Digital VLSI System Design

Dr. S. Ramachandran

Department of Electrical Engineering

Indian Institute of Technology, Madras

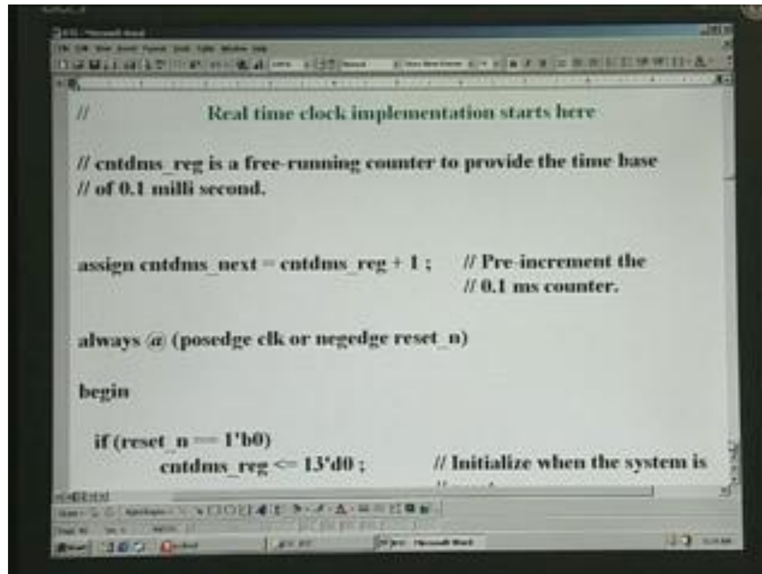
Lecture No. 52

System Design Examples using FPGA Board

(Refer Slide Time: 01:39)

We were looking at the implementation of a real-time clock using Verilog. We will continue the same. Before this, we have seen how to declare the various signals. Right now, we are about to start with the real-time clock implementation from here.

(Refer Slide Time: 02:16)



As pointed before, we have a deci-millisecond register, which is to keep track of 0.1 millisecond. This is the basic time base provided by this counter going by the name cntdms. Even a counter is a register. So it has the symbol _reg here. To start with, we have put an advance increment here. This is the actual register. The incremented value is put in this and it will be reckoned only when the positive edge of the clock strikes. That is done here with an always block, which you already familiar with.

(Refer Slide Time: 03:00)

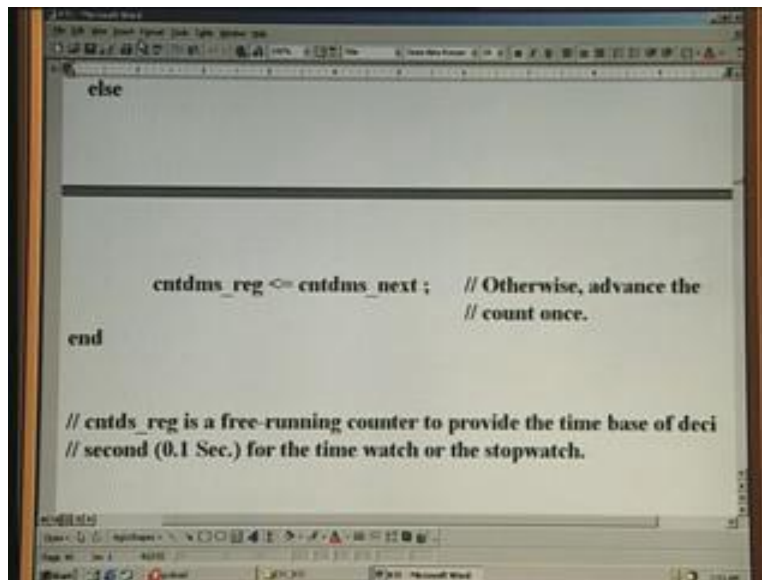
We have the block commencing here and this is for the reset condition. Whenever system reset is encountered, you need to reset this particular running counter, which is 13 bits in width. Look at the comment.

(Refer Slide Time: 03:16)

If this is satisfied, it does this operation; otherwise, it takes this. If it finds a match, that is, when the running counter cntdms (dms means deci-milliseconds) equals the set value (we have set this

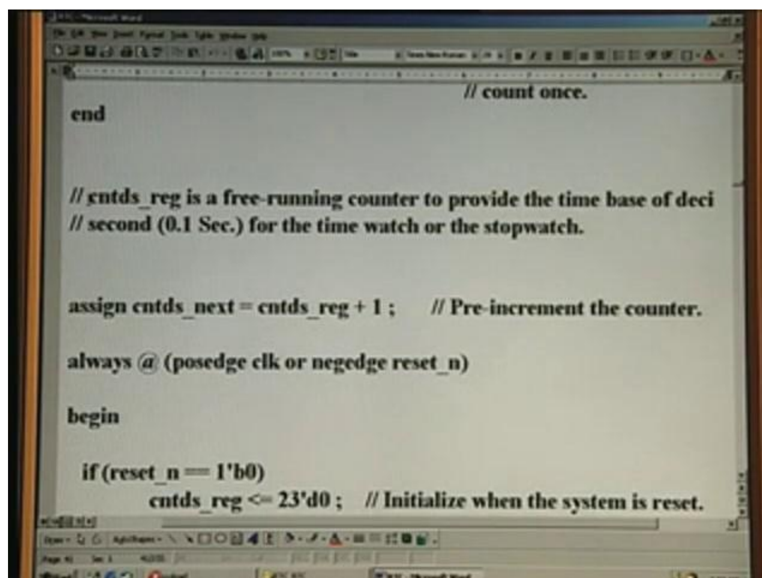
in the beginning by defining) and when it matches with this, what you need to do is reset the counter once again. Otherwise, what you need to do is increment the counter.

(Refer Slide Time: 03:47)



That is what we are doing here. But we have done this implementing earlier by using an assign statement, which is advance pre-increment. Thus, this counter is complete.

(Refer Slide Time: 04:02)



```
// second (0.1 Sec.) for the time watch or the stopwatch.

assign cntds_next = cntds_reg + 1 ;     // Pre-increment the counter.

always @ (posedge clk or negedge reset_n)

begin

  if (reset_n == 1'b0)
          cntds_reg <= 23'd0 ;   // Initialize when the system is reset.

  else if (cntds_reg == `ds_base )   // Also reset if terminal count is
                                     // reached.
          cntds_reg <= 23'd0 ;
```

Similarly, we have many counters to do time in deci-second – that means 0.1 second time base. This is exactly same as that, except that the signals are different. For example, cntds_next will be the advance increment for the actual counter, which is cntds_reg. This is the always block for doing this incrementing, resetting.

(Refer Slide Time: 04:24)

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

begin

  if (reset_n == 1'b0)
          cntds_reg <= 23'd0 ;   // Initialize when the system is reset.

  else if (cntds_reg == `ds_base )   // Also reset if terminal count is
                                     // reached.
          cntds_reg <= 23'd0 ;

  else
```

When the time meets the set values such as ds_base, it has to reset again. Note that this counter is 23 bits in width.

(Refer Slide Time: 04:34)

```
        cntds_reg <= cntds_next ; // Otherwise, advance the count
                                  // once.
end

assign tbsec = (cntb_reg == `time_base)&(cntds_reg == `ds_base) ;

                                  // Time base in seconds.

assign cntb_next = cntb_reg + 1 ;  // Pre-increment the counter.
```

```
                                  // Time base in seconds.

assign cntb_next = cntb_reg + 1 ;  // Pre-increment the counter.

always @ (posedge clk or negedge reset_n)

begin

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

  else if (tbsec == 1)
        cntb_reg <= 4'd0 ;     // Reset if terminal count (1 sec.) is
                               // reached.
```

Otherwise, what we need to do is assign the pre-increment value to the counter register. We have one more register, which is a 1 second time base register or rather a counter that is being taken only if this condition is met. For example, tbsec implies that when this running counter is equal to the set time base, for example, we have set 9 to give 9 second and when it matches and also matches with deci-second (0.1 second) – this is to get a more accurate timing. Otherwise, for this value, multiple times will be encountered. We want to service only once – the very first time it encounters. Therefore, we have included this as well. As usual, we have a pre-increment of the same register and then the actual register takes place here.

(Refer Slide Time: 05:34)

```
begin

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

  else if (tbsec == 1)
        cntb_reg <= 4'd0 ;      // Reset if terminal count (1 sec.) is
                                // reached.

  else if (cntds_reg == `ds_base)
```

We reset and when the time base is 1 second, at every 1 second, we have to take this action. What you should do is just reset that running counter.

(Refer Slide Time: 05:50)

