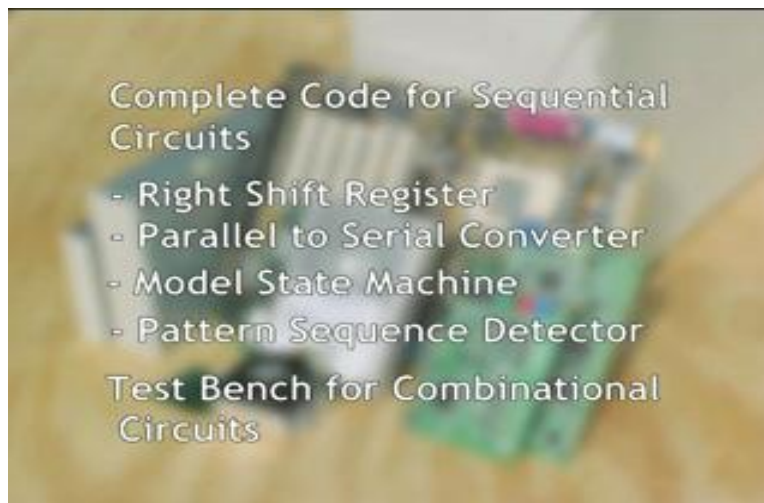


Digital VLSI System Design
Dr. S. Ramachandran
Dept of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 15

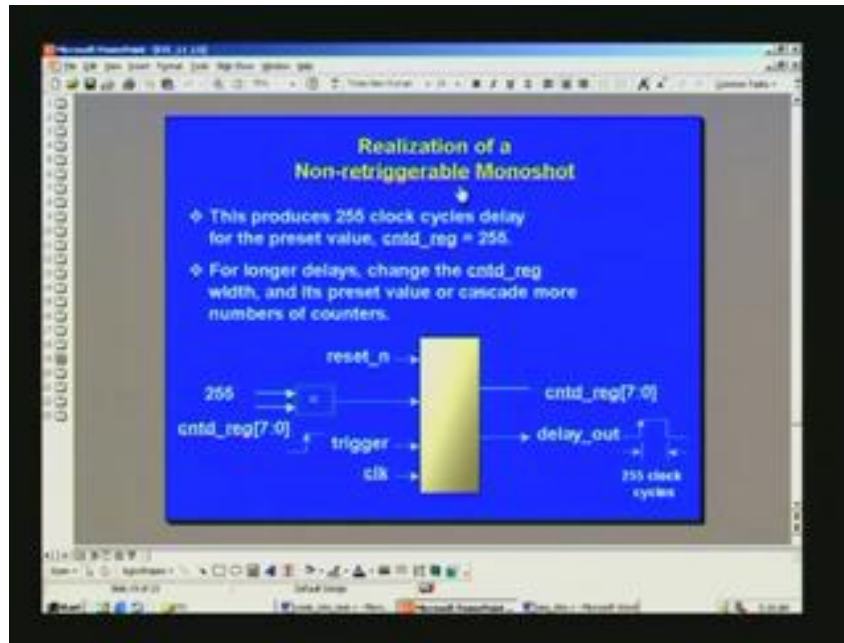
Coding Organization – Complete Realization (Continued)

(Refer Slide Time: 02:09)

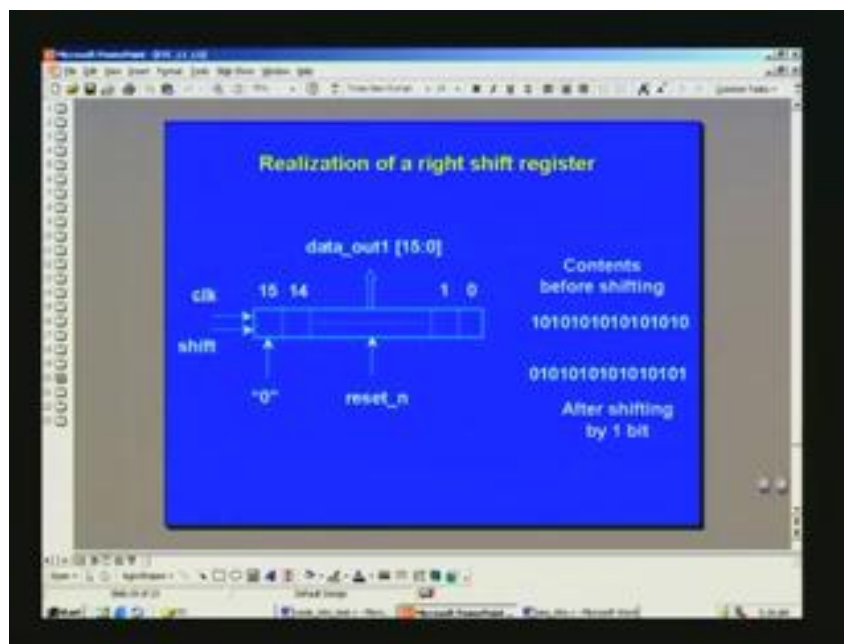


We have been looking into the organization of our Verilog code. We had covered non-retriggerable monoshot in a sequential circuit. Today, we will continue from this and we will go on to see how a shift register is implemented.

(Refer Slide Time: 02:23)



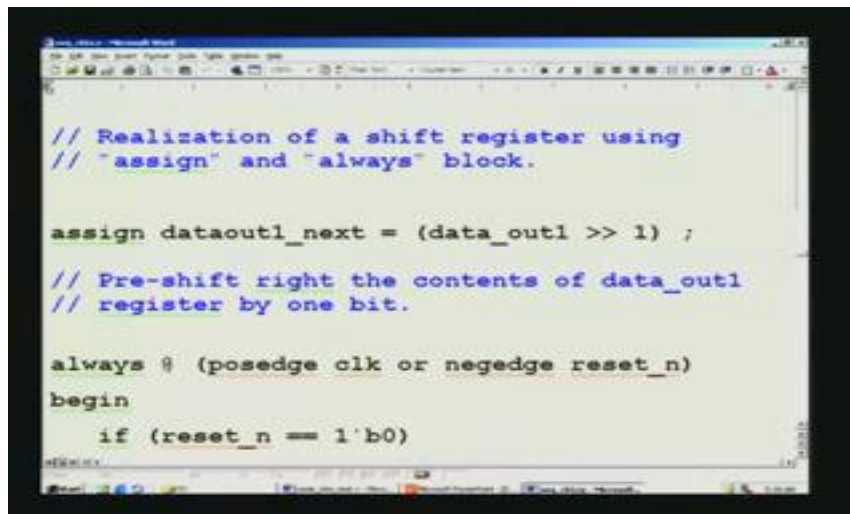
(Refer Slide Time: 02:31)



This has been explained earlier. We will have a look once again. The code for this is here, and as usual, this is the line comment. We have an assign statement which produces a combinational output as such. Here, what we want to do is a right shift by one bit. We use

a symbol like this and two greater than symbol. It implies that it is a shift. That is the right shift, since you can regard this as an arrow pointing to the right.

(Refer Slide Time: 03:36)



```
// Realization of a shift register using
// "assign" and "always" block.

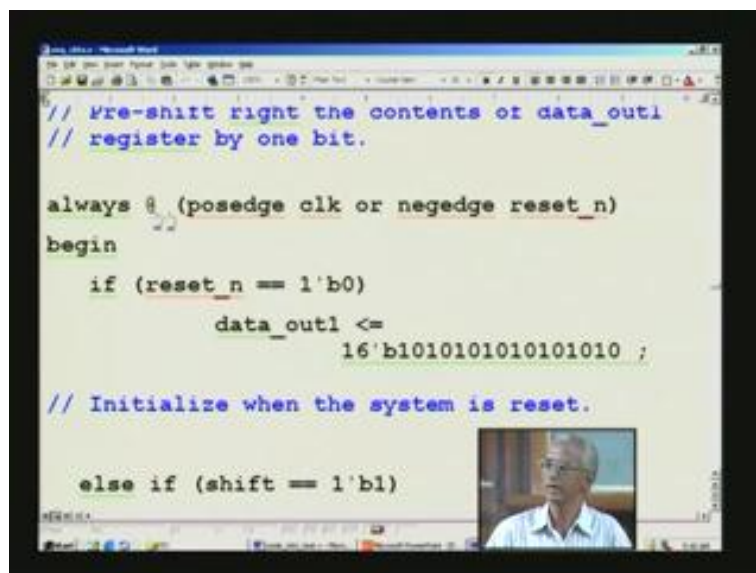
assign dataout1_next = (data_out1 >> 1) ;

// 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)
```

How many bits you want to shift can be indicated here. If you want a left shift, you just invert this to the mirror reflection of this. We assign this shifted value onto this. It goes by the same name with an extension called 'next' here. Implying thereby that, we are pre-computing this shifting so as to register it, when the event happens.

(Refer Slide Time: 03:44)



```
// Pre-shift right the contents of data_out1
// register by one bit.

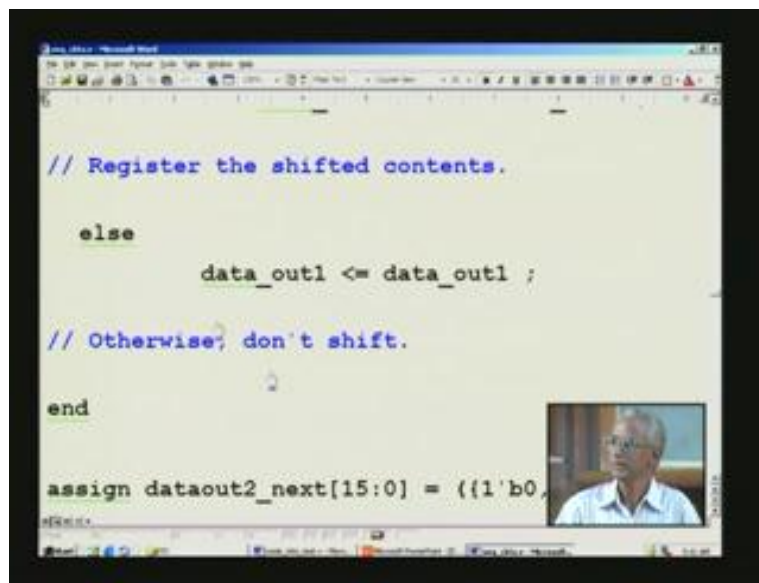
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        data_out1 <=
            16'b1010101010101010 ;

    // Initialize when the system is reset.

    else if (shift == 1'b1)
```

The event is a positive edge of the clock. Here also, based upon the reset, we initialize or rather preset the decided value here for the shift register. In hexadecimal you can regard it as all AAs, so it is a normal convention for designers to use AA and five five for bit test. If a shift is encountered, this is one time effort at the time of power on condition. In normal working we look for a shift to go high. If that happens, only then, whatever was shifted earlier using that we assign it here at this point. That means this will be assigned transferred to data_out1 only at the positive edge of the clock.

