

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

Lecture – 53

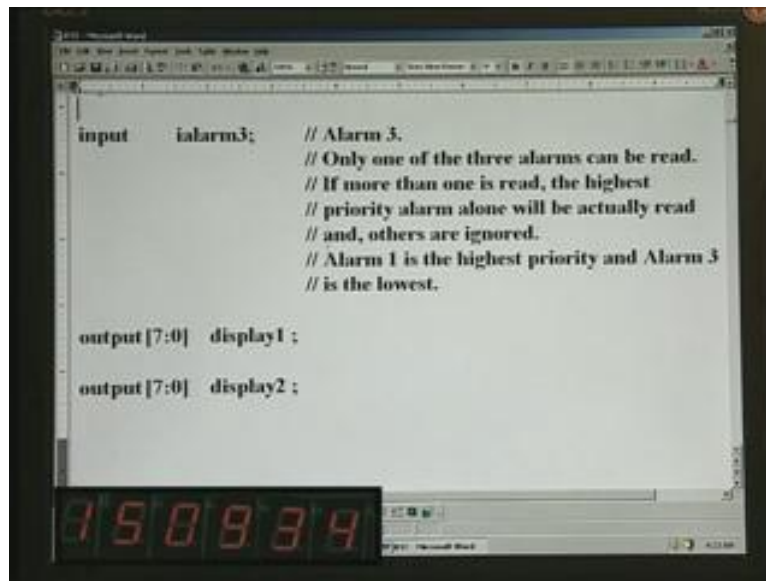
System Design Examples using FPGA Board (Continued...)

(Refer Slide Time: 01:25)



We will continue with the real-time clock Verilog coding. I crosschecked once again what I have written. Some students expressed there is some mistake. Please note the correction for the same.

(Refer Slide Time: 02:07)



```
input    ialarm3;    // Alarm 3.
                // Only one of the three alarms can be read.
                // If more than one is read, the highest
                // priority alarm alone will be actually read
                // and, others are ignored.
                // Alarm 1 is the highest priority and Alarm 3
                // is the lowest.

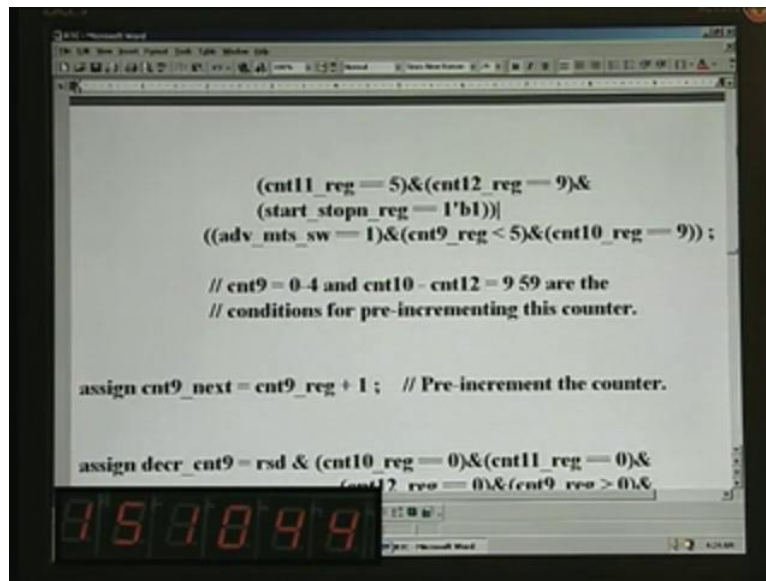
output [7:0]  display1 ;
output [7:0]  display2 ;
```

The screenshot shows a code editor window with the above Verilog code. Below the code editor, a 7-segment display is visible, showing the number '150934' in red LEDs.

This is right at the input declaration. When you come to **alarm3**, the comments are like this. There was a mistake is not in the code but only in the comment I have written. I will read out the correct version of the comment. Only one of the three alarms can be read. I had put it as set earlier. That is what you mean, right? Set is not right, it can only be read. You can set all the alarms if you wish, but while reading, you can only read one because there is only one display available. If you happen to set all the three alarms, naturally you cannot read. If you do not read any of them, then that is also not right.

What we gave is priority to these three alarms. We gave **alarm1** the top priority and then we are enabling that to be read. If more than one is read, the highest priority alarm alone will be actually read and others are ignored. This is the only change and all other things are the same. You remember that we stopped at implementation of the stopwatch. We will go to that point and then continue from where we left.

(Refer Slide Time: 03:16)



```
(cnt11_reg == 5)&(cnt12_reg == 9)&
(start_stoptn_reg == 1'b1))
((adv_mts_sw == 1)&(cnt9_reg < 5)&(cnt10_reg == 9));

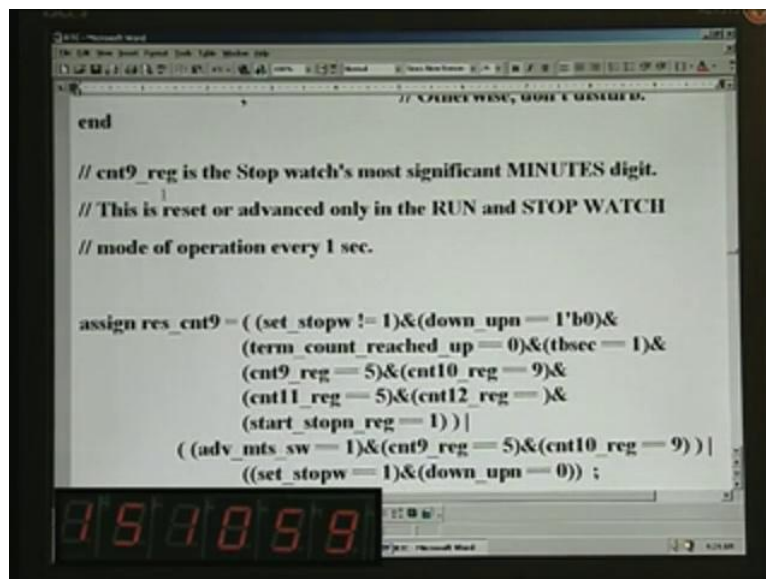
// cnt9 = 0-4 and cnt10 = cnt12 = 9 59 are the
// conditions for pre-incrementing this counter.

assign cnt9_next = cnt9_reg + 1; // Pre-increment the counter.

assign decr_cnt9 = rsd & (cnt10_reg == 0)&(cnt11_reg == 0)&
(cnt12_reg == 0)&(cnt9_reg > 0);
```

I think this is the one here.

(Refer Slide Time: 03:30)



```
end

// CASEWISE, DON'T CHANGE.

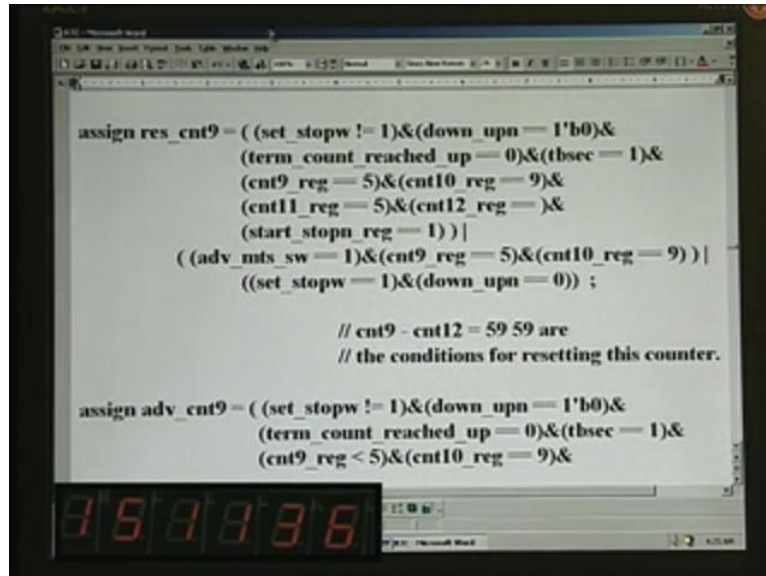
// cnt9_reg is the Stop watch's most significant MINUTES digit.
// This is reset or advanced only in the RUN and STOP WATCH
// mode of operation every 1 sec.

assign res_cnt9 = ( (set_stopw != 1)&(down_upn == 1'b0)&
(term_count_reached_up == 0)&(tbsec == 1)&
(cnt9_reg == 5)&(cnt10_reg == 9)&
(cnt11_reg == 5)&(cnt12_reg == )&
(start_stoptn_reg == 1)) |
( (adv_mts_sw == 1)&(cnt9_reg == 5)&(cnt10_reg == 9)) |
((set_stopw == 1)&(down_upn == 0)) );
```

We were looking at **cnt9**. Is that right? This is where we stopped. Here, it is precisely the same. **cnt9** is for the stopwatch's most significant minutes digit. What we have covered so far 7 and 8 for stopwatch are hours display. For minutes, it is precisely the same, except that the conditions are going to be different for each of these counters. For example, this is reset or advanced only in the RUN and STOP WATCH mode of operation every 1 second. This is precisely the same for other counter modes. When you want to reset the counter9, this is the

logic that we will have to solve. As usual, we will see only the comments because there is no point in repeating what you already know.

(Refer Slide Time: 04:09)



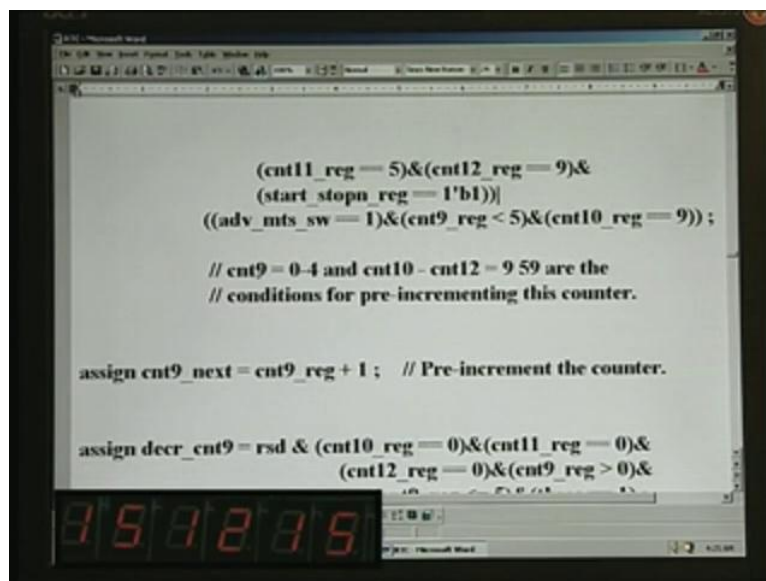
```
assign res_cnt9 = ((set_stopw != 1) & (down_upn == 1'b0) &
    (term_count_reached_up == 0) & (tbsec == 1) &
    (cnt9_reg == 5) & (cnt10_reg == 9) &
    (cnt11_reg == 5) & (cnt12_reg == 9) &
    (start_stopn_reg == 1)) |
    ((adv_mts_sw == 1) & (cnt9_reg == 5) & (cnt10_reg == 9)) |
    ((set_stopw == 1) & (down_upn == 0));

// cnt9 - cnt12 = 59 59 are
// the conditions for resetting this counter.

assign adv_cnt9 = ((set_stopw != 1) & (down_upn == 1'b0) &
    (term_count_reached_up == 0) & (tbsec == 1) &
    (cnt9_reg < 5) & (cnt10_reg == 9) &
```

Only thing is you have only to crosscheck – since you have this with you, you can always crosscheck at any point of time if you happen to have the CD. cnt9 to cnt12 must be 59, 59. This is the condition for resetting this counter. We are talking of resetting the minutes counter – that is the third counter from the left, 7, 8 and this is 9. How to reset? What condition determine the resetting will be precisely as per this comment.

(Refer Slide Time: 04:42)



```
(cnt11_reg == 5) & (cnt12_reg == 9) &
    (start_stopn_reg == 1'b1)) |
    ((adv_mts_sw == 1) & (cnt9_reg < 5) & (cnt10_reg == 9));

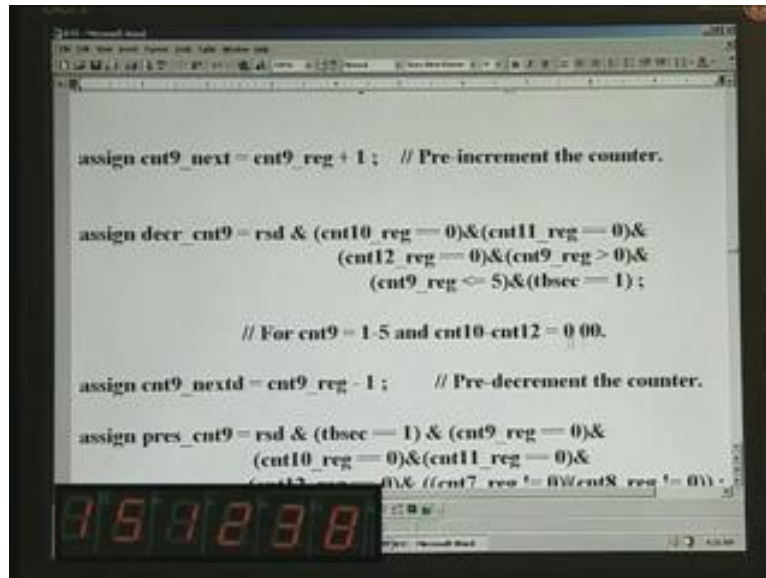
// cnt9 = 0-4 and cnt10 - cnt12 = 9 59 are the
// conditions for pre-incrementing this counter.

assign cnt9_next = cnt9_reg + 1; // Pre-increment the counter.

assign decr_cnt9 = rsd & (cnt10_reg == 0) & (cnt11_reg == 0) &
    (cnt12_reg == 0) & (cnt9_reg > 0) &
```

You can also advance **cnt9** for this particular status. For example, **cnt9** can be 0 to 4 and **cnt10 to cnt12** can be 9, 59. This is the condition when you can advance by 1. This is the pre-incrementing of counter9.

(Refer Slide Time: 05:02)



```
assign cnt9_next = cnt9_reg + 1; // Pre-increment the counter.

assign decr_cnt9 = rsd & (cnt10_reg == 0) & (cnt11_reg == 0) &
    (cnt12_reg == 0) & (cnt9_reg > 0) &
    (cnt9_reg <= 5) & (tbsec == 1);

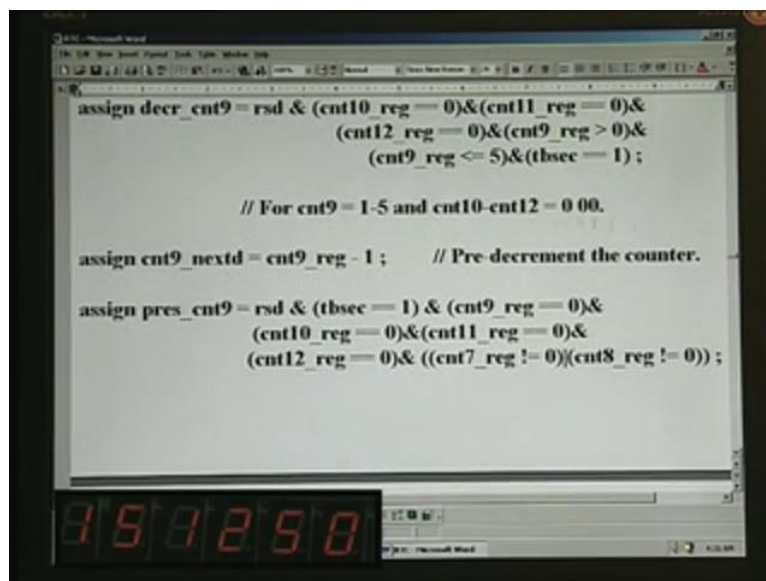
// For cnt9 = 1-5 and cnt10-cnt12 = 0 00.

assign cnt9_nextd = cnt9_reg - 1; // Pre-decrement the counter.

assign pres_cnt9 = rsd & (tbsec == 1) & (cnt9_reg == 0) &
    (cnt10_reg == 0) & (cnt11_reg == 0) &
    (cnt12_reg == 0) & ((cnt7_reg != 0) & (cnt8_reg != 0));
```

Once again, there is a decrement counter if you happen to be in the down-counting mode, for which the conditions are 1 to 5 for **cnt9** and **cnt10 to cnt12** must be all 0s. We are counting down. If it is 1 to 5, then only we need to decrement **cnt9**, which is precisely this. If it is 5, you will be decrementing it to 4.

(Refer Slide Time: 05:23)



```
assign decr_cnt9 = rsd & (cnt10_reg == 0) & (cnt11_reg == 0) &
    (cnt12_reg == 0) & (cnt9_reg > 0) &
    (cnt9_reg <= 5) & (tbsec == 1);

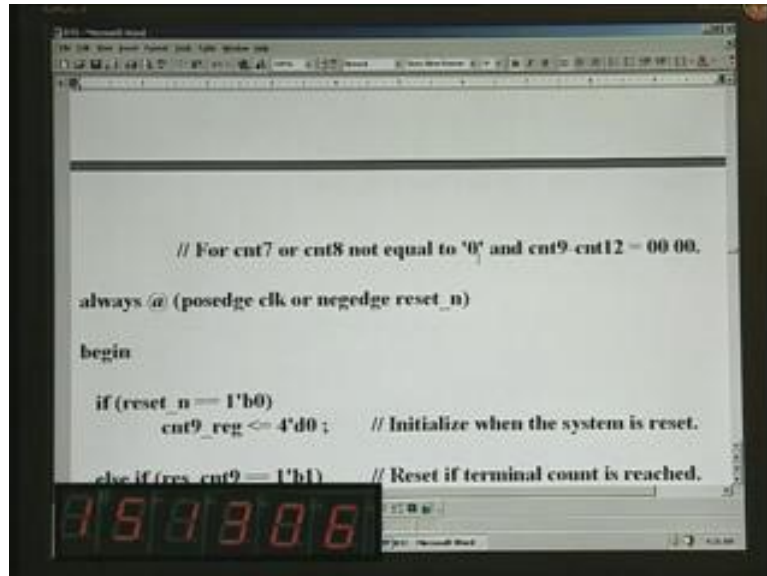
// For cnt9 = 1-5 and cnt10-cnt12 = 0 00.

assign cnt9_nextd = cnt9_reg - 1; // Pre-decrement the counter.

assign pres_cnt9 = rsd & (tbsec == 1) & (cnt9_reg == 0) &
    (cnt10_reg == 0) & (cnt11_reg == 0) &
    (cnt12_reg == 0) & ((cnt7_reg != 0) & (cnt8_reg != 0));
```

Note that the other digits must be 0s and pre-decrement once again and this is the condition for presetting **cnt9**.

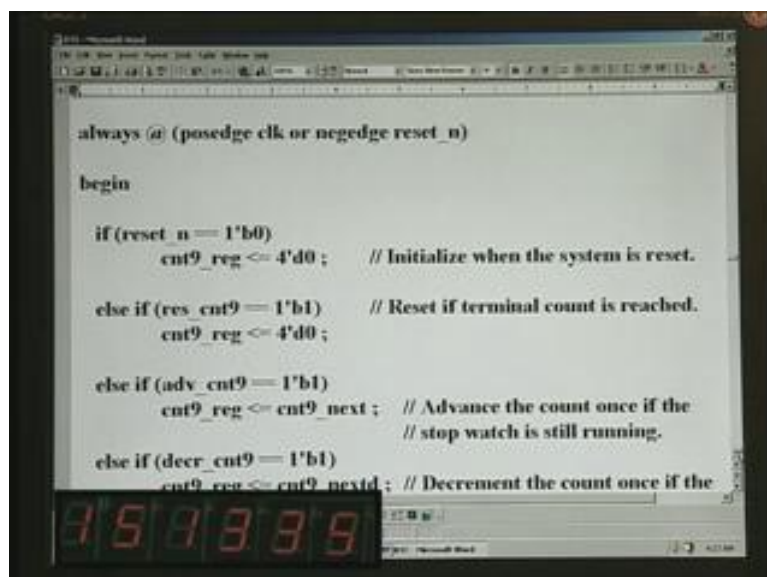
(Refer Slide Time: 05:34)



```
// For cnt7 or cnt8 not equal to '0' and cnt9 - cnt12 = 00 00.
always @(posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        cnt9_reg <= 4'd0; // Initialize when the system is reset.
    else if (res_cnt9 == 1'b1) // Reset if terminal count is reached.
        cnt9_reg <= 4'd0;
end
```

The condition is **cnt7 or cnt8** must not be equal to 0 and **cnt9** through **cnt12** must be 0s. This is clear. If all of them are 0, it has come to the terminal value and there is no further decrementing. In order to take care of that only we are putting it here that at least one of them must not be 0.

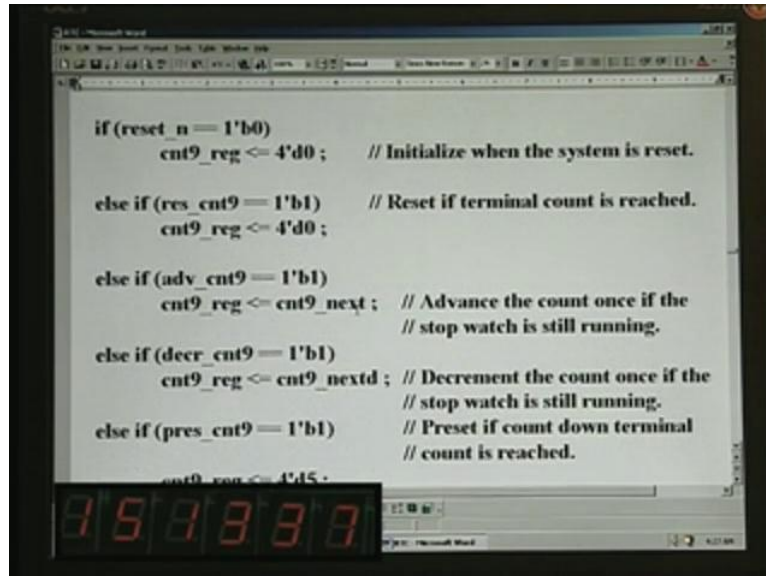
(Refer Slide Time: 06:01)



```
always @(posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        cnt9_reg <= 4'd0; // Initialize when the system is reset.
    else if (res_cnt9 == 1'b1) // Reset if terminal count is reached.
        cnt9_reg <= 4'd0;
    else if (adv_cnt9 == 1'b1) // Advance the count once if the
        cnt9_reg <= cnt9_next; // stop watch is still running.
    else if (decr_cnt9 == 1'b1) // Decrement the count once if the
        cnt9_reg <= cnt9_nextd;
end
```

As usual, **cnt9** implementation is here, reset condition, then advance and then decrement condition.

(Refer Slide Time: 06:10)



```
if (reset_n == 1'b0)
    cnt9_reg <= 4'd0; // Initialize when the system is reset.

else if (res_cnt9 == 1'b1) // Reset if terminal count is reached.
    cnt9_reg <= 4'd0;

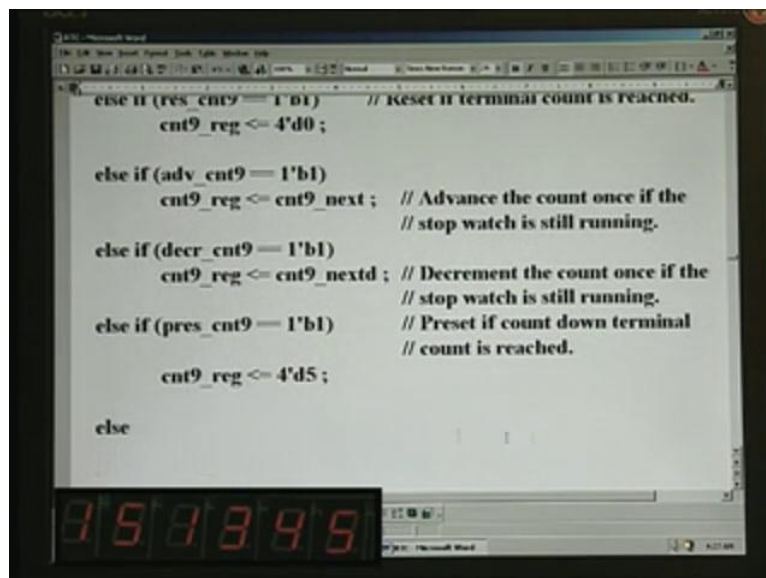
else if (adv_cnt9 == 1'b1)
    cnt9_reg <= cnt9_next; // Advance the count once if the
    // stop watch is still running.

else if (decr_cnt9 == 1'b1)
    cnt9_reg <= cnt9_nextd; // Decrement the count once if the
    // stop watch is still running.

else if (pres_cnt9 == 1'b1)
    cnt9_reg <= 4'd5; // Preset if count down terminal
    // count is reached.
```

When this condition takes place, only then it will do the advancing or decrementing or presetting.

(Refer Slide Time: 06:17)



```
else if (res_cnt9 == 1'b1) // Reset if terminal count is reached.
    cnt9_reg <= 4'd0;

else if (adv_cnt9 == 1'b1)
    cnt9_reg <= cnt9_next; // Advance the count once if the
    // stop watch is still running.

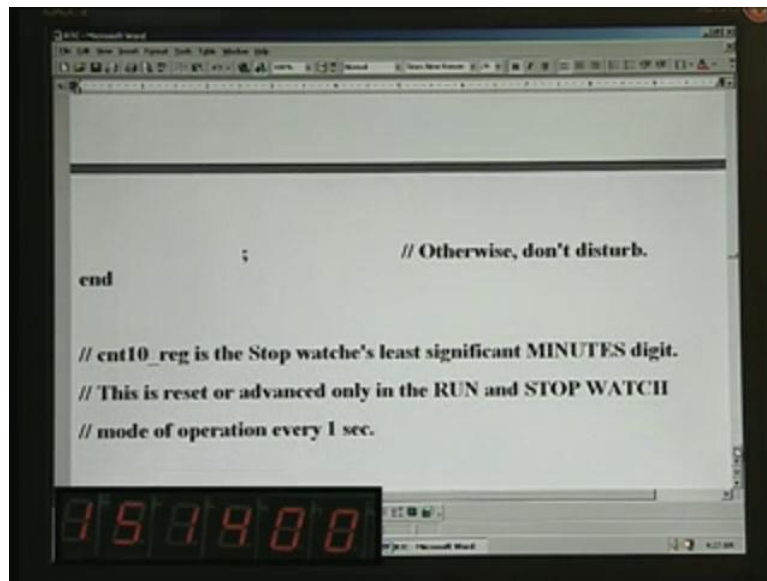
else if (decr_cnt9 == 1'b1)
    cnt9_reg <= cnt9_nextd; // Decrement the count once if the
    // stop watch is still running.

else if (pres_cnt9 == 1'b1)
    cnt9_reg <= 4'd5; // Preset if count down terminal
    // count is reached.

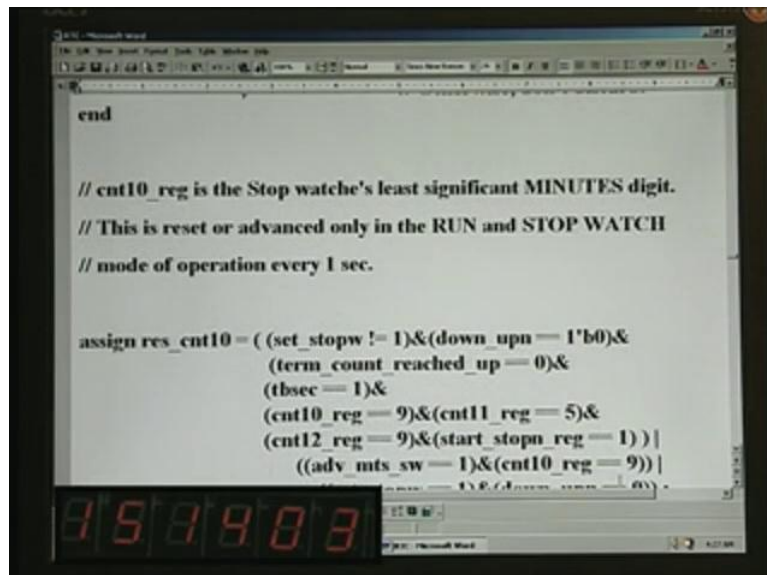
else
```

In this case, this is minutes, so you can go right up to 5. This is the preset value. After 0, it should roll back to 5 and that is why when the preset count is 1, it has been preset to 5.

(Refer Slide Time: 06:33)



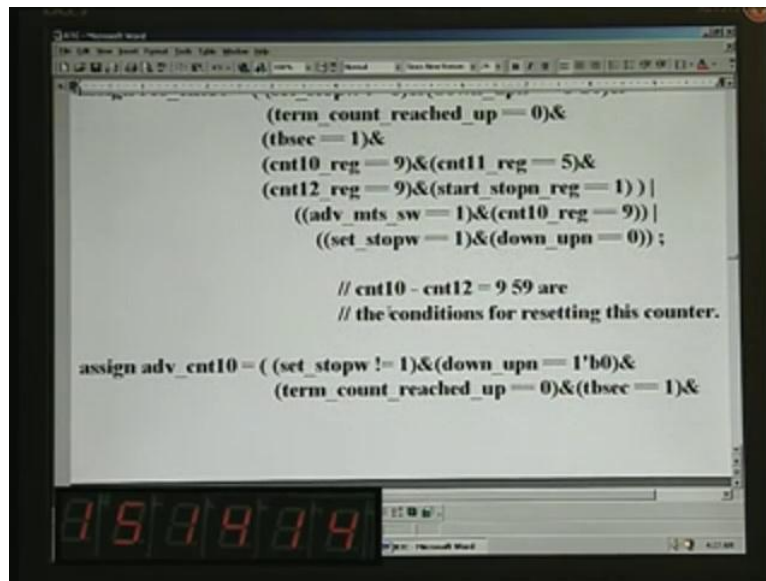
```
end ; // Otherwise, don't disturb.  
  
// cnt10_reg is the Stop watche's least significant MINUTES digit.  
// This is reset or advanced only in the RUN and STOP WATCH  
// mode of operation every 1 sec.
```



```
end  
  
// cnt10_reg is the Stop watche's least significant MINUTES digit.  
// This is reset or advanced only in the RUN and STOP WATCH  
// mode of operation every 1 sec.  
  
assign res_cnt10 = ((set_stopw != 1)&(down_upn == 1'b0)&  
    (term_count_reached_up == 0)&  
    (tbsec == 1)&  
    (cnt10_reg == 9)&(cnt11_reg == 5)&  
    (cnt12_reg == 9)&(start_stopn_reg == 1) ) |  
    ((adv_mts_sw == 1)&(cnt10_reg == 9)) |  
    (adv_mts_sw == 1)&(cnt10_reg == 0))
```

The next one is **cnt10**. It is the stopwatch's least significant minutes digit. This is advanced or reset only in the RUN and STOP WATCH mode every 1 second. This is once again the same thing.