```
  else if (cntds_reg == `ds_base)
        cntb_reg <= cntb_next ;   // Advance the count once every
                                  // 0.1 sec.
  else                  ;         // Otherwise, ignore.

end

assign dmsec = (deb_cnt_reg == `debounce_time)&
```

The counter is incremented in the next step, when the counter time base is 0.1 second. That is what you have to do. That means you have to advance this particular counter every 0.1 second and that is what we are doing here.

(Refer Slide Time: 06:05)

```
end

assign dmsec = (deb_cnt_reg == `debounce_time)&
                              (cntdms_reg == `dms_base) ;

                              // Debounce time in milli seconds.

assign deb_cnt_next = deb_cnt_reg + 1 ;   // Pre-increment the
                                          // counter.

always @ (posedge clk or negedge reset_n)

begin
```

For debouncing, we have another counter that is running and it is called debounce counter. This is the condition that we want to take action for. For example, we want deci-millisecond and let us say we have one more signal when the debounce counter is equal to the set debounce time. For example, if you remember, it is 29 that we set corresponding to 29 into 0.1 millisecond. Deci-millisecond is also to be reckoned with together for the same reason that we have seen for the earlier ones. That means to say 29 into 0.1 millisecond and that is 3 millisecond. 29 is actually to be read as 30, because actually 30 clocks are involved although you set at 29. It is because we start counting right from 0 – 0, 1, 2, 3 up to 29, so all-inclusive is 30 actually. 30 into 0.1 milliseconds would give you 3 millisecond. That is how you get 3 millisecond debounce time.

That means whenever a switch is pressed, it will make and break for some time, say 2 to 2.5 milliseconds based on real-time experience. In order to get rid of these movements or oscillations, we want to give some cushion or time delay so that we sense only the steady state. For example, if you push a button, it will make and break for a while and then settle down finally because the pressure that you have applied on the button is much greater than its kicking back and breaking the contact. That is what we mean by the debounce time, which you have already become familiar with in the previous lectures.

(Refer Slide Time: 07:52)

```
assign deb_cnt_next = deb_cnt_reg + 1 ;   // Pre-increment the
                                          // counter.

always @ (posedge clk or negedge reset_n)

begin

  if (reset_n == 1'b0)
         deb_cnt_reg <= 5'd0 ;          // Initialize when the system is
                                        // reset.
```

Here, we need to once again pre-increment this debounce counter and once again, we reset when system reset is encountered.

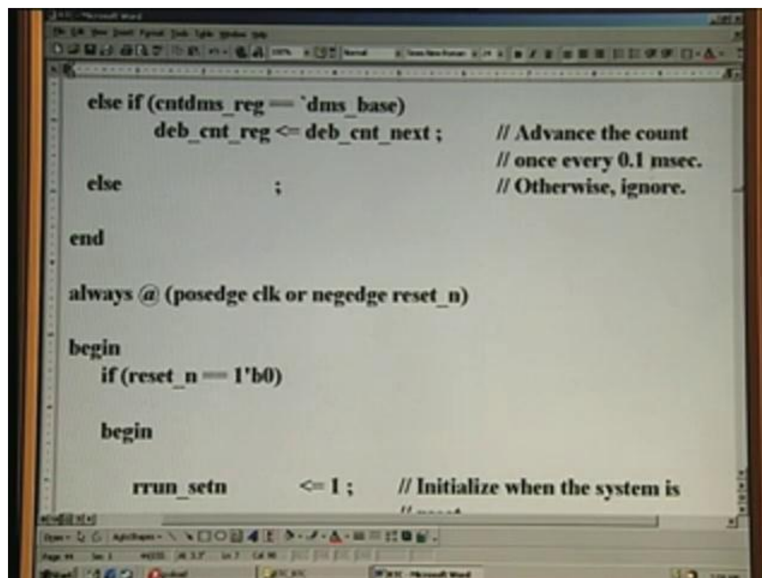
(Refer Slide Time: 08:05)

```
  else if (dmsec == 1)
         deb_cnt_reg <= 5'd0 ;          // Reset after 3 msec. delay.

  else if (cntdms_reg == `dms_base)
         deb_cnt_reg <= deb_cnt_next ;       // Advance the count
                                             // once every 0.1 sec.
  else                    ;                  // Otherwise, ignore.

end

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

After 3 millisecond delay, we want to reset the same counter once again. This is how it keeps on going from 0 to 3 millisecond and revolves around that. Every time the clock strikes, for example if it is 20 Megahertz, the clock will be arriving at every 50 nanoseconds. At every clock second,

this is going to take place. Although we have seen several always blocks, it implies that they are all parallel circuits. It is not a sequential circuit as I have been impressing time and again. When the cntdms that is 0.1 second time base, it is actually 0.1 millisecond, but we are going to check for 30. 29 is the setting value here. This is only 0.1 second. We advance the count once every 0.1 second, because we are trying to advance here. Resetting only will take 3 milliseconds, whereas you have to do advancing every 0.1 second. I am afraid there is some mistake.

(Refer Slide Time: 09:27)



I think it is 0.1 millisecond. There is a mistake there. 0.1 into 29 is 30 and 30 into 0.1 is 3 millisecond. In order to get 3 millisecond here, we are going to advance 30 times. The incrementing will take place only here, because this pre-increment is done before and that incremented value is assigned here. This assignment takes place only once in 0.1 millisecond. There was a mistake there. Please correct that. I have corrected the same here.

(Refer Slide Time: 10:02)

```
end

always @ (posedge clk or negedge reset_n)

begin
    if (reset_n == 1'b0)

    begin

        rrun_setn       <= 1 ;    // Initialize when the system is
                                  // reset.
        rtime_stopwn    <= 1 ;    // 'r' stands for register or store.
        rdown_upn       <= 0 ;
```

We need to take action for different switches that we have seen in the layout earlier. run_set was one such, time stop watch, down_up count – all were there earlier. I said that we will have to read that and we have to preserve it as a register. That is why r has been used. Whenever you reset, these are all the signals, rather they are all flip-flops fundamentally in the hardware. We need to reset them whenever you apply power on reset right at the starting of the system.

(Refer Slide Time: 10:38)

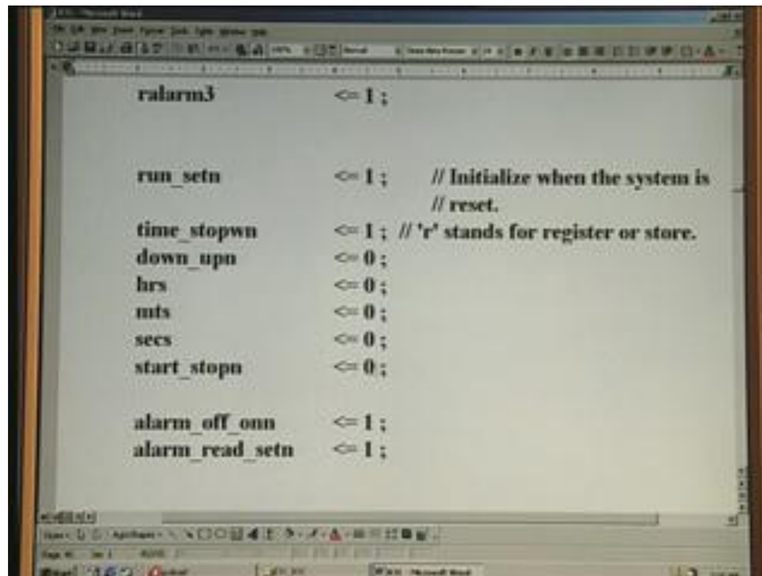
```
        rhrs            <= 0 ;
        rmts            <= 0 ;
        rsecs           <= 0 ;
        rstart_stopn    <= 0 ;

        ralarm_off_onn      <= 1 ;
        ralarm_read_setn    <= 1 ;
        ralarm1             <= 1 ;
        ralarm2             <= 1 ;
        ralarm3             <= 1 ;


        run_setn            <= 1 ;    // Initialize when the system is
                                      // reset.
        time_stopwn         <= 1 ;  // 'r' stands for register or store.
```

```
rhrs            <= 0 ;
rmts            <= 0 ;
rsecs           <= 0 ;
rstart_stopn    <= 0 ;

ralarm_off_onn      <= 1 ;
ralarm_read_setn    <= 1 ;
ralarm1             <= 1 ;
ralarm2             <= 1 ;
```

```
ralarm_off_onn      <= 1 ;
ralarm_read_setn    <= 1 ;
ralarm1             <= 1 ;
ralarm2             <= 1 ;
ralarm3             <= 1 ;


run_setn            <= 1 ;      // Initialize when the system is
                                // reset.
time_stopwn         <= 1 ; // 'r' stands for register or store.
down_upn            <= 0 ;
hrs                 <= 0 ;
mts                 <= 0 ;
secs                <= 0 ;
start_stopn         <= 0 ;
```