(Refer Slide Time: 04:50)



```
// Register the shifted contents.

else
    data_out1 <= data_out1 ;

// Otherwise, don't shift.

end

assign dataout2_next[15:0] = ((1'b0,
```

(Refer Slide Time: 04:56)

```
end

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
```

Otherwise, do not disturb it. This is the usual thing that we have already seen. The same shift register can also be implemented in this fashion. It is exactly the same except that the assign statement is slightly different. Our goal is to right shift by one bit. So, we can take that 15 to 0 bit. If you right shift, what will happen? Zero bit will get lost. Then what remains is, 15 is to one. So, we just concatenate two numbers that is 0 on the msb, because vacated bit will always be 0.

That is the nomenclature adapted earlier for that arrow that you have seen there. This does precisely the same thing as that one. It has been named two, just to distinguish from that. When we stimulate, we can easily compare the two and see that, it is basically same. This is the concatenation of 0 for the msb. This is the bit vacated and shifted is on right here. So, fifteen bits plus one bit here, in total, it makes sixteen bits. That is precisely, what is here.

(Refer Slide Time: 06:09)

```
always # (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        data_out2 <= 16'b1010101010101010 ;

    // Initialize when the system is reset.

    else if (shift == 1'b1)
        data_out2 <= dataout2_next ;

    // Register the shifted contents.
```

(Refer Slide Time: 06:27)

```
// Register the shifted contents.

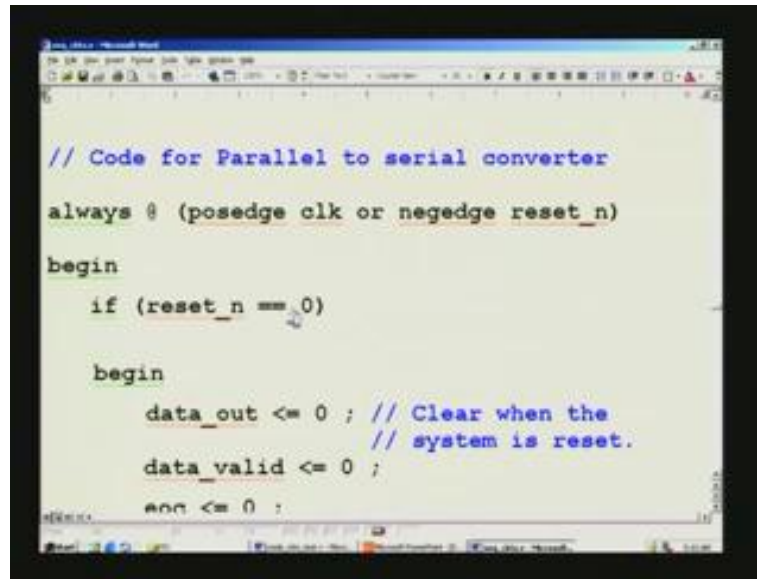
else
    data_out2 <= data_out2 ;

// Otherwise, don't shift.

end

// Code for Parallel to serial converter
```

(Refer Slide Time: 06:37)

A screenshot of a code editor window displaying Verilog code for a parallel-to-serial converter. The code is as follows:

```
// Code for Parallel to serial converter
always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 0)
  begin
    data_out <= 0 ; // Clear when the
                  // system is reset.
    data_valid <= 0 ;
    en0 <= 0 ;
```

Once again, as in the previous case, this is precisely the same as the data one case. So, I will not go into the details. Again preset here and then shifted contents are moved into that. This is not to disturb, if any other condition occurs here. Next, we had taken a parallel to serial converter. This is what we had in parallel to the serial converter. We have a power on reset or system reset and whatever values that you want to serialize; you have parallel information that can be entered into a shift register, inside in this box through a sixteen bit data bus. So, whatever data you want to send out, serialize and send it out. You can load through this, provided you assert a load input as such.

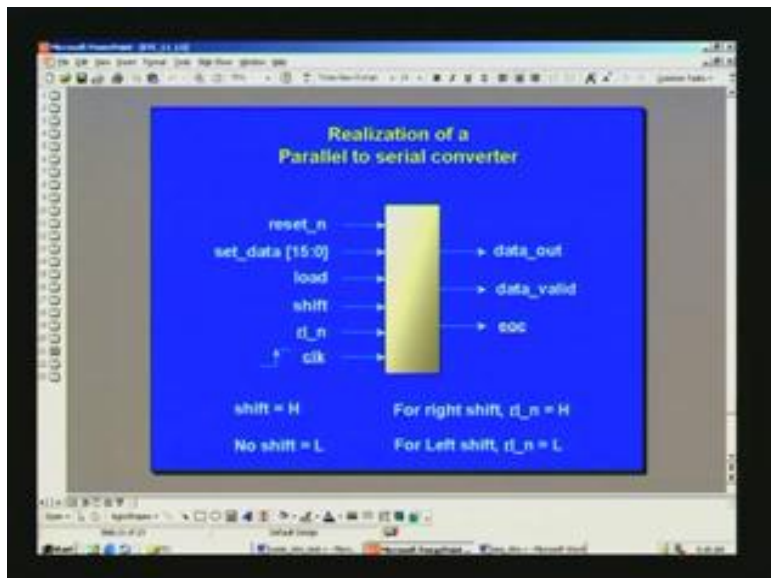
Once you have loaded, then you have one more control called 'shift control'. Together with that, you have a right or left shift available. This is basically a shift register and that is why we use this. All this shifting will happen only at the positive edge of the clock and out comes a single bit data, because that is what we want, parallel to serial. This is the parallel information and we want to serialize. Every clock pulse at the positive raising edge of the clock, you get a data out. When the data is valid, it is indicated by another signal called 'data valid'. When all the sixteen bits have been converted, thus, one end of conversion is asserted. As far as right and left shift are concerned, the right is active high and left is low. That is what is mentioned here in this. Looking at the code, we had one

always block here, because we want to take action at the positive edge of the clock. As usual, we have a system reset here, which resets all the registers that we have.

(Refer Slide Time: 08:31)

```
begin
  if (reset_n == 0)
  begin
    data_out <= 0 ; // Clear when the
                  // system is reset.
    data_valid <= 0 ;
    soc <= 0 ;
  end
  else if (load == 1)
  begin
```

(Refer Slide Time: 08:38)



(Refer Slide Time: 08:45)

```
begin
    data_out <= 0 ; // Clear when the
                    // system is reset.
    data_valid <= 0 ;
    eoc <= 0 ;
end

else if (load == 1)

begin
    sr[15:0] <= set_data[15:0] ;
    cnt_ps_reg <= 16 ;
    data_out <= 0 ;
```

The registers are: data out, data valid, and end of conversion, which is same as the output here. After resetting it, we look for loading condition. When it is asserted, what we do is, as I said, internally, it has a shift register nomenclature as sr. It is sixteen bit in width. This is the input actually here.

(Refer Slide Time:

```
end

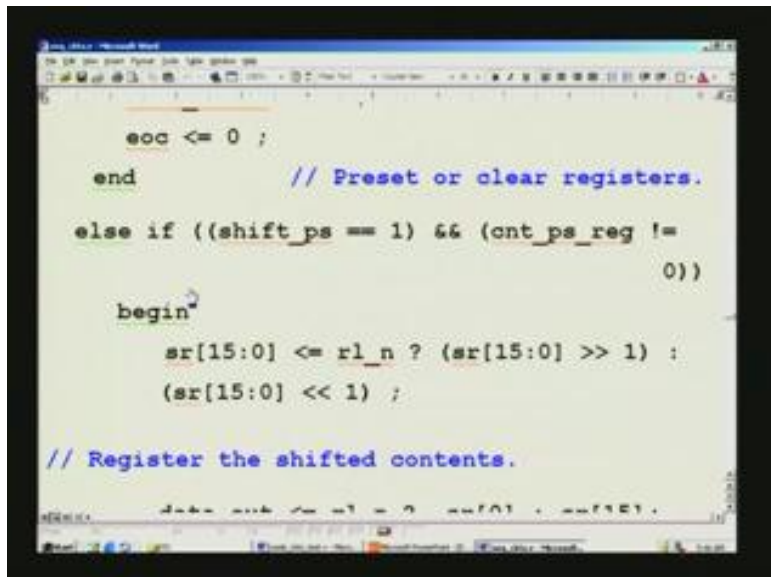
else if (load == 1)

begin
    sr[15:0] <= set_data[15:0] ;
    cnt_ps_reg <= 16 ;
    data_out <= 0 ;
    data_valid <= 0 ;
    eoc <= 0 ;
end // Preset or clear

else if (shift == 1) // cnt ps reg is
```

Whatever is at the input, it is assigned to the shift register; it is just copied inside. At the positive edge of the clock, provided the load is one, which we want. We also have a counter inside. Here, that counter is preset to sixteen. This keeps track of the number of bits that have been sent out. You want to send sixteen bits of information, that is the reason why, it is sixteen here. Data out and these things are not to be active. So, we just retain it as zero.

(Refer Slide Time: 09:48)



```
    eoc <= 0 ;
end          // Preset or clear registers.

else if ((shift_ps == 1) && (cnt_ps_reg !=
                                0))
begin
    sr[15:0] <= r1_n ? (sr[15:0] >> 1) :
                (sr[15:0] << 1) ;

    // Register the shifted contents.

    data_out <= sr[15:0] ;
```

Here, we want to shift only if this signal goes high. Also, we should have that counter non-zero indicating some more bits are waiting for transmission. So, if it becomes zero, we stop the transmission. For this condition, we naturally have to do the shifting and output the result. So, this is basically a mix to do that job based upon right or left signal. Right means one here. If this signal is one, then we do the right shift using the conventional statement and then assign this one back to itself. So, it is overwriting itself. Then there is one more possibility here. If it is zero, you left shift here. That is what we want here. Whether we want left shift or right shift, it is nearly given by this, and that is what is being done here. Right shift is done here and left shift is done here. That shifted results will be transferred once again, back to the shift register.

(Refer Slide Time: 11:05)

```
// Register the shifted contents.
    data_out <= rl_n ? sr[0] : sr[15];
// Select LSB or MSB.
    cnt_ps_reg[4:0] <= cnt_ps_reg[4:0] - 1;
// Keep track of the bits to be sent.
    data_valid <= 1 ; eoc <= 0 ;
end
```

This is actually for registering. We see here that the shift register used internally does not come out; the output you have to route that one to the actual data out. That is what we do once again based upon the right or left shift here. If it is the right shift, what we have to do is, we have to assign sr 0, because when you right shift, LSB will be sent out first. So, that is being done here. That is being sent to data out and remember that data out is only a single bit output. On the other hand, if it were left shift, it will be de-asserted. That is logic 0 low. In this case, we have to send MSB out. That is what is being done here. Having done this, you have to decrement the counter that which will keep track of the number of bits sent. Earlier, it started with 16 and now, it will decrement to 15. Now, we have started outputting the data. Therefore, the data valid is also asserted here. End of the conversion is de-asserted, because it is not a time for it. On the other hand, if this condition is satisfied, that is, when it has an output, all the 16 bits, each time it will go only through this path and it keeps on outputting bit after bit after shifting.