(Refer Slide Time: 06:47)



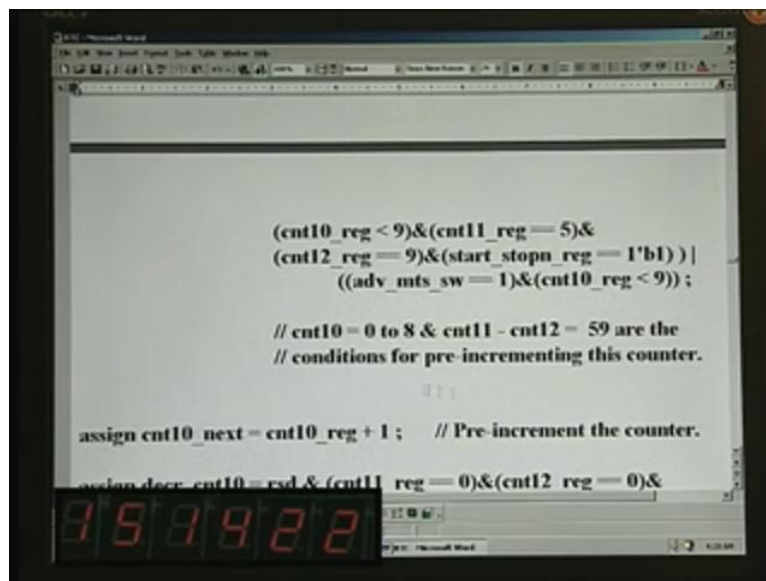
```
(term_count_reached_up == 0)&
(thsec == 1)&
(cnt10_reg == 9)&(cnt11_reg == 5)&
(cnt12_reg == 9)&(start_stopn_reg == 1) |
((adv_mts_sw == 1)&(cnt10_reg == 9)) |
((set_stopw == 1)&(down_upn == 0));

// cnt10 - cnt12 = 9 59 are
// the conditions for resetting this counter.

assign adv_cnt10 = (set_stopw != 1)&(down_upn == 1'b0)&
(term_count_reached_up == 0)&(thsec == 1)&
```

cnt10 to cnt12 must be 9, 59 – condition for reset.

(Refer Slide Time: 06:55)



```
(cnt10_reg < 9)&(cnt11_reg == 5)&
(cnt12_reg == 9)&(start_stopn_reg == 1'b1) |
((adv_mts_sw == 1)&(cnt10_reg < 9));

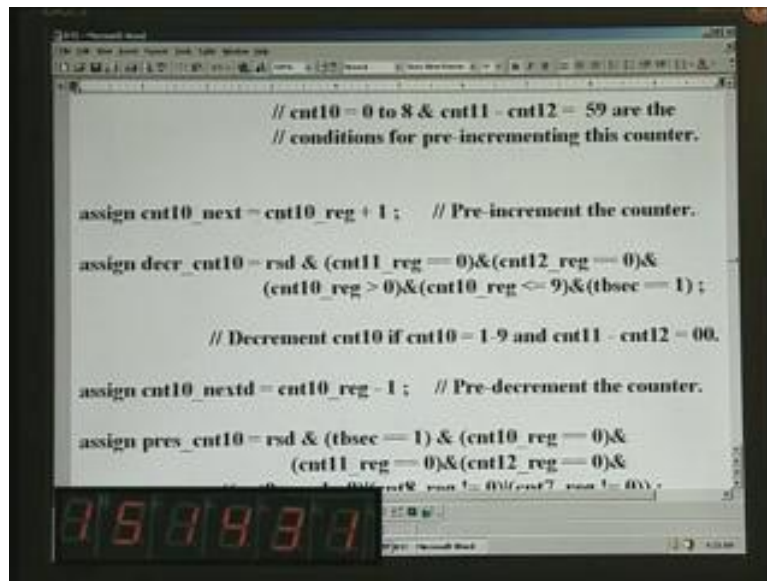
// cnt10 = 0 to 8 & cnt11 - cnt12 = 59 are the
// conditions for pre-incrementing this counter.

// ;

assign cnt10_next = cnt10_reg + 1; // Pre-increment the counter.
assign decr_cnt10 = rsd & (cnt11_reg == 0)&(cnt12_reg == 0)&
```

The condition for advancing cnt10 is 0 to 8 and cnt11 and cnt12 must be 59.

(Refer Slide Time: 07:03)



```
// cnt10 = 0 to 8 & cnt11 - cnt12 = 59 are the
// conditions for pre-incrementing this counter.

assign cnt10_next = cnt10_reg + 1; // Pre-increment the counter.

assign decr_cnt10 = rsd & (cnt11_reg == 0) & (cnt12_reg == 0) &
(cnt10_reg > 0) & (cnt10_reg <= 9) & (tbsec == 1);

// Decrement cnt10 if cnt10 = 1-9 and cnt11 - cnt12 = 00.

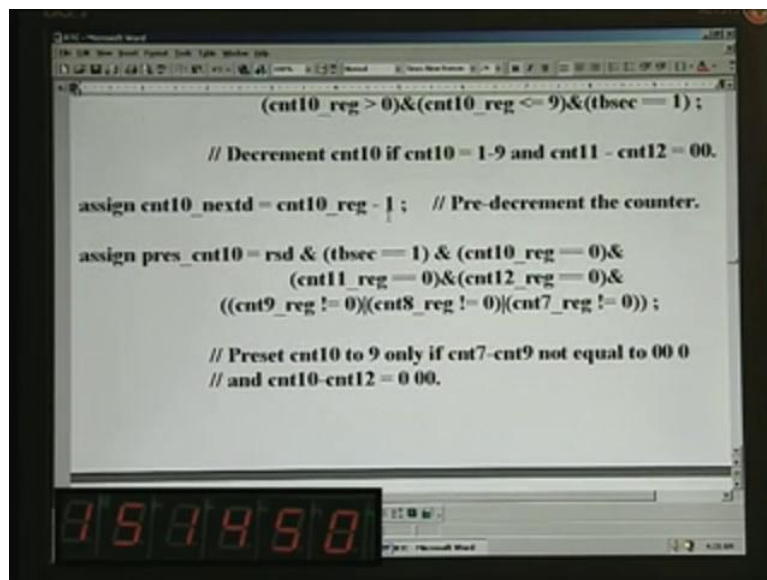
assign cnt10_nextd = cnt10_reg - 1; // Pre-decrement the counter.

assign pres_cnt10 = rsd & (tbsec == 1) & (cnt10_reg == 0) &
(cnt11_reg == 0) & (cnt12_reg == 0) &
((cnt9_reg != 0) & (cnt8_reg != 0) & (cnt7_reg != 0));

// Preset cnt10 to 9 only if cnt7-cnt9 not equal to 00 0
// and cnt10-cnt12 = 0 00.
```

Once again, pre-increment is there and pre-decrement is there. Decrement once again applies only for the run stopwatch down-count mode. The condition is for 10 equal to 1 to 9 and 11 and 12 must be 0. This is the decrement counter.

(Refer Slide Time: 07:23)



```
(cnt10_reg > 0) & (cnt10_reg <= 9) & (tbsec == 1);

// Decrement cnt10 if cnt10 = 1-9 and cnt11 - cnt12 = 00.

assign cnt10_nextd = cnt10_reg - 1; // Pre-decrement the counter.

assign pres_cnt10 = rsd & (tbsec == 1) & (cnt10_reg == 0) &
(cnt11_reg == 0) & (cnt12_reg == 0) &
((cnt9_reg != 0) & (cnt8_reg != 0) & (cnt7_reg != 0));

// Preset cnt10 to 9 only if cnt7-cnt9 not equal to 00 0
// and cnt10-cnt12 = 0 00.
```

Pre-decrement again and the condition for presetting the counter is here. cnt10 to cnt9 must be preset to 9 only if cnt7 through cnt9 is not equal to 00 0. Once again, I hope this is clear. Also, cnt10 to cnt12 must be 0s – all of them 0 except for this condition. This is for presetting the counter.


(Refer Slide Time: 07:55)

```
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        cnt10_reg <= 4'd0; // Initialize when the system is reset.

    else if (res_cnt10 == 1'b1) // Reset if terminal count is reached.
        cnt10_reg <= 4'd0;

    else if (adv_cnt10 == 1'b1)
        cnt10_reg <= cnt10_next; // Advance the count once if the
        // stop watch is still running.

    else if (decr_cnt10 == 1'b1)
        cnt10_reg <= cnt10_nextd; // Decrement the count once if
        // the stop watch is still running.
end
```



Realization of cnt10 is here. As usual, reset, then advance, decrement and preset.

(Refer Slide Time: 08:02)


```
cnt10_reg <= 4'd0;

else if (adv_cnt10 == 1'b1)
    cnt10_reg <= cnt10_next; // Advance the count once if the
    // stop watch is still running.

else if (decr_cnt10 == 1'b1)
    cnt10_reg <= cnt10_nextd; // Decrement the count once if
    // the stop watch is still running.

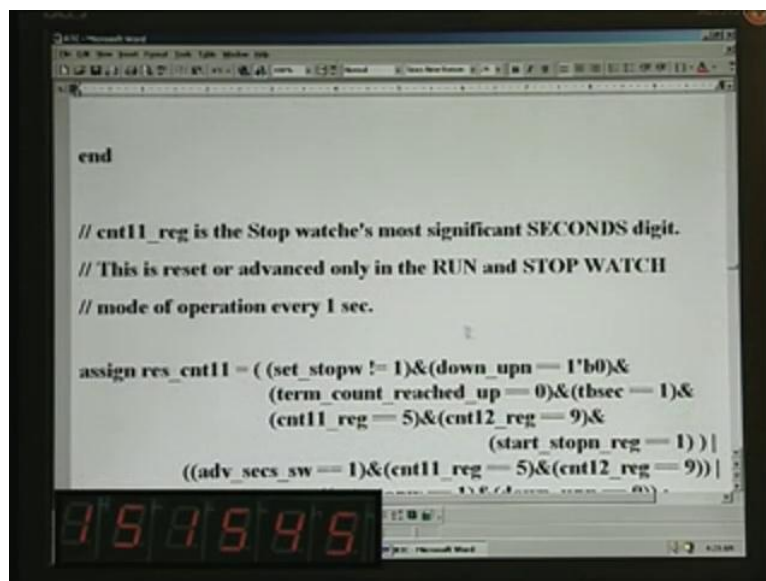
else if (pres_cnt10 == 1'b1) // Preset if count down terminal
    // count is reached.
    cnt10_reg <= 4'd9;

else
    ; // Otherwise, don't disturb.
```



Preset in this case is when it is 9. What we are doing is for 10. 59 is the maximum and 9 corresponds to cnt10. We are concerned with only the minutes LSD.

(Refer Slide Time: 08:18)



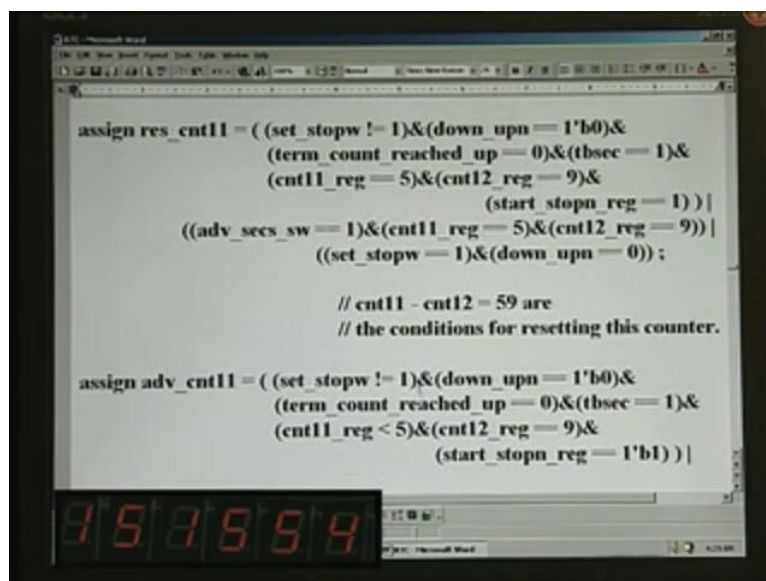
```
end

// cnt11_reg is the Stop watche's most significant SECONDS digit.
// This is reset or advanced only in the RUN and STOP WATCH
// mode of operation every 1 sec.

assign res_cnt11 = ((set_stopw != 1)&(down_upn == 1'b0)&
    (term_count_reached_up == 0)&(tbsec == 1)&
    (cnt11_reg == 5)&(cnt12_reg == 9)&
    (start_stopn_reg == 1)) |
    ((adv_secs_sw == 1)&(cnt11_reg == 5)&(cnt12_reg == 9)) |
```

11 is the stopwatch's most significant seconds digit and the other conditions are the same.

(Refer Slide Time: 08:23)



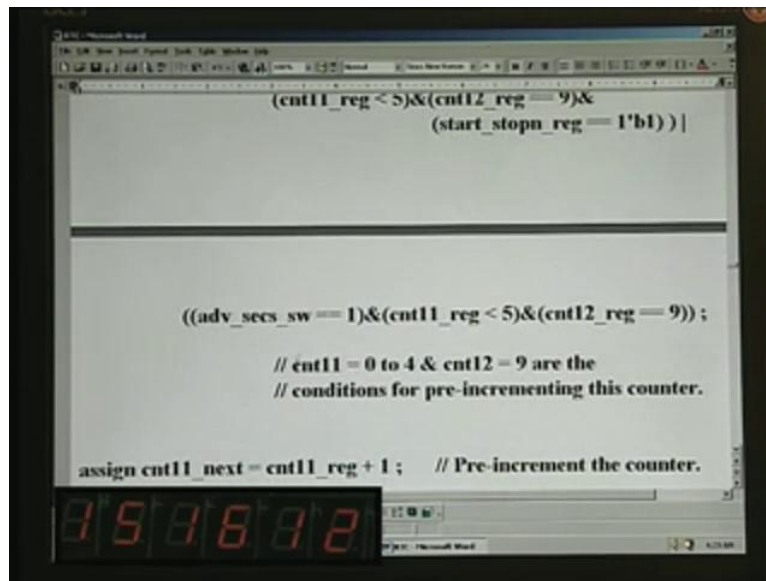
```
assign res_cnt11 = ((set_stopw != 1)&(down_upn == 1'b0)&
    (term_count_reached_up == 0)&(tbsec == 1)&
    (cnt11_reg == 5)&(cnt12_reg == 9)&
    (start_stopn_reg == 1)) |
    ((adv_secs_sw == 1)&(cnt11_reg == 5)&(cnt12_reg == 9)) |
    ((set_stopw == 1)&(down_upn == 0));

// cnt11 - cnt12 = 59 are
// the conditions for resetting this counter.

assign adv_cnt11 = ((set_stopw != 1)&(down_upn == 1'b0)&
    (term_count_reached_up == 0)&(tbsec == 1)&
    (cnt11_reg < 5)&(cnt12_reg == 9)&
    (start_stopn_reg == 1'b1)) |
```

The condition for resetting is once again 11 and 12. You can see that the logic is progressively getting easier. It is 59 as far as the 11 and 12 counters are concerned. The same is the case for the advance counter except for a slight difference.

(Refer Slide Time: 08:45)

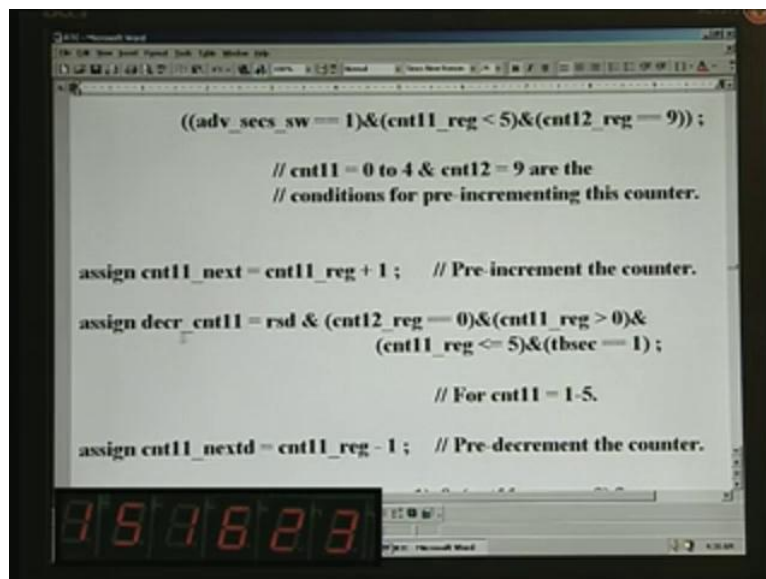


```
(cnt11_reg < 5) & (cnt12_reg == 9) &
(start_stpn_reg == 1'b1) |

((adv_secs_sw == 1) & (cnt11_reg < 5) & (cnt12_reg == 9));
// cnt11 = 0 to 4 & cnt12 = 9 are the
// conditions for pre-incrementing this counter.
assign cnt11_next = cnt11_reg + 1; // Pre-increment the counter.
```

That is, **cnt11** must be 0 to 4 and **cnt12** must be 9. This is the precondition.

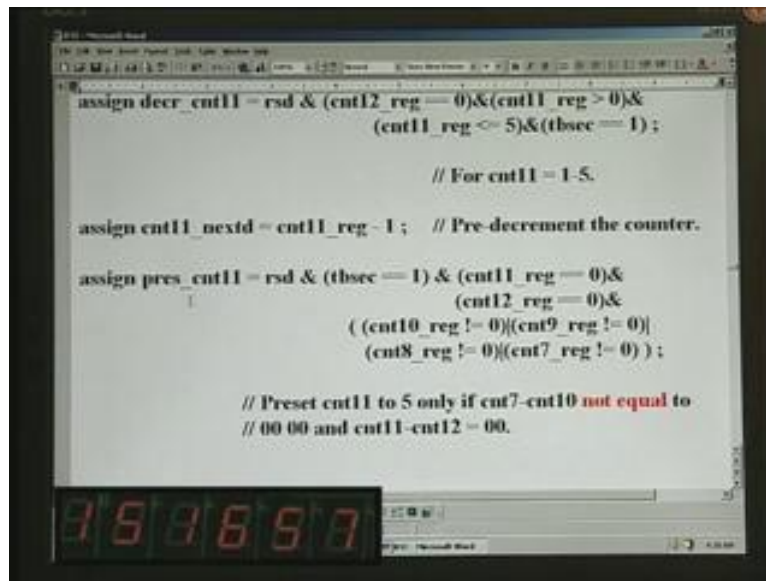
(Refer Slide Time: 08:56)



```
((adv_secs_sw == 1) & (cnt11_reg < 5) & (cnt12_reg == 9));
// cnt11 = 0 to 4 & cnt12 = 9 are the
// conditions for pre-incrementing this counter.
assign cnt11_next = cnt11_reg + 1; // Pre-increment the counter.
assign decr_cnt11 = rsd & (cnt12_reg == 0) & (cnt11_reg > 0) &
(cnt11_reg <= 5) & (tbsec == 1);
// For cnt11 = 1-5.
assign cnt11_nextd = cnt11_reg - 1; // Pre-decrement the counter.
```

Advance increment is here, then decrement **cnt11**. This is the condition for decrementing. As usual, the **rsd signal** has been used here and **cnt11** is from 1 to 5. This is for **seconds MSD**. The maximum that you can go is 5 here and this is the condition for decrement. If it is 1, 1 can go to 0 but if it is 0, it cannot go down further. You can decrement only with the relevant digit. That is the implication that we have been looking into all along.

(Refer Slide Time: 09:11)



```
assign decr_cnt11 = rsd & (cnt12_reg == 0) & (cnt11_reg > 0) &
                    (cnt11_reg <= 5) & (tbsec == 1);

                    // For cnt11 = 1-5.

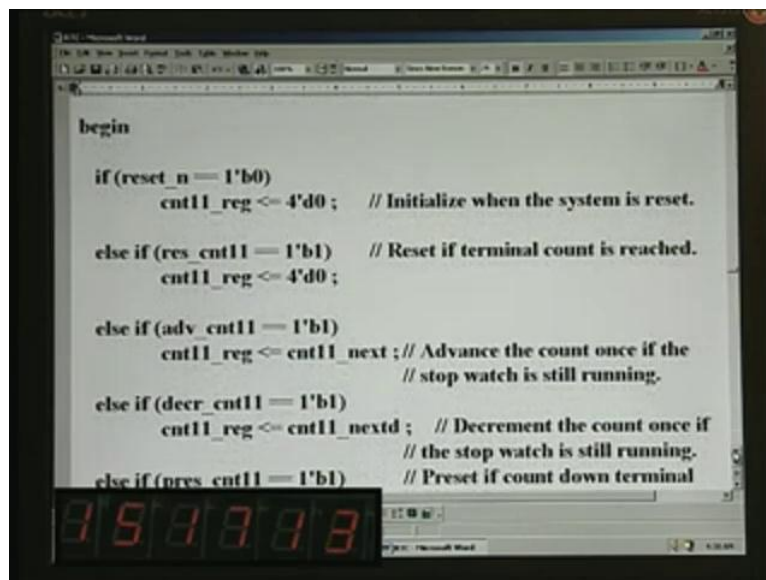
assign cnt11_nextd = cnt11_reg - 1; // Pre-decrement the counter.

assign pres_cnt11 = rsd & (tbsec == 1) & (cnt11_reg == 0) &
                    (cnt12_reg == 0) &
                    ((cnt10_reg != 0) | (cnt9_reg != 0) |
                    (cnt8_reg != 0) | (cnt7_reg != 0));

                    // Preset cnt11 to 5 only if cnt7-cnt10 not equal to
                    // 00 00 and cnt11-cnt12 = 00.
```

Once again, the pre-decrement, then presetting **cnt11**. You preset the counter to 5 only if **cnt7** and **cnt10** are not equal to all 0s. Of course, 11 and 12 must be 0 for the same condition that we have been seeing.

(Refer Slide Time: 09:46)



```
begin

if (reset_n == 1'b0)
    cnt11_reg <= 4'd0; // Initialize when the system is reset.

else if (res_cnt11 == 1'b1) // Reset if terminal count is reached.
    cnt11_reg <= 4'd0;

else if (adv_cnt11 == 1'b1)
    cnt11_reg <= cnt11_next; // Advance the count once if the
                            // stop watch is still running.

else if (decr_cnt11 == 1'b1)
    cnt11_reg <= cnt11_nextd; // Decrement the count once if
                            // the stop watch is still running.

else if (pres_cnt11 == 1'b1) // Preset if count down terminal
```

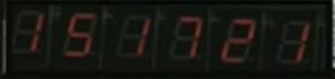
Again, reset, advance, decrement counter and preset **cnt11** of course.

(Refer Slide Time: 09:54)

```
cnt11_reg <= 4'd0 ;

else if (adv_cnt11 == 1'b1)
    cnt11_reg <= cnt11_next ; // Advance the count once if the
                               // stop watch is still running.
else if (decr_cnt11 == 1'b1)
    cnt11_reg <= cnt11_nextd ; // Decrement the count once if
                               // the stop watch is still running.
else if (pres_cnt11 == 1'b1) // Preset if count down terminal
                               // count is reached.
    cnt11_reg <= 4'd5 ;

else
    ; // Otherwise, don't disturb.
end
```

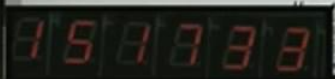


For 11, the maximum number is 5 and so we roll back to 5. Prior to this, it was 0 in the decrement mode.

(Refer Slide Time: 10:04)

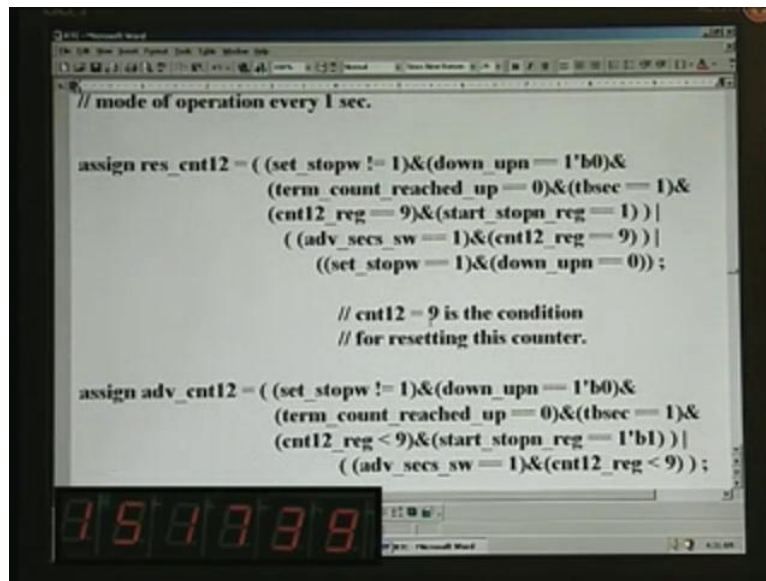
```
// cnt12_reg is the Stop watche's least significant SECONDS digit.
// This is reset or advanced only in the RUN and STOP WATCH
// mode of operation every 1 sec.

assign res_cnt12 = ((set_stopw != 1) & (down_upn == 1'b0) &
    (term_count_reached_up == 0) & (tbsec == 1) &
    (cnt12_reg == 9) & (start_stopn_reg == 1)) |
    ((adv_secs_sw == 1) & (cnt12_reg == 9)) |
    ((set_stopw == 1) & (down_upn == 0));
```



Now the last digit, seconds digit LSD is here.

(Refer Slide Time: 10:11)



```
// mode of operation every 1 sec.

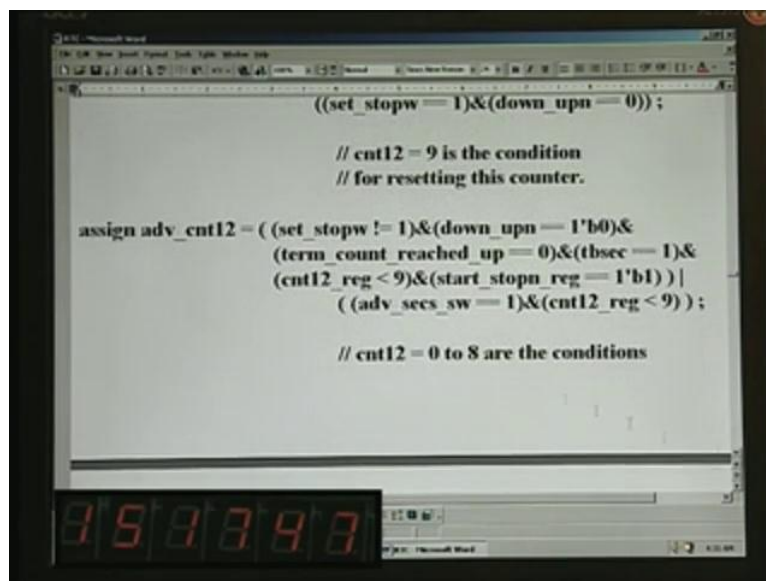
assign res_cnt12 = ( (set_stopw != 1) & (down_upn == 1'b0) &
                    (term_count_reached_up == 0) & (tbsec == 1) &
                    (cnt12_reg == 9) & (start_stopn_reg == 1) ) |
                  ( (adv_secs_sw == 1) & (cnt12_reg == 9) ) |
                  ((set_stopw == 1) & (down_upn == 0));

// cnt12 = 9 is the condition
// for resetting this counter.

assign adv_cnt12 = ( (set_stopw != 1) & (down_upn == 1'b0) &
                    (term_count_reached_up == 0) & (tbsec == 1) &
                    (cnt12_reg < 9) & (start_stopn_reg == 1'b1) ) |
                  ( (adv_secs_sw == 1) & (cnt12_reg < 9) );
```

Once again, the reset condition is **cnt12** must be 9 – it is the condition for resetting this counter and **cnt12 is itself**.

(Refer Slide Time: 10:19)



```
((set_stopw == 1) & (down_upn == 0));

// cnt12 = 9 is the condition
// for resetting this counter.

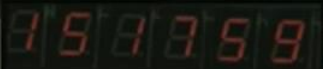
assign adv_cnt12 = ( (set_stopw != 1) & (down_upn == 1'b0) &
                    (term_count_reached_up == 0) & (tbsec == 1) &
                    (cnt12_reg < 9) & (start_stopn_reg == 1'b1) ) |
                  ( (adv_secs_sw == 1) & (cnt12_reg < 9) );

// cnt12 = 0 to 8 are the conditions
```

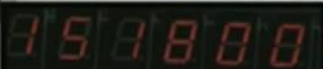
Advancing the counter is for **cnt12** equal to 0 to 8 – this is for advance. If it is 0, it can become 1 and if it is 8, it can become 9. Therefore, the valid number is 0 to 8.

(Refer Slide Time: 10:31)

```
// for pre-incrementing this counter.
assign cnt12_next = cnt12_reg + 1; // Pre-increment the counter.
assign decr_cnt12 = rsd & (cnt12_reg > 0) & (cnt12_reg <= 9) &
                    (tbsec == 1);
// Decrement cnt12 every sec. if cnt12 = 1-9.
assign cnt12_nextd = cnt12_reg - 1; // Pre-decrement the counter.
assign pres_cnt12 = rsd & (tbsec == 1) & (cnt12_reg == 0) &
                    ((cnt11_reg != 0) | (cnt10_reg != 0) | (cnt9_reg != 0) |
                     (cnt8_reg != 0) | (cnt7_reg != 0));
```



```
assign cnt12_next = cnt12_reg + 1; // Pre-increment the counter.
assign decr_cnt12 = rsd & (cnt12_reg > 0) & (cnt12_reg <= 9) &
                    (tbsec == 1);
// Decrement cnt12 every sec. if cnt12 = 1-9.
assign cnt12_nextd = cnt12_reg - 1; // Pre-decrement the counter.
assign pres_cnt12 = rsd & (tbsec == 1) & (cnt12_reg == 0) &
                    ((cnt11_reg != 0) | (cnt10_reg != 0) | (cnt9_reg != 0) |
                     (cnt8_reg != 0) | (cnt7_reg != 0));
// Preset cnt12 to 9 only if cnt7-cnt11 not equal to
// 00 00 0 and cnt12 = 0.
```




Advance, pre-increment, then decrement in the down-count mode in which case **cnt12** is decremented every second only if **cnt12** is 1 to 9. So 1 can become 0, **9 can become 1** and so on.