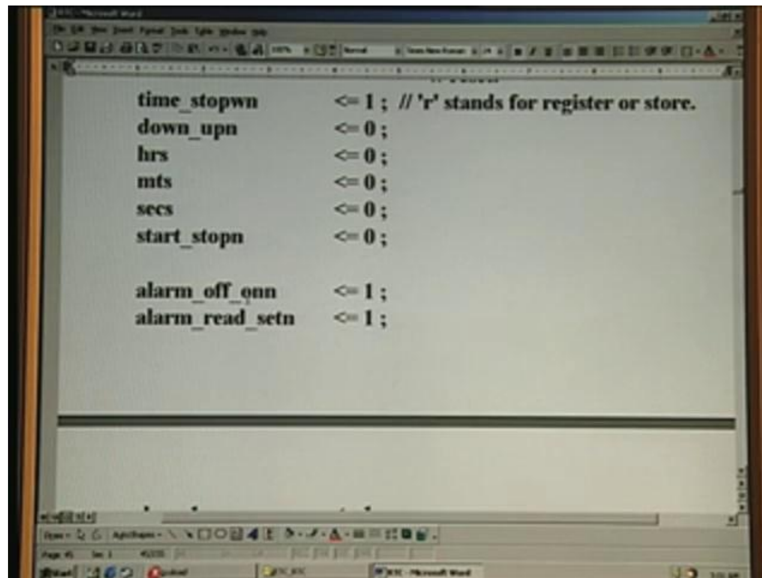
All these signals such as hours, minutes, seconds, start, stop are registers, including alarm_off_on, read_set and three alarms. They need to be set to either 0 or 1 as the case may be, depending upon what is active or what is inactive. It should be inactive at this point of time.

(Refer Slide Time: 11:06)



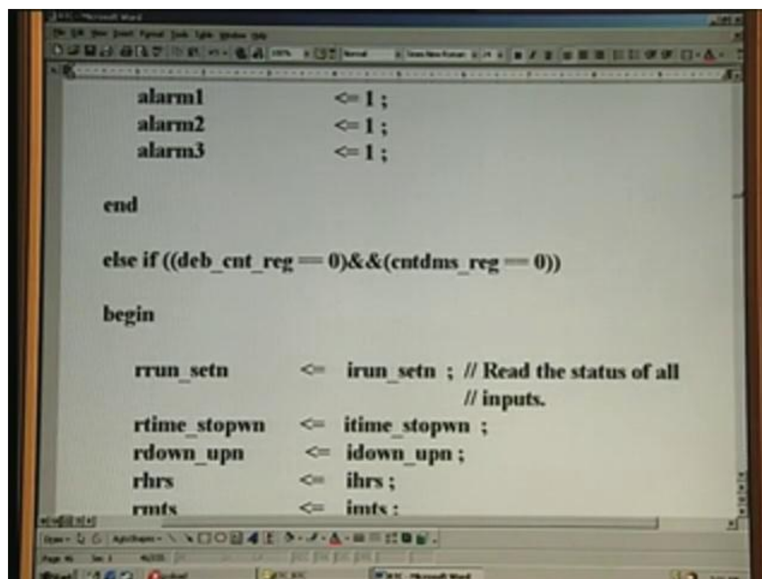
So is the case for runset, which we are going to use after debouncing. This has also been explained earlier. These are all the true reflections of your actual corresponding inputs here. There is a run_set as well as time stop watch, then down_up, and then hours, minutes, seconds, start stop watch. stop_n implies that if it is 0, it means it is in stop. Right now, it has been initialized to the stop condition. You can advance the up counter or down counter only if start_stop is in stop mode, which means when it is in 1. We are right now in the reset mode and that is why we need to stop the thing to start with so that when you switch on the system, it will not start running automatically – it will run only when you push the start button.

(Refer Slide Time: 12:03)



Similarly, alarm_off_on switch also has been taken to the off position and off is 1 here. Similarly, alarm_read_set has been put in read mode and not in set mode.

(Refer Slide Time: 12:16)



Similarly, alarm1, alarm2 and alarm3 are all taken to active low. That means they are not set; only 0 will set in this condition here.

(Refer Slide Time: 12:30)

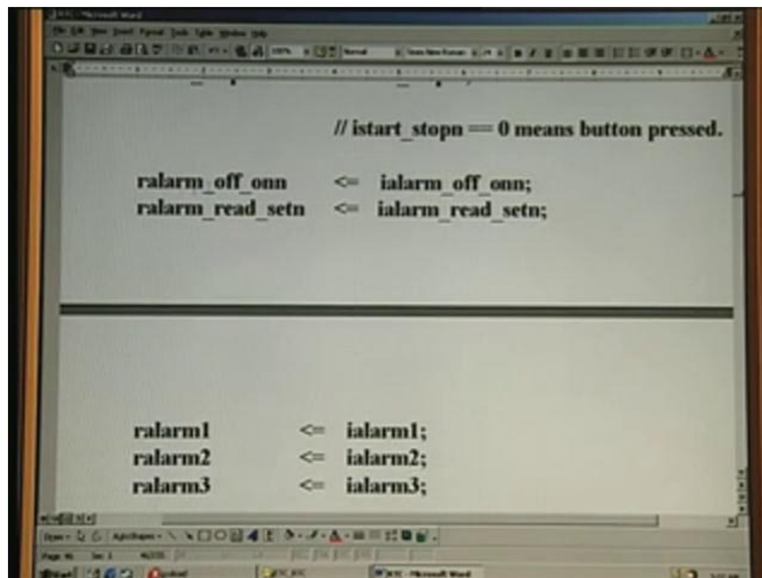
```
end

else if ((deb_cnt_reg == 0)&&(cntdms_reg == 0))

begin

    rrun_setn        <=  irun_setn  ; // Read the status of all
                                       // inputs.
    rtime_stopwn     <=  itime_stopwn  ;
    rdown_upn        <=  idown_upn ;
    rhrs             <=  ihrs ;
    rmts             <=  imts ;
    rsecs            <=  isecs ;
    rstart_stopn     <=  istart_stopn ;
```

```
begin

    rrun_setn        <=  irun_setn  ; // Read the status of all
                                       // inputs.
    rtime_stopwn     <=  itime_stopwn  ;
    rdown_upn        <=  idown_upn ;
    rhrs             <=  ihrs ;
    rmts             <=  imts ;
    rsecs            <=  isecs ;
    rstart_stopn     <=  istart_stopn ;

                     // istart_stopn == 0 means button pressed.

    ralarm_off_onn      <=  ialarm_off_onn;
    ralarm_read_setn    <=  ialarm_read_setn;
```

The actual working starts from here. The work starts for changing the different signals in accordance with what is really happening. For example, this run_set switch is being read here and assigned to this register only at this point when the clock strikes and also when this condition is met. For example, this condition is the debounce counter must be 0, that is to say we are just starting. What we need to do is to debounce a particular switch, either a push button switch or any other switch. This can be done only by reading the input first right at the 0 time – that is what is implied here (both mean 0 time), then read from the actual input (i stands for the input,

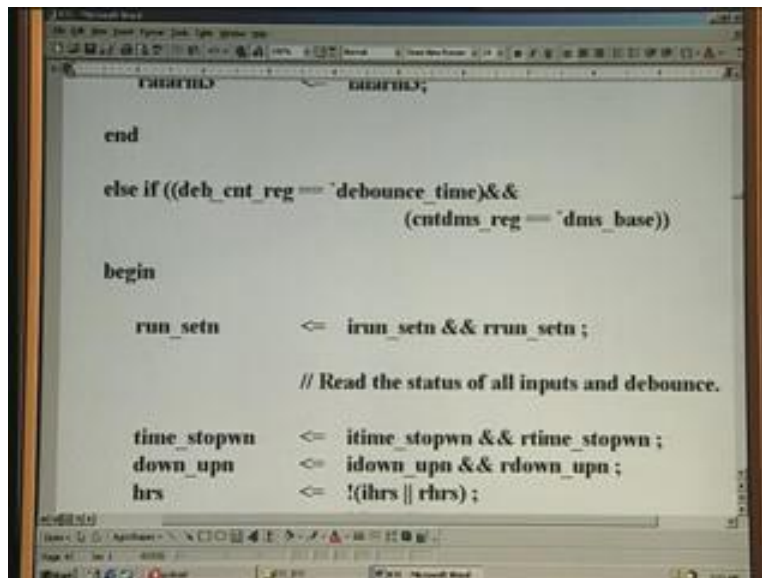
which we have already seen in the simplified architecture) and assign that to the register r here. So is the case for time, stopwatch, then down_up, then the three push buttons and start_stop also. This is also a push button switch whose status we will have to keep track of. These are all the registered conditions of the actual input here.