(Refer Slide Time: 13:01)

```
// Select LSB or MSB.
cnt_ps_reg[4:0] <= cnt_ps_reg[4:0] - 1;

// Keep track of the bits to be sent.
data_valid <= 1 ; eoc <= 0 ;
end

else if ((shift == 1) && (cnt_ps_reg == 0))
begin
```

When it touches one here, even then it will do, because, one more bit is to be processed. The next time, it comes over there; this condition will not be satisfied. A similar condition was there, which was not zero there and that condition will not be satisfied. Therefore, it will not reach this. Instead, it will come to this and the process from here. It says that, as long as a shift is still asserted, the counter is zero, signifying that all the bits have already been sent. Then we take this action.

(Refer Slide Time: 13:30)

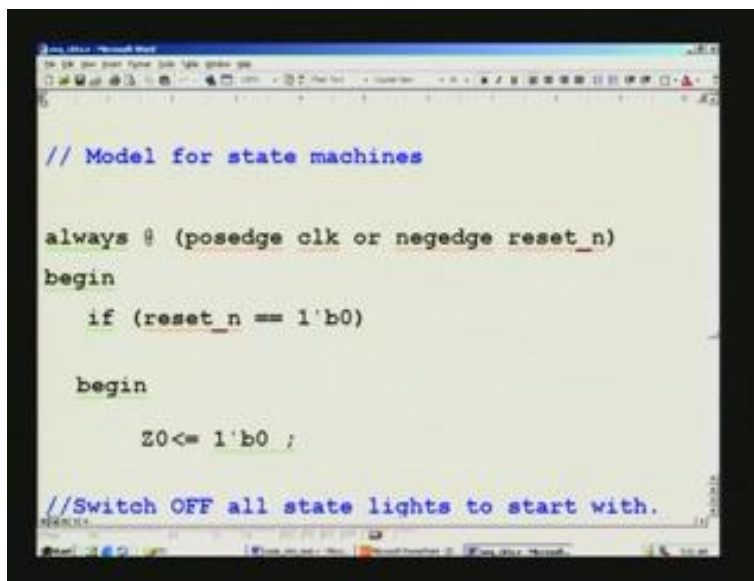
```
else if ((shift == 1) && (cnt_ps_reg == 0))
begin
data_out <= 0 ;
data_valid <= 0 ;
eoc <= 1 ;
end

else ;

end
```

In this case, we have finished the job. In fact, the previous positive edge of the clock itself, we have completed the job. Now, in this edge, we need to just de-assert the data valid, because we are not outputting any data. That is what is here. The data is zero here and naturally the end of the conversion is over. Therefore, we assert here. If no other conditions are satisfied, nothing is to be done here. That is why no statement has been put. In this, a semicolon is put. We started with an always block, so there was a beginning and it is matching end.

(Refer Slide Time: 14:13)

A screenshot of a code editor window displaying Verilog code. The code is as follows:

```
// Model for state machines

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

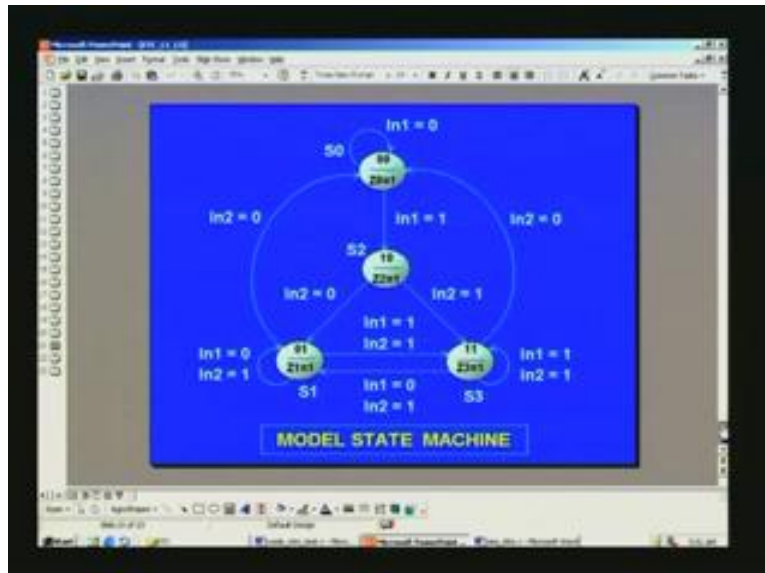
        begin

            Z0 <= 1'b0 ;

//Switch OFF all state lights to start with.
```

The code is displayed in a monospaced font with a light background. The editor window has a standard toolbar and a title bar at the top.

(Refer Slide Time: 14:19)



We also saw a model for a state machine; that is what is here. You have zee not to zee three states and these are all the outputs. For example, this is only a simple output to indicate in which state you are in, either in S not state or in S 1 and so on. Based upon two inputs, in 1 and in 2, you can make the transition from one state to another. For example, so long as this In is zero, you continue to be in the same state. If it is one, you go to S 2 states and so on. We have already seen this and we will look at the code.

(Refer Slide Time: 15:02)

```
// Model for state machines

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
  begin
    z0 <= 1'b0 ;
  end
end

//Switch OFF all state lights to start with.
```

(Refer Slide Time: 15:09)

```
    Z0<= 1'b0 ;

    //Switch OFF all state lights to start with.

    Z1<= 1'b0 ;
    Z2<= 1'b0 ;
    Z3<= 1'b0 ;

    state <= `S0 ;

    // Initialize the state when the system is
    // reset.
```

Once again, when a reset is encountered we clear all the outputs here in this case and initialize it to S not state. Note that I have made it 0. You could have made even one, because that is what the starting state is, S not state. Anyway, this will automatically take care, because with the arrival of the next pulse, it will go into S not state, in which case we would have already covered there.

(Refer Slide Time: 15:36)

```
    else
    case(state)
        `S0:
            begin
                Z0<= 1'b1 ;
                // Switch ON state 00 light and
                // switch OFF all other lights.

                Z1<= 1'b0 ;
```

(Refer Slide Time: 16:17)

```
    Z1 <= 1'b0 ;
    Z2 <= 1'b0 ;
    Z3 <= 1'b0 ;

    if (in1 == 1'b0)
// If input 1 is not active,
        state <= `S0 ;
// continue to remain in the state 00.
```

Here is a case statement. In order to go to different states and process, whatever is required for that particular state, as per the state diagram, we have already seen and that is based upon the state here. This has been defined using a define statement. Here, in this state, we need to switch on Z not and switch off all other outputs here. Suppose, In is 0, then, what we need to do is, to remain in the same state. That is what is here, so remain, in the same state so long as this is zero (referring to 14:18). If In is one, it has to go to S 2 state. That is what is happening here.

(Refer Slide Time: 16:31)

```
else
// However, if input 1 is active,
// go to the next state.
state <= `S2 ;
end
`S2:
begin
Z0 <= 1'b0 ;
```

We compare in one. If it is not zero, then only it will go to this statement. That is what we want, if input is active then it goes to state S 2.

(Refer Slide Time: 16:53)

```
`S2:
begin
Z0 <= 1'b0 ;

// Switch ON state 10 light and
// switch OFF all other lights.

Z1 <= 1'b0 ;
Z2 <= 1'b1 ;
Z3 <= 1'b0 ;
```

(Refer Slide Time:

```
        Z3 <= 1'b0 ;

        if (in2 == 1'b0)

// If input 2 is not active,
// go to the state 01.

            state <= `S1 ;

        else // Otherwise,

            state <= `S3; //go to the state 11.
        end
```

(Refer Slide Time: 17:09)

```
// switch OFF all other lights.

        Z1 <= 1'b0 ;
        Z2 <= 1'b1 ;
        Z3 <= 1'b0 ;

        if (in2 == 1'b0)

// If input 2 is not active,
// go to the state 01.

            state <= `S1 ;
```

In this case, Z 2 must be set; all others reset. We are in S 2 states. So, let us see what to do beyond this. If In 2 is 0, we need to go to S 1 state; we are in S 2 state right now. Here, if In 2 is 0, we have to go to S 1 state. That is what is here. Therefore, we assigned to S 1 here. If not, that is, In 2 is 1, we need to go to S 3 state.

(Refer Slide Time: 17:44)

```
state `S1:
begin
    Z0 <= 1'b0 ;

    // Switch ON state 01 light and
    // switch OFF all other lights.

    Z1 <= 1'b1 ;
    Z2 <= 1'b0 ;
```

Similarly, in S 1 state, it all depends upon the inputs. It may come to this state, only when those conditions are satisfied. Suppose it has come to this state, then you need to switch off all except one, which is here.

(Refer Slide Time: 18:02)

```
Z1 <= 1'b1 ;
Z2 <= 1'b0 ;
Z3 <= 1'b0 ;

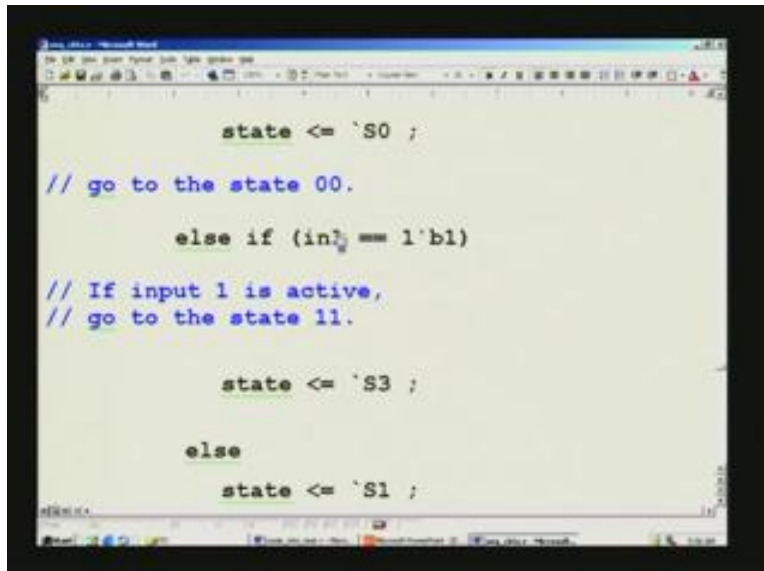
if (in2 == 1'b0)
    // If input 2 is not active,
    state <= `S0 ;

// go to the state 00.
```

Once again, In2 condition, if In2 is 0, go to S not state. We are in this one, if In2 is 0, take to S not state. Otherwise, let us say In2 equal to 1. For 0, 1, you remain here and for 1, 1,

you remain in this. Inputs will take you to either this state or this state, depending upon this two being one. So, note that this and this are same precisely, and this state is same. That is what is here, if In2 is zero, it takes you to the initial state.

(Refer Slide Time: 18:49)

A screenshot of a code editor window displaying Verilog code. The code defines state transitions based on input values. It starts with 'state <= `S0 ;'. A comment '// go to the state 00.' follows. Then, an 'else if (in2 == 1'b1)' block contains a comment '// If input 1 is active, // go to the state 11.' and the assignment 'state <= `S3 ;'. An 'else' block contains the assignment 'state <= `S1 ;'. The code is highlighted in a light blue color.

```
state <= `S0 ;  
  
// go to the state 00.  
  
else if (in2 == 1'b1)  
  
// If input 1 is active,  
// go to the state 11.  
  
state <= `S3 ;  
  
else  
state <= `S1 ;
```

Otherwise, once again we check In1 also, because two conditions are there. If it is 1, it will make an entry only with In2 equal to 1, because if it were 0, it would have gone there. So, here at this point, In 2 is 1 that means both are one here. So, automatically it has to be taken to state three. That is what is being done here. If not, that is In2 is 1 and In1 is 0, 0, 1 must take you this S 1 state. That is what is done here.