(Refer Slide Time: 10:49)

```
// Decrement cnt12 every sec. if cnt12 = 1-9.
assign cnt12_nextd = cnt12_reg - 1; // Pre-decrement the counter.
assign pres_cnt12 = rstd & (tbsec == 1) & (cnt12_reg == 0) &
((cnt11_reg != 0) & (cnt10_reg != 0) & (cnt9_reg != 0) &
(cnt8_reg != 0) & (cnt7_reg != 0));

// Preset cnt12 to 9 only if cnt7-cnt11 not equal to
// 00 00 0 and cnt12 = 0.

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




Once again, pre-decrement and preset the counter. This happens only if **cnt7 to cnt11** are not equal to all 0s; **cnt12** also must be equal to 0 and it is preset to 9 in this case.

(Refer Slide Time: 11:09)

```
if (reset_n == 1'b0)

cnt12_reg <= 4'd0; // Initialize when the system is reset.
else if (res_cnt12 == 1'b1) // Reset if count up terminal count
// is reached.
cnt12_reg <= 4'd0;
else if (adv_cnt12 == 1'b1)
cnt12_reg <= cnt12_next; // Advance the count once if the
// stop watch is still running
```



```

cnt12_reg <= 4'd0;

else if (adv_cnt12 == 1'b1)
    cnt12_reg <= cnt12_next; // Advance the count once if the
                             // stop watch is still running.


else if (decr_cnt12 == 1'b1)
    cnt12_reg <= cnt12_nextd; // Decrement the count once if
                              // the stop watch is still running.

else if (pres_cnt12 == 1'b1) // Preset if count down terminal
                              // count is reached.
    cnt12_reg <= 4'd9;

else
    ; // Otherwise, don't disturb.

end

```



The block for `cnt12` is this. The preset is that 9 will be reflected here. All the other things are similar.

(Refer Slide Time: 11:20)

```

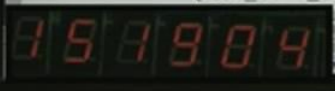
end

assign res_term_count_reg1 = (adv_hrs_tcr == 1) &
                             (term_count_reg1 == 2) & (term_count_reg2 == 3);

// Reset Terminal count register for Up counter.

assign adv_term_count_reg1 = (adv_hrs_tcr == 1) &
                             (term_count_reg1 < 2) & (term_count_reg2 == 9);

```



Next, what we have to see is the terminal count register for the up counter. In the up counter mode, what we have to look for is you are going to set a particular value and it counts right from 0, 1, 2, 3 and so on. When the terminal count (that is what you have set) is reached, a match is found. That is what we mean by terminal count here. For the up counter, it is one thing and for down counter, it is another. For that, you need this signal also, which implies reset `term_count_reg1`. This must take place only when `adv_hrs_tcr` is equal to 1 and when `term_count_reg1` is equal to 2 and `term_count_reg2` is equal to 3 (tcr is terminal count

register). Remember that `reg1` through `reg6` are there, so this is for up count, is it not? We are looking for terminal count equal to 2, 3. Only then, you need to energize this signal. It is a reset terminal count `reg1`.

(Refer Slide Time: 12:37)

```


// Reset Terminal count register for Up counter.
assign adv_term_count_reg1 = (adv_hrs_tcr == 1) &
    (term_count_reg1 < 2) & (term_count_reg2 == 9);

assign term_count_reg1_next = term_count_reg1 + 1;

always @ (posedge clk or negedge reset_n)

begin
    if (reset_n == 0)
        term_count_reg1 <= 0;

```



```

// Reset Terminal count register for Up counter.

assign adv_term_count_reg1 = (adv_hrs_tcr == 1) &
    (term_count_reg1 < 2) & (term_count_reg2 == 9);

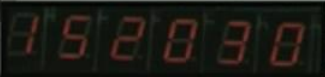
assign term_count_reg1_next = term_count_reg1 + 1;

always @ (posedge clk or negedge reset_n)

begin
    if (reset_n == 0)
        term_count_reg1 <= 0;

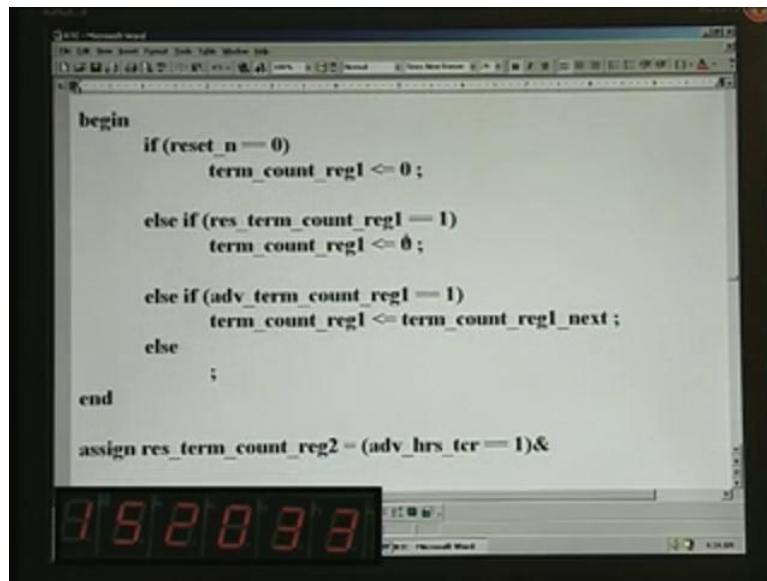
    else if (res_term_count_reg1 == 1)
        term_count_reg1 <= 0;

```



Next is advance terminal. The same `reg1` you need to advance also. For that, exactly the same condition there. This terminal count `reg` will be less than 2 and equal to 9. That means 09 or 19 is implied there and it is similar to your `cnt1` or `cnt7` that we have already seen before. Once again, you need a pre-increment here and this terminal count `reg1` is realized as we have done before.

(Refer Slide Time: 13:05)



```
begin
  if(reset_n == 0)
    term_count_reg1 <= 0;

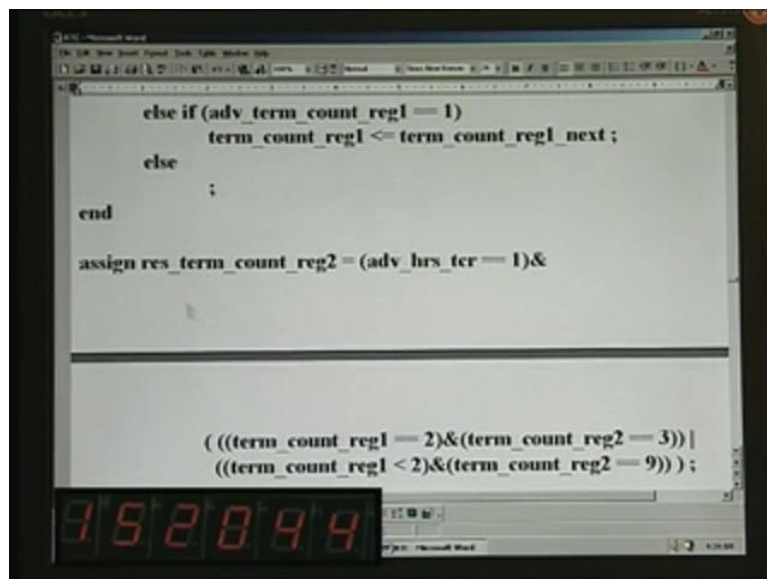
  else if(res_term_count_reg1 == 1)
    term_count_reg1 <= 0;

  else if(adv_term_count_reg1 == 1)
    term_count_reg1 <= term_count_reg1_next;
  else
    ;
end

assign res_term_count_reg2 = (adv_hrs_ter == 1) &
```

You need to reset or advance it and there is no decrement as such here.

(Refer Slide Time: 13:16)



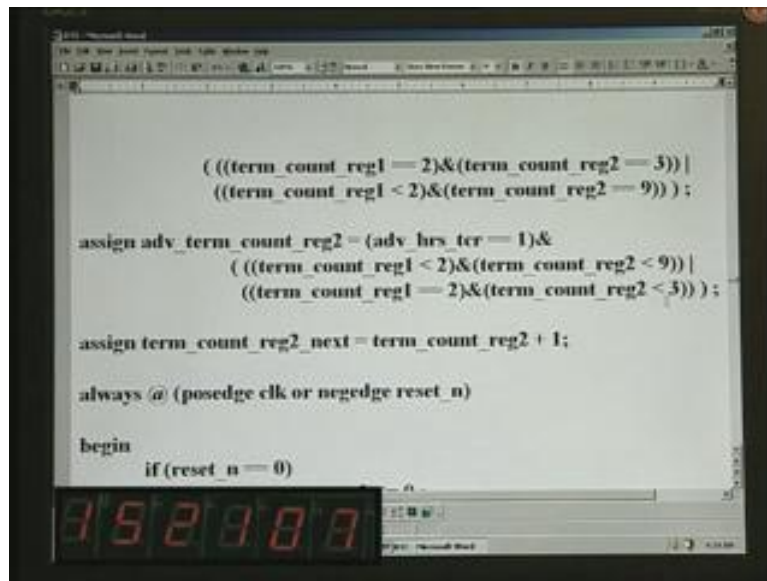
```
else if(adv_term_count_reg1 == 1)
  term_count_reg1 <= term_count_reg1_next;
else
  ;
end

assign res_term_count_reg2 = (adv_hrs_ter == 1) &

(((term_count_reg1 == 2) & (term_count_reg2 == 3)) |
((term_count_reg1 < 2) & (term_count_reg2 == 9)));
```

Similarly, for reset `term_count_reg2`, it is precisely same as what we have seen for `reg1`.

(Refer Slide Time: 13:22)



```
(((term_count_reg1 == 2)&(term_count_reg2 == 3)) |
((term_count_reg1 < 2)&(term_count_reg2 == 9)));

assign adv_term_count_reg2 = (adv_hrs_ter == 1) &
((term_count_reg1 < 2) & (term_count_reg2 < 9)) |
((term_count_reg1 == 2) & (term_count_reg2 < 3));

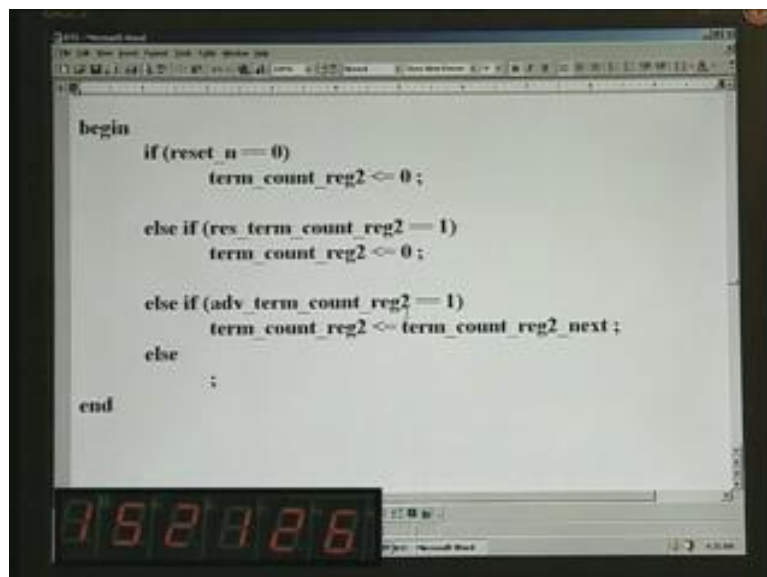
assign term_count_reg2_next = term_count_reg2 + 1;

always @ (posedge clk or negedge reset_n)

begin
    if (reset_n == 0)
```

I think I have not even put the comment here. Advance counter is also here and you can very easily reason out what they are. Less than 2 means 0 or 1, less than 9 means 8. You can easily find out what they are. Terminal count equal to 2 and here, it is less than 3. Only for that, you need to advance. Mind you, we are just trying to advance the second register, which corresponds to the hours LSD. Pre-incrementing is there as usual for cnt2.

(Refer Slide Time: 13:57)



```
begin
    if (reset_n == 0)
        term_count_reg2 <= 0;

    else if (res_term_count_reg2 == 1)
        term_count_reg2 <= 0;

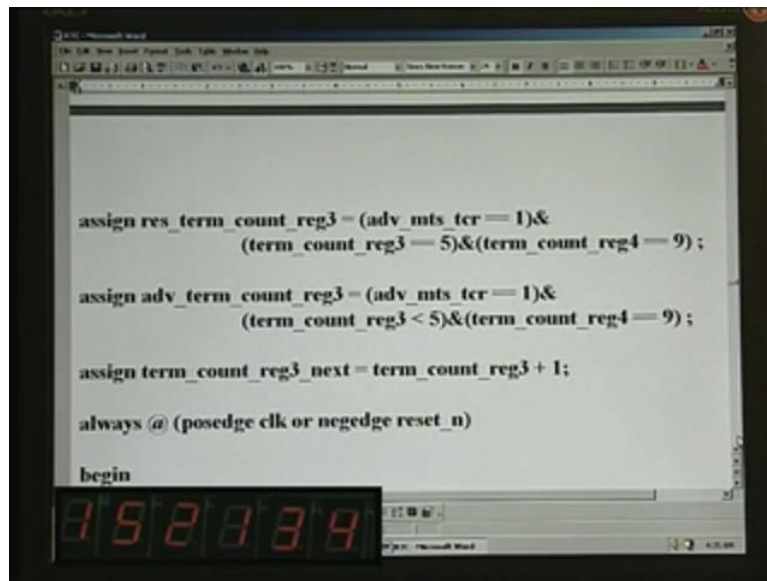
    else if (adv_term_count_reg2 == 1)
        term_count_reg2 <= term_count_reg2_next;

    else
        ;

end
```

Once again, the register advance is there, we assign the next value here when the clock strikes.

(Refer Slide Time: 14:06)



```
assign res_term_count_reg3 = (adv_mts_tcr == 1) &
    (term_count_reg3 == 5) & (term_count_reg4 == 9);

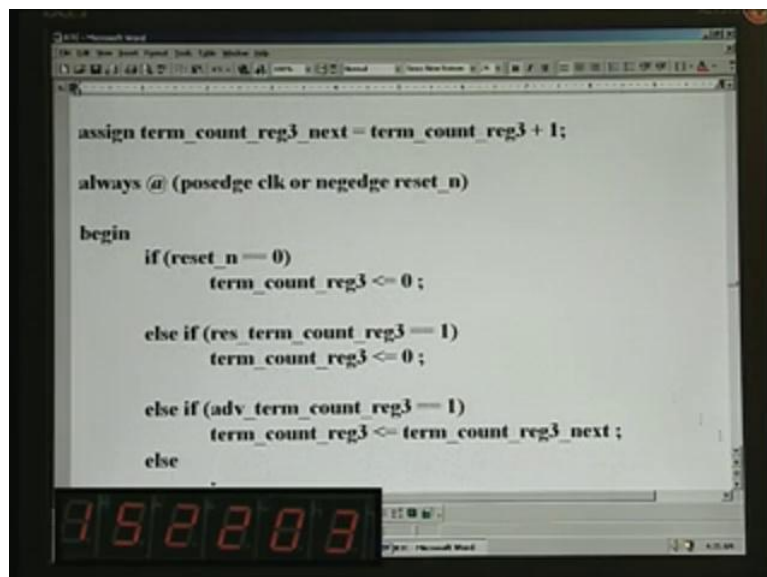
assign adv_term_count_reg3 = (adv_mts_tcr == 1) &
    (term_count_reg3 < 5) & (term_count_reg4 == 9);

assign term_count_reg3_next = term_count_reg3 + 1;

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

Next is reg3. Once again, reset term count, advance term count and then pre-incrementing. These are all precisely the same except for the conditions shown here, which you can easily reason out, except that some difference is there. Earlier, I think it was hours, now minutes will have to be taken into account. This condition also I think we have already covered, this **minutes_tcr**.

(Refer Slide Time: 14:35)



```
assign term_count_reg3_next = term_count_reg3 + 1;

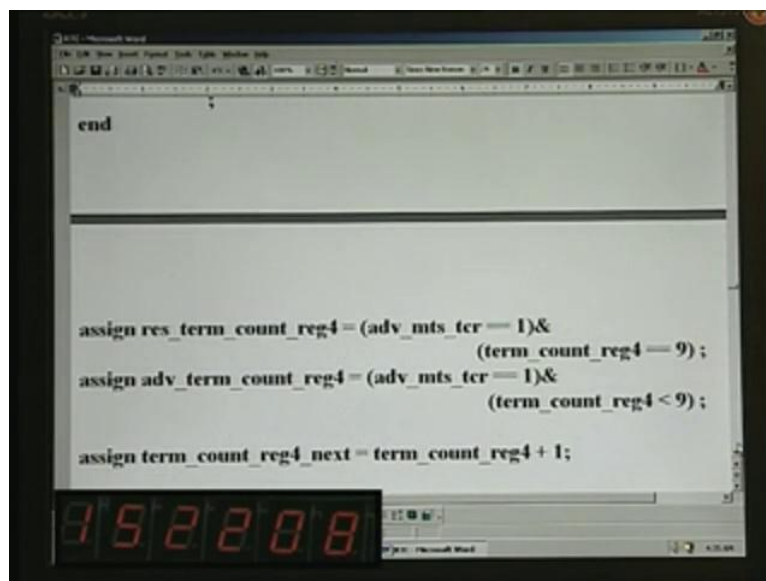
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 0)
        term_count_reg3 <= 0;

    else if (res_term_count_reg3 == 1)
        term_count_reg3 <= 0;

    else if (adv_term_count_reg3 == 1)
        term_count_reg3 <= term_count_reg3_next;
    else
```

Once again, the realization of that particular register term_count_reg3 and incrementing is happening here.

(Refer Slide Time: 14:40)



```
end

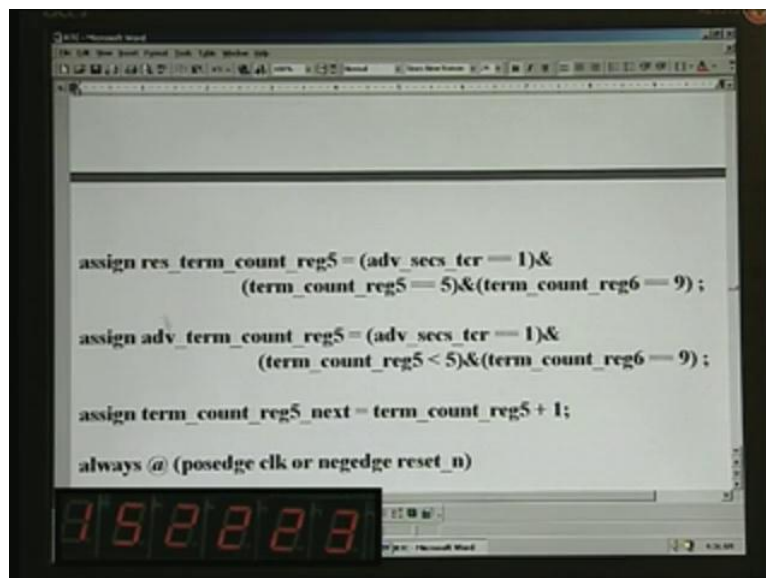
assign res_term_count_reg4 = (adv_mts_tcr == 1) &
                             (term_count_reg4 == 9);
assign adv_term_count_reg4 = (adv_mts_tcr == 1) &
                             (term_count_reg4 < 9);

assign term_count_reg4_next = term_count_reg4 + 1;
```

The screenshot shows a code editor window with the above Verilog code. Below the code, a 7-segment display is visible, showing the number 152208 in red digits.

Similarly for **reg4** with all that reset, advance, pre-increment and then the actual register implementation – these are all precisely the same, I do not have to repeat the same thing, which you are already familiar with by now.

(Refer Slide Time: 14:53)



```
assign res_term_count_reg5 = (adv_secs_tcr == 1) &
                             (term_count_reg5 == 5) & (term_count_reg6 == 9);
assign adv_term_count_reg5 = (adv_secs_tcr == 1) &
                             (term_count_reg5 < 5) & (term_count_reg6 == 9);

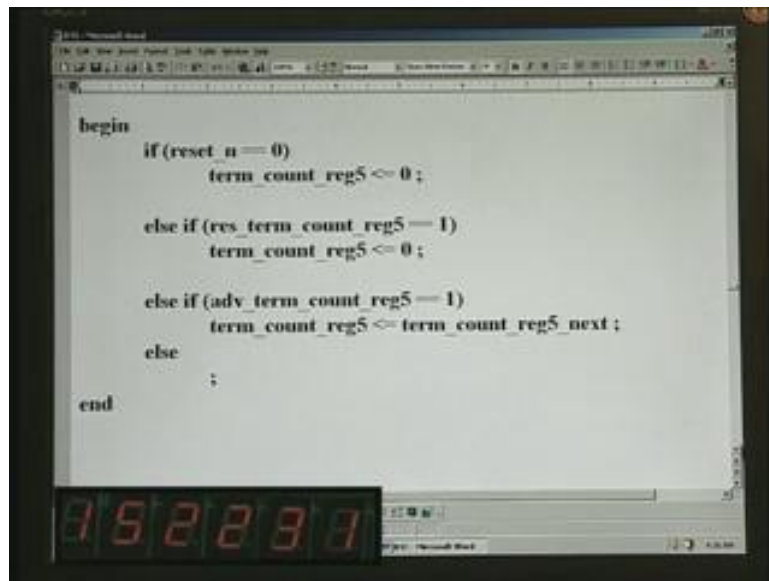
assign term_count_reg5_next = term_count_reg5 + 1;

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

The screenshot shows a code editor window with the above Verilog code. Below the code, a 7-segment display is visible, showing the number 152223 in red digits.

Reset, advance, once again pre-increment for **cnt5**.

(Refer Slide Time: 15:01)



```
begin
  if(reset_n == 0)
    term_count_reg5 <= 0;

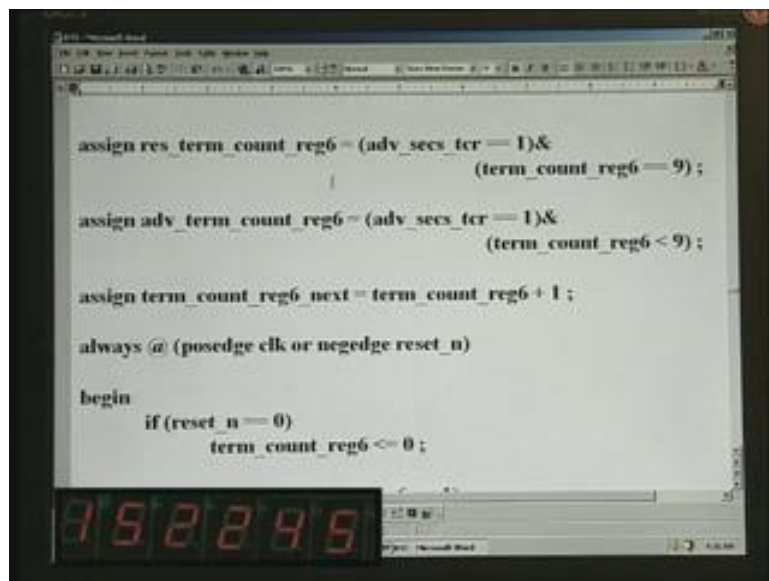
  else if(res_term_count_reg5 == 1)
    term_count_reg5 <= 0;

  else if(adv_term_count_reg5 == 1)
    term_count_reg5 <= term_count_reg5_next;
  else
    ;
end
```

The screenshot shows a Verilog code editor with the following code. Below the code, a 7-segment display shows the number 152233.

This is for the seconds MSD. This is the realization for the same.

(Refer Slide Time: 15:07)



```
assign res_term_count_reg6 = (adv_secs_tcr == 1) &
                             (term_count_reg6 == 9);
assign adv_term_count_reg6 = (adv_secs_tcr == 1) &
                             (term_count_reg6 < 9);
assign term_count_reg6_next = term_count_reg6 + 1;
always @ (posedge clk or negedge reset_n)
begin
  if(reset_n == 0)
    term_count_reg6 <= 0;
```

The screenshot shows a Verilog code editor with the following code. Below the code, a 7-segment display shows the number 152245.


The last one in this sequence is reg6 and that will happen for seconds. That is why this has been taken into account. These signals are already shown earlier. It means advance seconds and terminal count reg all being matching.

(Refer Slide Time: 15:32)

```
begin
  if (reset_n == 0)
    term_count_reg6 <= 0;

  else if (res_term_count_reg6 == 1)
    term_count_reg6 <= 0;

  else if (adv_term_count_reg6 == 1)
    term_count_reg6 <= term_count_reg6_next;
  else
    ;
end
```




Once again, pre-increment realization of the counter is here. All the six are almost identical, except for the signal change.

(Refer Slide Time: 15:39)

```
// Timer out is set when the terminal count (Up/Down) is reached.
assign timer_out_alarm_counter_next = timer_out_alarm_counter
                                     + 1
                                     // 30 secs. audio alarm counter.

assign timer_out_alarm = (timer_out) &
                          (timer_out_alarm_counter != 31);
// This signal is high for 30 secs. after terminal count
// is reached, i.e., timer_out = 1.
```

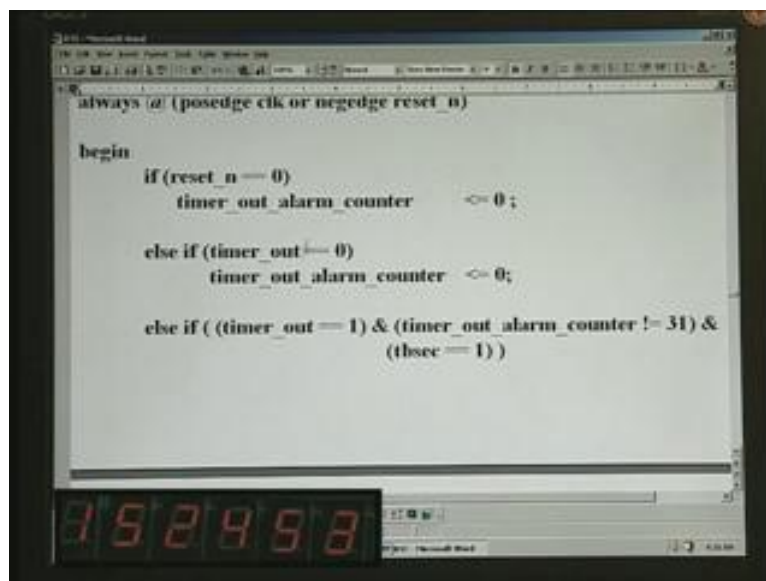


Timer out is set when the terminal count (up or down) is reached. You remember that we have a timer out, for example, to fire a rocket, as we have already seen. You can do it in two ways: either up mode or down mode. When the set value is reached, then the timer out goes high and remains high there. Concurrent to this, a sound alarm is also energized but that will be on only for 30 seconds – you will get a beeping alarm for that. As far as timer out is

concerned, once the time is over, it goes high and remains high – it is unlike the beeping output.

This is also as a counter basically and pre-increment is done in the same fashion. Why we need this counter is we need to keep track of the audio alarm. We have already set it for say 30 seconds (if you want 30 seconds) but you can change that also. In fact, you can change it from this statement you see here. If you say timer_out_alarm, this is nothing other than timer_out, have we declared earlier? This is the output. When timer_out is 1 and when timer_out_alarm_counter (that is the counter we are going to see below) is not equal to 31, only then timer_out_alarm must be on. That means to say this must be on only for 30 seconds, not beyond. Is that right? This signal is high for 30 seconds after terminal count is reached, that is timer_out equal to 1. This is the condition.

(Refer Slide Time: 17:25)

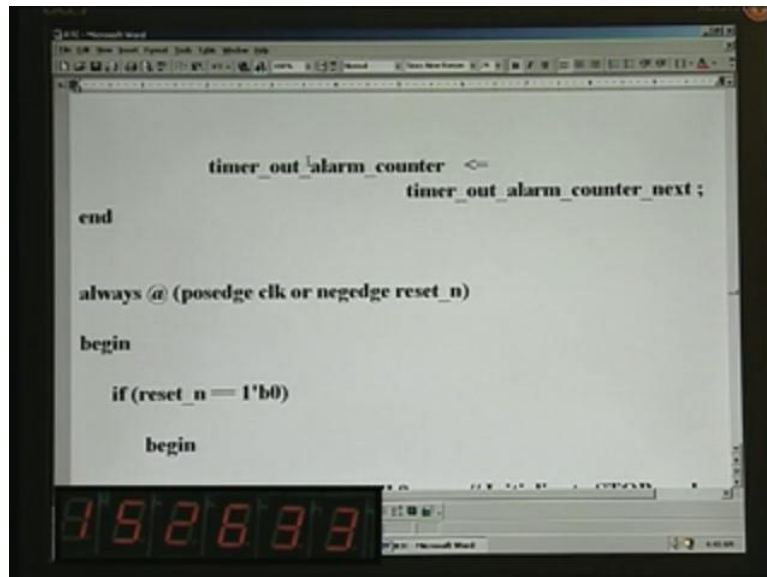


The timer out counter is like any other counter except for a few changes. We have an else-if here. If timer_out is equal to 0, that means it is not yet energized. What you need to do is the counter is also cleared. If timer_out is 1, that means timer_out is high and this can be high only when the terminal count is reached, is it not? That terminal count is different, this counter is different – this counter is only for sounding the alarms. We need to sound the alarm just for 30 seconds; otherwise, you get annoyed with the buzzing sound all the time.

When the running time or stopwatch matches with the set value in the count mode, be it up or down, then only timer_out is set – only after the lapse. After it is set, then only the 30 seconds

will come into play. That is why time `timer_out_alarm` not equal to 31. `30 means when it is equal, it has` already finished sounding for 30 seconds – that is what it means. Then, this will go low and this will not be satisfied. We need to satisfy this, so that we may sound the alarm only for 30 seconds – that is the implication of this and this should happen only every second.

(Refer Slide Time: 19:06)

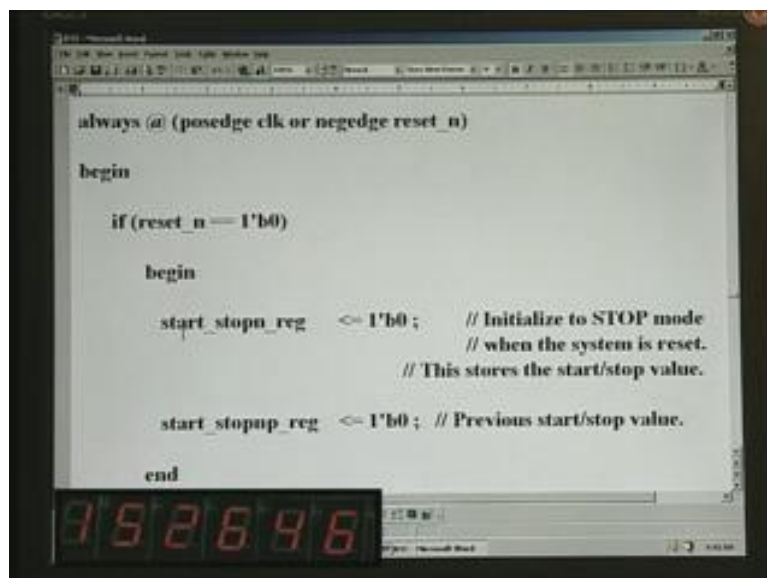


```
timer_out_alarm_counter <=
    timer_out_alarm_counter_next;
end

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

Otherwise, you increment the actual alarm counter.