(Refer Slide Time: 13:39)



These are the registers for alarm_off_on, the three alarms, etc. All are registers here. It is precisely the same as input, but we are registering here only when time is 0, which we have already seen before, and when the clock strikes, which will happen every 15 nanoseconds if it is a 20 Megahertz oscillator.

(Refer Slide Time: 14:02)



If that is not satisfied, it may satisfy this condition. This condition is debounce counter is equal to debounce time. We have seen that the debounce time is 3 milliseconds. When it is 3 milliseconds, then what action should we take? This is not only reckoned with this, but also with reference to a deci-millisecond also. It is to have finer tuning and it should happen at precisely the time when it is ripe and not subsequent times. This condition may be met for several conditions of this but only for the first encounter that we have here, we need to take action and that is why these have also been taken into account. Normally, designers ignore these and get into trouble later on when they do not reckon finer timing, etc.

(Refer Slide Time: 14:59)

```
else if ((deb_cnt_reg == `debounce_time)&&
                                (cntdms_reg == `dms_base))

begin

    run_setn        <=  irun_setn && rrun_setn ;

                    // Read the status of all inputs and debounce.

    time_stopwn     <=  itime_stopwn && rtime_stopwn ;
    down_upn        <=  idown_upn && rdown_upn ;
    hrs             <=  !(ihrs || rhrs) ;
    mts             <=  !(imts || rmts) ;
    secs            <=  !(isecs || rsecs) ;
    start_stopn     <=  !(istart_stopn || rstart_stopn) ;
```

When this condition is met, what we need to do is the actual debouncing of the push button switches or any other switch. Let us say we have not pushed the button. When you read it, it may read 0 state and if you push a button, it may read 1 state. How will you sense the transition that it has been just pushed? It is this particular thing that gives this intelligence to reckon with the pushing of the button. This can be very easily done. This is the present value of the push button and we have just now seen that rrun was the previous value in the previous else condition. This is the same condition, but in the previous clock and when 0.1 second or whatever time base that matches has been met. Now in the subsequent sample, that is the current time now. Suppose the push button has been pressed in between. Suppose it was 0 0.1 second back. That means this was 0.

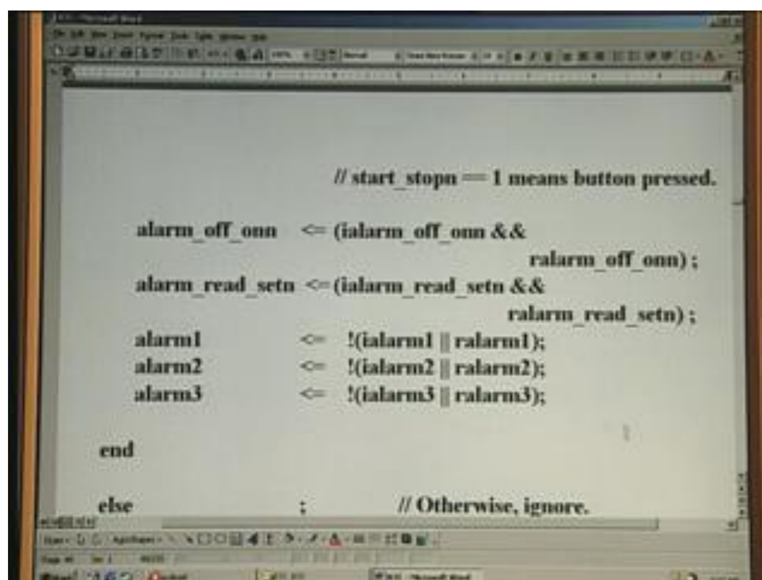
Now, in between after 0.1 second, it is going to be read; in between whatever be the time this goes high, it will be reckoned only at 0.1 second now. It may not be really 0.1 second, but here it is as far as I think debounce time, 3 millisecond delay. The first sample is rrun and the second sample is the present value here. The time lag between these two is 3 millisecond and that is how you are giving the debounce time of 3 millisecond. When you AND this together, if both are 1, then you can say with confidence that the push button has been pressed previously – it was already pressed before 3 milliseconds and now, it is continuing to be in the pressed condition;

that is how you recognize it. So is the case for all the other signals such as time_stopw, down_up.

These are all the signal names given for the debounce logic operated upon. The logic for hours, minutes and seconds is slightly different, because these are all active high signals and that is why we have used AND. If it is active low signal, we need to use r and since we want that particular pressing of the button to be recognized as 1 for further processing. If you want it as 0, you are free to do so. I have chosen to make it as 1 – active high. This signal is active high.

If you want this active high, note that we need to OR the present one and this is the previous value of the push button. When both are same, for example if both are 0, 0, that means it has been pressed 3 milliseconds or more prior to this. That means when you press, this push button is 0 and not 1. If its previous value is 0 and present value also is 0, the OR-ing of this will be 0 and NOT 0 is 1. That means the two samples have been reflecting that a push button has been pressed. This is how you debounce. So is the case for minutes and seconds push buttons and so also for start_stop. It is precisely the same condition that you have here.

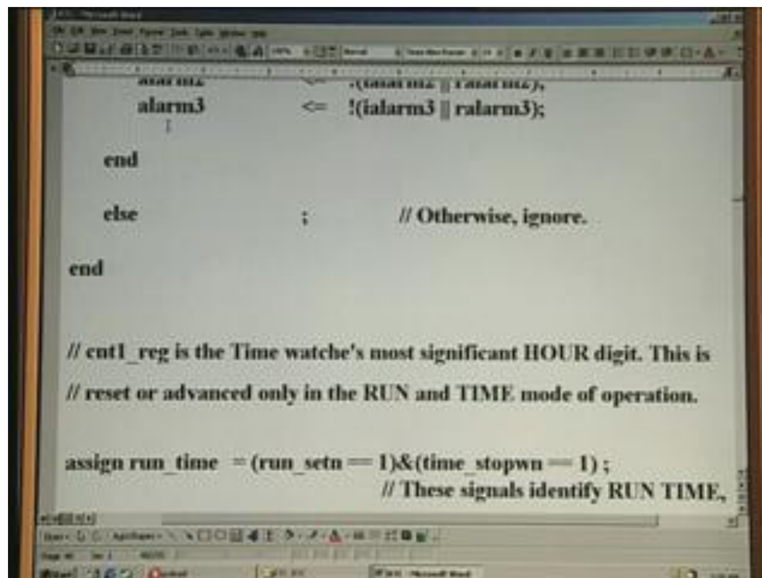
(Refer Slide Time: 18:24)



The comment says start_stopn = 1 means the button is pressed. The push button has been pressed for start. We will have to deal with this a little more in depth and as we go on, we will see more about this. You need to do this for the alarm_off_on switch also, but in this case, I am treating it

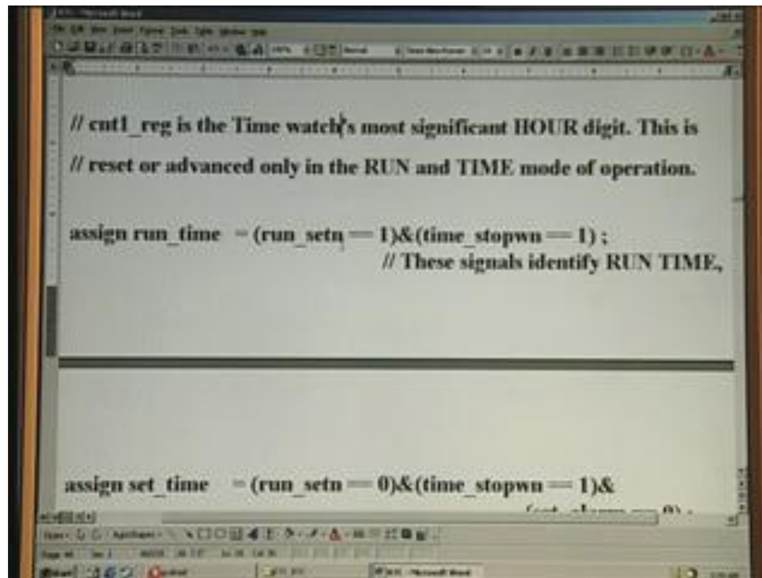
as active high signal. Therefore, I have AND-ed the two: past and present. So is the case for alarm_read_set switch, which is right there on the front panel of the real-time clock. alarm1 through alarm3 are all active low and that is the reason why NOR has been used for the present and past samples.