(Refer Slide Time: 19:35)

```
// Otherwise, remain in the state 01.  
  
    end  
  
    `S3:  
    begin  
        Z0 <= 1'b0 ;  
  
        // Switch ON state 11 light and  
        // switch OFF all other lights.
```

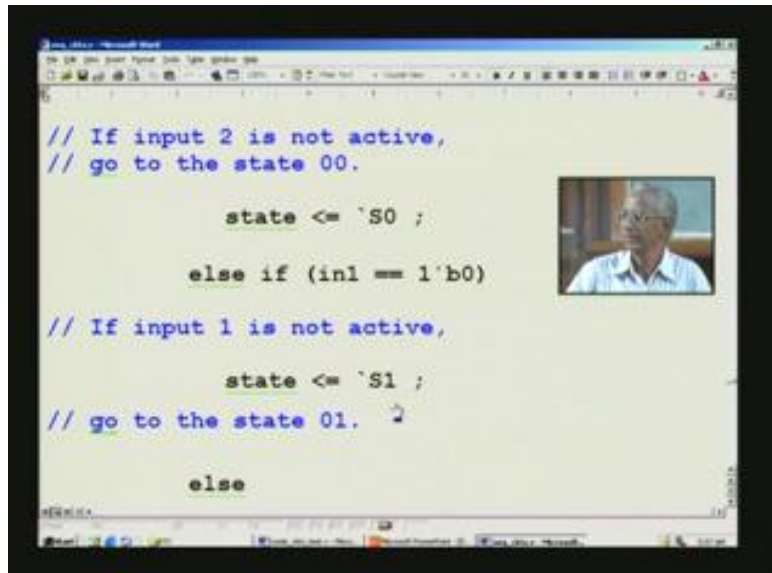
Otherwise, you may encounter next state. So, similarly process for S 3 states. Once again, that LED is on and once again you will look for the In2 input. Based on that, you go either to S not or S 1. If both conditions are not satisfied, go to S 3. That is what we have here.

(Refer Slide Time: 19:44)

```
// Switch ON state 11 light and  
// switch OFF all other lights.  
  
    Z1 <= 1'b0 ;  
    Z2 <= 1'b0 ;  
    Z3 <= 1'b1 ;  
  
    if (in2 == 1'b0)  
  
// If input 2 is not active,
```

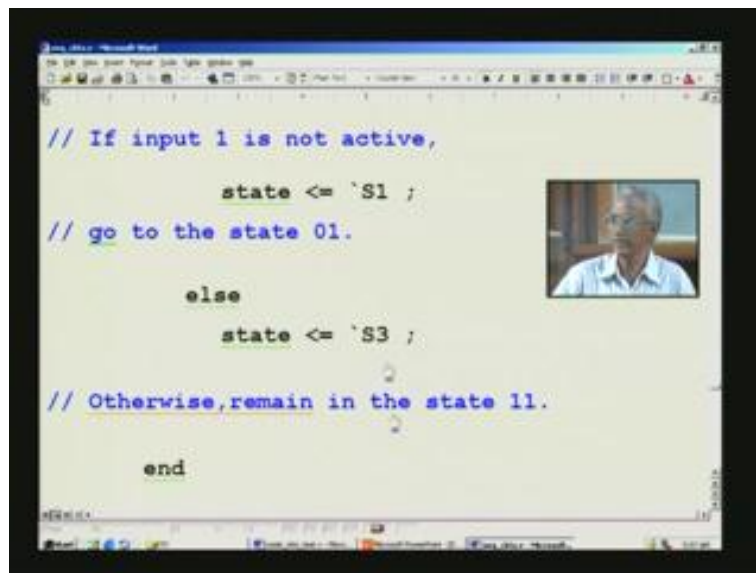
(Refer Slide Time: 19:55)

```
// If input 2 is not active,  
// go to the state 00.  
  
state <= `S0 ;  
  
else if (in1 == 1'b0)  
  
// If input 1 is not active,  
  
state <= `S1 ;  
// go to the state 01. 2  
  
else
```



(Refer Slide Time: 20:01)

```
// If input 1 is not active,  
  
state <= `S1 ;  
// go to the state 01.  
  
else  
state <= `S3 ;  
2  
  
// Otherwise, remain in the state 11.  
2  
  
end
```



(Refer Slide Time: 20:29)

```
state <= `S3 ;

// Otherwise, remain in the state 11.

end

default: state <= `S0 ;

// Otherwise, remain in the state 00.

endcase

end
```

If In2 were 0, it went to S not and if it were same 1 1, it remained as such. When it became 0 1, it went to S 1, that is what we have already seen, S 1 and then S 3. We also put a default state to take care of this try state or do not care occurring in this. We started with a case and therefore an end case. Then one begin matching end is also there.

(Refer Slide Time: 20:48)

```
end

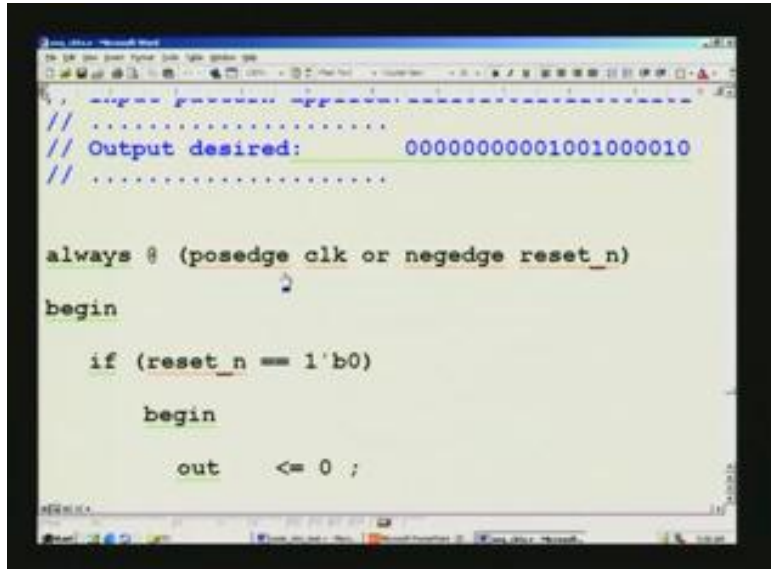
// Another design example - Pattern sequence
// detector
// Pattern to be detected: 0110
// Input pattern applied: 11110100110110001101
// .....
// Output desired: 00000000001001000010
// .....

always @ (posedge clk or negedge reset_n)
```

Next, we had considered a pattern sequence detector and this detected 0 1 1 0 sequence. For example, we may consider one of the input patterns in this fashion. The very first

occurrence 0 1 1 0 happens here. Right at that last zero, the output must be one there. Once again, 0 1 1 0 is occurring here and corresponding one is available here. Once again 0 1 1 0 and one more 1 here. This is what we have already seen.

(Refer Slide Time: 21:23)

A screenshot of a code editor window showing Verilog code. The code includes a comment for the desired output sequence: "Output desired: 00000000001001000010". The main logic is an always block triggered by the positive edge of 'clk' or the negative edge of 'reset_n'. Inside this block, there is an if-statement that checks if 'reset_n' is equal to '1'b0'. If true, it sets the output 'out' to 0.

```
// Output desired: 00000000001001000010

always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        begin
            out <= 0 ;
```

In order to this, being a sequential circuit, we once again use an always block with positive edge clock. As usual, we have a reset and we have only one input and one output, because this is a sequence detector.

(Refer Slide Time: 21:30)

```
begin
    if (reset_n == 1'b0)
        begin
            out    <= 0 ;

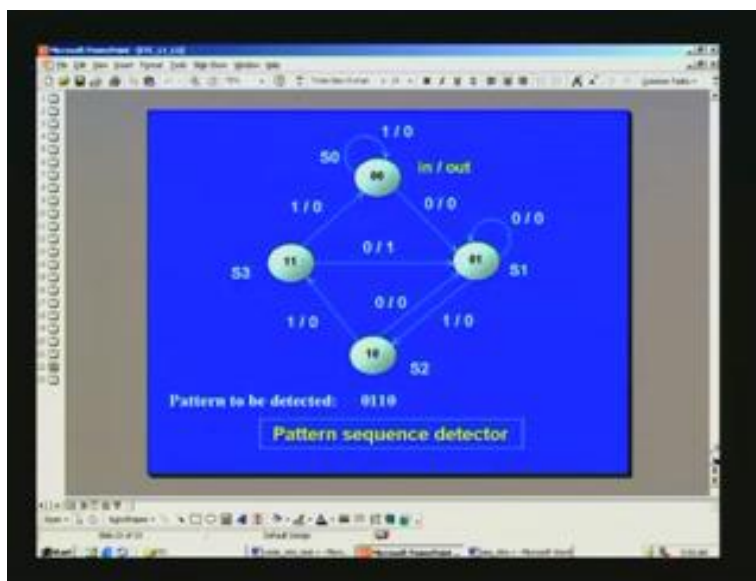
            // Switch OFF output to start with.

            psd_state<= 0 ;

            // Initialize the state when the system is
            // reset.
```

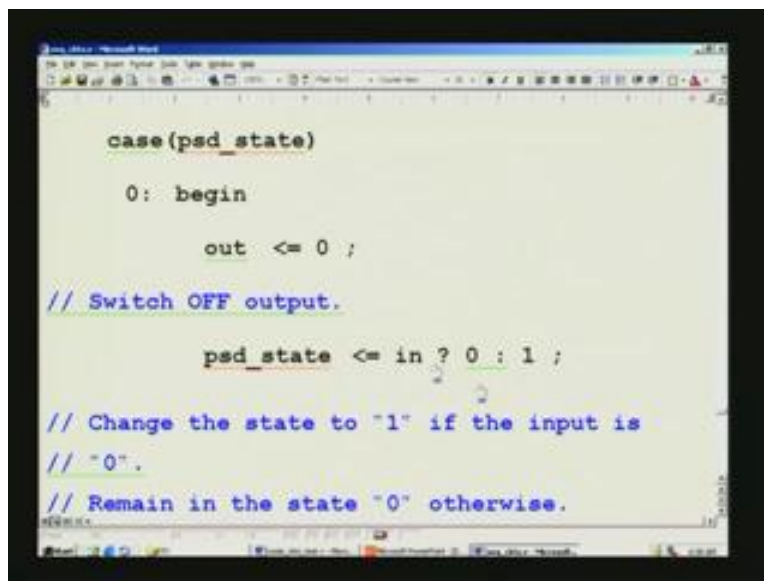
We initialize the output to be 0, whenever as reset condition is encountered. We also make this state 0 here. We will deal this state separately. In earlier example, we had used an actual name itself. We called it by name, say, S not, S 1 etc., for states. Now, let us call it by decimal number which is easy to deal with.