(Refer Slide Time: 19:13)



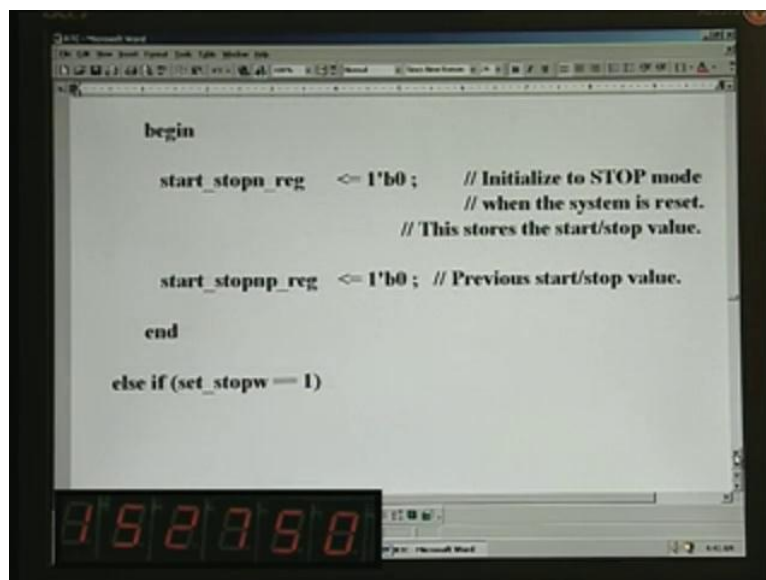
```
always @ (posedge clk or negedge reset_n)
begin
    if(reset_n == 1'b0)
        begin
            start_stopn_reg <= 1'b0; // Initialize to STOP mode
            // when the system is reset.
            // This stores the start/stop value.

            start_stopup_reg <= 1'b0; // Previous start/stop value.
        end
end
```

We have another set of always blocks for various purposes. For example, you may remember that we have a push button for start and stop. Why is this register necessary? We have only

one push button switch and when we push for the first time, we should take the system into start mode. If you push the same push button again, it should be turned into stop mode. Start and stop are applicable only for up counter and down counter. Whenever you want to start the counter, all you have to do is press the button. It will start and when you want to stop, press the same button again. You can use for dark room – you can develop your films, etc., using this same real-time clock in the down-count mode. You can even use an up counter and there is going to be a buzzer after the set time is lapsed and it will freeze at the last value. If it is down count, it will be all 0s and stay there. The buzzer will sound for 30 seconds and be silent thereafter, but the output itself will remain high.

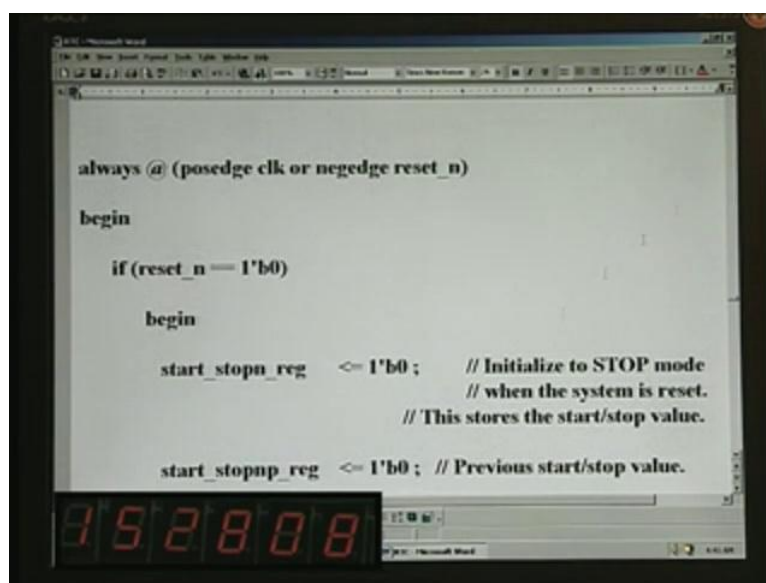
(Refer Slide Time: 20:23)



```
begin
    start_stopn_reg <= 1'b0; // Initialize to STOP mode
                        // when the system is reset.
                        // This stores the start/stop value.

    start_stopnp_reg <= 1'b0; // Previous start/stop value.
end

else if (set_stopw == 1)
```

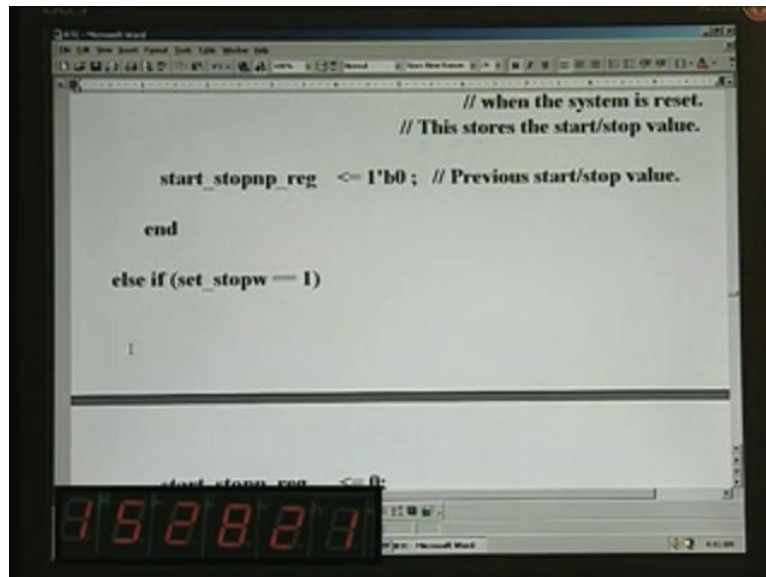


```
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
    begin
        start_stopn_reg <= 1'b0; // Initialize to STOP mode
                                // when the system is reset.
                                // This stores the start/stop value.

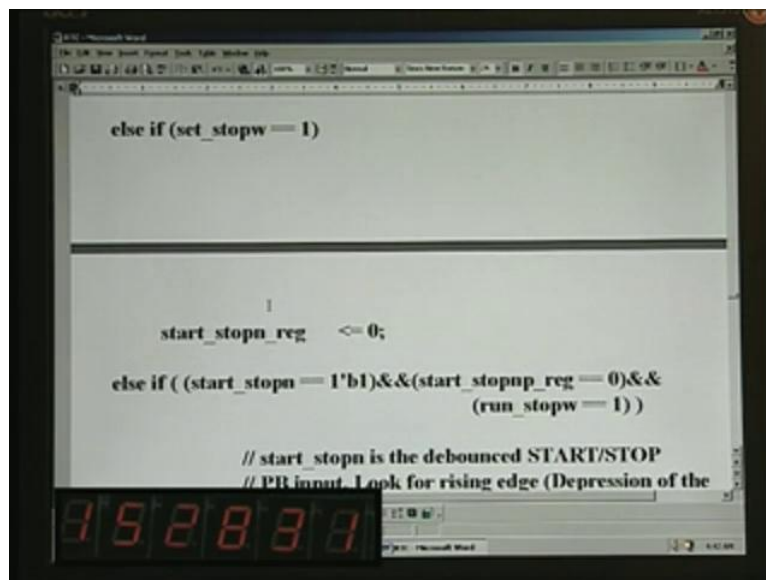
        start_stopnp_reg <= 1'b0; // Previous start/stop value.
```

Then what we need is for the same signal, we also need to know the previous value so that we can sense **when the start stopwatch push**. This will be known only if you register at every clock pulse or every time this logic is satisfied. Here, this is only clearing the logic when **power on system reset** is encountered. This is the previous of value of this.

(Refer Slide Time: 20:55)



```
// when the system is reset.  
// This stores the start/stop value.  
  
start_stopnp_reg <= 1'b0; // Previous start/stop value.  
  
end  
  
else if (set_stopw == 1)  
  
1  
  
start_stopnp_reg <= 0;
```



```
else if (set_stopw == 1)  
  
1  
  
start_stopnp_reg <= 0;  
  
else if ((start_stopw == 1'b1) && (start_stopnp_reg == 0) &&  
        (run_stopw == 1))  
  
// start_stopw is the debounced START/STOP  
// PB input. Look for rising edge (Depression of the
```

If set stopwatch is 1, **that is the literal meaning you should take here**, when you are in the setting mode of stopwatch, what you should do is you have to stop the **running up-down counter**, because we do not have to do any action at that point of time.

(Refer Slide Time: 21:16)

```
start_stopn_reg <= 0;

else if ( (start_stopn == 1'b1) && (start_stopnp_reg == 0) &&
         (run_stopw == 1) )

    // start_stopn is the debounced START/STOP
    // PB input. Look for rising edge (Depression of the
    // push button switch).

begin

    start_stopn_reg <= !start_stopn_reg ;
        // Toggle between START & STOP.

    start_stopnp_reg <= start_stopn ;

152853
```

On the other hand, if **start_stopn** is 1, it implies that you want to start and it is already in start mode after you push for the first time. We have just now seen that the previous value must be 0. 01 senses the rising edge of the push button and that means the push button has been pressed. It must be in this run stopwatch mode and note that there is no n. **This is straight.** start_stopn is the debounced START/STOP push button input and this is the comment for this. Look for rising edge (depression of the push button switch). That is the comment that we have just now explained.

(Refer Slide Time: 21:55)

```
// start_stopn is the debounced START/STOP
// PB input. Look for rising edge (Depression of the
// push button switch).

begin

    start_stopn_reg <= !start_stopn_reg ;
        // Toggle between START & STOP.

    start_stopnp_reg <= start_stopn ;
        // Preserve as the previous start/stop value.

end

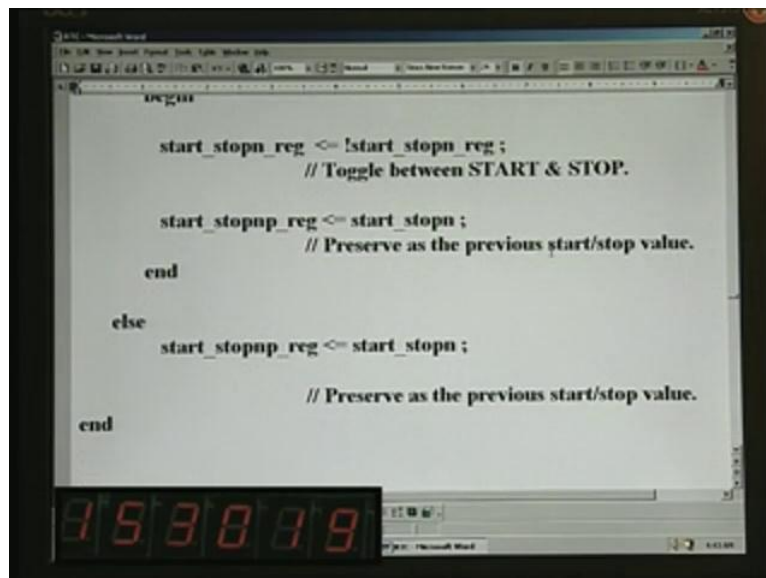
else

    start_stopnp_reg <= start_stopn ;

152923
```

Here, what we should do is whenever we push the button, it has to toggle. This is because we have only one register called start_stopn. The same will indicate whether it is in start mode or stop mode. The first time when your system **initializes resetting**, this will be cleared. Subsequently, when you push the start button once, this will be made 1 and subsequent pushing will have to be inverted. What you need is an inversion and that is precisely **what this statement is for**. The same signal is inverted and assigned to itself. In other words, it is toggling between start and stop. This is how you achieve with single button rather than having to use two different buttons. Another thing is that you should not forget to load this previous value with the present value. This is an important thing and if you miss this, it will not work.

(Refer Slide Time: 22:52)



```
begin
    start_stopn_reg <= !start_stopn_reg ;
    // Toggle between START & STOP.

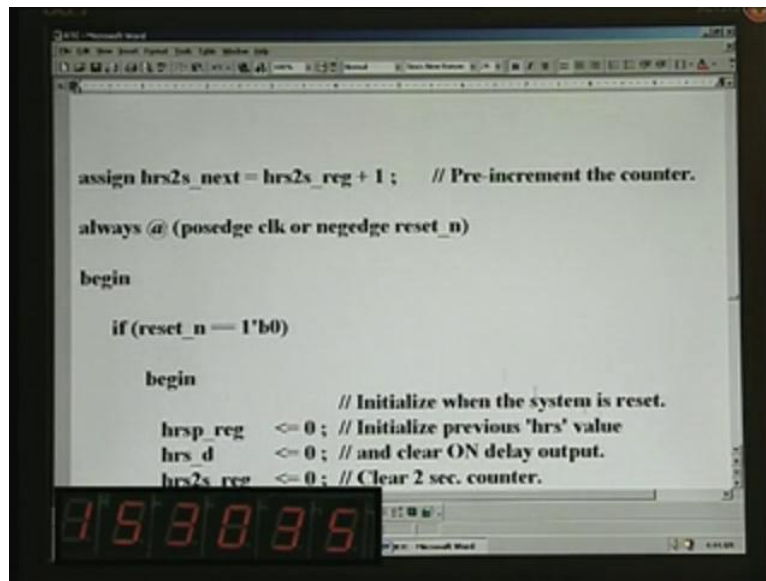
    start_stopnp_reg <= start_stopn ;
    // Preserve as the previous start/stop value.
end

else
    start_stopnp_reg <= start_stopn ;
    // Preserve as the previous start/stop value.
end
```

153019

Preserve as the previous start/stop value. This one is also precisely the same. If none of these conditions are met, then also you should not forget to preserve the present value into the past value.

(Refer Slide Time: 23:08)



```
assign hrs2s_next = hrs2s_reg + 1; // Pre-increment the counter.

always @ (posedge clk or negedge reset_n)

begin

    if (reset_n == 1'b0)

        begin

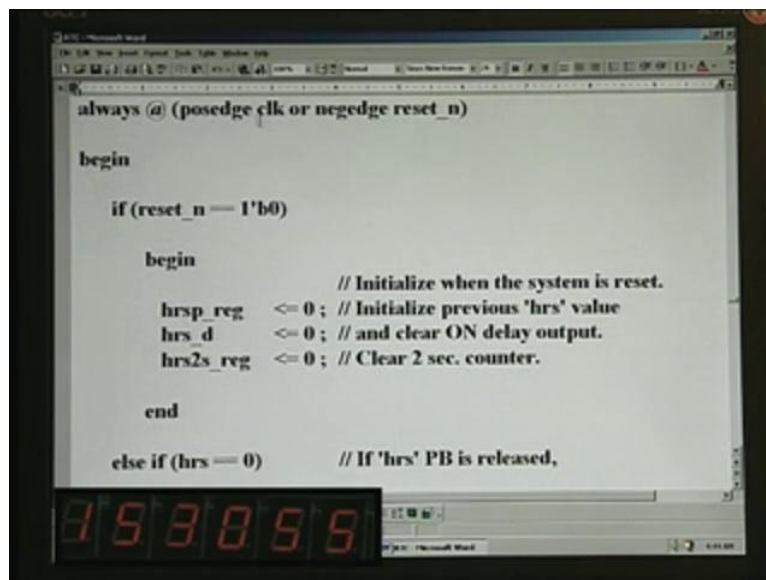
            // Initialize when the system is reset.
            hrsp_reg <= 0; // Initialize previous 'hrs' value
            hrs_d <= 0; // and clear ON delay output.
            hrs2s_reg <= 0; // Clear 2 sec. counter.

        end

end
```

Next we need to generate 2 seconds. One of your questions was how long I should hold. We need to hold it for 2 seconds and that is precisely what we are going to implement now using this `hrs2s_next` and `hrs2s_reg`. This is the pre-increment for that.

(Refer Slide Time: 23:27)



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

begin

    if (reset_n == 1'b0)

        begin

            // Initialize when the system is reset.
            hrsp_reg <= 0; // Initialize previous 'hrs' value
            hrs_d <= 0; // and clear ON delay output.
            hrs2s_reg <= 0; // Clear 2 sec. counter.

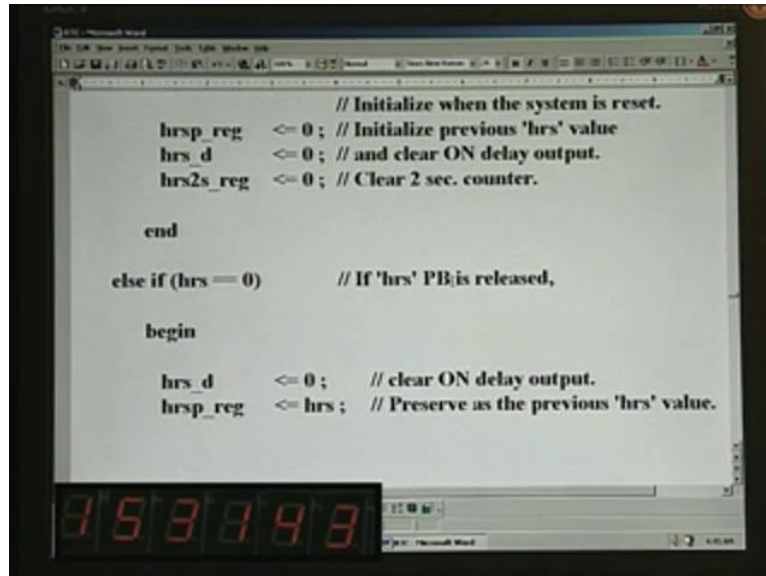
        end

    else if (hrs == 0) // If 'hrs' PB is released,
```

Once again there being a counter, we had to have it in a `positive edge clock always block`. This is the usual resetting and this is the actual register here. We also need a delayed output so as to keep track of pushing continuously for 2 seconds. This is what is going to keep track of that. This is basically derived from the hours push button. If you push the hours button in order to set, `only then` all these come into play. The de-bounce condition of the push button

hrs is hrs and its previous value is what we earmark as hrsp. These are all to be initialized to start with.

(Refer Slide Time: 24:16)



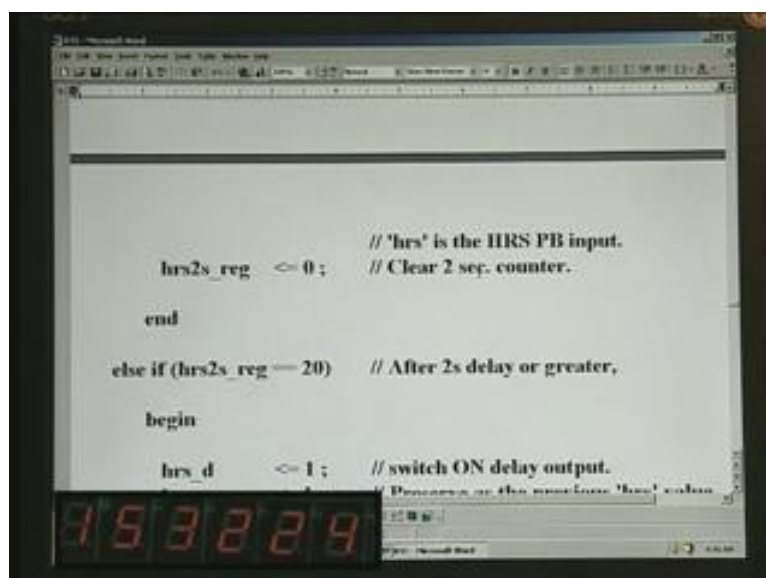
```
// Initialize when the system is reset.
hrsp_reg <= 0; // Initialize previous 'hrs' value
hrs_d <= 0; // and clear ON delay output.
hrs2s_reg <= 0; // Clear 2 sec. counter.

end

else if (hrs = 0) // If 'hrs' PB is released,
begin
hrs_d <= 0; // clear ON delay output.
hrsp_reg <= hrs; // Preserve as the previous 'hrs' value.
```

If the hours push button is released, what will happen is hrs is going to become 0. If you push also, I think it should go to 0 – just examine this comment yourselves. Clear the ON delay output. These are all preconditions. In a release condition, this is the one. If it is push, I think it is 1, let us clarify that. That is why we are trying to initialize this. This is 0, mind you. The previous value is the actual hours. The current value is taken here and assigned as the previous value.

(Refer Slide Time: 24:56)



```
hrs2s_reg <= 0; // 'hrs' is the HRS PB input.
// Clear 2 sec. counter.

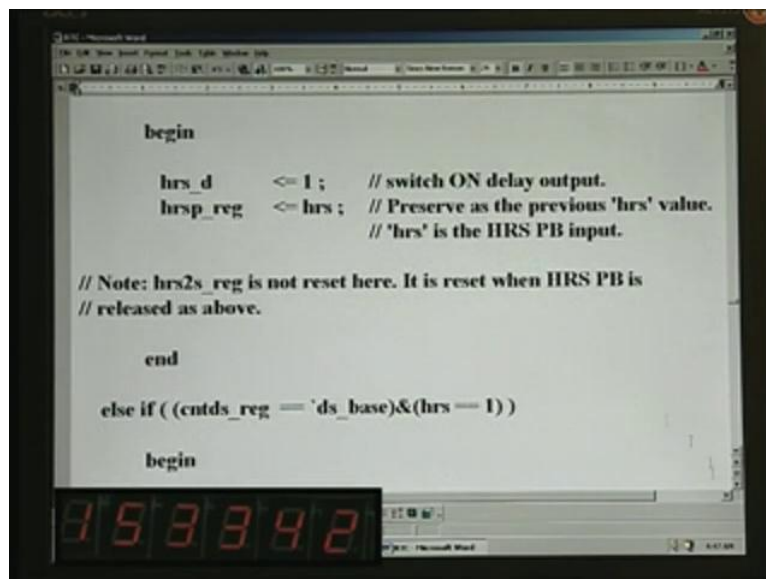
end

else if (hrs2s_reg = 20) // After 2s delay or greater,
begin
hrs_d <= 1; // switch ON delay output.
// Preserve as the previous 'hrs' value.
```

hrs is the hours push button input. Clear the two-second counter. This is the two-second counter as I mentioned and it needs to be cleared. That means the push button is not pressed for incrementing that display. Next, suppose the **hrs2s** starts running (we have not spoken of running yet) and **we look for the value 20**, this will give exactly this time basis 0.1 second once again here and 20 would mean a delay of 2 seconds or greater. This is how you reckon the actual delay. This is the running counter for keeping track.

It will start from 0, 1, 2, 3. It will start advancing from 0, 1 and so on only if you had pressed for at least 2 seconds – only then this will come into play. Here, when this happens, that is, after 2 seconds delay, this **hours delay** goes high. That is what I was saying when you start the thing, **hours output delay** will still continue to be 0. Only after the lapse of 2 seconds delay, it will go to 1. That is what is happening here and you should not forget to preserve the current hours push button – you have to preserve that.

(Refer Slide Time: 26:14)



Note **hrs2s_reg** is not reset here. I am just inviting your attention by saying that it is not reset here – it is reset when hrs push button is released as above. Earlier, we have seen **release here**.


(Refer Slide Time: 26:29)

```
hrs_d    <= 0 ; // clear ON delay output.
hrs_reg  <= hrs ; // Preserve as the previous 'hrs' value.

hrs2s_reg <= 0 ; // Clear 2 sec. counter.

end

else if (hrs2s_reg == 20) // After 2s delay or greater,
```



Here only it is reset but not here.

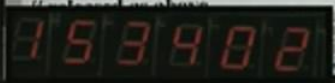
(Refer Slide Time: 26:34)

```
hrs2s_reg <= 0 ; // Clear 2 sec. counter.

end

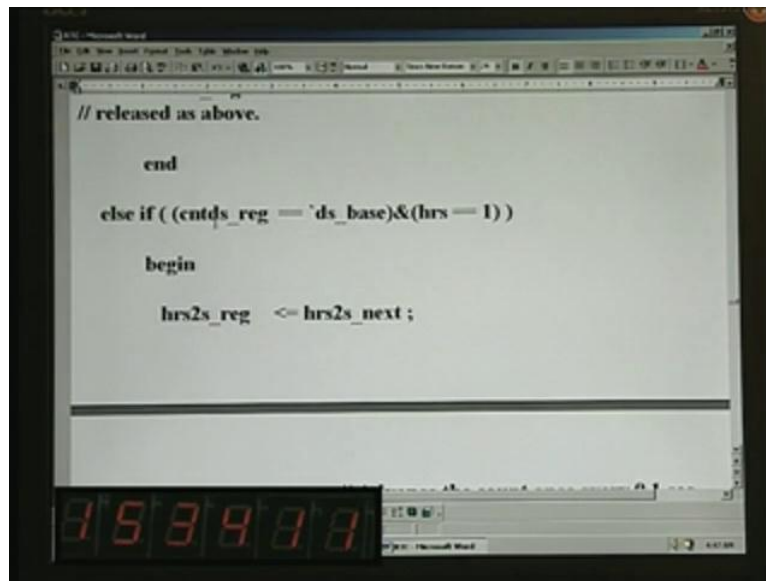
else if (hrs2s_reg == 20) // After 2s delay or greater,
begin
hrs_d    <= 1 ; // switch ON delay output.
hrs_reg  <= hrs ; // Preserve as the previous 'hrs' value.
// 'hrs' is the HRS PB input.

// Note: hrs2s_reg is not reset here. It is reset when HRS PB is
// closed again.
```



After the time is lapsed, only after the two-second delay, you need to do this.

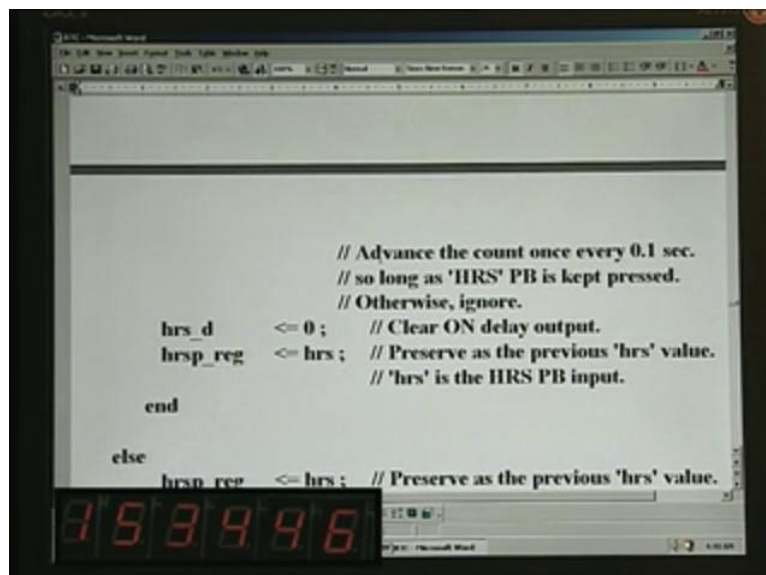
(Refer Slide Time: 26:42)



```
// released as above.  
  
end  
  
else if ( (cntds_reg == `ds_base)&(hrs == 1) )  
  
begin  
  
hrs2s_reg <= hrs2s_next ;
```

The next condition is **counter decisecond**. We want 0.1 second. Only when that is equal to the set value we have defined earlier and the hours push button is also 1, we need to increment the counter. That means we are incrementing every 0.1 second. 20 is what we have put earlier, just now we have seen. 20 into 0.1 will be 2 second. That is how you get that. Is it clear?

(Refer Slide Time: 27:17)

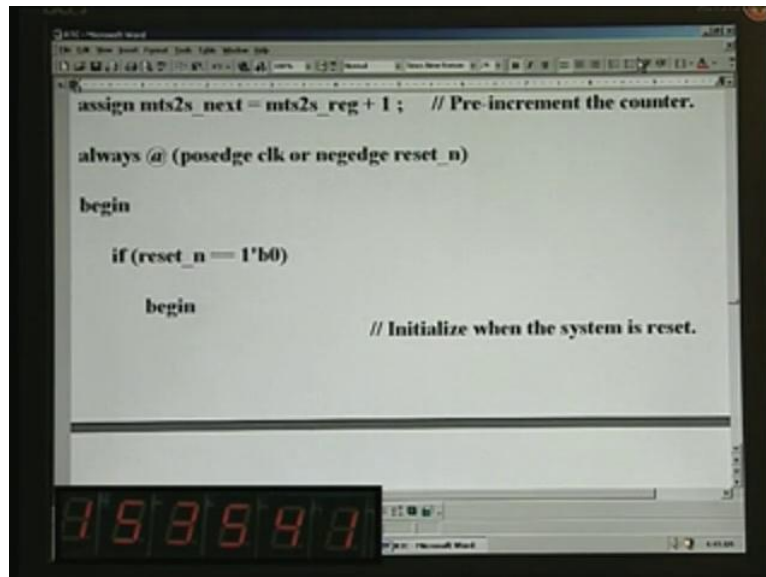


```
// Advance the count once every 0.1 sec.  
// so long as 'HRS' PB is kept pressed.  
// Otherwise, ignore.  
  
hrs_d <= 0 ; // Clear ON delay output.  
hrsp_reg <= hrs ; // Preserve as the previous 'hrs' value.  
// 'hrs' is the HRS PB input.  
  
end  
  
else  
hrsp_reg <= hrs ; // Preserve as the previous 'hrs' value.
```

Advance the count once every 0.1 second so long as the hours push button switch is pressed, otherwise ignore. Did I convey the same thing or did I make a mistake? Is it okay? 0.1 is the time base. You are trying to count 20 each time. What we have seen is merely incrementing

when the condition is 0.1 second. Whenever the 0.1 second condition occurs, you merely increment this counter. When it is equal to 20, stop that. We have forced it to 0, remember? Here also, clear the ON delay output and preserve as the previous value – you have to do this also. If none of these conditions are met, you have to just preserve once again.

(Refer Slide Time: 28:14)



```
assign mts2s_next = mts2s_reg + 1; // Pre-increment the counter.

always @ (posedge clk or negedge reset_n)

begin

    if (reset_n == 1'b0)