(Refer Slide Time: 19:20)



This always block is complete and it has serviced all the inputs and assigned, debounced and then preserved in the respective variables, which are exactly the same as the layout nomenclature that we have adopted earlier.

(Refer Slide Time: 19:23)



Here starts the actual running time counting. For this purpose, we need six counters: cnt1_reg through cnt6_reg. cnt1 is the time watch's most significant hour digit and the LSD is cnt2. This is reset or advanced only in the RUN and TIME mode of operation. This is obvious because we want only the running of the time. We have another signal called run_time, which precisely reflects what mode this is in. When run_set switch is in 1, it means RUN and if it is 0, it means SET. That is why the n is given here. So is the case for any other variable. For example, time_stopw is in time mode, because it is 1. It means RUNTIME. This logic is assigned to a single variable call run_time. That is the reason why we have several wire declarations. I think that was your question offline. You can easily keep track of the logic pertaining to a particular state. For example, run_time has been kept like this so that you do not have to put the same logic again and again later on. In order to avoid that, we just assign it to some other signal and use that particular signal, which has a meaningful name, run_time for example.

(Refer Slide Time: 20:55)

```
assign set_time  = (run_setn == 0)&(time_stopwn == 1)&
                                      (set_alarm == 0) ;
                        // SET TIME &
assign set_stopw = (run_setn == 0)&(time_stopwn == 0)&
                                      (set_alarm == 0) ;
                        // SET STOPWATCH modes respectively.

assign run_stopw    = (run_setn == 1)&(time_stopwn == 0) ;
                        // Run stopwatch mode.

assign set_alarm  = (alarm_read_setn == 0)&(set_stopw == 1);
                        // Set or Read alarm in set stopwatch mode.
```

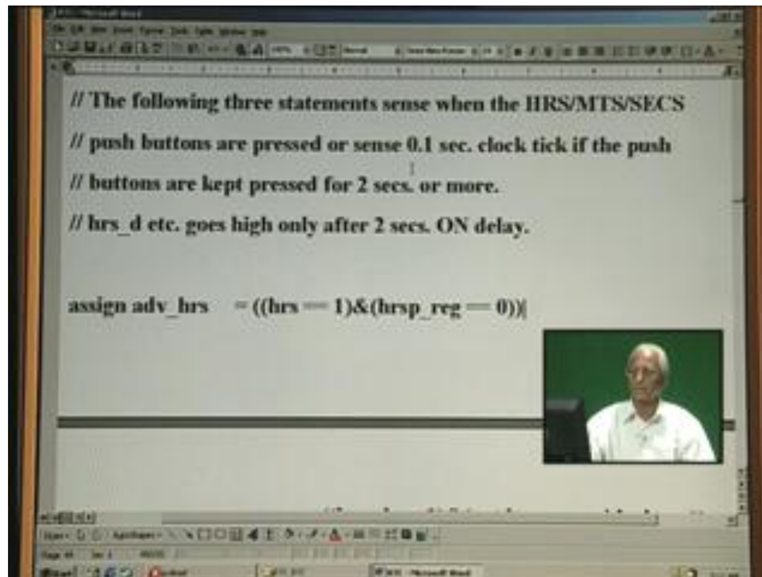
The next thing is set_time. It has exactly the same meaning and you can just verify for yourself. For example, set is 0, and then time is 1. Set_time is what we have given here and set_alarm must be in 0 condition, because we do not want to set the alarm. SET TIME and SET STOPWATCH modes respectively. This is exactly the counterpart of this and you can just reason out yourself.

(Refer Slide Time: 21:26)

```
                                      (set_alarm == 0) ;
                        // SET STOPWATCH modes respectively.

assign run_stopw    = (run_setn == 1)&(time_stopwn == 0) ;
                        // Run stopwatch mode.

assign set_alarm  = (alarm_read_setn == 0)&(set_stopw == 1);
                        // Set or Read alarm in set stopwatch mode.

// The following three statements sense when the HRS/MTS/SECS
// push buttons are pressed or sense 0.1 sec. clock tick if the push
// buttons are kept pressed for 2 secs. or more.
// hrs_d etc. goes high only after 2 secs. ON delay.
```

Similarly, run_stopw and set_alarm. You can once again see here that run is 1, then time_stopw is 0. So it is run stopwatch and that is how we have got it here. We can see set_alarm once again. It is 0, so it reads set here. Stopwatch is one thing, while alarm is for the time. If it is set stopwatch, then you want to have … this is for the alarm setting, so set or read the alarm in set stopwatch mode.

(Refer Slide Time: 22:10)

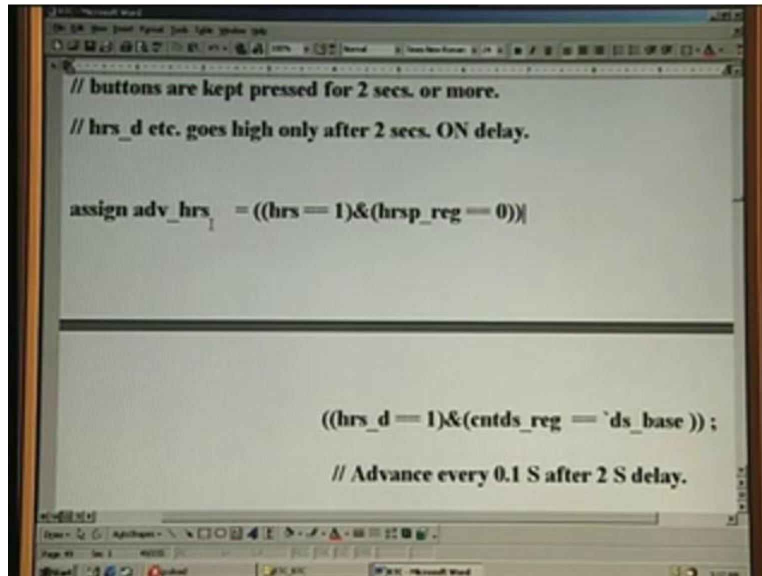


Here, we have three statements that we are going to follow. They are for hours, minutes and seconds. Push buttons are pressed or sense 0.1 second clock tick if the push buttons are kept pressed for 2 seconds or more. This was the question addressed earlier by one of you last time. What we do is every time we want to set something, we can either go into auto mode, wherein the system looks for 2 seconds a push button being pressed. Then, it runs very fast so that large delays can be easily set, because it is running faster. If you close in, you can release the button and advance slowly by pressing the push button each time you want to count just 1 – that is what it means here. The following statements sense when the hours, minutes and seconds push buttons are pressed or sense 0.1 second clock tick if the push buttons are kept pressed for 2 seconds or more. I hope this clears your doubt.

We also said we needed 2 seconds ON delay timer. For that, we are using this signal called hrs_d, where d stands for delay. If you want, you can say on delay. On delay means when you

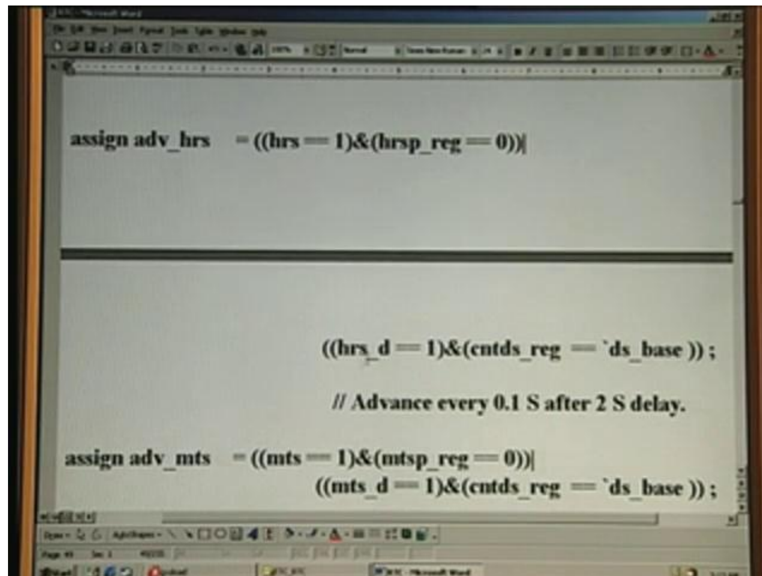
close the switch, the timing starts but the output is not yet energized. Only after the expiry of 2 seconds, the output goes high and remains like that. That is what is known as ON delay timer. We will see why it is required when we come to the appropriate point.

(Refer Slide Time: 23:48)



The next signal is adv_hrs. Once again, we use exactly the same logic. hrs is the push button that we have pressed, of course when it is in debounce condition. Hereafter, it is going to be the debounce condition and we have already examined how we arrived at this. When the hrs push button is pressed, then it will be 1. If you see the previous value for the same signal and if it is 0, then you recognize that a push button has been pressed. The present value is 1 and the previous value of the same switch is 0. That implies that the push button has been just pressed.