(Refer Slide Time:



As per this diagram, this implies that the first one is input and the output is the next one and the conditional outputs. Basically, we have four states here; S 0, S 1, S 2, S 3. We will deal with straight away decimal number, instead of a binary number, although the binary is put here. As long as one is input, then, we need to stay in the same state. If it is 0, we need to go to 0 1 state, and in both the cases, outputs are 0. So, that is in case 0. We have a decimal number, instead of a name, S 0, S 1, etc., the output needs to be 0. That is what we have seen.

(Refer Slide Time: 22:56)



```
case(psd_state)
  0: begin
    out <= 0 ;
    // Switch OFF output.
    psd_state <= in ? 0 : 1 ;
    // Change the state to "1" if the input is
    // "0".
    // Remain in the state "0" otherwise.
  end
endcase
```

Once again, we use a mux for determining the next state. It all depends upon the in. If it is 1, we need to remain in the same state. That is what we have seen in the figure. Otherwise, we wish to go to S 1 state.

(Refer Slide Time: 23:30)

```
// Remain in the state "0" otherwise.

end

1: begin
    out <= 0 ; // Switch OFF output.
    psd_state<= in ? 2 : 1 ;

// Change the state to "2" if the input is
// "1".
// Remain in the state "1" otherwise
```

When we come to this one state, we will still need to output only zero; because that 0 1 1 0 pattern has not yet come. State will have to be either two or one depending upon whether in is equal to one or zero. Here, if it is one, it has go to S 2 and if it is 0, it has to remain there. The outputs once again are zero here. In this state, you can go back to s one state, provided the input is 0. The output corresponding to that will be zero. Here, in S 2, the state input is 0. So, it should take you to S 1 state and output is 0.

(Refer Slide Time: 24:25)

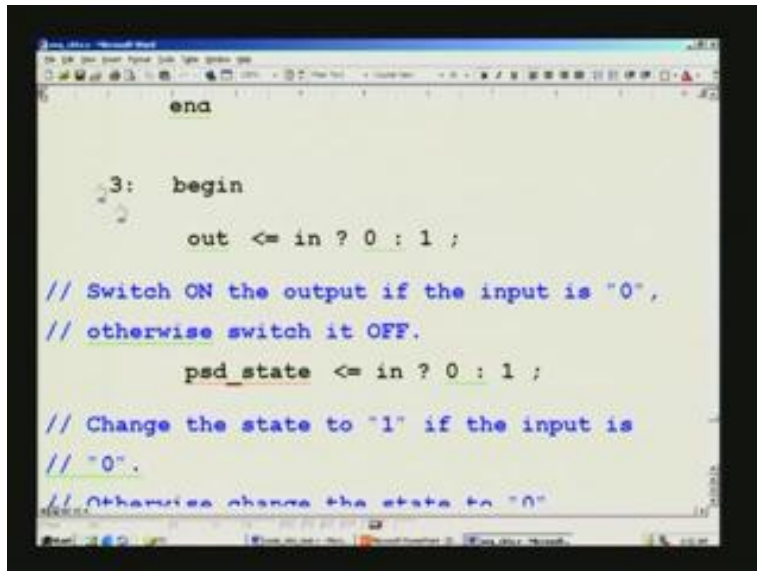
```
2: begin
    out <= 0 ; // Switch OFF output.
    psd_state<= in ? 3 : 1 ;

// Change the state to "3" if the input is
// "1",
// otherwise change the state to "1".

end
```

If the input is zero, it should take you to one state. Otherwise, it should take you to the state three. So, it should take you to state one or state three. Here also, the output is 0. That is what is here.

(Refer Slide Time: 24:59)



```
end

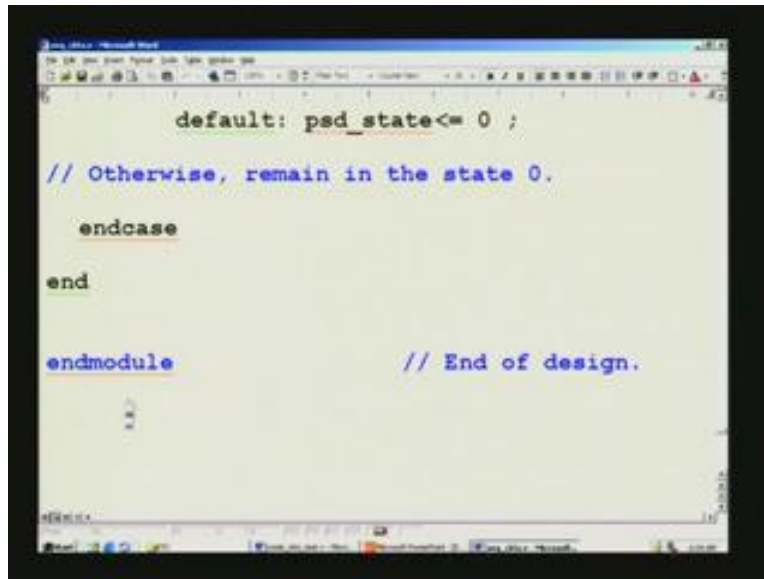
3: begin
    out <= in ? 0 : 1 ;

    // Switch ON the output if the input is "0",
    // otherwise switch it OFF.
    psd_state <= in ? 0 : 1 ;

    // Change the state to "1" if the input is
    // "0".
    // Otherwise change the state to "0"
```

Then, in the third state, if in is one, you need to go to zero from the third state; otherwise one. Just remember, 0 or one, depending upon in being equal to 1 or 0. That is, as far as the output is concerned (Refer Slide Time: 22:10). Here, as far as the output is concerned, this is 1 here, otherwise it is 0 here. Depending upon the condition, for input equal to 1, output should be 0, whereas it should output 1 otherwise. That is what we saw here. Looking at the state here, we are in s three, so if the input is one, it should take you to s not state, and otherwise, it should take you to S 1. Just remember, S not and S 1 for input equal to one and 0. Input equal to 1, it takes you to 0 and for 0, it takes you to 1 and state also depends once again on input. If it is 1, it should take you to 0 and otherwise 1 state. If in equal to 1, it should take you to 0. Here, the input is one, so it should take you to 0. Otherwise, it should take you to S 1. That is what we have seen here.

(Refer Slide Time: 27:04)

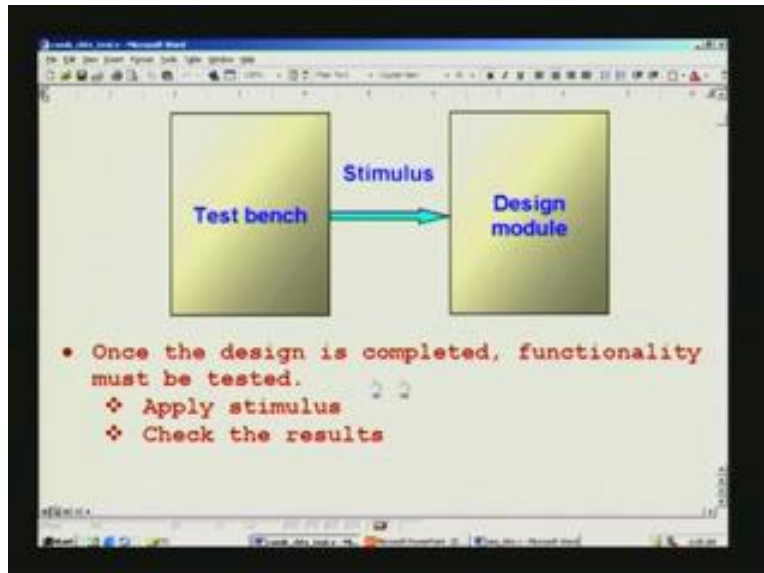


```
default: psd_state<= 0 ;  
  
// Otherwise, remain in the state 0.  
  
endcase  
  
end  
  
endmodule // End of design.
```

The default condition is also accounted for, which takes you to a safe zero state. This is a matching end case that we have here, and an end also. This will be the end of the sequential circuit design.

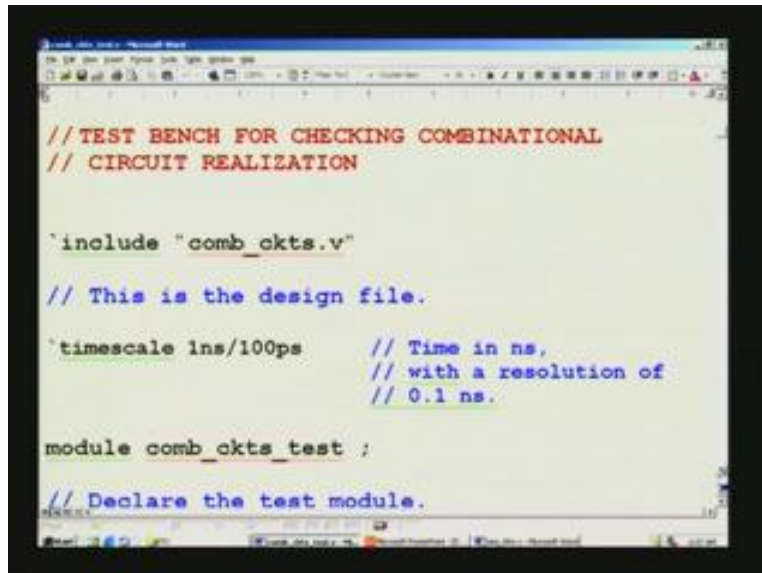
We have written a combination as well as sequential circuit, and the question is how do we test them? We need to write another Verilog code called the test bench. It also goes by the name stimulus. We will stick on to this nomenclature of calling it the test bench. Notice that, what we are going to write is a test bench, and this is actually a file. As I mentioned earlier, we try to retain the very same name that was given for the file. This one is a test bench for combinational circuits, so I give the same name here with an extension of test, signifying that this is a test bench. Some people are in the habit of giving t b- that is also alright. Whatever the designer is familiar with, he can adapt that. This is a dot v file, once again signifying that it is a Verilog file. Now, we will see how to write a test bench.

(Refer Slide Time: 28:15)



Prior to that, let us understand what a test bench is little more elaborately. We have a design module, so we have already seen a combinational circuit which is a design module. Similarly, another design module called sequential circuits exists. If you take a combinational circuit, your design module will be the combinational circuit. That is, the file that you have to test and that is actually called the design. The code which we will test the design is called the test bench. That is what is in the present file. You need to apply a stimulus. A stimulus is nothing but the application of different inputs at various points of time. That is what we are going to do in this particular exercise. You apply a stimulus and look for the results. Being a designer, you have to know how a circuit functions. You can always cross-check whether it is functioning as per your requirement. That is how you have to do. That is what this says. Once a design is completed, the functionality must be tested. That is what we have already seen. To test it, apply a stimulus from the test bench onto the design module and then check the results. How to check the results? We will be covering in a simulation later on. Right now, we will concentrate on how to write a test bench. Once you start using simulator as your method, it will be a very painful thing. So, we use a timing diagram, as it will be very convenient and you can quickly look into that.