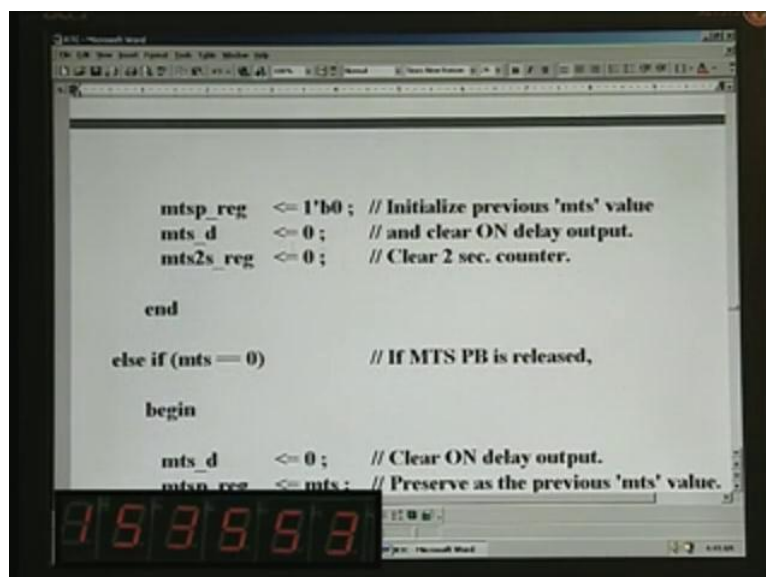
        begin

            // Initialize when the system is reset.


```

What we have seen so far is for hours. Similarly, we need for minutes as well as seconds. I am not going into many details.

(Refer Slide Time: 28:26)



```
    mtsp_reg <= 1'b0; // Initialize previous 'mts' value
    mts_d <= 0; // and clear ON delay output.
    mts2s_reg <= 0; // Clear 2 sec. counter.

end

else if (mts == 0) // If MTS PB is released,

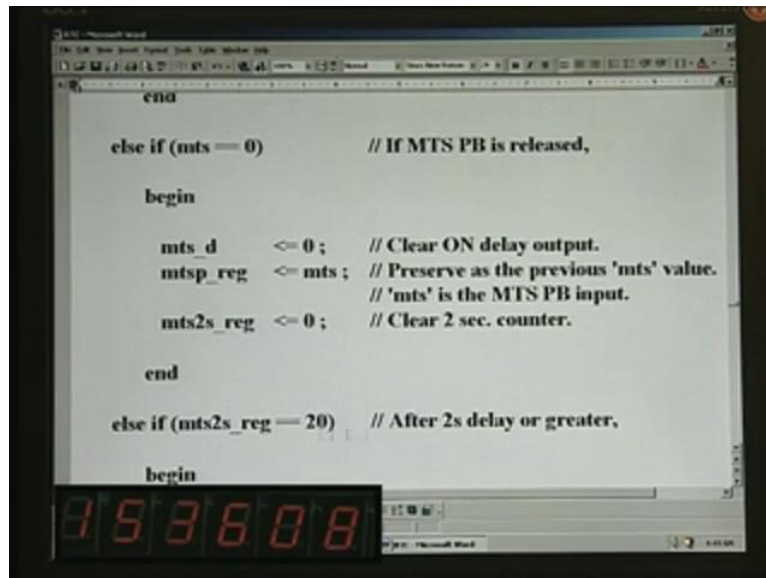
begin

    mts_d <= 0; // Clear ON delay output.
    mtsn_reg <= mts; // Preserve as the previous 'mts' value.


```

I will leave it to you, you can just go through. We have got the very same thing here. If minutes instead of hours is encountered, the action you have to do is precisely the same thing.

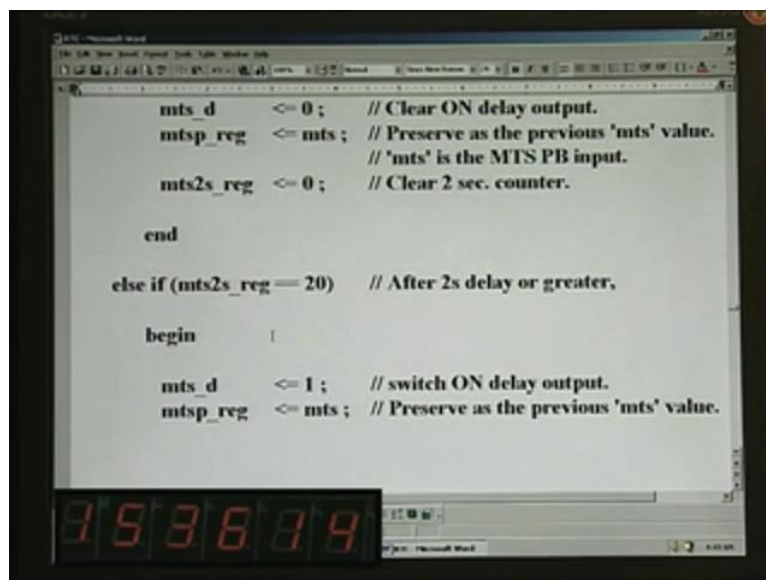
(Refer Slide Time: 28:41)



```
end  
  
else if (mts == 0) // If MTS PB is released,  
begin  
  
    mts_d <= 0; // Clear ON delay output.  
    mtsp_reg <= mts; // Preserve as the previous 'mts' value.  
                // 'mts' is the MTS PB input.  
    mts2s_reg <= 0; // Clear 2 sec. counter.  
  
end  
  
else if (mts2s_reg == 20) // After 2s delay or greater,  
begin
```

Again if it is 20, it means 2 seconds.

(Refer Slide Time: 28:48)




```
    mts_d <= 0; // Clear ON delay output.  
    mtsp_reg <= mts; // Preserve as the previous 'mts' value.  
                // 'mts' is the MTS PB input.  
    mts2s_reg <= 0; // Clear 2 sec. counter.  
  
end  
  
else if (mts2s_reg == 20) // After 2s delay or greater,  
begin  
  
    mts_d <= 1; // switch ON delay output.  
    mtsp_reg <= mts; // Preserve as the previous 'mts' value.
```

Earlier, what we spoke was for hours, **only two digits** hours display will be managed there. Now, we are going to manage the next two digits for the minutes alone. The next two digits will be for the seconds, which is going to come over here.

(Refer Slide Time: 29:03)

```
end
else if ((cntds_reg == `ds_base)&(mts == 1))
begin
    mts2s_reg <= mts2s_next ;
                // Advance the count once every 0.1 sec.
                // so long as 'MTS' PB is kept ressed.
                // Otherwise, ignore.

    mts_d      <= 0 ; // Clear ON delay output.
    mtsp_reg   <= mts ; // Preserve as the previous 'mts' value.
                // 'mts' is the MTS PB input.
end
```



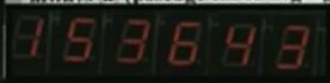
This is precisely the same condition, **decisecond meeting** and this time, it is the minutes push button switch. Based on the minutes push button switch only, we need to take action. These are all precisely the same.

(Refer Slide Time: 29:16)

```
else
    mtsp_reg <= mts ; // Preserve as the previous 'mts' value.
                // 'mts' is the MTS PB input.

end

assign secs2s_next = secs2s_reg + 1 ; // Pre-increment the counter.
always @(posedge clk or negedge reset_n)
```




```
begin
  if (reset_n == 1'b0)
    begin
      // Initialize when the system is reset.
      secs_reg <= 1'b0; // Initialize previous 'secs' value
      secs_d <= 0; // and clear ON delay output.
      secs2s_reg <= 0; // Clear 2 sec. counter.
    end
  else if (secs == 0) // If secs PB is released,
```

So is the case for seconds here. Advance increment, then reset condition, all this and once again instead of hours or minutes equal to 0, we do this.

(Refer Slide Time: 29:27)

```
secs_d <= 0; // Clear ON delay output.
secs_reg <= secs; // Preserve as the previous 'secs' value.
// 'secs' is the SECS PB input.
secs2s_reg <= 0; // Clear 2 sec. counter.
end
else if (secs2s_reg == 20) // After 2s delay or greater,
  begin
    secs_d <= 1; // switch ON delay output.
    secs_reg <= secs; // Preserve as the previous secs value.
    // secs is the SECS PB input.
```

We have set or reset the relevant registers. Once again, [29:33] 20 here, the delay is again made 1 and previous value set here.

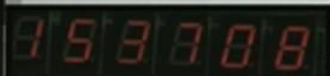
(Refer Slide Time: 29:41)

```
end

else if ((cntds_reg == 'ds_base') & (secs == 1))

begin

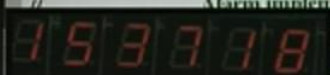
secs2s_reg <= secs2s_next ; // Advance the count once
// every 0.1 sec. so long as 'SECS' PB
// is kept pressed. Otherwise, ignore.
```



```
secs2s_reg <= secs2s_next ; // Advance the count once
// every 0.1 sec. so long as 'SECS' PB
// is kept pressed. Otherwise, ignore.
secs_d <= 0 ; // Clear ON delay output.
sejsp_reg <= secs ; // Preserve as the previous 'secs' value.
// 'secs' is the SECS PB input.
end

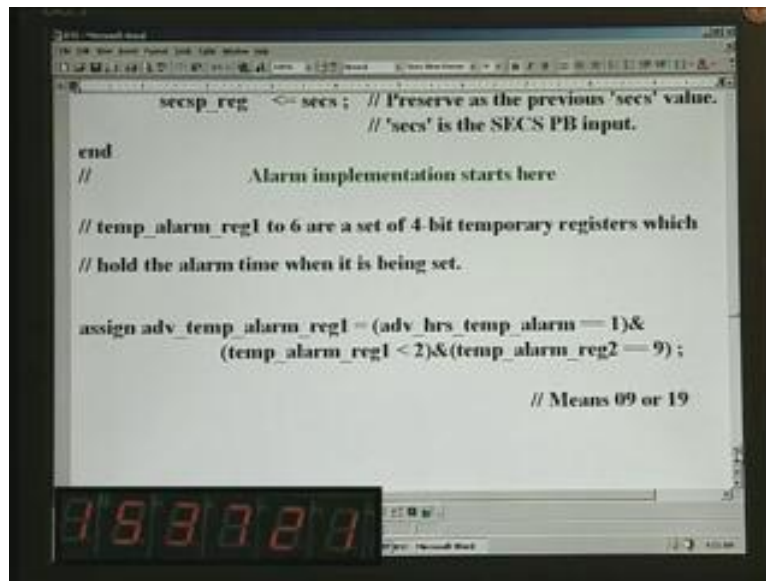
else
sejsp_reg <= secs ; // Preserve as the previous 'secs' value.
// 'secs' is the SECS PB input.
end

// Alarm implementation starts here
```



Once again, if this condition is met, only then we increment. As usual, it is precisely the same. Otherwise, preserve the value.

(Refer Slide Time: 29:54)



```
secsp_reg <= secs; // Preserve as the previous 'secs' value.
// 'secs' is the SECS PB input.

end
// Alarm implementation starts here

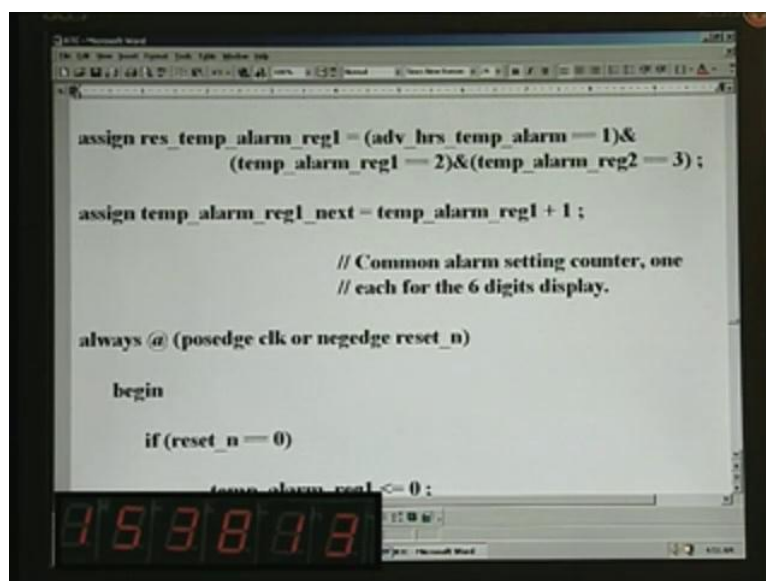
// temp_alarm_reg1 to 6 are a set of 4-bit temporary registers which
// hold the alarm time when it is being set.

assign adv_temp_alarm_reg1 = (adv_hrs_temp_alarm == 1) &
(temp_alarm_reg1 < 2) & (temp_alarm_reg2 == 9);

// Means 09 or 19
```

Now, what is left is alarm implementation. It starts right here. For this, we have a set of four-bit temporary registers **temp_alarm_reg1 to 6**, which hold the alarm time when it is being set. Then, assign this particular signal, which is **adv_temp_alarm_reg1**. We do it only when **adv_hrs_temp_alarm** is 1 and **temp_alarm_reg1** is less than 2 – that means 0 or 1. The next alarm must be 9, which means 09 or 19. Only then, you advance. It is the alarm setting we are talking of now. Once again, reg1 will correspond to the hours MSD. That is what we are talking about here.

(Refer Slide Time: 30:46)



```
assign res_temp_alarm_reg1 = (adv_hrs_temp_alarm == 1) &
(temp_alarm_reg1 == 2) & (temp_alarm_reg2 == 3);

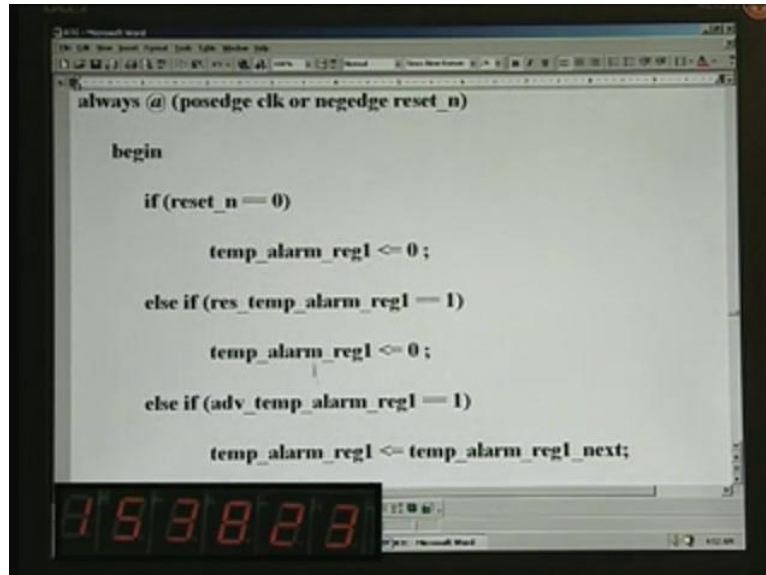
assign temp_alarm_reg1_next = temp_alarm_reg1 + 1;

// Common alarm setting counter, one
// each for the 6 digits display.

always @ (posedge clk or negedge reset_n)
begin
if (reset_n == 0)
temp_alarm_reg1 <= 0;
```

Similarly, reset condition is here for 2 and 3. This is the same condition and this is pre-incrementing the register.

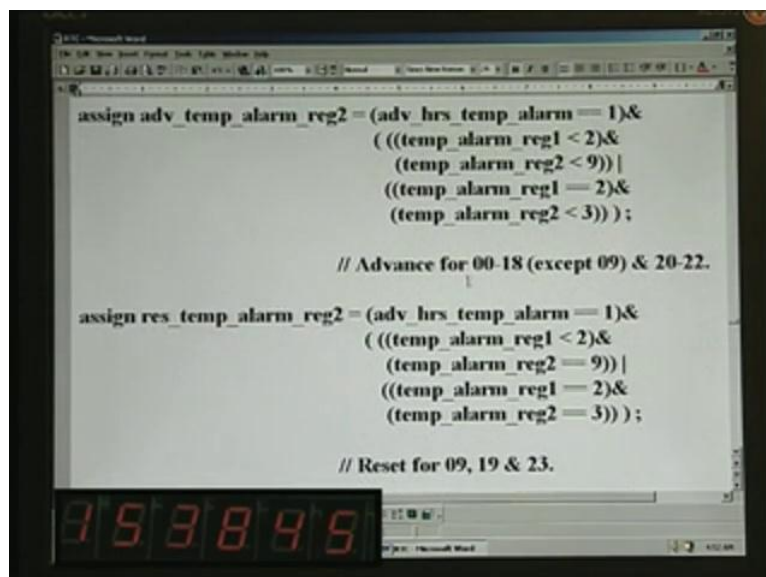
(Refer Slide Time: 30:57)



```
always @(posedge clk or negedge reset_n)
begin
    if(reset_n == 0)
        temp_alarm_reg1 <= 0;
    else if(res_temp_alarm_reg1 == 1)
        temp_alarm_reg1 <= 0;
    else if(adv_temp_alarm_reg1 == 1)
        temp_alarm_reg1 <= temp_alarm_reg1_next;
end
```

Once again, as usual, we have the reset condition for realizing the **register 1 temporary alarm**. What we do is we first have a temporary alarm for all the six registers, corresponding to the six counters that we had or six displays we had. Pre-incrementing is assigned actually here.

(Refer Slide Time: 31:19)



```
assign adv_temp_alarm_reg2 = (adv_hrs_temp_alarm == 1) &
    (((temp_alarm_reg1 < 2) &
      (temp_alarm_reg2 < 9)) |
     ((temp_alarm_reg1 == 2) &
      (temp_alarm_reg2 < 3)));

// Advance for 00-18 (except 09) & 20-22.

assign res_temp_alarm_reg2 = (adv_hrs_temp_alarm == 1) &
    (((temp_alarm_reg1 < 2) &
      (temp_alarm_reg2 == 9)) |
     ((temp_alarm_reg1 == 2) &
      (temp_alarm_reg2 == 3)));

// Reset for 09, 19 & 23.
```

Similarly, for reg2 – it is precisely the same. I am going to read only the comment. Here, you are going to advance this alarm setting only if it is 00 to 18, other than 09 and 20 to 22. Only then, you need to advance, not otherwise. Similarly, you can reset only when it is 09 or 19 or 23. This is as far as the second register is concerned, which is corresponding to 9 or this 9 or 3, because this is the terminal count. Whether it is up counter or down counter setting or alarm counter, all will go only in the forward direction – incrementing only.

(Refer Slide Time: 31:58)

```

(temp_alarm_reg2 == 9) |
((temp_alarm_reg1 == 2) &
(temp_alarm_reg2 == 3));

// Reset for 09, 19 & 23.

assign temp_alarm_reg2_next = temp_alarm_reg2 + 1;

always @ (posedge clk or negedge reset_n)

```

```

begin
if (reset_n == 0)
temp_alarm_reg2 <= 0;
else if (res_temp_alarm_reg2 == 1)
temp_alarm_reg2 <= 0;
else if (adv_temp_alarm_reg2 == 1)
temp_alarm_reg2 <= temp_alarm_reg2_next;
else


```

Once again, pre-increment for that and the temporary alarm reg2 is precisely the same as the previous thing.

(Refer Slide Time: 32:07)

```
end

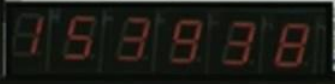
assign adv_temp_alarm_reg3 = (adv_mts_temp_alarm == 1) &
                               (temp_alarm_reg3 < 5) & (temp_alarm_reg4 == 9);
```



```
// Advance for 09, 19, 29, 39, 49.
assign res_temp_alarm_reg3 = (adv_mts_temp_alarm == 1) &
                               (temp_alarm_reg3 == 5) & (temp_alarm_reg4 == 9);

// Reset for 59.
assign temp_alarm_reg3_next = temp_alarm_reg3 + 1;

always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 0)
```



```
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 0)
        temp_alarm_reg3 <= 0 ;
    else if (res_temp_alarm_reg3 == 1)

```

Once again, for 3 you have advance as well as reset here. The condition is 59 here. Then once again, pre-increment and a block for **temporary alarm reg** here. This is exactly the same.

(Refer Slide Time: 32:24)

```
        temp_alarm_reg3 <= 0 ;
    else if (adv_temp_alarm_reg3 == 1)
        temp_alarm_reg3 <= temp_alarm_reg3_next ;
    else
        ;
end

assign adv_temp_alarm_reg4 = (adv_mts_temp_alarm == 1) &
    (temp_alarm_reg4 < 9) ;
```

```
end

assign adv_temp_alarm_reg4 = (adv_mts_temp_alarm == 1) &
                             (temp_alarm_reg4 < 9);

// Advance for 0-8.

assign res_temp_alarm_reg4 = (adv_mts_temp_alarm == 1) &
                             (temp_alarm_reg4 == 9);

// Reset for 9.

temp_alarm_reg4 + 1;
```

So is the case for advance reg4. The condition is 0 to 8 in this and reset for 9 as far as 4 is concerned. Once again, there is the **pre-increment block** for realizing the same.

(Refer Slide Time: 32:39)

```
begin

if(reset_n == 0)

temp_alarm_reg4 <= 0;

else if(res_temp_alarm_reg4 == 1)

temp_alarm_reg4 <= 0;

else if(adv_temp_alarm_reg4 == 1)

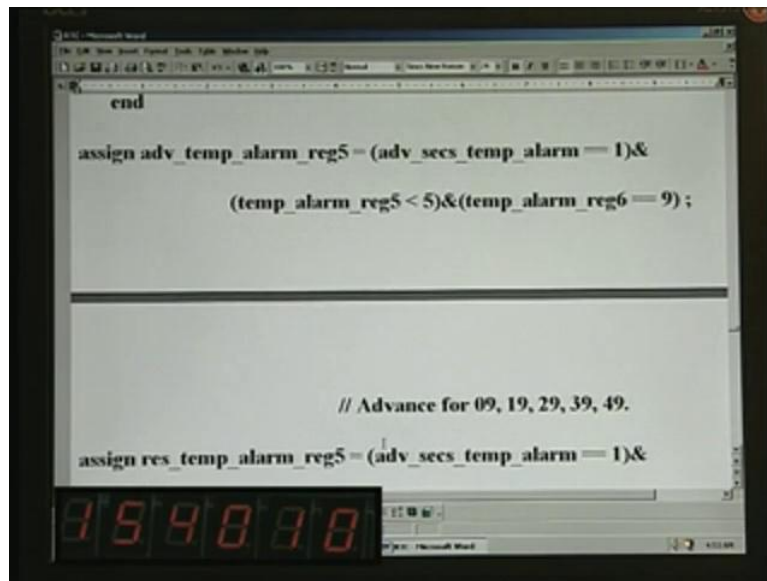
temp_alarm_reg4 <= temp_alarm_reg4_next;

else

;
```

This is exactly the same.

(Refer Slide Time: 32:43)



```
end

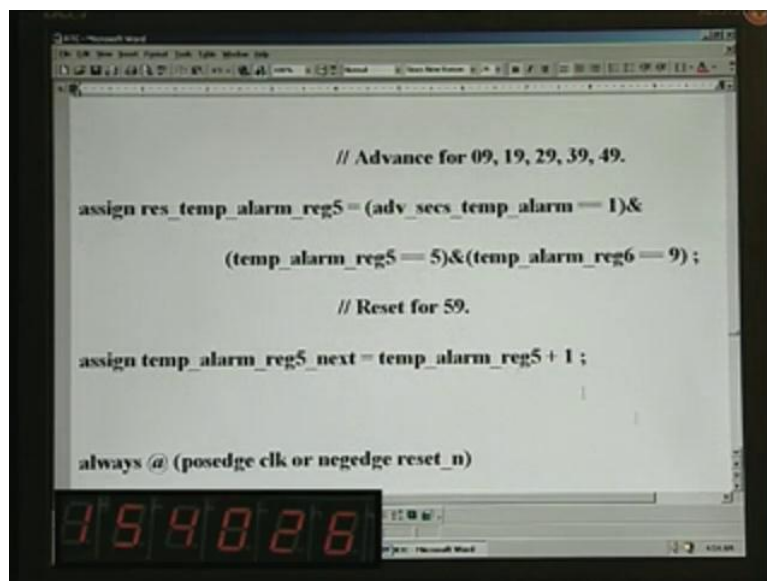
assign adv_temp_alarm_reg5 = (adv_secs_temp_alarm == 1) &
    (temp_alarm_reg5 < 5) & (temp_alarm_reg6 == 9);

// Advance for 09, 19, 29, 39, 49.

assign res_temp_alarm_reg5 = (adv_secs_temp_alarm == 1) &
```

The only thing is that it will advance for 09 or 19 or 29, 39, 49. This is clear. We are in 5 and one more is there. This is for seconds.

(Refer Slide Time: 32:59)



```
// Advance for 09, 19, 29, 39, 49.

assign res_temp_alarm_reg5 = (adv_secs_temp_alarm == 1) &
    (temp_alarm_reg5 == 5) & (temp_alarm_reg6 == 9);

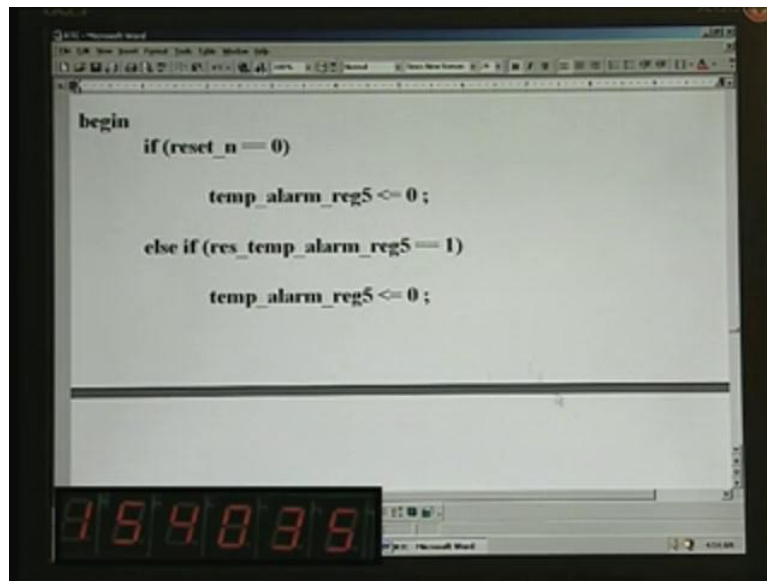
// Reset for 59.

assign temp_alarm_reg5_next = temp_alarm_reg5 + 1;

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

This is assign. This is also required for resetting for 59. Again, pre-increment.

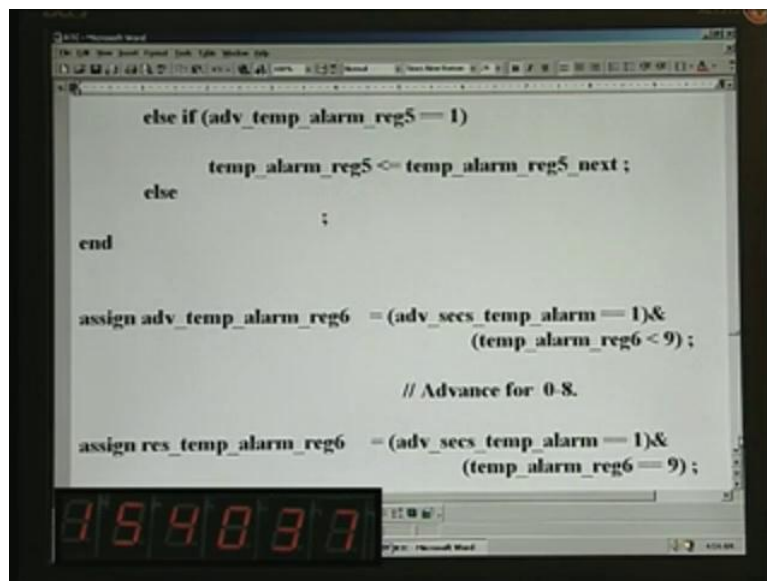
(Refer Slide Time: 33:09)



```
begin
  if (reset_n == 0)
    temp_alarm_reg5 <= 0 ;
  else if (res_temp_alarm_reg5 == 1)
    temp_alarm_reg5 <= 0 ;
```

The block for 5 is here, this is once again the same.

(Refer Slide Time: 33:10)



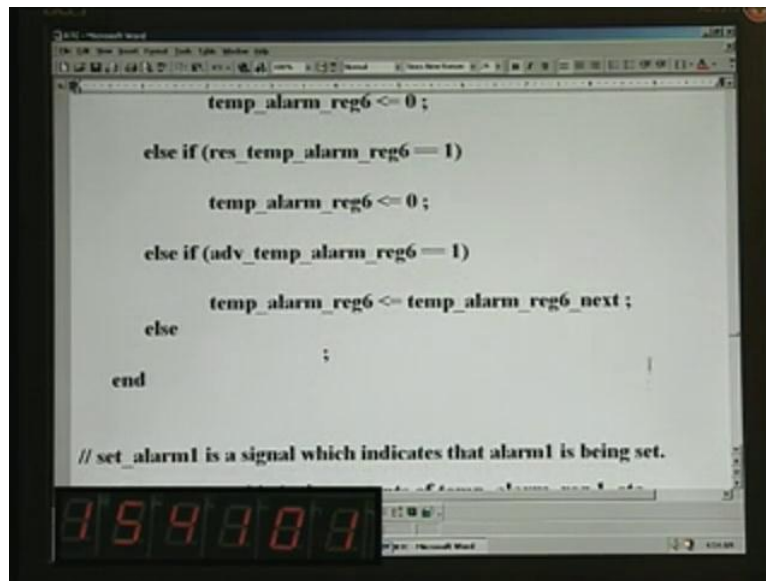
```
else if (adv_temp_alarm_reg5 == 1)
  temp_alarm_reg5 <= temp_alarm_reg5_next ;
else
  ;
end

assign adv_temp_alarm_reg6 = (adv_secs_temp_alarm == 1) &
  (temp_alarm_reg6 < 9) ;
// Advance for 0-8.

assign res_temp_alarm_reg6 = (adv_secs_temp_alarm == 1) &
  (temp_alarm_reg6 == 9) ;
```

The last digits for the seconds LSD is here. Then, advance, reset and pre-increment are there. Advance for 0 to 8 or reset for 9.

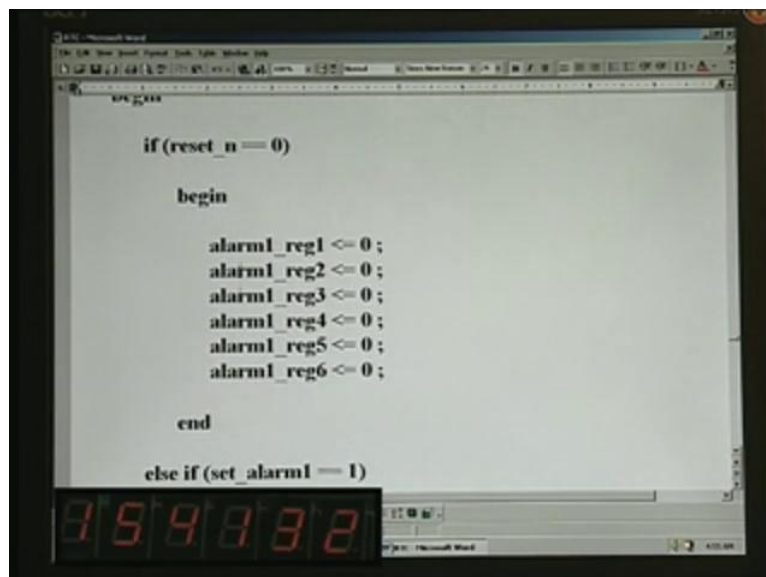
(Refer Slide Time: 33:30)



```
temp_alarm_reg6 <= 0 ;  
  
else if (res_temp_alarm_reg6 == 1)  
  
    temp_alarm_reg6 <= 0 ;  
  
else if (adv_temp_alarm_reg6 == 1)  
  
    temp_alarm_reg6 <= temp_alarm_reg6_next ;  
else  
    ;  
end  
  
// set_alarm1 is a signal which indicates that alarm1 is being set.
```

This is the block for **alarm_reg6** here. You assign it here and as usual, you have reset, etc. Next, we need to set alarm1. If you want to set alarm1, what you should do is you should set the stopwatch and **alarm read/set** in set mode with alarm1 set. We have already seen **set_alarm**. alarm1 must be in position 1. This is the de-bounced switch position. That is what we are calling as **set_alarm1**.

