next action will be taken only at the next positive edge of the clock. With the arrival of the second clock pulse second in the sense relatively speaking. Once again it will see now let us say the reset has disappeared.

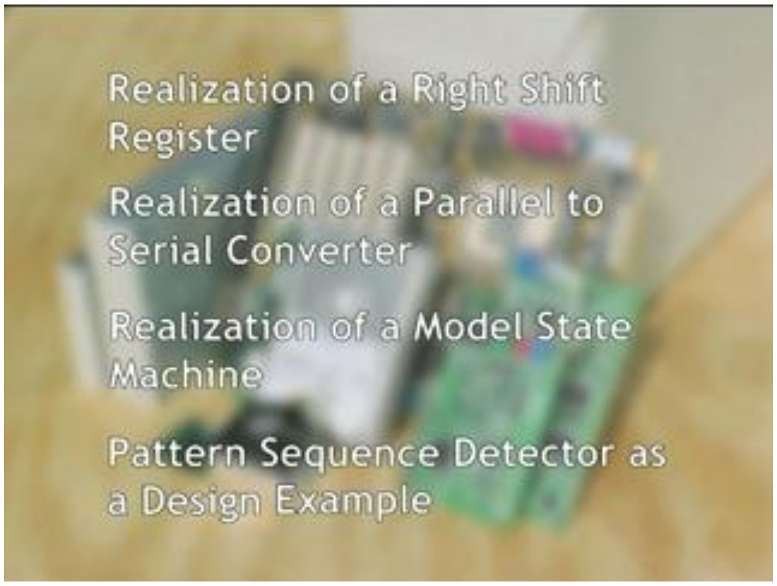
What happens is this will not be satisfied this is not executed and this was this is therefore executed only one time at the power on condition. When you switch on the system there will be a pulse created and that will reset and so long as it is asserted that long it will keep on rolling no matter how many pulses have lapsed. It will keep on doing this and get back if there are 100 clock pulses, naturally 100 times it will revolve around. Do only this. The role is only to make out and state 0. Beyond this after the reset is removed, only then it can come to this portion so having come here it was already initialized to 0.

This state therefore naturally it will go to this block execute this block and then again exit without bothering about any other states as it is priority encoder. Again this state depends upon In. If In is 1, it will be assigned 0. The state is going to be 0 but we are already in 0 state. It continues to be 0 state. So with the arrival of the next clock pulse, it will again come to the same state and continues in the same state as long as In is 1. If In is 0, state will get updated to 1. So first let us say some nth clock it as entered this for the first time. It has gone to 1 and similar explanation holds good for any other thing. Let us say for like that it has come from 1 state to another depending upon In and out. Thereby gradually takes. Everything happens depending upon the clock. Clock is too fast a thing to go in a system. Suppose it works in 100 mega hertz, even in FPGA it can have 100 mega hertz, depending on device. ASIC it can touch 300 mega hertz or beyond. Depending on the technology we can have different speeds' but for these applications that may not be a desirability. This completes this application and before we wind up we will see what assignment we can have on this.

We have seen this input. Input in this fashion but at what time should the input be valid? That is the question. How are you to feed the inputs? You have to feed well in advance before the clock strikes. Before the positive edge of the clock you should have fed this and mind you every clock you have to feed that different inputs 1 1 1 1. All should keep track of the clock speed at every clock. You should change this so it is humanly impossible thing to do that so what we can do is you can mix. We have already seen how to use the shift register in the previous example parallel to serial conversion. That is precisely what you can do. It is a very handy thing application here is it not? You as a user can give let us say 16 bits and put it in that other module what we have already done. Then serialize and that serialized pattern is what is applied here. Is that clear? When we write the simulation we will not go to the depths. We will show you that it can be done in a much more simple fashion because of the tools. The power of the tools notice that this whenever 0, you have 1 output here, which you can easily trace and convince yourself. Precisely at that point of time you have this and this can be verified very easily once we take the simulation of this later on.

Summary of Lecture 12

(Refer Slide Time: 52:10)



Realization of a Right Shift Register

Realization of a Parallel to Serial Converter

Realization of a Model State Machine

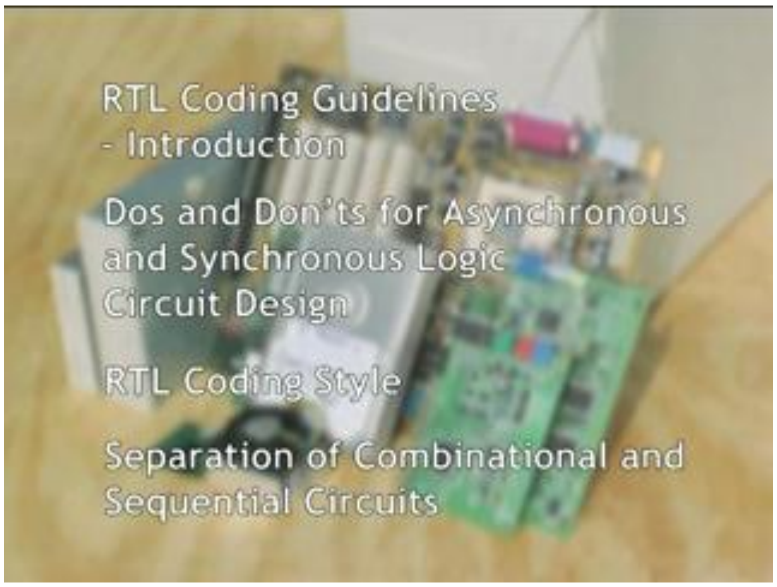
Pattern Sequence Detector as a Design Example

(Refer Slide Time: 52:36)

Next Lecture

RTL Coding Guidelines

(Refer Slide Time: 52:39)



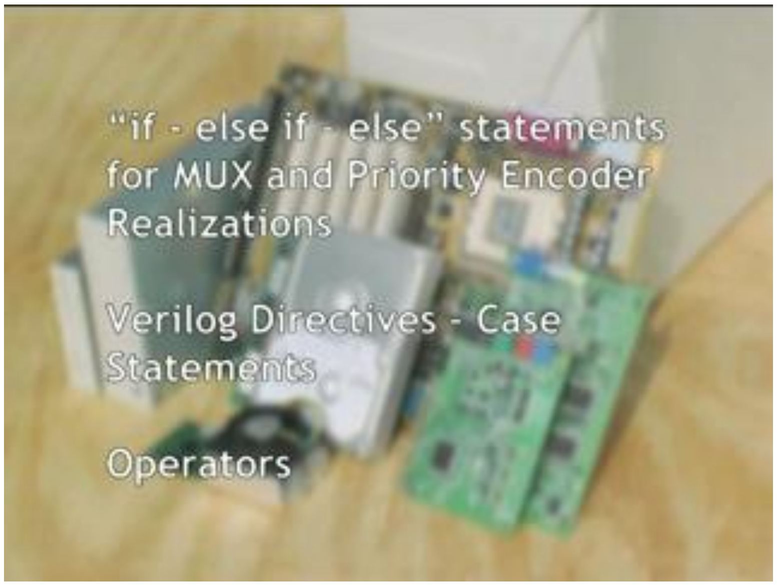
RTL Coding Guidelines
- Introduction

Dos and Don'ts for Asynchronous and Synchronous Logic Circuit Design

RTL Coding Style

Separation of Combinational and Sequential Circuits

(Refer Slide Time: 52:43)



"if - else if - else" statements for MUX and Priority Encoder Realizations

Verilog Directives - Case Statements

Operators