(Refer Slide Time: 29:58)

A screenshot of a text editor window showing Verilog code for a test bench. The code is as follows:

```
// TEST BENCH FOR CHECKING COMBINATIONAL  
// CIRCUIT REALIZATION  
  
'include "comb_ckt.v"  
  
// This is the design file.  
  
'timescale 1ns/100ps // Time in ns,  
// with a resolution of  
// 0.1 ns.  
  
module comb_ckt_test ;  
  
// Declare the test module.
```

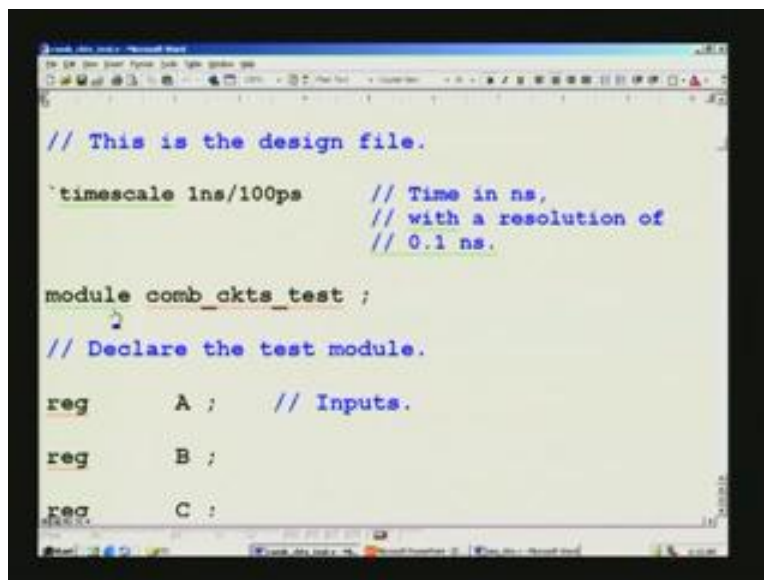
Analysis will be faster using timing diagram. That is why, it is generally preferred. With designers, it is very popular. It has also picked up in institutions, just as it is popular within the industries. The very first statement in the test bench is, we have a design called combinational circuits and that particular thing was a different file. It is not this file. This file is the combinational circuits underscore test; so you should be very clear. That is why, earlier we put a block separating out the test bench and the design. How does this know unless you mention so? This is the statement that you have to give in order to identify that there is a design called combinational circuits and that is external to this present file.

In order to do that, you say “include”. This is a very same thing as a c command instruction. We have a tick here, reverse tick here; this is a must here. This is a keyword that you cannot ignore. Separate with one or more blanks or tabs and then put within quotes. Quotes are also required and then it is a note that is a dot v. So, we have to completely specify the file name. If it is in some other directory other than this present thing, you have to give the entire path. But, it is a good practice for you to keep everything in the same directory, in which case you do not have to put that. So, we have another thing. We are going to simulate your code. So, we are going to simulate it, that is, to apply different pulses with reference to different times. There must be some time base

to tell the system that you want a time base of say, one nano second. In that case, you have another statement. Once again, a tick and a time scale, a space must be given, and then say one nano second or you want one micro second or one u s you can put here. You must mention the resolution you want, that also can be specified here. For example, it is a 0.1 nano second resolution here; hundred pica seconds means 100 by 1000. So, it is a 0.1 nano second. That is what is put here, with a resolution of 0.1 nano second.

Next thing is, as we have designed for the design file, we need to declare the module. So, the module actually starts here. These are all extraneous things to the module. That means, there are the defining time and earlier things defined, which is the input file and design file. This is outside this module.

(Refer Slide Time: 33:02)



```
// This is the design file.

`timescale 1ns/100ps      // Time in ns,
                          // with a resolution of
                          // 0.1 ns.

module comb_ckts_test ;
// Declare the test module.

reg    A ;    // Inputs.

reg    B ;

reg    C ;
```

Test bench is the module with the file name. It is a good practice to retain the very same name as you have the module name. There is no compulsion that you should have same. In fact, I mentioned that, you have to have one, but it is a cross-check. You can change the name, but readability will be easier. If you are maintaining so many files, if you have the same file, you can easily spot it out. That is why I prefer to retain the very same name. Here, you declare the module with a space and then the actual module name you


give here. This happens to be the test bench; so you will give the full thing. Notice that, dot v is not to be given here. If you give dot v, the compiler will complain.

Notice that, in design, we had listing of I/Os. Here, this is only a test bench. So, there is no other higher module for this. That is why there are no I/Os here. We dispense with the brackets as well and listing of I/Os; because there are no I/Os for the test bench. Whatever are I/Os in the design module, that itself can be viewed right from this top. This is the top of the design. Actually, when you say design, when you refer to the top module, you refer to the actual design module and top. But, in this case, when you test and have a test bench, the test bench becomes an even higher level than the design module, because the design is being called by this test bench. That is why we do not have I/Os, and therefore, have not listed that. Do not forget the semicolon, which tells at least module definition has ended. In order to communicate that module has ended, you have to have an end module which will appear at the fag end of this code. Now, the difference is in the design file.

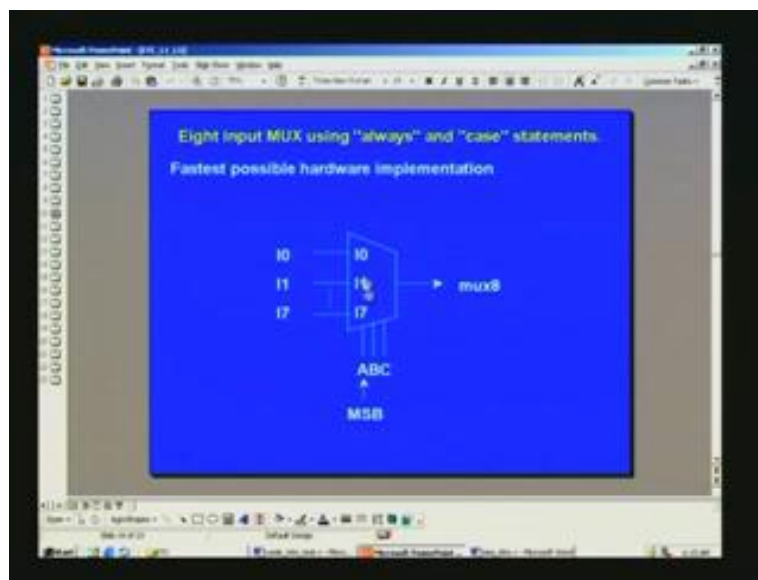
We said that the input, etc., we had declared, because that was appearing right here when we are listing. Therefore, you will not find any inputs and outputs listed separately. Instead, you have reg and wire here. We will explain why a reg and why a wire. For example, in the design, you have had all these as inputs such as A, B, C, then, IO, and so on. These are all general inputs that we have had for A, B, C. What we are doing in test bench is, that we declare it as reg. That is, wherever inputs are concerned, it is a reg. You should not confuse with the register that we have had using always a positive edge clock earlier for implementing flip flops. That reg is totally different from this reg. What all this means is that, we want to hold this value, because we are going to test our system. Testing implies that we are going to go from one point of time to another. So, we may be right now at the zero time. We will move on to say, next 10 nano second and 20 nano second and so on. We may go step by step advance in terms of time. When we do that, unless this is held firmly, you will not get a stable input for your simulation. Therefore, it is essential that you declare it as reg. If you declare it as any other thing, it may complain again. Here, these are all general inputs we have had for the combinational circuit.

(Refer Slide Time: 37:30)

```
reg I0 ; // MUX inputs.
reg I1 ;
reg I2 ;
reg I3 ;
reg I4 ;
reg I5 ;
reg I6 ;
```



(Refer Slide Time: 37:56)



We remember that we had a 8 input mux. That is, I0 through I7 are those inputs for this mux. In fact, you can have a look at what we had. Then, we had a magnitude comparator. It had contained N1 and N2 as two 8 bit numbers. That is what is here, say N1 and N2, it is a magnitude comparator.

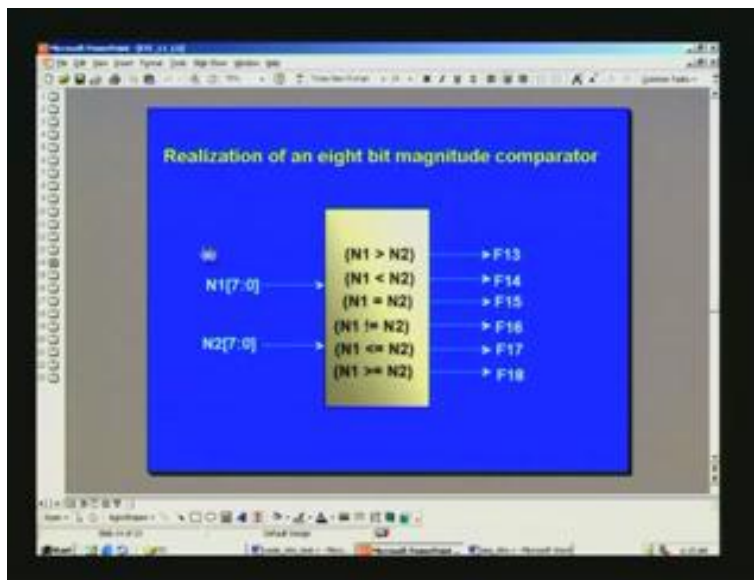
(Refer Slide Time: 38:28)

```
reg      I7 ;

reg      [7:0]      N1 ; // Inputs of magnitude
           // comparator.
reg      [7:0]      N2 ;

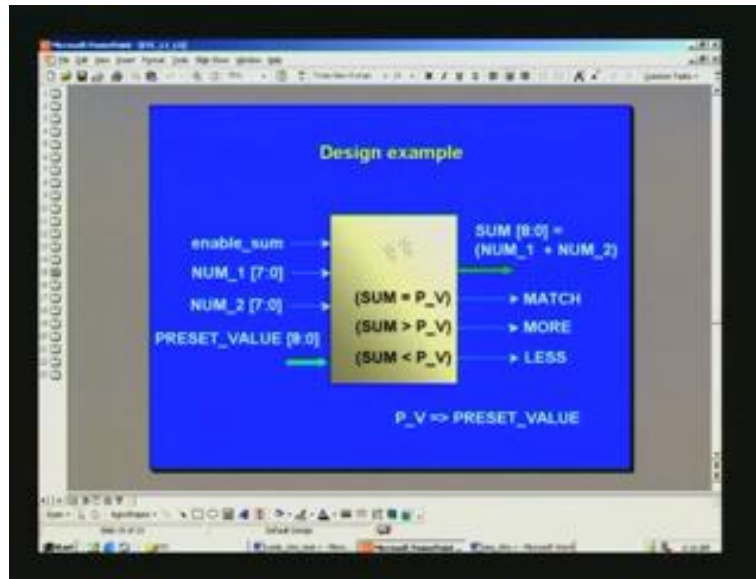
reg      enable_sum ;           // Inputs of
reg      [7:0]      NUM_1 ; // sum & compare
reg      [7:0]      NUM_2 ;
```

(Refer Slide Time: 39:20)



We will see shortly, how to change the time. Once again, we have a multi-bit representation. This, we have already seen earlier. We also had another design example, in which case we just summed two numbers, Num one and Num two. Each of them are eight bits, and once again these are all inputs. Therefore, reg has been declared here and there was also a preset value, which was compared with the sum of these numbers. When a match was found, we reported that is what the example here.