(Refer Slide Time: 24:31)



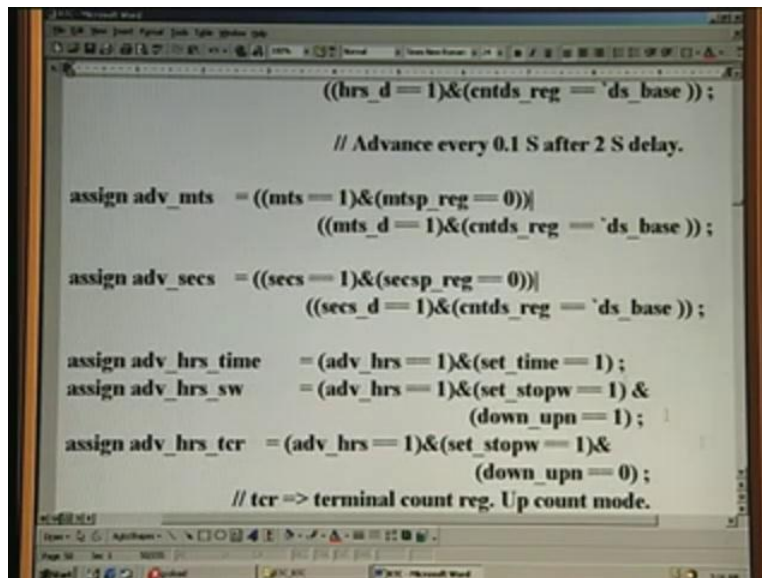
When that happens or this happens…. For example, the 2 second delay is going to come. How is this 0.1 single increment incremented in a single manner, whenever you push the push button? When you push the button, it will go to 1 here. The previous value would be 0, because you have just now pressed and it has been just now recognized. Whenever you make a single push button press and release, this will come into force, whereas if you keep it pressed for 2 seconds or more, this statement will come into play. When this comes into play, then this delay will go to 1 only after 2 seconds. That is how this circuit works.

This will remain at 0 for 2 seconds and then go to 1. This will happen only if you had pressed the button for 2 seconds or more. Had you released in between, this would have taken effect once because when you push the button, this will be met and when you release, it is not met – it is reckoned as just one push. That is how you can advance the hours by 1 count or it can be continuous count at 10 times the rate. How does 10 times come? It is because we are comparing with the counter deci-second register with the deci-second base that we already have set in (ds stands for deci-seconds, 0.1 second).

This will be as fast as you press. If you are slow in pressing, it will advance only in pace with your pressing. You can leisurely press and take any amount of time, whereas in this auto mode, it will be rapidly pressed because this 0.1 second will keep on arriving ad infinitum. This will be

multiple, but this will come into effect only if 2 seconds have lapsed and this has gone high. As long as it is high, this will keep on continuing; this is how we do that. As I mentioned, in the example of the Philips radio, they have implemented something like this so that the setting can be either manual (one after another) or high speed.

(Refer Slide Time: 26:52)



The same argument holds good for adv_mts and adv_secs. I am not going into the details. It is exactly the same, except that the signals are different. Similarly, you have adv_hrs_time. We have already seen the same adv_hrs and here, it is that together with set_time. We are also in set_time mode; we have examined this also. When these two conditions are met, we call that particular signal condition as adv_hrs_time.

Similarly, adv_hrs_sw, adv_hrs_tcr (terminal count register) for up count mode. What we mean by terminal count register is in the up count mode, it is capable of counting up right from 0 – 0, 1, 2, 3, etc. When we look at the demo after a lecture or two, you will understand it better. But right now, this is sufficient. As it advances, it will finally reach this set value. You can set any up count value by using those switches we have already seen and I have explained in answer to your question last time.

When it matches with that, it is a match here to just signal that it is in advance hours terminal count register mode. You can see here that down_up is 0, so it is in up mode and it is set

stopwatch. Notice that there is no n here. Set stopwatch is a single meaning – you are going to set the stopwatch, that is the meaning there. Do not take it as 0 stands for stopwatch. This means that it is in advance hours step stopwatch mode in up count mode.

(Refer Slide Time: 28:51)

```
assign adv_hrs_sw      = (adv_hrs == 1)&(set_stopw == 1) &
                                     (down_upn == 1) ;
assign adv_hrs_tcr    = (adv_hrs == 1)&(set_stopw == 1)&
                                     (down_upn == 0) ;
                    // tcr => terminal count reg. Up count mode.

assign adv_mts_time    = (adv_mts == 1)&(set_time == 1) ;
assign adv_mts_sw      = (adv_mts == 1)&(set_stopw == 1) &
                                  (down_upn == 1) ;

assign adv_mts_tcr     = (adv_mts == 1)&(set_stopw == 1)&
                                  (down_upn == 0) ;
```

Similarly, advance minutes time and advance minutes stopwatch are all self-explanatory. Again, advance minutes tcr just like you had for hours. Similarly, you are going to have for seconds.

(Refer Slide Time: 29:03)

```
                                  (down_upn == 1) ;
assign adv_secs_tcr     = (adv_secs == 1)&(set_stopw == 1)&
                                  (down_upn == 0) ;

assign adv_hrs_temp_alarm    = (adv_hrs == 1)&(set_alarm == 1) ;
assign adv_mts_temp_alarm    = (adv_mts == 1)&(set_alarm == 1) ;
assign adv_secs_temp_alarm   = (adv_secs == 1)&(set_alarm == 1) ;

assign display_alarm     = ((alarm1 == 1)|(alarm2 == 1)|
                  (alarm3 == 1))&(alarm_read_setn == 1) ;

assign display_time   = (time_stopwn == 1)&(display_alarm == 0) &
                                        (set_alarm == 0) ;

assign display_stopw = (time_stopwn == 0)&(display_alarm == 0)&
                                        (set_alarm == 0) ;
```

I do not have to go into these details.

(Refer Slide Time: 29:09)

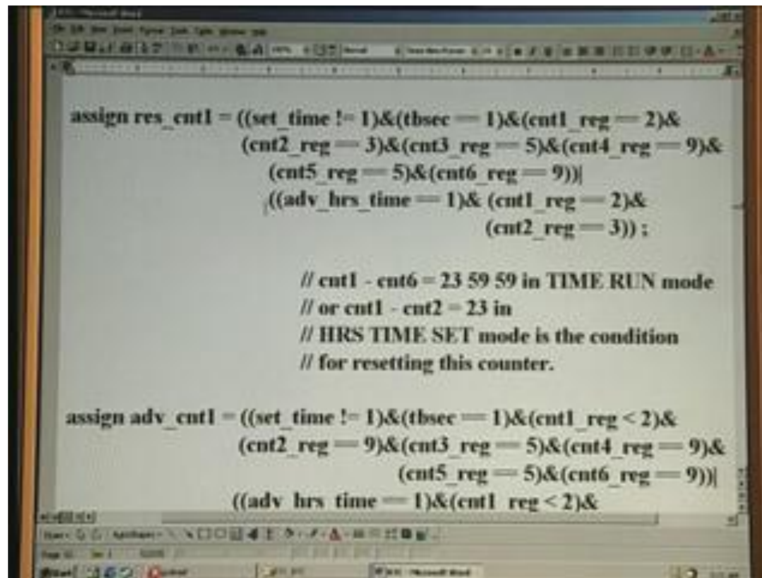
Next, you have adv_hrs_temp_alarm. This temporary alarm is there in order to set three different alarms, but before we set, we need some register for setting in a temporary manner. We will not overwrite our up/down counter, because the same setting has been used for up counter, down counter as well as for the three different alarm settings. Since there are many settings, we will have to distinguish .That is why so many signals are there. This is precisely for minutes and seconds as well. This is self-explanatory. Next, we have what is called display_alarm. In

simplified architecture, we have seen this. What it means is you want to display just the alarm, you want to set or read the alarm back.

You have three different alarms, for example, alarm1, alarm2 and alarm3. This is a debounce switch and there are three switches for alarm1 through alarm3 – ialarm, etc. This is nothing other than debounced input. Even if one or more alarm is set and if alarm_read_set is in 1 condition, what is to be done? We want to just display the alarm. If you want to display time, you see that it is time or stopwatch. Now you see that n stands for low. It is not in stopwatch mode but in time mode. This 1 corresponds to time.

display_alarm has been used here. That is the reason why…. Otherwise, the logic becomes too long and it will be difficult for you to keep track of the same. That is why the signals have been separately defined. display_alarm must not be energized – this is obvious. When you want to display the alarm, you should not display time. That is the implication there. Make sure that it is not there. set_alarm must also be 0, because you are interested only in time, not the alarm or displaying the alarm. You want to neither set the alarm nor display the alarm but you want to have only when the time is encountered. Similarly, for stopwatch, display_stopw is there. time_stopw here, the stopwatch is 0, so this is stopwatch mode. display_alarm must not be there and set_alarm is also similar to this. Only one of the three displays alarm/time/stopwatch is possible at one time. This has been explained before also.