(Refer Slide Time: 34:05)

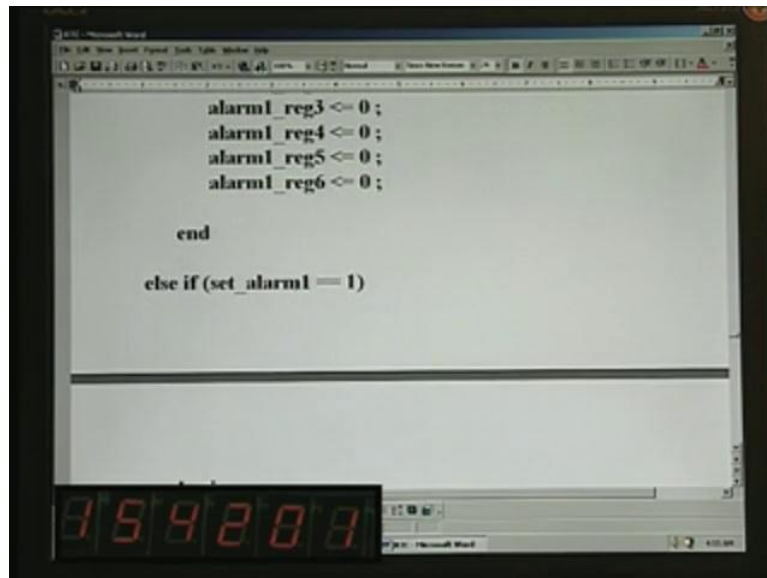


```
begin  
  
    if (reset_n == 0)  
  
        begin  
  
            alarm1_reg1 <= 0 ;  
            alarm1_reg2 <= 0 ;  
            alarm1_reg3 <= 0 ;  
            alarm1_reg4 <= 0 ;  
            alarm1_reg5 <= 0 ;  
            alarm1_reg6 <= 0 ;  
  
        end  
  
        else if (set_alarm1 == 1)
```

Once again, the block is there **for alarm1 and there are three alarms**. There are once again six displays, corresponding to the same order, this being the hours and so on. This is for resetting. We need separate registers for setting the independent alarms. We cannot have the

same temporary alarm. The temporary alarm is first set and then transferred here. That is what we are going to see later on.

(Refer Slide Time: 34:34)

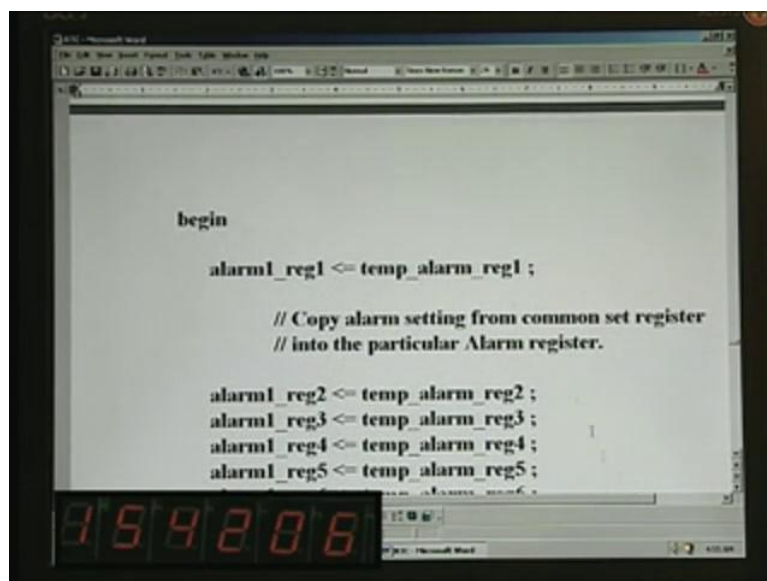


```
alarm1_reg3 <= 0 ;
alarm1_reg4 <= 0 ;
alarm1_reg5 <= 0 ;
alarm1_reg6 <= 0 ;

end

else if (set_alarm1 == 1)
```

A screenshot of a code editor window showing Verilog code. The code initializes alarm registers 3 through 6 to 0. Below the code, a red digital display shows the number 154203.



```
begin

alarm1_reg1 <= temp_alarm_reg1 ;

// Copy alarm setting from common set register
// into the particular Alarm register.

alarm1_reg2 <= temp_alarm_reg2 ;
alarm1_reg3 <= temp_alarm_reg3 ;
alarm1_reg4 <= temp_alarm_reg4 ;
alarm1_reg5 <= temp_alarm_reg5 ;
```

A screenshot of a code editor window showing Verilog code. The code transfers settings from temporary registers to alarm registers 1 through 5. Below the code, a red digital display shows the number 154206.

```
alarm1_reg6 <= temp_alarm_reg6 ;

end

end

assign set_alarm2 = (set_alarm == 1) & (alarm2 == 1) ;

always @ (posedge clk or negedge reset_n)

begin
```

If set_alarm1 is 1 (we have seen the condition), if it is set, what we should do is we have to set that temporary alarm. From temporary alarm, we are setting to the independent alarm. That is what we are doing here – copy alarm setting from common set register into the particular alarm register. It is precisely the same here for the six registers and set_alarm2 is similar to set_alarm1. The condition is clear except that alarm2 is now 1.

(Refer Slide Time: 35:01)

```
if (reset_n == 0)

begin

alarm2_reg1 <= 0 ;

alarm2_reg2 <= 0 ;

alarm2_reg3 <= 0 ;

alarm2_reg4 <= 0 ;

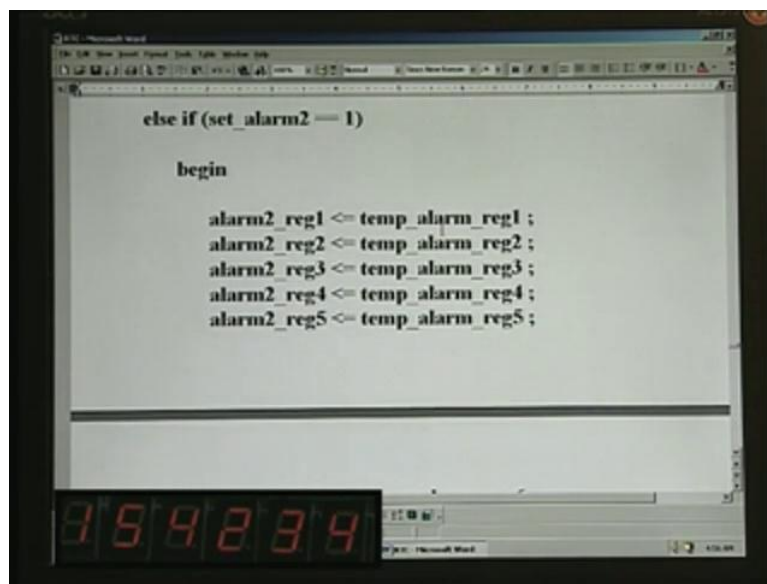
alarm2_reg5 <= 0 ;

alarm2_reg6 <= 0 ;

end
```

This is the block for realizing the same. Once again, power on reset here for alarm2.

(Refer Slide Time: 35:06)



```
else if (set_alarm2 == 1)

begin

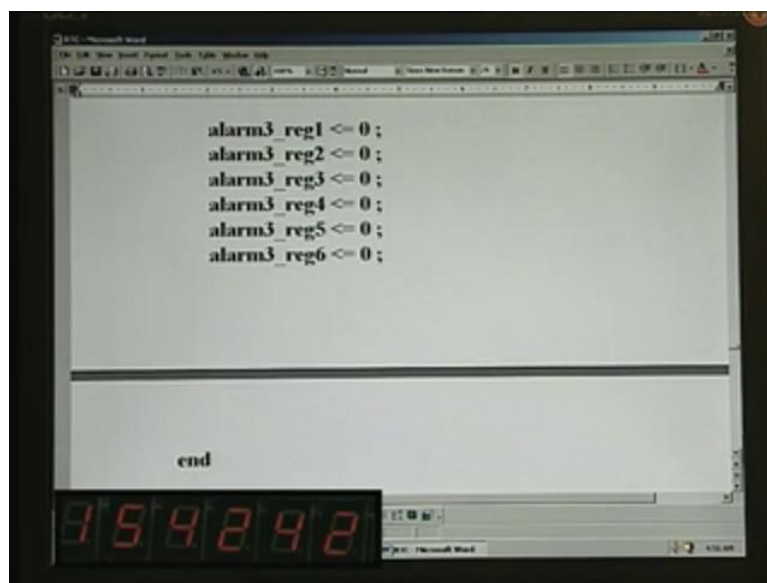
    alarm2_reg1 <= temp_alarm_reg1 ;
    alarm2_reg2 <= temp_alarm_reg2 ;
    alarm2_reg3 <= temp_alarm_reg3 ;
    alarm2_reg4 <= temp_alarm_reg4 ;
    alarm2_reg5 <= temp_alarm_reg5 ;


```

The screenshot shows a code editor window with the above code. Below the editor, a 7-segment display shows the number 154234 in red.

Here, temporary alarm is set to this, provided `set_alarm2` is there.

(Refer Slide Time: 35:13)



```
alarm3_reg1 <= 0 ;
alarm3_reg2 <= 0 ;
alarm3_reg3 <= 0 ;
alarm3_reg4 <= 0 ;
alarm3_reg5 <= 0 ;
alarm3_reg6 <= 0 ;

end
```

The screenshot shows a code editor window with the above code. Below the editor, a 7-segment display shows the number 154242 in red.

```
else if (set_alarm3 == 1)
begin
    alarm3_reg1 <= temp_alarm_reg1 ;
    alarm3_reg2 <= temp_alarm_reg2 ;
    alarm3_reg3 <= temp_alarm_reg3 ;
    alarm3_reg4 <= temp_alarm_reg4 ;
    alarm3_reg5 <= temp_alarm_reg5 ;
    alarm3_reg6 <= temp_alarm_reg6 ;

end

always @ (alarm1 or alarm2 or alarm3 or reset_n)
```

The same is the case for alarm3. It is precisely the same thing. Here, we are **setting the temporary**.

(Refer Slide Time: 35:26)

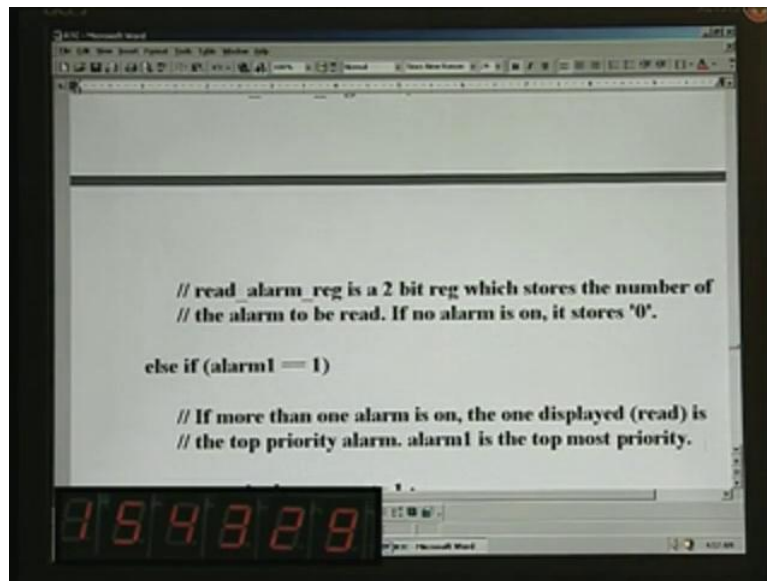
```
alarm3_reg5 <= temp_alarm_reg5 ;
alarm3_reg6 <= temp_alarm_reg6 ;

end

always @ (alarm1 or alarm2 or alarm3 or reset_n)
begin
    if (reset_n == 0)
        read_alarm_reg <= 0 ;
```

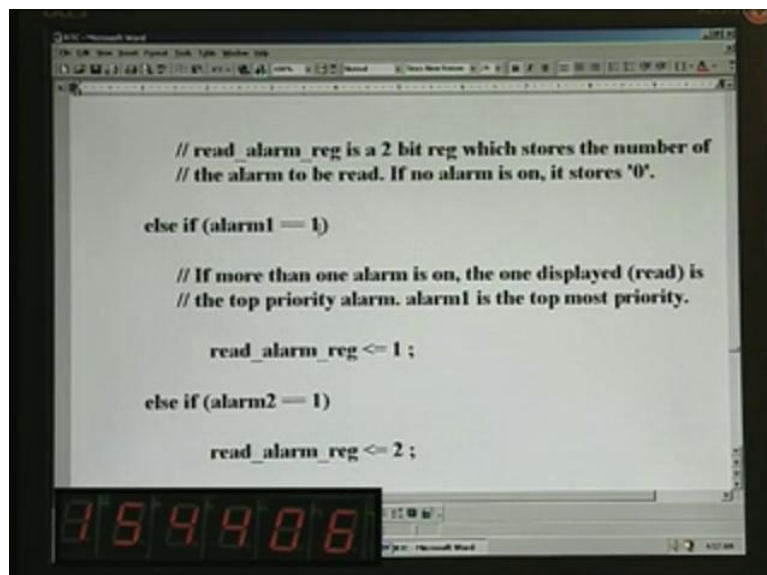
This is the block. I have lost track of this, let us have a look at this. **When alarm1 or any of the alarms**, then we need to take some action. What is that? We have to read the switch positions for the alarm. There are three switches, am I right? Please correct me if I am wrong. This is the power on reset. read_alarm_reg, there is a reg for keeping track of read alarm.

(Refer Slide Time: 36:02)



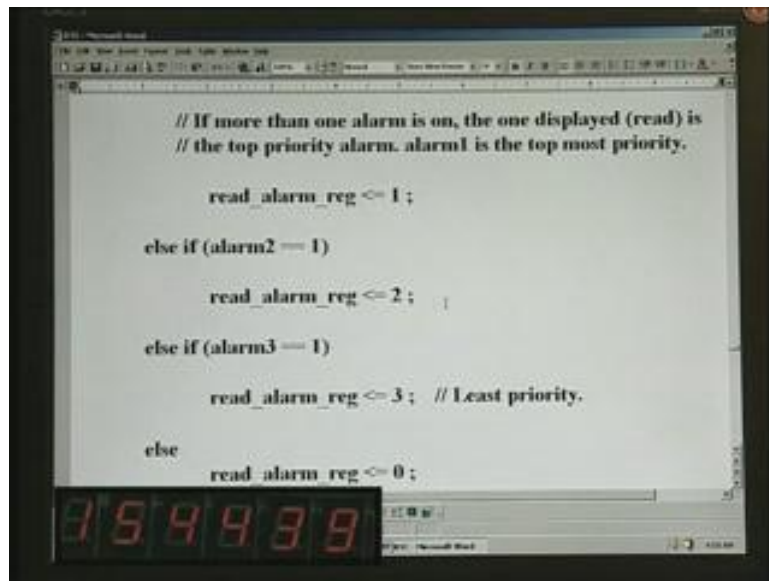
I will just read the comment. read_alarm_reg is the **two-bit register** that stores the number of the alarm to be read. If there are three alarms, you need to keep track of which alarm we are speaking. Reading means displaying. Each of the alarms has been assigned some unique number. For example, if it is 0, you need two bits to represent three alarms. If no alarm is on, it stores 0. So 0 corresponds to no alarm being set.

(Refer Slide Time: 36:37)



If alarm1 is 1, that is, the first alarm is set (this is the physical switch after de-bouncing), if more than one alarm is on, the one displayed (read) is the top priority alarm. alarm1 is the topmost priority and I hope there is no problem in this – earlier, you remember we corrected.

(Refer Slide Time: 37:06)



```
// If more than one alarm is on, the one displayed (read) is
// the top priority alarm. alarm1 is the top most priority.

read_alarm_reg <= 1;

else if (alarm2 == 1)

    read_alarm_reg <= 2;

else if (alarm3 == 1)

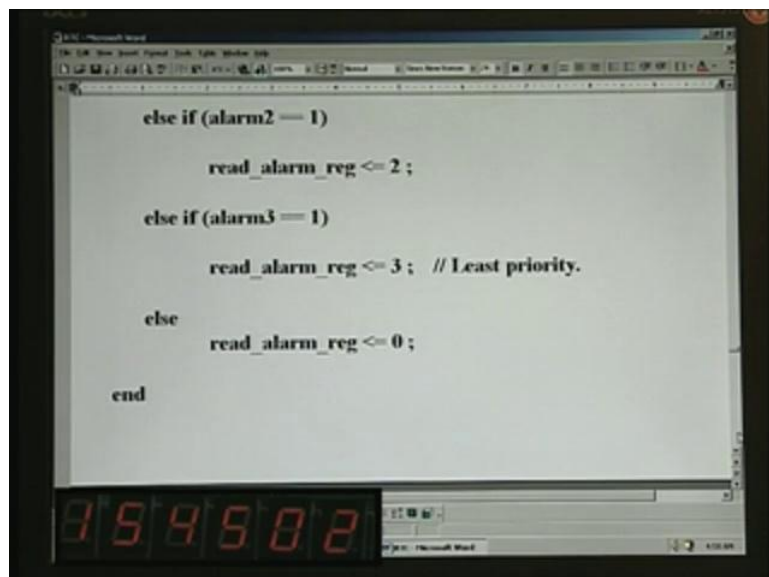
    read_alarm_reg <= 3; // Least priority.

else

    read_alarm_reg <= 0;
```

Here we need to set this register, **read_alarm_reg to 1** for that condition. That was for first alarm. If alarm2 is set on the other hand, this reg will be forced to the value 2. If alarm3 is encountered on the other hand, this will be set. Note that this is a priority **encoder**. This gives the topmost priority to alarm1 because that was the very first statement and that is how the priority is assigned. This is clear to you.

(Refer Slide Time: 37:35)



```
else if (alarm2 == 1)

    read_alarm_reg <= 2;

else if (alarm3 == 1)

    read_alarm_reg <= 3; // Least priority.

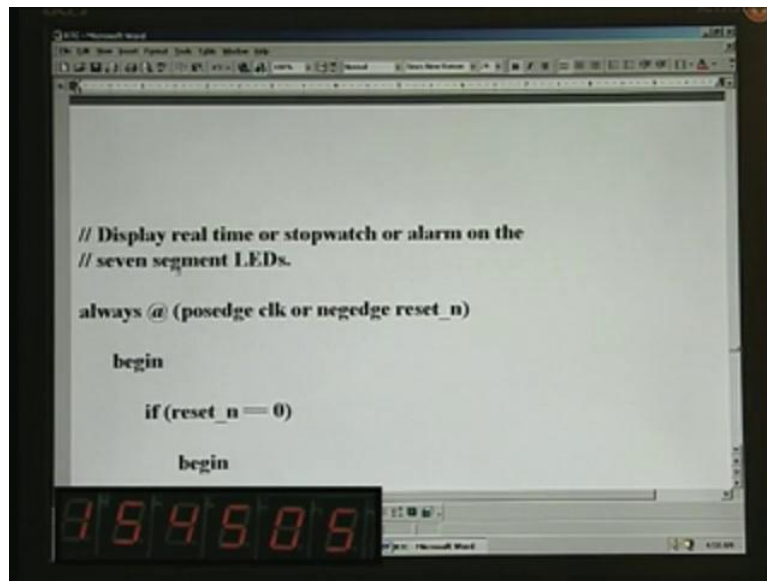
else

    read_alarm_reg <= 0;

end
```

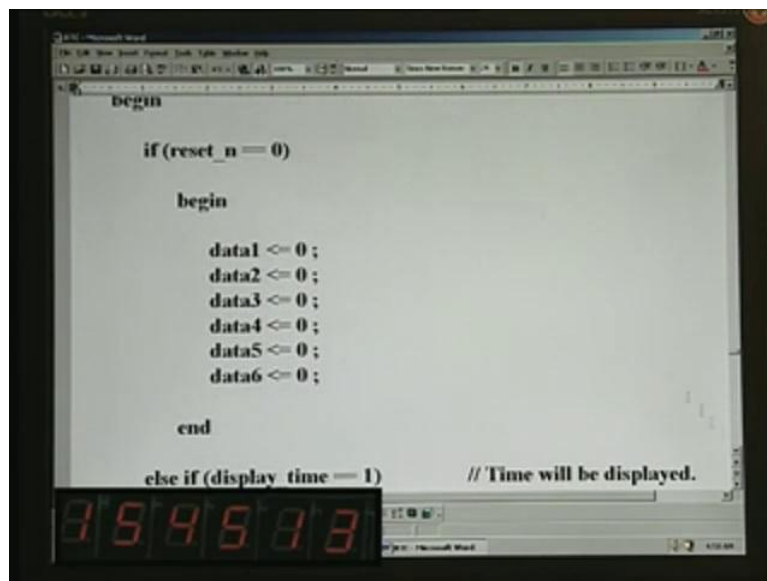
If none of this is met, simply make it 0, which implies that no switch has been set for that.

(Refer Slide Time: 37:38)



The next block is display real time or stopwatch or alarm on the seven-segment LEDs.

(Refer Slide Time: 37:46)



We have data1 through data6, which we saw in the simplified architecture earlier just for the display. We need to clear first.

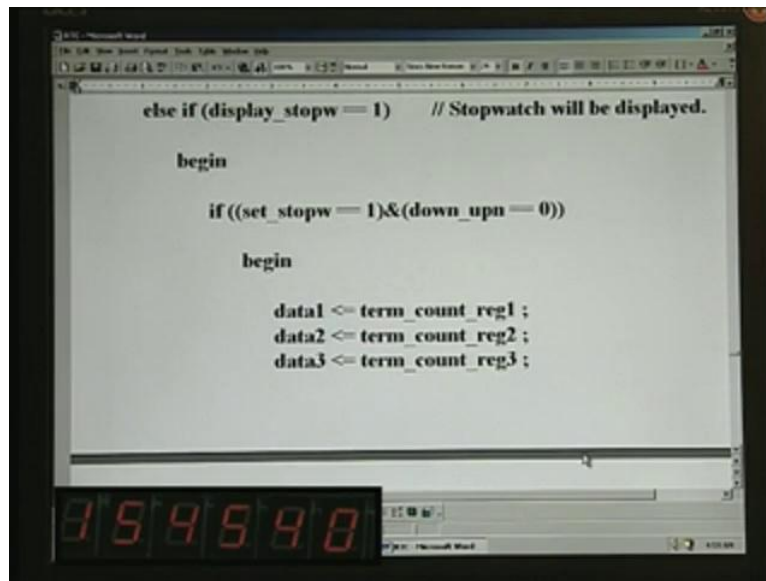
(Refer Slide Time: 38:00)

```
data6 <= 0 ;  
  
end  
  
else if (display_time == 1) // Time will be displayed.  
  
  
  
  
begin  
  
data1 <= cnt1_reg ;  
data2 <= cnt2_reg ;  
  
154528
```

```
  
  
begin  
  
data1 <= cnt1_reg ;  
data2 <= cnt2_reg ;  
data3 <= cnt3_reg ;  
data4 <= cnt4_reg ;  
data5 <= cnt5_reg ;  
data6 <= cnt6_reg ;  
  
end  
  
154534
```

If you are in display time mode, it implies that we need to display the actual running time. If you are in that mode, what we have to do is take **cnt1** through **cnt6**, which is the running counter and assign it to data1 through data6.

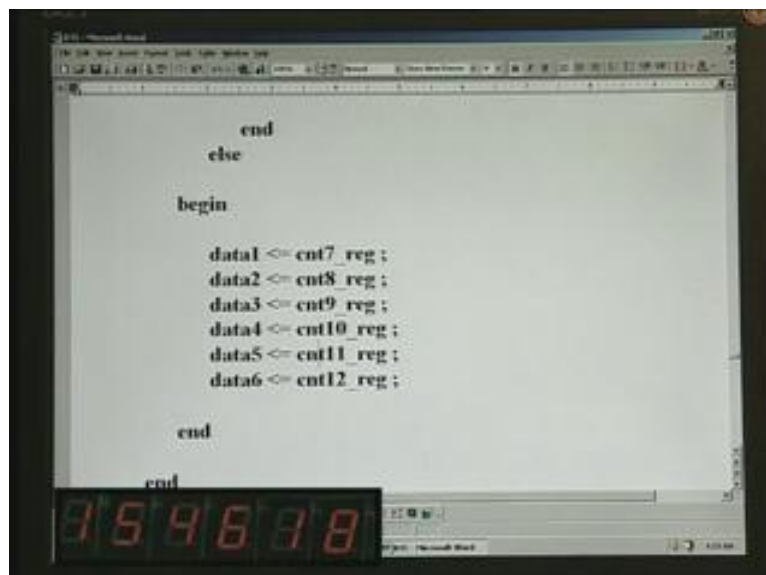
(Refer Slide Time: 38:14)



```
else if (display_stopw == 1) // Stopwatch will be displayed.
begin
    if ((set_stopw == 1) & (down_upn == 0))
    begin
        data1 <= term_count_reg1 ;
        data2 <= term_count_reg2 ;
        data3 <= term_count_reg3 ;
    end
end
```

Otherwise, if display stopwatch mode is set now, what you have to do is... Once again you have to check the condition whether it is in set stopwatch mode and also in down or up. It is in up mode. If it is in set stopwatch mode for up counter, then what you need to do is you have to take term_count_reg1 and then assign it and push it to the display. data1 through data6 are nothing other than the display.

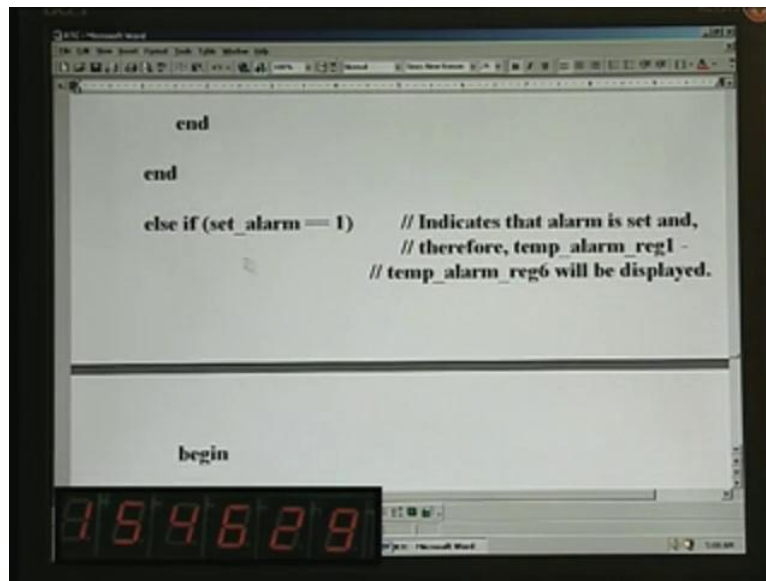
(Refer Slide Time: 38:50)



```
end
else
begin
    data1 <= cnt7_reg ;
    data2 <= cnt8_reg ;
    data3 <= cnt9_reg ;
    data4 <= cnt10_reg ;
    data5 <= cnt11_reg ;
    data6 <= cnt12_reg ;
end
end
```

Otherwise, what we have to do is this. This is for stopwatch. cnt7 through cnt12 is earmarked for the stopwatch. We need to push it to data1, which in turn will be displayed.

(Refer Slide Time: 39:02)

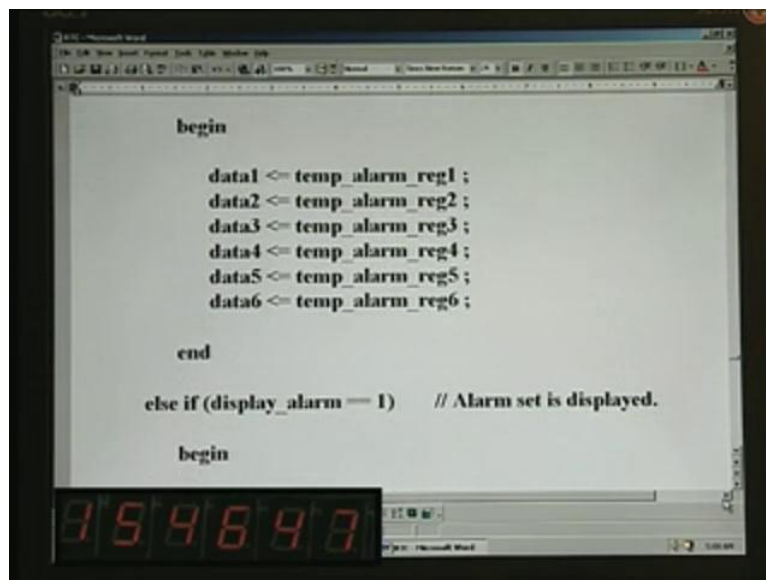


```
end
end
else if (set_alarm == 1) // Indicates that alarm is set and,
                        // therefore, temp_alarm_reg1 -
                        // temp_alarm_reg6 will be displayed.

begin
```

If set_alarm is 1, that indicates the alarm is set and therefore, temp_alarm_reg1 through reg6 will be displayed. This is what you want, because alarm is set and therefore you want to display the set alarm. This is the condition for set alarm.

(Refer Slide Time: 39:21)



```
begin
    data1 <= temp_alarm_reg1 ;
    data2 <= temp_alarm_reg2 ;
    data3 <= temp_alarm_reg3 ;
    data4 <= temp_alarm_reg4 ;
    data5 <= temp_alarm_reg5 ;
    data6 <= temp_alarm_reg6 ;

end
else if (display_alarm == 1) // Alarm set is displayed.

begin
```

Only here, we do the same thing and the temporary alarm is pushed to the display.