(Refer Slide Time: 39:24)



We had two numbers here. We added this one, provided enable sum is energized and only then, the sum will be non-zero. The actual sum equals preset value here. This I had just abbreviated, because there is no space to put here. When it is equal or greater or less, then you get a match more or less as the output for this design example. That preset value is nine bits; that is what is here.

(Refer Slide Time: 39:58)

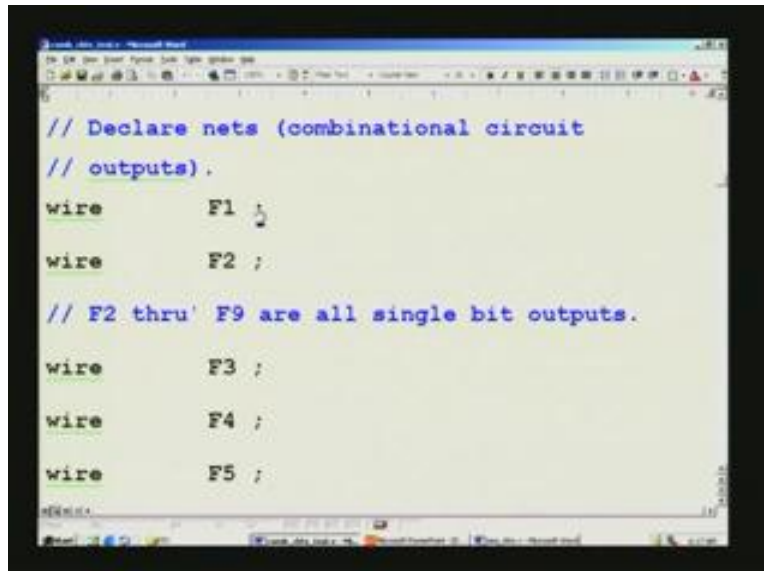
```
reg [1:0] N1 ; // inputs of magnitude
// comparator.
reg [7:0] N2 ;

reg enable_sum ; // Inputs of
reg [7:0] NUM_1 ; // sum & compare
reg [7:0] NUM_2 ;
reg [8:0] PRESET_VALUE ; // example.
2

// Declare nets (combinational circuit
```

As far as the outputs are concerned, this is basically a combinational circuit outputs, using either always block or assign statement. Here, we have to declare it as a wire. We will see why it is a wire, later on.

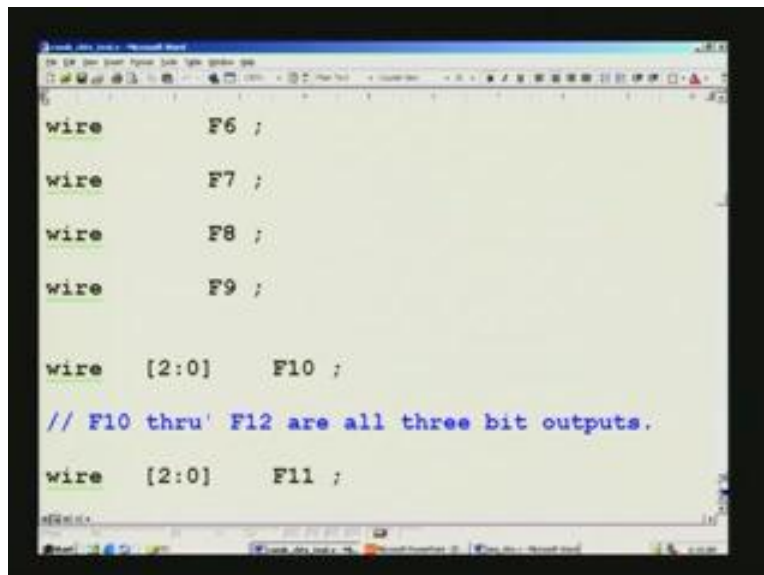
(Refer Slide Time: 40:26)



```
// Declare nets (combinational circuit
// outputs).
wire      F1 ;
wire      F2 ;

// F2 thru' F9 are all single bit outputs.
wire      F3 ;
wire      F4 ;
wire      F5 ;
```

(Refer Slide Time: 40:41)



```
wire      F6 ;
wire      F7 ;
wire      F8 ;
wire      F9 ;

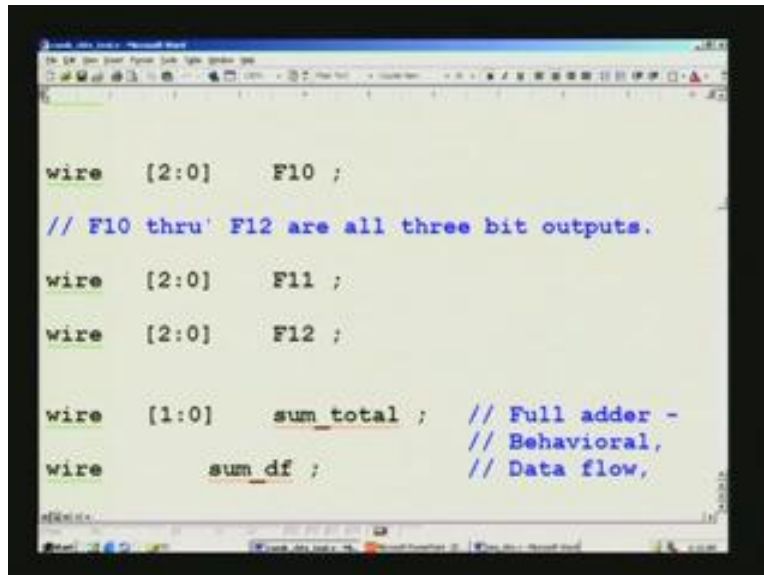
wire [2:0] F10 ;

// F10 thru' F12 are all three bit outputs.
wire [2:0] F11 ;
```

We had F1 through F12 earlier for buffer inverter etc., and that is what I do not have to show it I suppose here. All of them are outputs. So, it is declared as wires. We will come

shortly to that. We had some other outputs which are three bits in width and therefore are mentioned as two to 0. All the outputs are combinational outputs, and are all wire. So, it is very simple in test bench.

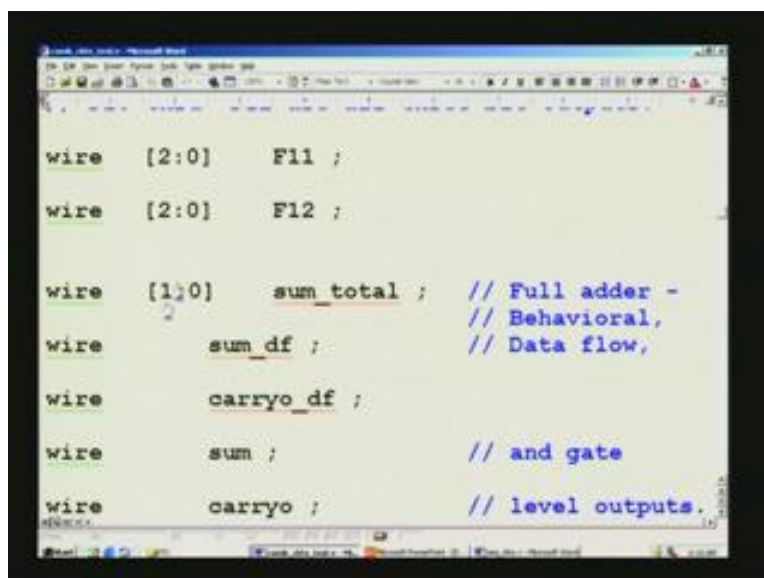
(Refer Slide Time: 40:59)



```
wire [2:0] F10 ;  
  
// F10 thru' F12 are all three bit outputs.  
  
wire [2:0] F11 ;  
wire [2:0] F12 ;  
  
wire [1:0] sum_total ; // Full adder -  
// Behavioral,  
wire sum_df ; // Data flow,
```

All the inputs are declared as reg and all outputs are declared as wire. Note the difference between a design file and this.

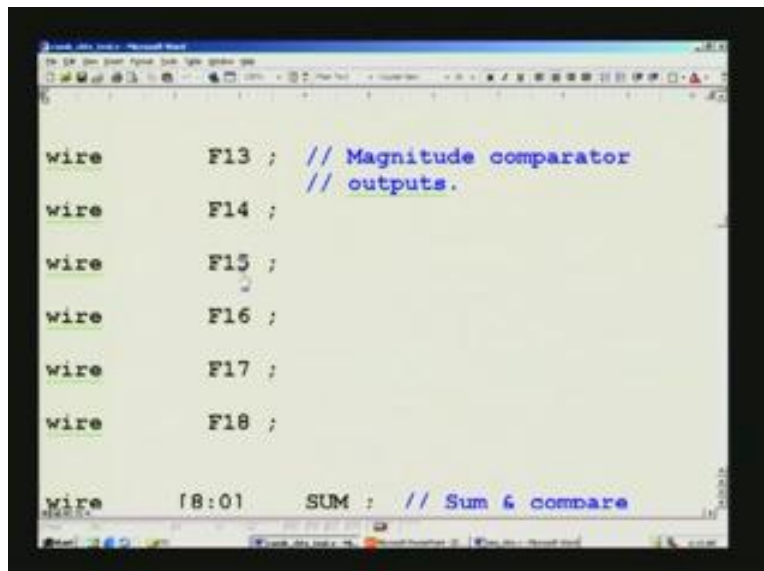
(Refer Slide Time: 41:13)



```
wire [2:0] F11 ;  
wire [2:0] F12 ;  
  
wire [1:0] sum_total ; // Full adder -  
// Behavioral,  
wire sum_df ; // Data flow,  
wire carryo_df ;  
wire sum ; // and gate  
wire carryo ; // level outputs.
```

One was behavioral statement. That is here. The corresponding output sum is two bits. The first bit signified it was a sum and carry the one. This is a behavioral thing. We have also covered a data flow here that is differently named so as not to overwrite when we simulate. Corresponding carry out is also here. We also had a gate level implementation of the same and that is here sum one. We also had a magnitude comparator and those outputs are F3 to F18 here and that is as per this.

(Refer Slide Time: 42:16)



```
wire      F13 ; // Magnitude comparator
           // outputs.
wire      F14 ;
wire      F15 ;
wire      F16 ;
wire      F17 ;
wire      F18 ;

wire      [8:0] SUM : // Sum & compare
```

This is the one, F13 to F18 for various conditions which you have already seen. I do not have to go into this; based upon two numbers N1 and N2. We also had summing that which we have already seen there. So, that was previous slide we have seen.