(Refer Slide Time: 31:56)



Now, the logic for the very first counter. We need to reset the counter. At what time would we like to reset? I will read the comment first so that you will get an idea and then, I will explain the logic. We are going to use nothing more than assign statements and always block – it is quite easy to follow. Although it is like a labyrinth, you have to look into all the aspects. These will be especially tiring. If you are interested in the working of the hardware, you have to take care of everything; even a single mistake will make it go totally out of order. cnt1 through cnt6 is designated for real-time display and the maximum time that you can have is 23 hours, 59 minutes and 59 seconds. It is in TIME RUN mode. cnt1 to cnt2 must be 23 in HRS TIME SET mode. This is the condition for resetting this counter.

You want this time. For example, if it is this time, what you have to look for is cnt1 and cnt2 must be 23. It is exactly the same – cnt1 is 2, cnt2 is 3 and so on. This is the condition for resetting this first counter alone. cnt1 is this corresponding 2. When do you want to reset? After 23:59:59 runtime, what is the next time? It is going to be 00. In other words, cnt1 is going to be reset, that is, after 2, it has to become 0. That is what we mean here. Let us see the actual logic here.

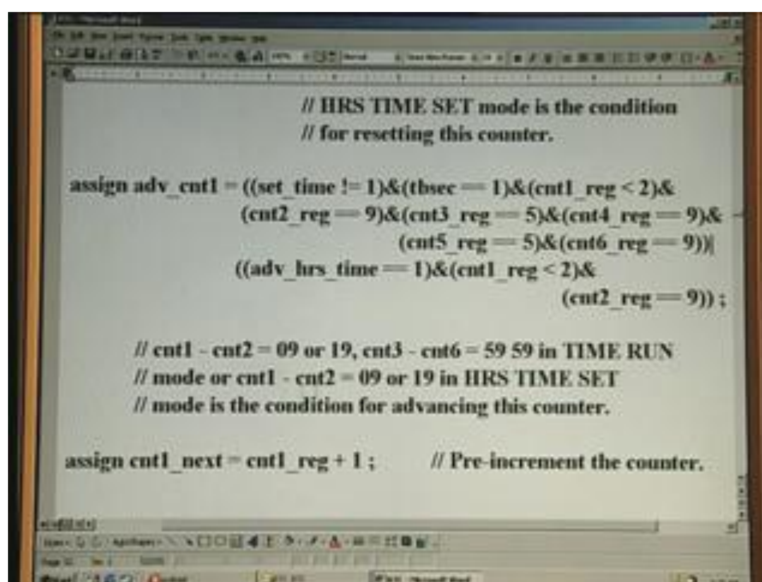
For example, set_time is not equal to 1 means it should not be in the setting mode, but it must be in the run mode and only then, you should take action and it should be 1 second now. Every 1

second or 0.1 second, there is always a clock ticking all the time and when it is exactly 1 second, then only you take the action, otherwise no. cnt1_reg must be 2 – that is the condition we have put. This is coming right from this condition. Then, cnt3 must be 3. Similarly, other counters must be 5, 9. That is why we have 5 here, then 9 here and cnt5, cnt6 are 5, 9. The whole thing is actually reflecting the comment that we have put here. This is one option. There is another option also. This is while the timer is running, but you have not taken care for the setting.

In order to do the setting, you have to take that logic also into account and that is being done here. For example, cnt1 is earmarked only for the hours' MSD, but when you want to advance while setting, you are going to do MSD and LSD together. LSD starts first and so, it must be in advance hours time whose logic we have already seen – a big logic expression. That must be [34:55] and also, cnt1 and cnt2 must be 23.

When you want to set an auto setting after 23, what do you expect? It should become 00, but we are concerned with only cnt1. As far as cnt1 is concerned, after 23, you do not bother about 59, 59 – that is only setting the runtime during the time being run, whereas now we are bothered about setting the hours display alone. For hours, we do not have to bother about minutes and seconds. That is why we are stopping right here. This statement stops just with two counters, because we are interested only in the setting of the two digit hours.

(Refer Slide Time: 35:41)



Same is the case for advancing the counter here – exactly the same thing. Here, it is slightly different. I will just read out the comments hereafter so that you can easily make out – everything will be precisely in the same fashion. I am going to read only the comment part just to accelerate. cnt1 and cnt2 must be 09 or 19. Let us say we are now in the TIME RUN mode. What will happen after 09? This is cnt1 and cnt2. What do you expect? Total time is 09:59:59. After this, you expect 10 o'clock. As far as cnt1 is concerned, it should go from 0 to 1. Let us not worry about all other counters. When we come to that counterpart, we will take action for that particular counter. That means our design goal is only concentrating on a single counter – that is what we are doing here.

If it is 09, then we need to advance it to 10. If it is 19, we need to advance it to 20. As far as cnt1 is concerned, 0 will be incremented just by 1. The adv_cnt1 will be advanced by 1 only when this condition is met. All other conditions are straightforward. 59:59 is still intact – that is all clear. Another thing is you have to advance hours time – this is the actual setting mode. This is the running mode and this is the setting mode. Both have been clubbed. You have to necessarily [37:18] because whenever you want to advance the cnt1, the advancing will take place both in normal running mode as well as while setting. So we have to take this into account. Here, less than 2 means 0 or 1 – that is why this has come; other things are straightforward equal to. That is how this statement is done. I will just read out the comment. cnt1 to cnt2 09 or 19, cnt3 to cnt6 59, 59 in TIME RUN mode or cnt1, cnt2 is equal to 09 or 19 in HRS TIME SET mode is the condition for advancing this counter. This is the pre-increment for cnt1.

(Refer Slide Time: 38:04)

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

begin

  if (reset_n == 1'b0)
         cnt1_reg <= 4'd0 ;      // Initialize when the system is reset.

  else if (res_cnt1 == 1'b1)     // Reset if terminal count is reached.
         cnt1_reg <= 4'd0 ;

  else if (adv_cnt1 == 1'b1)
         cnt1_reg <= cnt1_next ;   // Advance the count once if the
                                   // time watch is still running.
  else
                ;                  // Otherwise, don't disturb.
```

```
begin

  if (reset_n == 1'b0)
         cnt1_reg <= 4'd0 ;      // Initialize when the system is reset.

  else if (res_cnt1 == 1'b1)     // Reset if terminal count is reached.
         cnt1_reg <= 4'd0 ;

  else if (adv_cnt1 == 1'b1)
         cnt1_reg <= cnt1_next ;   // Advance the count once if the
                                   // time watch is still running.
  else
                ;                  // Otherwise, don't disturb.
end
```

The next block is the very first block for the timer cnt1. This is precisely the same as all the counters that we have been handling in all these lectures. Once again, for reset the system and reset when the terminal count is reached or advance it by 1. We have defined all these conditions previously and this is precisely the same.

(Refer Slide Time: 38:35)

```
// cnt2_reg is the Time watche's least significant HOUR digit. This is
// reset or advanced only in the RUN mode of operation.

assign adv_res_cnt2 = (set_time != 1)&(tbsec == 1)&(cnt3_reg == 5)&
                      (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9) ;

assign res_cnt2_time    = (adv_res_cnt2 == 1) & ( (cnt1_reg < 2)&
```

```
assign adv_res_cnt2 = (set_time != 1)&(tbsec == 1)&(cnt3_reg == 5)&
                      (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9) ;

assign res_cnt2_time    = (adv_res_cnt2 == 1) & ( (cnt1_reg < 2)&
                                                  (cnt2_reg == 9) |
                          ((cnt1_reg == 2)&(cnt2_reg == 3)) ) ;

assign res_cnt2_set  = (adv_hrs_time == 1) & ( (cnt1_reg < 2)&
                                               (cnt2_reg == 9) |
                       (cnt1_reg == 2)&(cnt2_reg == 3) ) ;
```

Exactly the same thing will go on for all the counters. So I will expedite a bit because everything is going to be the same except for minor variants, which is particular for that particular counter. For example, adv_reset_cnt2 here will be exactly the same, except that the relevant thing is being taken here.

(Refer Slide Time: 38:58)