(Refer Slide Time: 39:35)

```
data4 <= temp_alarm_reg4 ;
data5 <= temp_alarm_reg5 ;
data6 <= temp_alarm_reg6 ;

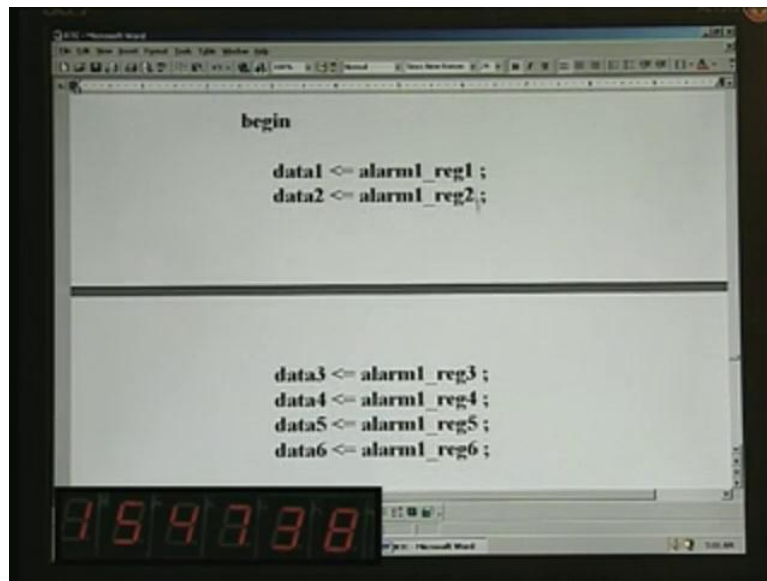
end

else if (display_alarm == 1) // Alarm set is displayed.
begin
    case (read_alarm_reg)
        1: // alarm1 is displayed.
            begin
```

```
            begin
                data1 <= alarm1_reg1 ;
                data2 <= alarm1_reg2 ;
            end
        end
    end
```

There is one more mode, many in fact inside this. If it is in display alarm mode, we have a few more clauses here. First, this alarm set is displayed. In this case, it would depend upon **read_alarm_reg**. You remember that corresponding to which alarm is set, we have given some number for that, 0 through 3. That is what is here. If it is 0, you do not have to do anything **hopefully**. It may come at the end, I think. If it is 1, it means alarm1. We need to display the alarm1.

(Refer Slide Time: 40:11)



```
begin

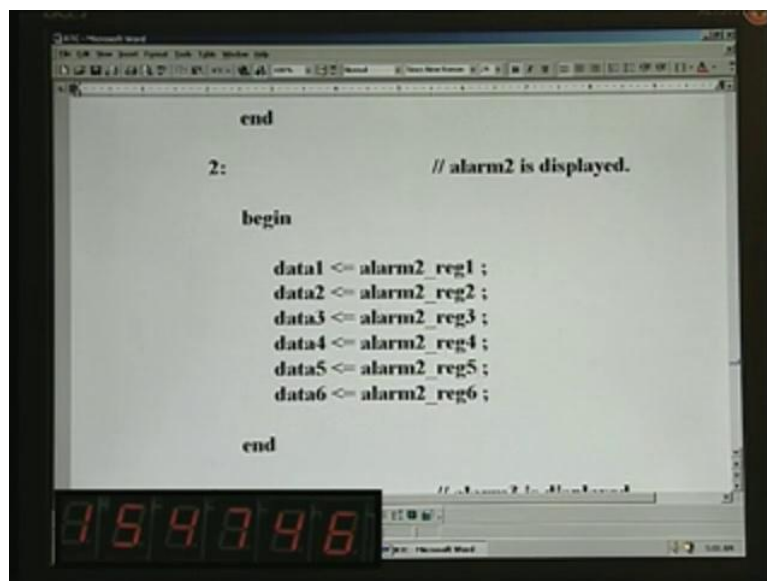
    data1 <= alarm1_reg1;
    data2 <= alarm1_reg2;

data3 <= alarm1_reg3;
data4 <= alarm1_reg4;
data5 <= alarm1_reg5;
data6 <= alarm1_reg6;
```

The screenshot shows a code editor window with the above Verilog code. Below the code, a 7-segment display is shown with the digits 1547338.

So all you have to do is push alarm1_reg1 through 6, which we have already set earlier, to the display.

(Refer Slide Time: 40:19)



```
end

2: // alarm2 is displayed.

begin

    data1 <= alarm2_reg1;
    data2 <= alarm2_reg2;
    data3 <= alarm2_reg3;
    data4 <= alarm2_reg4;
    data5 <= alarm2_reg5;
    data6 <= alarm2_reg6;


end
```

The screenshot shows a code editor window with the above Verilog code. Below the code, a 7-segment display is shown with the digits 1547348.

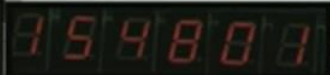
Otherwise, if it is 2, it means alarm2 and you do the same thing, but this time you do from alarm2 register and push it to the display.

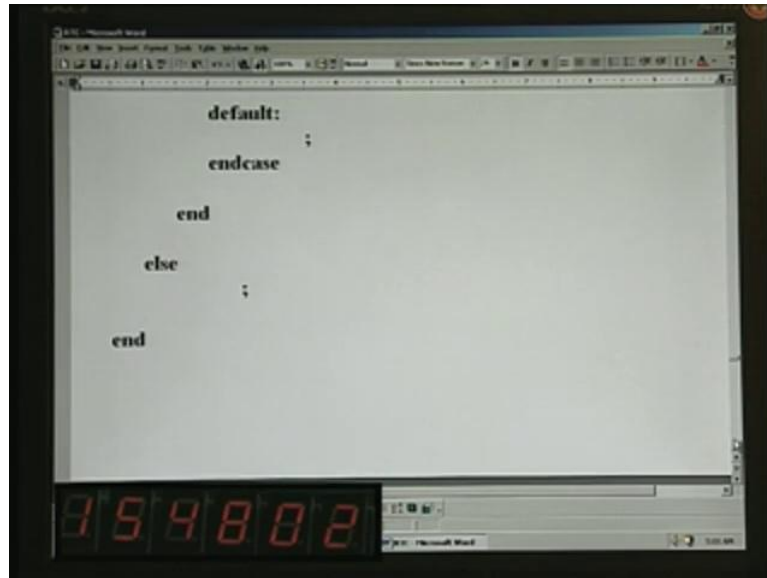
(Refer Slide Time: 40:30)

```
3: // alarm3 is displayed.  
  
begin  
    data1 <= alarm3_reg1 ;  
    data2 <= alarm3_reg2 ;  
    data3 <= alarm3_reg3 ;  
    data4 <= alarm3_reg4 ;  
    data5 <= alarm3_reg5 ;
```



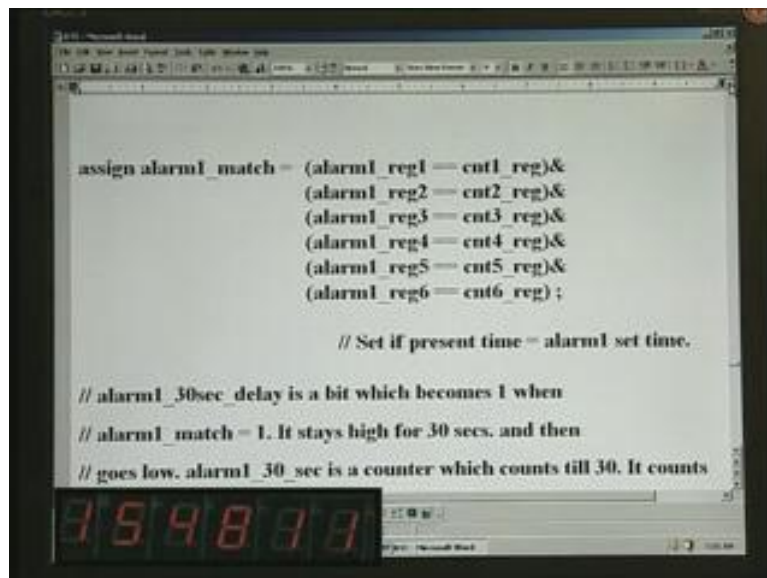
```
begin  
    data1 <= alarm3_reg1 ;  
    data2 <= alarm3_reg2 ;  
    data3 <= alarm3_reg3 ;  
    data4 <= alarm3_reg4 ;  
    data5 <= alarm3_reg5 ;  
    data6 <= alarm3_reg6 ;  
  
end
```





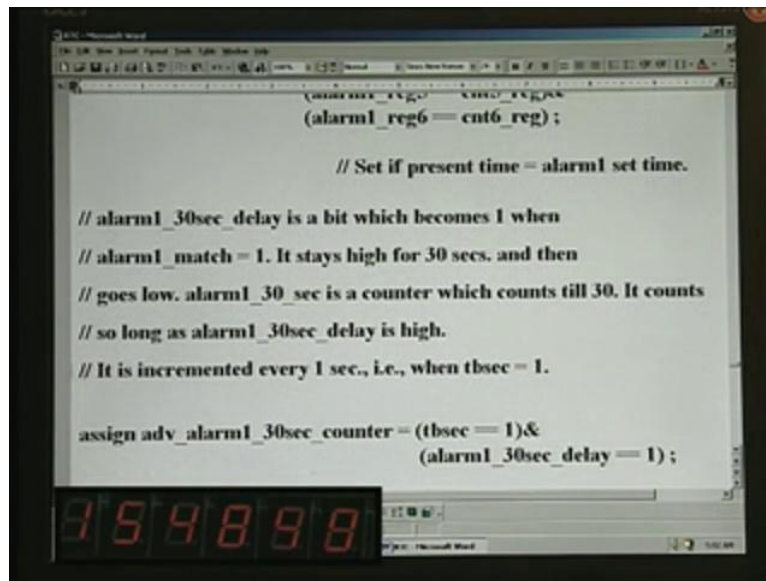
Otherwise, it may mean alarm3. Once again, you do from the alarm3 register onto the display. That ends that particular set alarm.

(Refer Slide Time: 40:42)



We have to go some more distance before we complete. What we need to do here is alarm1_match and for this, these are all the conditions. When alarm1 equal to cnt1 and alarm1_reg 2... this is the hours, then minutes, then seconds. When each of them equals the running counter, this is the runtime basically, then what should you do? The alarm has found its match, is it not. You have set an alarm time and when the running counter matches that set value.... This is precisely the statement is responsible for turning on the alarm. Set if present time is equal to alarm1 set time.

(Refer Slide Time: 41:23)



```
(alarm1_reg6 = cnt6_reg);

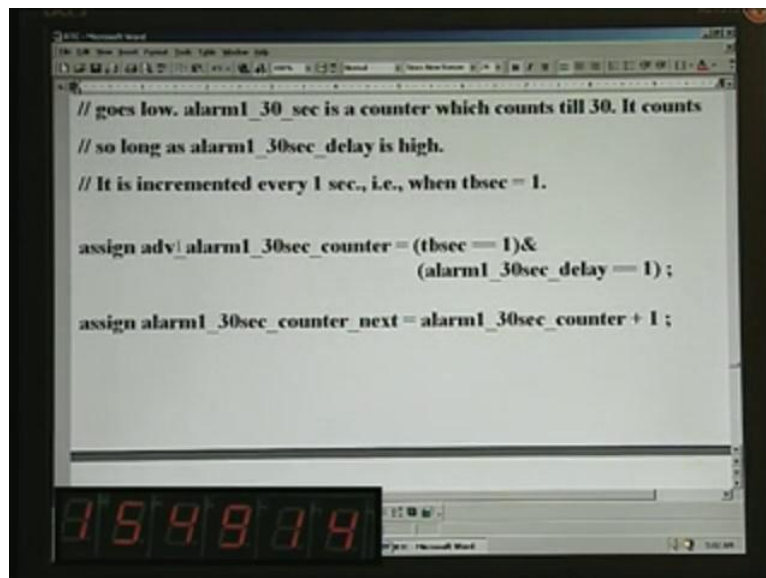
// Set if present time = alarm1 set time.

// alarm1_30sec_delay is a bit which becomes 1 when
// alarm1_match = 1. It stays high for 30 secs. and then
// goes low. alarm1_30_sec is a counter which counts till 30. It counts
// so long as alarm1_30sec_delay is high.
// It is incremented every 1 sec., i.e., when tbsec = 1.

assign adv_alarm1_30sec_counter = (tbsec == 1) &
    (alarm1_30sec_delay == 1);
```

You noticed that we also needed a 30 seconds buzzer activity. What we do is alarm1_30sec_delay is a bit that becomes 1 when alarm1_match is equal to 1. It stays high for 30 seconds and then goes low. alarm1_30 is a counter that counts till 30. It counts so long as alarm1_30sec_delay is high. It is incremented every 1 second when **time base** is equal to 1.

(Refer Slide Time: 41:48)



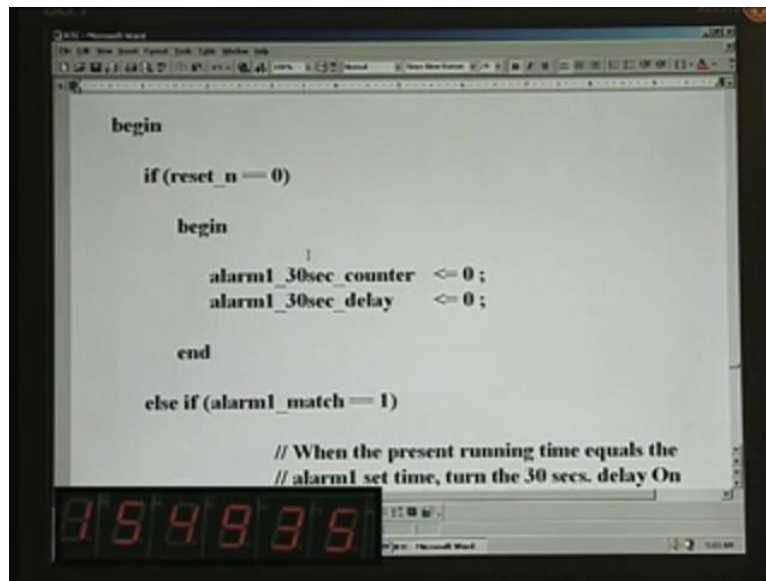
```
// goes low. alarm1_30_sec is a counter which counts till 30. It counts
// so long as alarm1_30sec_delay is high.
// It is incremented every 1 sec., i.e., when tbsec = 1.

assign adv_alarm1_30sec_counter = (tbsec == 1) &
    (alarm1_30sec_delay == 1);

assign alarm1_30sec_counter_next = alarm1_30sec_counter + 1;
```

We need a counter. For this alarm1_30sec_counter, advance is the signal that **we need to get**. This is for 1 second and when **alarm1_30sec_delay**. This is the advance counter for that.

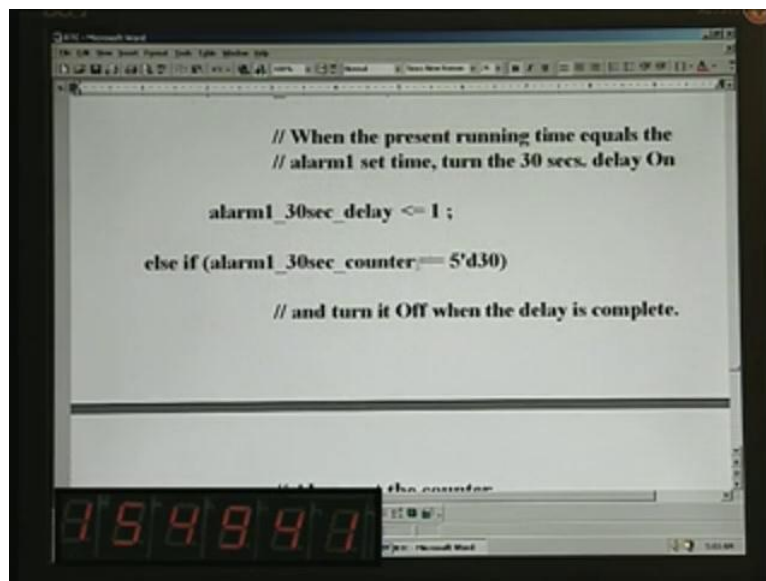
(Refer Slide Time: 42:06)



```
begin
    if (reset_n == 0)
        begin
            alarm1_30sec_counter <= 0;
            alarm1_30sec_delay <= 0;
        end
    else if (alarm1_match == 1)
        // When the present running time equals the
        // alarm1 set time, turn the 30 secs. delay On
```

Once again, this is the block that initializes these two – delay as well as the counter.

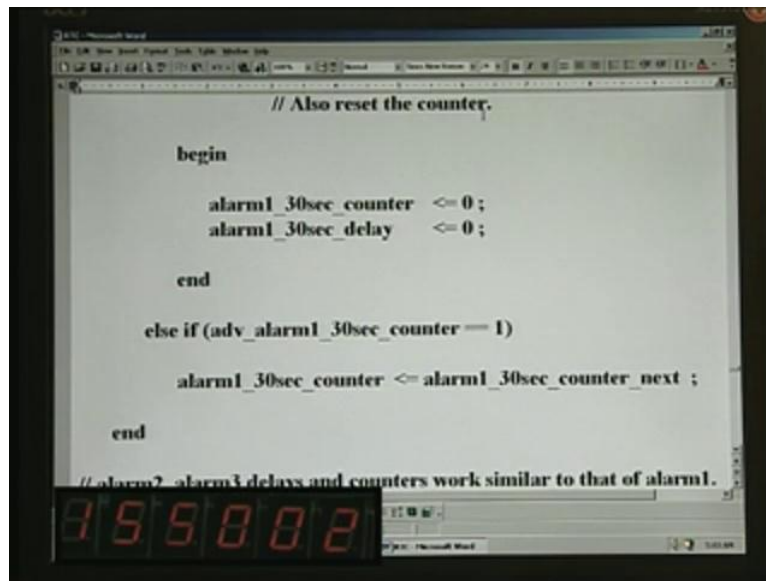
(Refer Slide Time: 42:15)



```
        // When the present running time equals the
        // alarm1 set time, turn the 30 secs. delay On
        alarm1_30sec_delay <= 1;
    else if (alarm1_30sec_counter == 5'd30)
        // and turn it Off when the delay is complete.
```

When alarm1 matches, only then we set the alarm1_30sec_delay1 as 1 here. Counter is different, delay is different. This is a single bit, whereas the counter is multi-bit here – 5 bit. When it matches with 30 seconds, then turn it off when the delay is complete. If delay is complete, that means turn it off.

(Refer Slide Time: 42:33)



```
// Also reset the counter.

begin

    alarm1_30sec_counter <= 0 ;
    alarm1_30sec_delay <= 0 ;

end

else if (adv_alarm1_30sec_counter == 1)

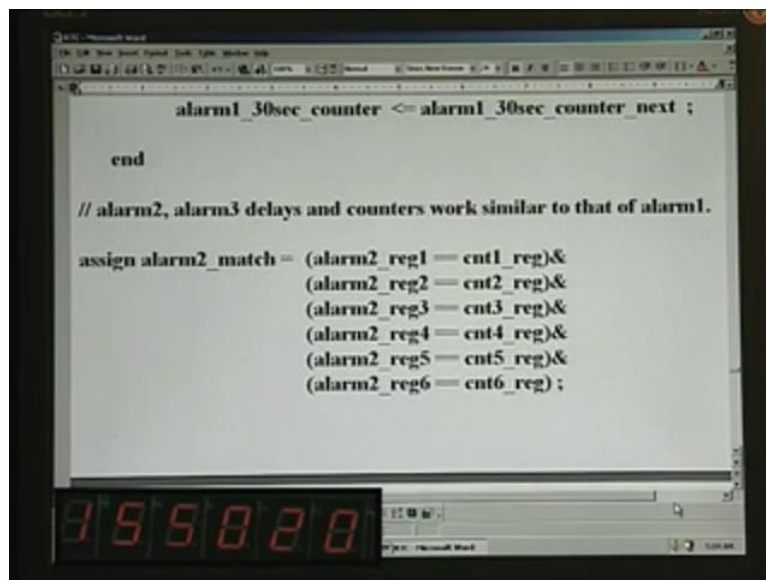
    alarm1_30sec_counter <= alarm1_30sec_counter_next ;

end

// alarm2, alarm3 delays and counters work similar to that of alarm1.
```

You should also not forget to reset the counter and that is what we are doing here. If alarm1_30sec_counter is 1, you have to only increment. We are counting that 1 second using that 30. **Every time tbsec we have used.** That is how you are counting for 30 seconds.

(Refer Slide Time: 42:54)



```
alarm1_30sec_counter <= alarm1_30sec_counter_next ;

end

// alarm2, alarm3 delays and counters work similar to that of alarm1.

assign alarm2_match = (alarm2_reg1 == cnt1_reg)&
    (alarm2_reg2 == cnt2_reg)&
    (alarm2_reg3 == cnt3_reg)&
    (alarm2_reg4 == cnt4_reg)&
    (alarm2_reg5 == cnt5_reg)&
    (alarm2_reg6 == cnt6_reg);
```

Similarly, alarm2 and alarm3 delays and counters work similar to that of alarm1. It is precisely the same **cnt1 to alarm2 this time.** This is for match.

(Refer Slide Time: 43:13)

```
alarm2_30sec_delay <= 1 ;

else if (alarm2_30sec_counter == 5'd30)
    // 30 secs. complete.

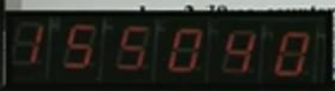
begin

    alarm2_30sec_counter <= 0 ;
    alarm2_30sec_delay <= 0 ;

end

else if (adv_alarm2_30sec_counter == 1)

    alarm2_30sec_counter <= alarm2_30sec_counter_next ;
```



```
begin

    alarm2_30sec_counter <= 0 ;
    alarm2_30sec_delay <= 0 ;


end

else if (adv_alarm2_30sec_counter == 1)

    alarm2_30sec_counter <= alarm2_30sec_counter_next ;

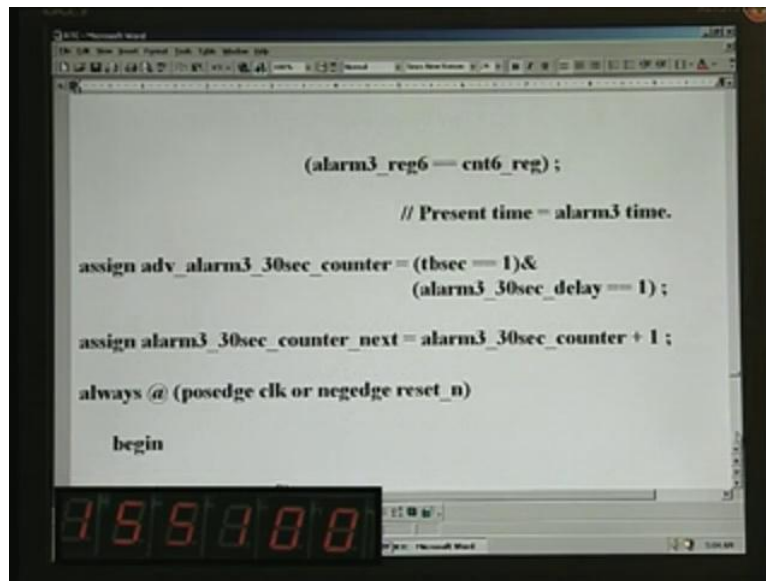
end

assign alarm3_match = (alarm3_reg1 == cnt1_reg) &
    (alarm3_reg2 == cnt2_reg) &
    (alarm3_reg3 == cnt3_reg) &
```



Then all this. When this match is 30 seconds, it does that. Then advance the counter and assign the pre-incremented value here. Then alarm3 match. When it is equal to the running value, then a match is found and **it then energizes** the signal.

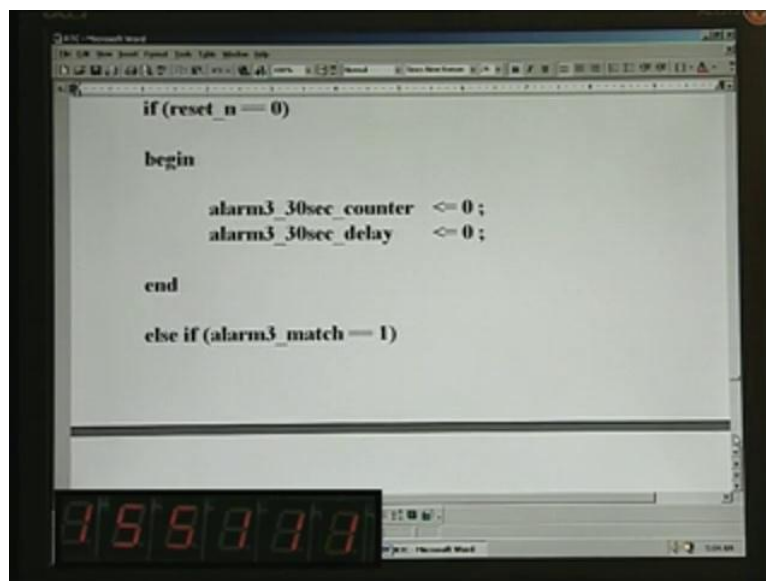
(Refer Slide Time: 43:34)



```
(alarm3_reg6 == cnt6_reg);  
  
// Present time = alarm3 time.  
assign adv_alarm3_30sec_counter = (tbsec == 1) &  
    (alarm3_30sec_delay == 1);  
assign alarm3_30sec_counter_next = alarm3_30sec_counter + 1;  
always @ (posedge clk or negedge reset_n)  
  
begin
```

Then you also need to advance the `alarm3_30sec_counter` corresponding to `alarm30`, which we have already seen earlier for 2 and 1. Then, pre-increment that counter.

(Refer Slide Time: 43:44)

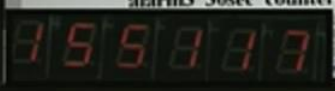


```
if (reset_n == 0)  
  
begin  
  
    alarm3_30sec_counter <= 0;  
    alarm3_30sec_delay <= 0;  
  
end  
  
else if (alarm3_match == 1)
```

```
alarm3_30sec_delay <= 1 ;

else if (alarm3_30sec_counter == 5'd30) // 30 secs. complete.
begin
    alarm3_30sec_counter <= 0 ;
    alarm3_30sec_delay <= 0 ;
end

else if (adv_alarm3_30sec_counter == 1)
    alarm3_30sec_counter <= alarm3_30sec_counter_next ;
```



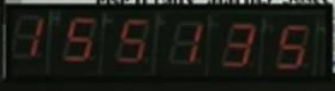
Once again, realize the same counter. There are two parameters once again. If the alarm matches, what you do is just set that particular thing to 1. delay is the one that keeps track of 30 seconds, whether it is over or not.

(Refer Slide Time: 44:09)

```
alarm3_30sec_delay <= 1 ;

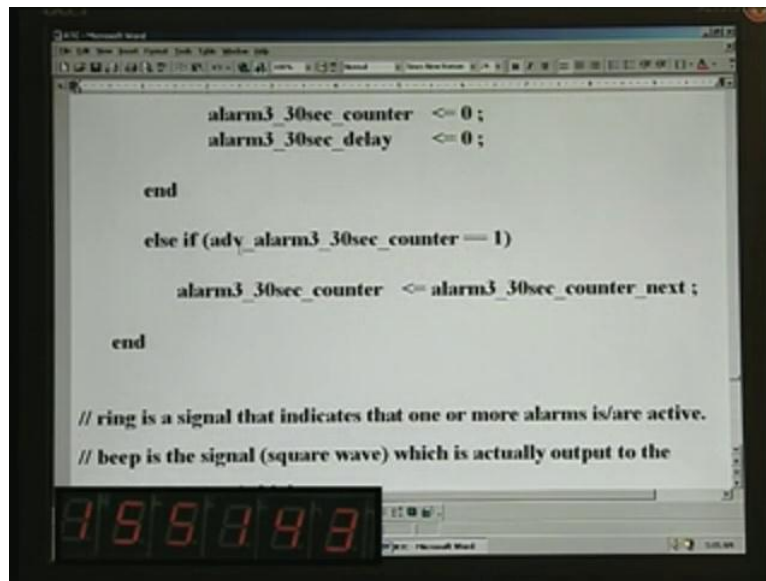
else if (alarm3_30sec_counter == 5'd30) // 30 secs. complete.
begin
    alarm3_30sec_counter <= 0 ;
    alarm3_30sec_delay <= 0 ;
end

else if (adv_alarm3_30sec_counter == 1)
    alarm3_30sec_counter <= alarm3_30sec_counter_next ;
```



Here, it implies, when this is equal to 30, it means 30 seconds complete and then, you have to just reset the counter – ready for the next event.

(Refer Slide Time: 44:17)



```
alarm3_30sec_counter <= 0 ;
alarm3_30sec_delay <= 0 ;

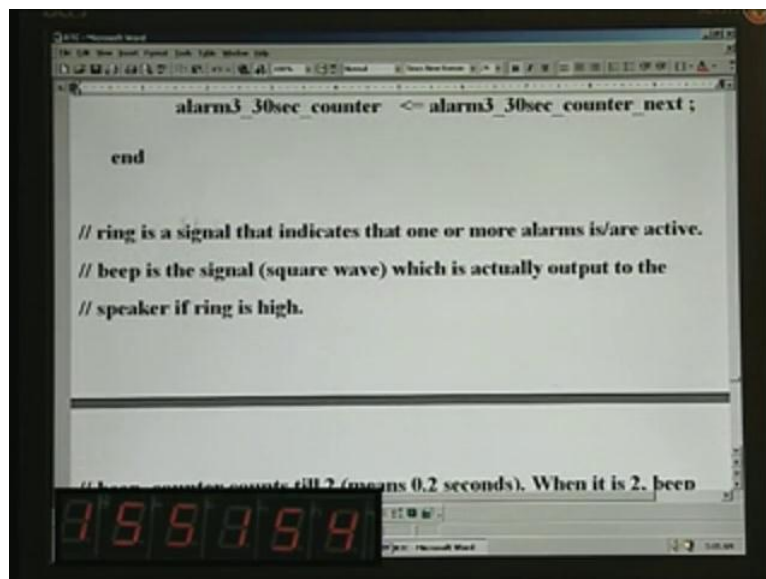
end

else if (adv_alarm3_30sec_counter == 1)
    alarm3_30sec_counter <= alarm3_30sec_counter_next ;
end

// ring is a signal that indicates that one or more alarms is/are active.
// beep is the signal (square wave) which is actually output to the
```

Then you advance the counter when this signal is 1 and assign the next.