(Refer Slide Time: 42:41)

```
wire    F17 ;
wire    F18 ;
wire    [8:0]    SUM ; // Sum & compare
                        // example.
wire    MATCH ;
wire    MORE ;
wire    LESS ;
```

We have seen corresponding outputs for the inputs there. Now, we list outputs here. Coming to that explanation of wire, before that, we will learn how to instantiate a design module. We are now right on the test bench and we are yet to call.

(Refer Slide Time: 42:55)

```
wire    MORE ;
wire    LESS ;

comb_cks  ul(

// Instantiate the design module.
// ul stands for the first instantiation.

    .A(A) , // Connect ports by name.
    .B(B) ,
    .C(C) .
```

We have identified I/Os rather of the design module. How are we to press the design module into service? That is the question. The way to call it is just name, that is here in this case, the design module is combinational circuits, underscore circuits, and not the

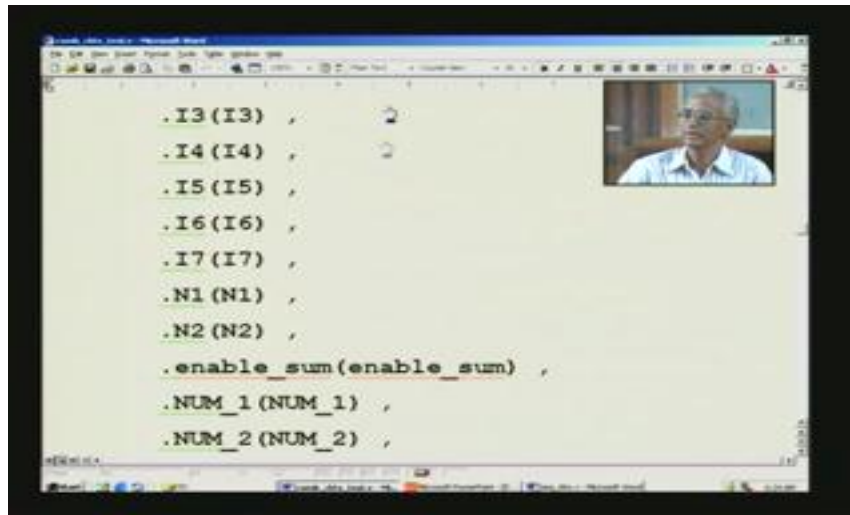
underscore test. The underscore test is this test bench, which we are right now on. We are going to call this one. What it means is when you convert this into an actual working circuit as an i c chip, then what happens is, this circuit will actually function because of this statement. You have to instantiate here. Whatever is the design module, you have to call that module. That is known as instantiation here. You just put that particular design module name, and that is all.

This is different from the test bench. We have declared a module called the combinational circuits underscore test. Whereas we are calling the actual design only here, and this you can call any number of times. Suppose you have circuit diagram, let us say, we have a t t l gates say seventy-four l s 0 0 0. There are ten numbers on a board. How do you identify them? You identify them as u1, u2, u3, and so on. That is the nomenclature you adopt and the same is the case. The very same thing has been adopted here also. You can just call this once and then say you treat this as one particular package, as one i c like this.

You can call any number, any number of times and any other module you can call here. Once you call this one and there are I/Os here, we have to list these I/Os. Those I/Os, how do you express them? We will see that also. Why should we declare that output as a wire? I have pointed out that like this, we can have a number of modules in the same test bench or even within a design. You can call any other design sub modules, you can call. So, in which case, there must be a way to identify the connection between one module and the other. How do you identify it? If you look at the circuit which you have hooked up in the earlier days, you physically connect one particular pin to another. It may be from one i c terminating on another i c. So, the connection is nothing but a net or simply a wire. We want to connect one i c with another, this is u1. You may have another u10 and you want to connect a particular pin to u10 from some other pin. You have to declare that as a net or a wire. That is the fundamental there in declaring all the outputs here as a wire. Depending upon the actual circumstance, you will have to do that. As far as if it is test bench. Now, we have one more new thing introduced here. This comment is simply that we are instantiating followed by i c so to say, which is called Identification number. Then, you list as usual with an opening bracket here and there will be an end bracket.

You list all I/Os, because this is the design module. As I said, you have a pretty long list here of I/Os. So, what is the guarantee that you have adapted the very same format?

(Refer Slide Time: 50:20)

A screenshot of a video lecture. The main part of the image shows a list of I/O signals in a design module, displayed in a code editor window. The signals are: .I3 (I3), .I4 (I4), .I5 (I5), .I6 (I6), .I7 (I7), .N1 (N1), .N2 (N2), .enable_sum(enable_sum), .NUM_1 (NUM_1), and .NUM_2 (NUM_2). Each signal is preceded by a dot and followed by its name in parentheses. To the right of the code editor, there is a small inset video of a man speaking. The background is dark, and the code editor window has a light background with a blue title bar.

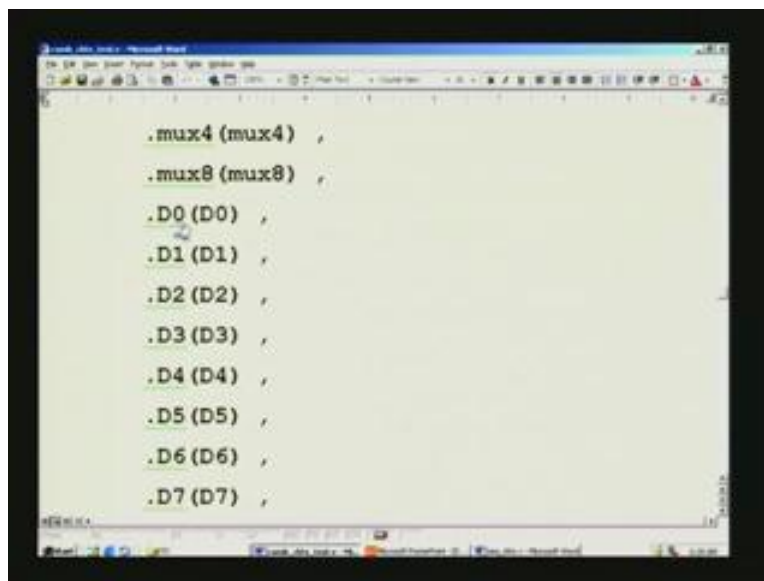
```
.I3 (I3) ,  
.I4 (I4) ,  
.I5 (I5) ,  
.I6 (I6) ,  
.I7 (I7) ,  
.N1 (N1) ,  
.N2 (N2) ,  
.enable_sum(enable_sum) ,  
.NUM_1 (NUM_1) ,  
.NUM_2 (NUM_2) ,
```

What is the guarantee you have put in the very same order? Suppose you violate the order, the whole thing will become a mess. Suppose you put A there and C next there in your design module and without this dot a convention, if you had used just A, and for the next you use B, whereas in the actual design module is C. So, there is a difference between the design module and the present module. If there is a difference, then it will report an error. The order matters, because you do not know whether it is an input or an output. As I said, you can have a free mix of inputs and outputs, unless you maintain the very same order. How will the tool know you can have many combinations? You can put one thing there and totally radically change from that combination. How does the tool know? It does not have that much intelligence, you have to inject that intelligence. The way to do this is, what is called connect ports by name.

So how do you connect the name? You use a dot here and these two need not be the same. This means that within brackets, you are implying what these signals are. A signal is within this test bench. What is covered within the brackets is within the test bench, and not in the design. Whereas this means, this is the nomenclature adopted in the design. That is how you tie between the various permutations combinations. I can just remove this and cut paste somewhere here at random. Still it will work in any combination, you can have a twisted

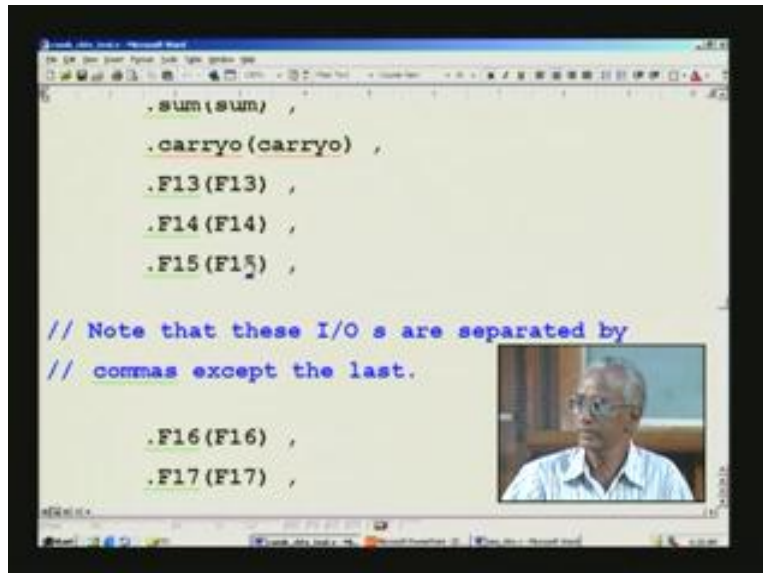
jumble of such variables that it will take care for the simple reason. That it is saying, which is the current signal for the test bench, which is the signal for the design you are discriminating. To repeat again, this one pertains to this test bench signal, whereas this pertains to the actual design. So, dot A means this parent or the other way. What this implies is, the design; so that is the nomenclature. What all we do here is, just list all the inputs that we have already seen right in the same fashion. By doing so, you can even if you stop the order by mistake, it will automatically take care. That is the purpose there and all good designers adopt this scheme. I highly recommend that you also adopt the very same thing. These are all what we have already seen pertaining to where inputs and outputs are all are listed here. This is for the d mux case here.

(Refer Slide Time: 50:32)



```
.mux4 (mux4) ,
.mux8 (mux8) ,
.D0 (D0) ,
.D1 (D1) ,
.D2 (D2) ,
.D3 (D3) ,
.D4 (D4) ,
.D5 (D5) ,
.D6 (D6) ,
.D7 (D7) ,
```

(Refer Slide Time: 51:05)



```
.sum(sum) ,  
.carryo(carryo) ,  
.F13(F13) ,  
.F14(F14) ,  
.F15(F15) ,  
  
// Note that these I/O s are separated by  
// commas except the last.  
  
.F16(F16) ,  
.F17(F17) ,
```

These are all the outputs that you have here. It can be with different names and need not be the same name. This pertains to the present one. I can give it as F25. So long as you wish, but that will have to be declared as a wire in this particular, whatever level whether it is a test bench or even in design module, you declare it as a wire. This can be seen when we deal with bigger applications, design applications which will take that. Now, we will see how to make compact modules. We have put here as a segregated thing which appears to be little out of place. The point is, I have just wanted to show in one platform as far as the simulation and the synthesis tool is concerned. That is why it has been clubbed, although some of them may be a disjointed thing. It is to have a sort of a ready reckoner that you combine all this. Otherwise, we could have had small modules, which we will be taking in the future. That is how it has to be designed. We will see consider depth later on and this exhausts the first module name. I think I will stop here and continue in the next class. We will see how to mention the time and so on. We have one more test bench for a sequential circuit also. That also will be covered in the next session.

Thank you.

(Refer Slide Time: 52:53)