(Refer Slide Time: 44:27)



```
alarm3_30sec_counter <= alarm3_30sec_counter_next ;

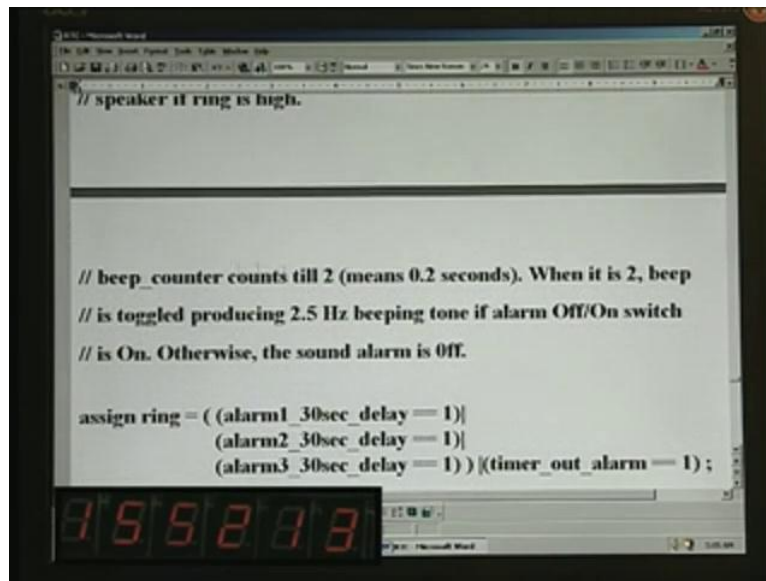
end

// ring is a signal that indicates that one or more alarms is/are active.
// beep is the signal (square wave) which is actually output to the
// speaker if ring is high.

// beep counter counts till 2 (means 0.2 seconds). When it is 2, beep
```

What is left is we need a ring signal, which is an intermediate signal. We actually need a beeping signal. Beep is the signal (it is a square wave that you want to create) that is actually output to the speaker if ring is high. Ring is an intermediate signal that indicates that one or more alarms are active.

(Refer Slide Time: 44:46)



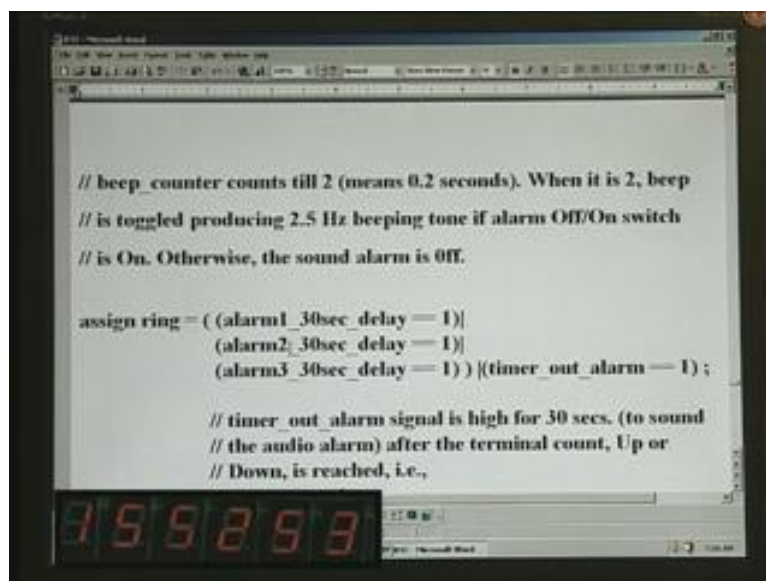
```
// speaker if ring is high.

// beep_counter counts till 2 (means 0.2 seconds). When it is 2, beep
// is toggled producing 2.5 Hz beeping tone if alarm Off/On switch
// is On. Otherwise, the sound alarm is Off.

assign ring = ( (alarm1_30sec_delay == 1)|
               (alarm2_30sec_delay == 1)|
               (alarm3_30sec_delay == 1) ) |(timer_out_alarm == 1);
```

beep_counter counts (we have a counter also) till 2 (means 0.2 seconds). When it is 2, a beep is toggled, producing 2.5 Hertz beeping tone. This is how you get a beeping tone, When you see the demo, you will clearly hear that; since we have not put a mike right now, we could not hear it. The beeping tone is heard if alarm off/on switch is on. There is another switch for off/on. If it is on, only then you can hear it. That does not mean activity is not going on. It does go on but you can hear only if it is in on mode. Otherwise, the sound alarm is off.

(Refer Slide Time: 45:22)



```
// beep_counter counts till 2 (means 0.2 seconds). When it is 2, beep
// is toggled producing 2.5 Hz beeping tone if alarm Off/On switch
// is On. Otherwise, the sound alarm is Off.

assign ring = ( (alarm1_30sec_delay == 1)|
               (alarm2_30sec_delay == 1)|
               (alarm3_30sec_delay == 1) ) |(timer_out_alarm == 1);

// timer_out_alarm signal is high for 30 secs. (to sound
// the audio alarm) after the terminal count, Up or
// Down, is reached, i.e.,
```

```

// is On. Otherwise, the sound alarm is off.

assign ring = ( (alarm1_30sec_delay == 1)|
               (alarm2_30sec_delay == 1)|
               (alarm3_30sec_delay == 1) )|(timer_out_alarm == 1);

// timer_out_alarm signal is high for 30 secs. (to sound
// the audio alarm) after the terminal count, Up or
// Down, is reached, i.e.,
// timer_out = 1.

assign beep_counter_next = beep_counter + 1;

always @ (posedge clk or negedge reset_n)

```

Then assign ring (ring is the signal). When alarm1_30sec_delay or the second or third delay plus timer_out_alarm, which we have already seen, is 1, then only a ring will be energized. The timer_out_alarm_signal is high for 30 seconds (to sound the audio alarm) after the terminal count, up or down is reached, that is, timer_out is equal to 1. Assign beep counter next – this is the pre increment.

(Refer Slide Time: 45:47)

```

begin

    beep_counter    <= 0;
    beep            <= 0;

end

else if (ring == 0) // ring = 0 means no alarm is active.

begin

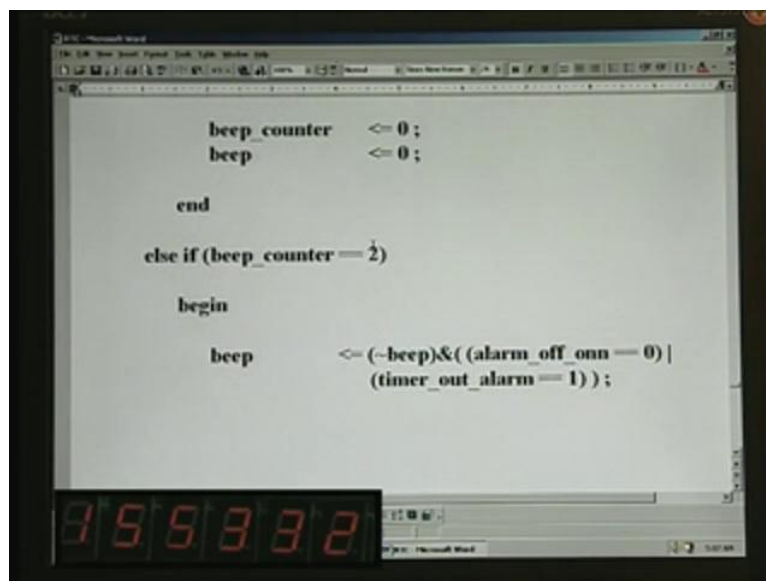
    beep_counter    <= 0;
    beep            <= 0;

end

```

The block for realizing the same is here. We have a beep counter here and we also have the beep output here. We initialize these. ring = 0 means no alarm is active. If ring is 0, once again you do the same thing.

(Refer Slide Time: 46:06)



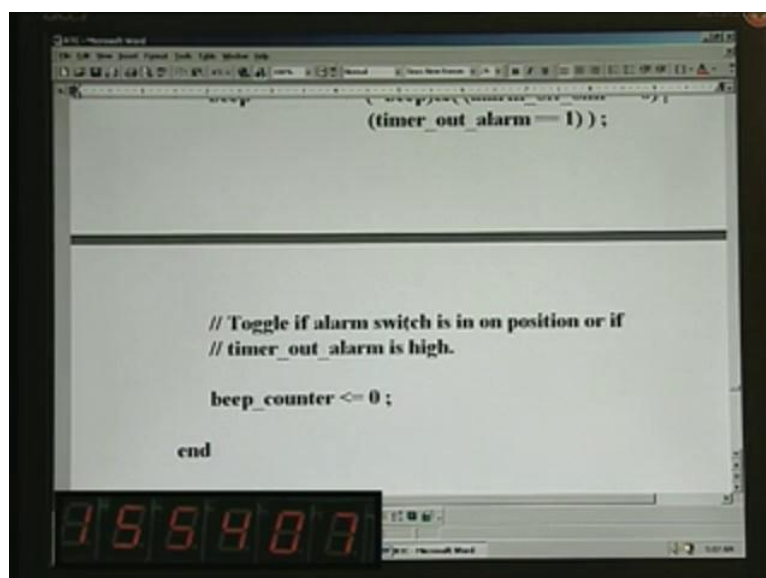
```
beep_counter <= 0 ;
beep <= 0 ;

end

else if (beep_counter == 2)
begin
beep <= (~beep) & ((alarm_off_on == 0) |
(timer_out_alarm == 1) );
```

If `beep_counter = 2`, what you have to do is the beep must be inverted and fed back only in `alarm_off_on`. That means the alarm is in the on condition and only then, you need to beep. Either this condition or `timer_out_alarm` is also 1. When either of the two conditions is met, what do you do is you assign it to beep. When this condition is met, it will assign beep equal to 1. It will set the beep.

(Refer Slide Time: 46:41)



```
(timer_out_alarm == 1) );

// Toggle if alarm switch is in on position or if
// timer_out_alarm is high.

beep_counter <= 0 ;

end
```

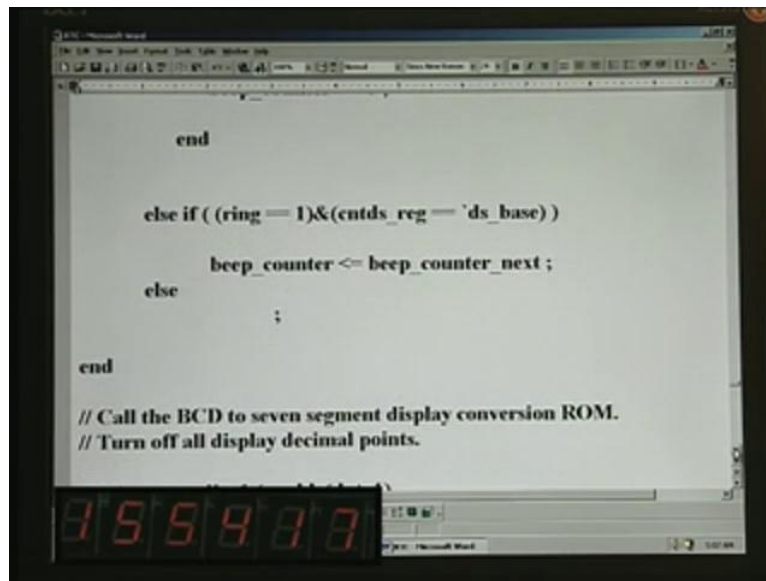
Toggle if alarm switch is in the on position or if `timer_out_alarm` is high. Then, you can `reset counter 0`.

(Refer Slide Time: 46:51)

```
end

else if ((ring == 1) & (cntds_reg == `ds_base))
    beep_counter <= beep_counter_next;
else
    ;
end

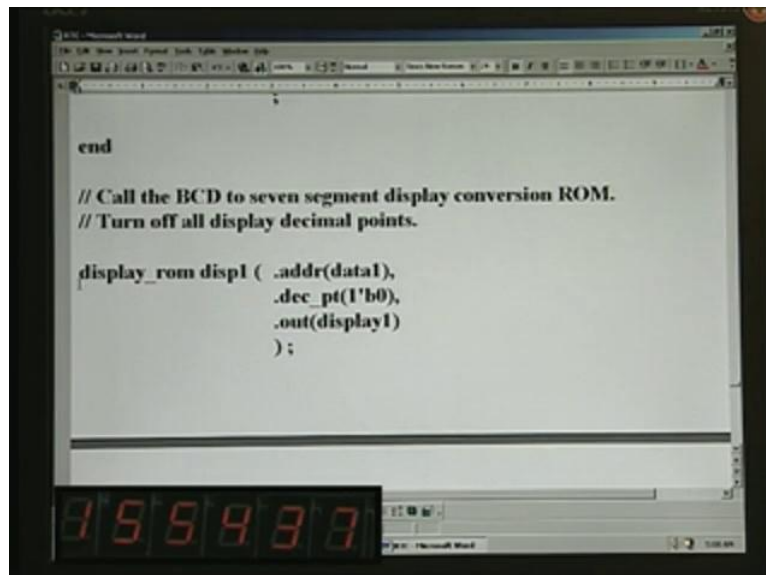
// Call the BCD to seven segment display conversion ROM.
// Turn off all display decimal points.
```



```
end

// Call the BCD to seven segment display conversion ROM.
// Turn off all display decimal points.

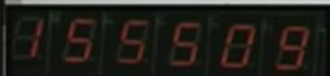
display_rom disp1 ( .addr(data1),
                    .dec_pt(1'b0),
                    .out(display1)
                    );
```



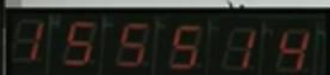
Otherwise if $ring = 1$ and this is decisecond once again, then only do the incrementing here. We are going to complete the design. Call the BCD to seven-segment display conversion ROM. This is the display ROM. We need to call that here by using this and this is the instantiation of the same. It is nothing but a ROM table. There is an address input, which can be given as data1, data2 and so on, so that **this will straightaway** do the code conversion from BCD to seven-segment display and **you can output that output, which is display1**. If you want to independently control the decimal point, you can put 1 here.

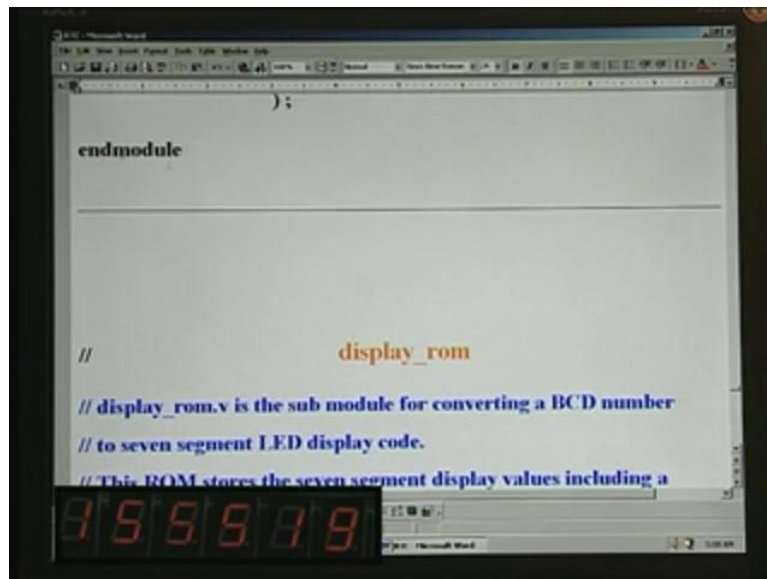
(Refer Slide Time: 47:43)

```
display_rom disp2 ( .addr(data2),  
                  .dec_pt(1'b0),  
                  .out(display2)  
                  );  
  
display_rom disp3 ( .addr(data3),  
                  .dec_pt(1'b0),  
                  .out(display3)  
                  );  
  
display_rom disp4 ( .addr(data4),  
                  .dec_pt(1'b0),  
                  .out(display4)  
                  );
```



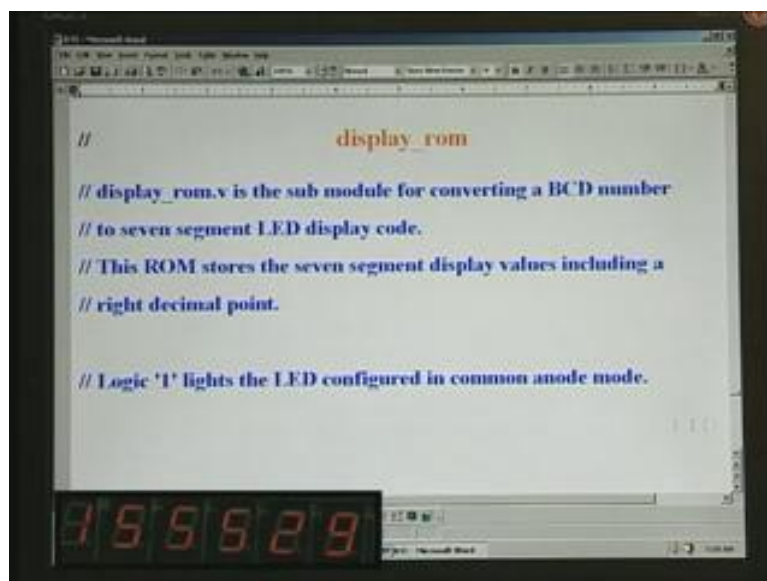
```
);  
display_rom disp5 ( .addr(data5),  
                  .dec_pt(1'b0),  
                  .out(display5)  
                  );  
  
display_rom disp6 ( .addr(data6),  
                  .dec_pt(1'b0),  
                  .out(display6)  
                  );
```





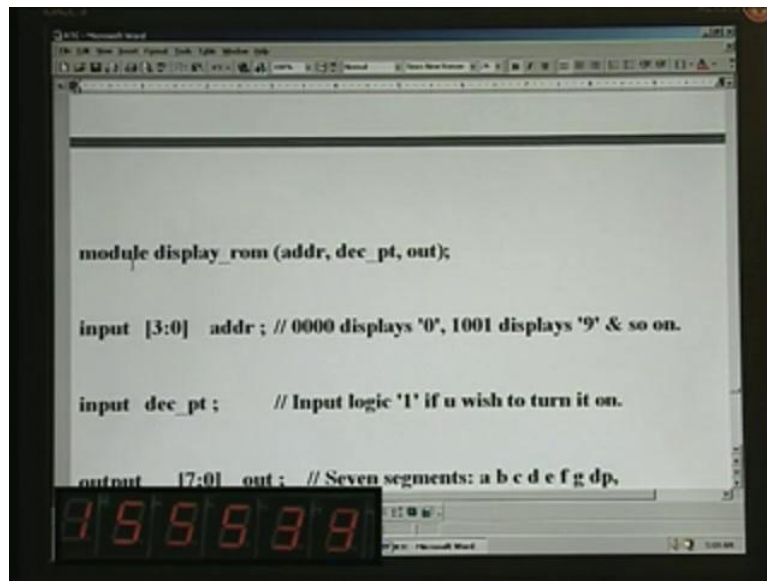
Like this, we have display1 through display6 here. Each time, data 1, data 2 – different data sent here and that is the reason why we get a display. That ends the design.

(Refer Slide Time: 47:51)



The submodule we have used for the display ROM is here. **display_rom.v** is the submodule for conversion here. It is a ROM table. Logic 1 is meant for turning on the segment. It is a common anode mode.

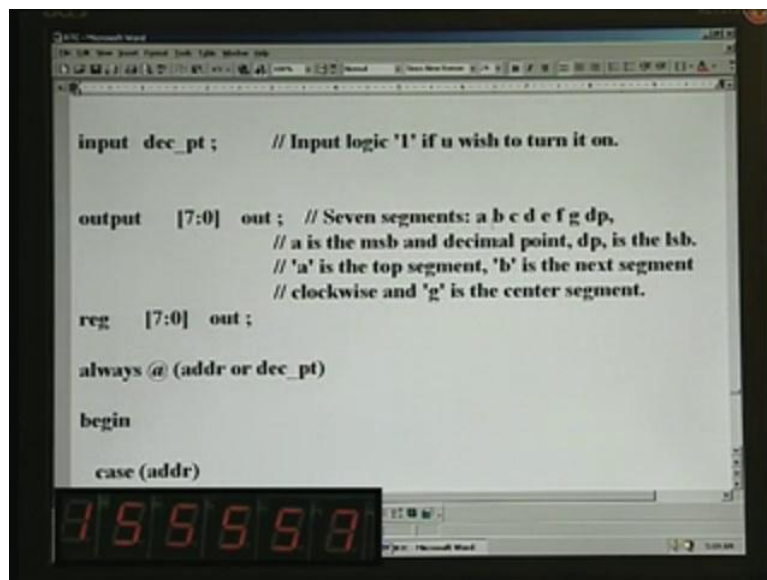
(Refer Slide Time: 48:07)



```
module display_rom (addr, dec_pt, out);  
  
input [3:0] addr; // 0000 displays '0', 1001 displays '9' & so on.  
  
input dec_pt; // Input logic '1' if u wish to turn it on.  
  
output [7:0] out; // Seven segments: a b c d e f g dp,  
  
endmodule
```

This is the declaration of the module. Address, decimal point and out are the I/Os here. We need to declare all the inputs, 0000 displays 0, 1001 displays 9 and so on. All other things are illegal values as far as the display is concerned. If you want decimal input, you have 1 here.

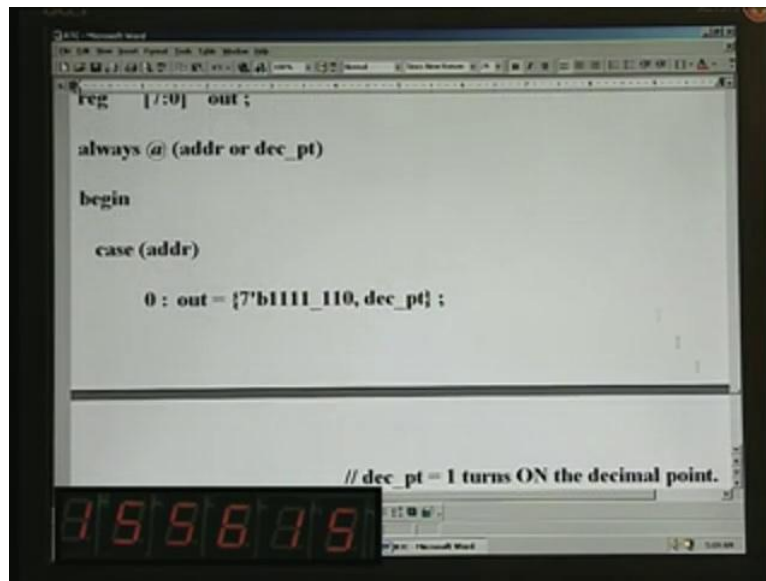
(Refer Slide Time: 48:31)



```
input dec_pt; // Input logic '1' if u wish to turn it on.  
  
output [7:0] out; // Seven segments: a b c d e f g dp,  
// a is the msb and decimal point, dp, is the lsb.  
// 'a' is the top segment, 'b' is the next segment  
// clockwise and 'g' is the center segment.  
reg [7:0] out;  
  
always @ (addr or dec_pt)  
  
begin  
case (addr)
```

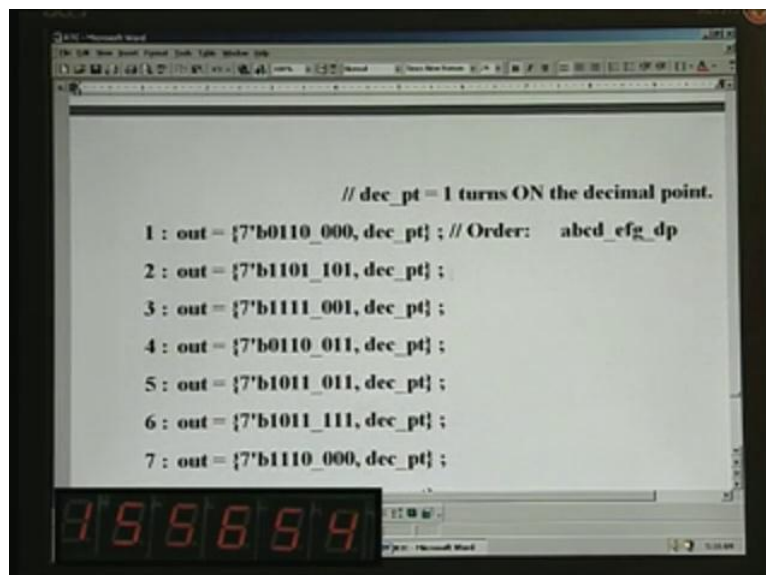
For output, you need eight bits. There are seven segments a, b, c, d, e, f and g – this is the order for decimal point. a is the MSB of this and dp, the decimal point, is the LSD. a is the top segment, b is the next segment clockwise and g is the center segment. You need to declare out as reg.

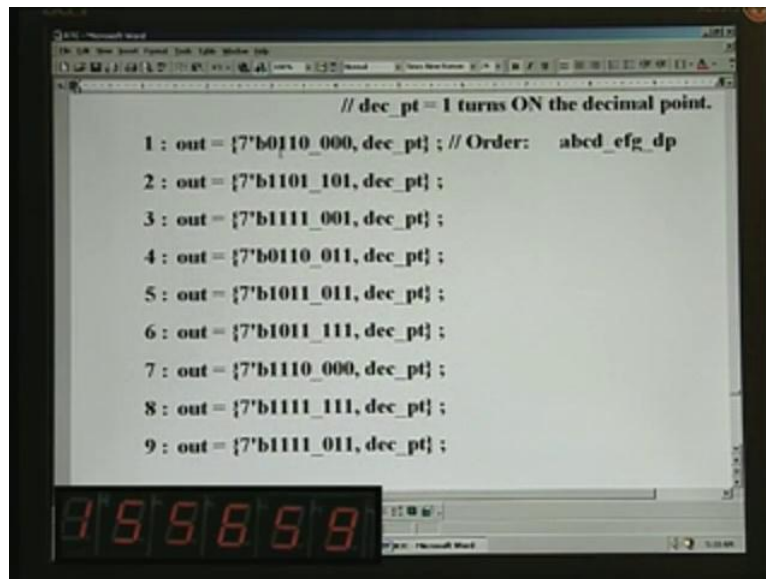
(Refer Slide Time: 48:49)



This is a **case statement using only** combinational logic. Case depends upon the address. Address is what you give as input. It needs to be converted from BCD to seven segments. This can go right from 0 through 9. For each, the output will be different and that is assigned here. For example, all 1s except for the last, **0 means...** a segment is the first, b, c and so on. This is g, is center segment and decimal point here. If you put 1, it will turn on.

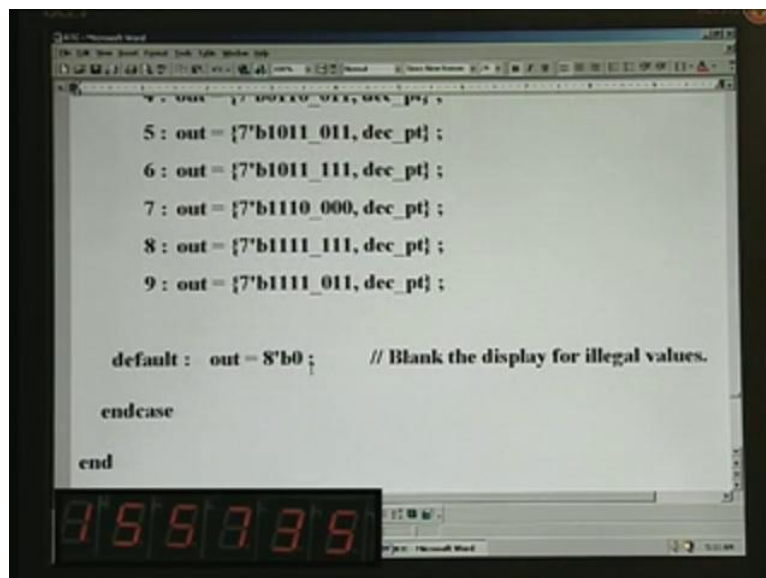
(Refer Slide Time: 49:28)





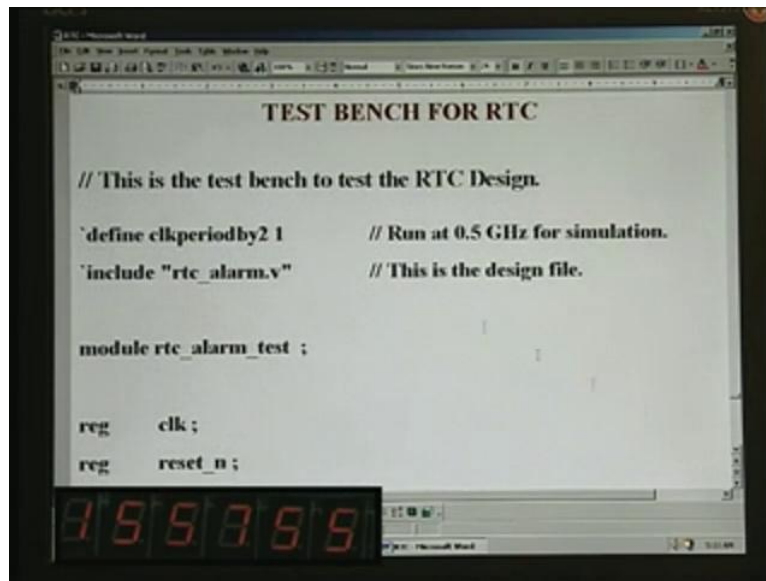
That is what is meant here. 1, 2, 3 up to 9 are precisely the same except that the value is changed here. You can just analyze and convince yourself that it is working. Let us take an example. For example, this is 9. What you should have is, it will go like this and all segments are there except for this segment. What is this segment? The center will be g, this will be f and this will be e. So e segment must be 0, 0 turns off, so a, b, c, d, e precisely. You can analyze like this and convince yourself that it really works.

(Refer Slide Time: 50:09)



We just put the default as 0, which will blank the display. For any value other than 0 through 9, it will blank the display. If you give any illegal data, it will blank the display. This ends the design.

(Refer Slide Time: 50:25)



```
TEST BENCH FOR RTC

// This is the test bench to test the RTC Design.

`define clkperiodby2 1      // Run at 0.5 GHz for simulation.
`include "rtc_alarm.v"     // This is the design file.

module rtc_alarm_test ;

reg    clk ;
reg    reset_n ;
```

We have a test bench for this real-time clock. This is the test bench. If you want, you can have a question here, so that we can wind up. We will take up the test bench a little later. If you have any specific question, you can ask me. What is left out is the test bench and it is going to be only a small thing, followed by the waveforms for this. This test bench is not going to be very elaborate – it is going to be very short. We have already entered the maze and come out of it – all intertwined counters. Naturally, we cannot write a very elaborate test bench. If you have any specific question, you can ask me. Do you have any?

Question (by Student): Why exactly do we do this debouncing, sir?

Debouncing of the switch?

Question (by Student) : Why do we have to do it?

If we do not debounce, multiple inputs will be given. It may not be all that necessary for everything. For example, you want to push a button. What do you expect? If I push once, it should increment only once, not hundred times. It will behave erratically if we do not debounce – it will give you multiple pulses because we are counting the pulse applied; it will be erratic. You expect just one increment – 1 must go to 2, but you will be annoyed to see 1 going to 8 or 10. That is the purpose of debouncing. Is your question answered? Do you have any more questions? Thank you.

(Refer Slide Time: 52:22)