```
                    (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9);

assign res_cnt2_time    = (adv_res_cnt2 == 1) & ( (cnt1_reg < 2)&
                                                  (cnt2_reg == 9) |
                                   ((cnt1_reg == 2)&(cnt2_reg == 3)) );

assign res_cnt2_set   = (adv_hrs_time == 1) & ( (cnt1_reg < 2)&
                                                  (cnt2_reg == 9) |
                                     (cnt1_reg == 2)&(cnt2_reg == 3) );

assign res_cnt2          = res_cnt2_time | res_cnt2_set ;

                    // cnt1 cnt2 = 23 or 09 or 19 and cnt3 - cnt6 = 59 59
                    // are the conditions for resetting this counter.
```

I will just read out the comments, if any. It is exactly the same, reset time; in time mode, you need one type of resetting and in set mode, you need another set of conditions. Similarly, this is only to reduce the logic. Rather than putting single logic in a long sentence, we have just bifurcated this. Here, it is nothing but res_cnt2_time here and set here. It means time mode and set mode. We are OR-ing. Either this condition is taking place or this condition is taking place. Then also, you need to reset the counter. In each, you have two such conditions. For example, there is one more r here for this condition. That is the 09 or 19 or 23 condition, depending upon whether it is in advance hours mode. This is common anyway. adv_reset_cnt is here. Whether it is cnt1 2 or this, then also, you need only to reset. Note that the bracket starts here and ends here. This is OR-ed with this expression and finally this. We need to set when this condition is satisfied: adv_hrs_time or this condition or again precisely the same condition. That is what we have already seen. I do not have to read the comments because I have already explained.

(Refer Slide Time: 40:43)

```
// are the conditions for resetting this counter.

assign adv_cnt2 = (adv_res_cnt2 == 1)|(adv_hrs_time == 1) ;

      // Other conditions are implied since res_cnt2 has a higher
      // priority than adv_cnt2. cnt1 cnt2 = 00 to 18 (except 09) or

      // 20 to 22 & cnt3 - cnt6 = 59 59 are the conditions for pre-
      // incrementing this counter.
```

Next is adv_cnt2, which we have seen just now. Either adv_reset_cnt2 is 1 or adv_hrs_time is 1. Then only you need to advance the cnt2. Other conditions are implied, since reset counter has higher priority than adv_cnt2. So, cnt1 to cnt2 is 00 through 18 for all these except 09 or 20 to 22. Of course, cnt3 to cnt6 must be 59, 59. This is the pre-condition for incrementing this particular counter.

(Refer Slide Time: 41:18)

```
      // incrementing this counter.

assign cnt2_next = cnt2_reg + 1 ;        // Pre-increment the counter.

always @ (posedge clk or negedge reset_n)

begin

  if (reset_n == 1'b0)
        cnt2_reg <= 4'd0 ;          // Initialize when the system is reset.

  else if (res_cnt2 == 1'b1)         // Reset if terminal count is reached.
        cnt2_reg <= 4'd0 ;
```

This is the pre-increment. The positive edge of the clock, we need to do all this. Once again, it is the same – reset, terminal count reset and then advance.

(Refer Slide Time: 41:34)

```
// cnt3_reg is the Time watche's most significant MINUTES digit.
// This is reset or advanced only in the RUN and TIME mode of
// operation every 1 sec.

assign res_cnt3 = ((set_time != 1)&(tbsec == 1)&(cnt3_reg == 5)&
                  (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9))|
                  ((adv_mts_time == 1)&(cnt3_reg == 5)&
                                                  (cnt4_reg == 9)) ;

                       // cnt3 - cnt6 = 59 59 are
                       // the conditions for resetting this counter.

assign adv_cnt3 = ((set_time != 1)&(tbsec == 1)&(cnt3_reg < 5)&
                  (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9))|
```

```
                  (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9))|
                  ((adv_mts_time == 1)&(cnt3_reg == 5)&
                                                  (cnt4_reg == 9)) ;

                       // cnt3 - cnt6 = 59 59 are
                       // the conditions for resetting this counter.

assign adv_cnt3 = ((set_time != 1)&(tbsec == 1)&(cnt3_reg < 5)&
                  (cnt4_reg == 9)&(cnt5_reg == 5)&(cnt6_reg == 9))|
                  ((adv_mts_time == 1)&(cnt3_reg < 5)&
                                                  (cnt4_reg == 9)) ;

                // cnt3 = 0-4 and cnt4 - cnt6 = 9 59
                // are the conditions for pre-incrementing this counter.
```

Similar is the case for resetting cnt3. These are all the conditions. I will leave it as an exercise for you to reason out. I will just read out the comments. cnt3 to cnt6 must be 59, 59. We are now at the cnt2. That means this is cnt3 prior to this. For advance count, cnt3 must be 0 to 4 and cnt1 and cnt2 are already exhausted. Now, we are in 3. If it is 0 to4, then only we need to advance, because this will go right up to 59. The other digits must be 9, 59.

(Refer Slide Time: 42:14)

```
assign cnt3_next = cnt3_reg + 1 ;          // Pre-increment the
counter.

always @ (posedge clk or negedge reset_n)

begin

  if (reset_n == 1'b0)
          cnt3_reg <= 4'd0 ;      // Initialize when the system is reset.
```

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

begin

  if (reset_n == 1'b0)
          cnt3_reg <= 4'd0 ;      // Initialize when the system is reset.

  else if (res_cnt3 == 1'b1)      // Reset if terminal count is reached.
          cnt3_reg <= 4'd0 ;

  else if (adv_cnt3 == 1'b1)
          cnt3_reg <= cnt3_next ;   // Advance the count once if the
                                    // time watch is still running.
  else
                    ;               // Otherwise, don't disturb.
end
```

This is the pre-increment. This is the usual block as we have seen before – reset and advance.

(Refer Slide Time: 42:22)

```
// cnt4_reg is the Time watche's least significant MINUTES digit.

// This is reset or advanced only in the RUN and TIME mode of
// operation every 1 sec.

assign res_cnt4 = ((set_time != 1)&(tbsec == 1)&(cnt4_reg == 9)&
```

```
// This is reset or advanced only in the RUN and TIME mode of
// operation every 1 sec.

assign res_cnt4 = ((set_time != 1)&(tbsec == 1)&(cnt4_reg == 9)&
                        (cnt5_reg == 5)&(cnt6_reg == 9)) |
                   ((adv_mts_time == 1)&(cnt4_reg == 9)) ;

                   // cnt4 - cnt6 = 9 59 are
                   // the conditions for resetting this counter.

assign adv_cnt4 = ((set_time != 1)&(tbsec == 1)&(cnt4_reg < 9)&
                        (cnt5_reg == 5)&(cnt6_reg == 9)) |
                   ((adv_mts_time == 1)&(cnt4_reg < 9)) ;
```

For cnt4, it is precisely the same. Notice that the logic keeps on reducing, because we have to keep track of less and less number of digits on the right. Here, the condition is 9, 59 from cnt4 to cnt6. There are only three – 4, 5, 6. That is for reset and this is for advance.

(Refer Slide Time: 42:42)

```
// cnt4 = 0 to 8 & cnt5 - cnt6 = 59 are the
// conditions for pre-incrementing this counter.

assign cnt4_next = cnt4_reg + 1 ;    // Pre-increment the counter.

always @ (posedge clk or negedge reset_n)

begin
```

This comment is again the same: 0 to 8 and then cnt5, cnt6 must be 59. Only then, we need to advance. This is again the pre-increment for cnt4.

(Refer Slide Time: 42:55)

```
if (reset_n == 1'b0)
        cnt4_reg <= 4'd0 ;       // Initialize when the system is reset.

else if (res_cnt4 == 1'b1)       // Reset if terminal count is reached.
        cnt4_reg <= 4'd0 ;

else if (adv_cnt4 == 1'b1)
        cnt4_reg <= cnt4_next ;    // Advance the count once if the
                                   // time watch is still running.
else
                    ;              // Otherwise, don't disturb.
end

// cnt5_reg is the Time watche's most significant SECONDS digit.
```

Counter realization is done using this block. Reset condition and then advance condition are there.

(Refer Slide Time: 43:02)

```
// This is reset or advanced only in the RUN and TIME mode of
// operation every 1 sec.

assign res_cnt5 = ((set_time !=1)&(tbsec == 1)&(cnt5_reg == 5)&
                                        (cnt6_reg == 9)) |
                  ((adv_secs_time == 1)&(cnt5_reg == 5)&
```

So is the case for cnt5.

(Refer Slide Time: 43:07)

```
assign res_cnt5 = ((set_time !=1)&(tbsec == 1)&(cnt5_reg == 5)&
                                        (cnt6_reg == 9)) |
                  ((adv_secs_time == 1)&(cnt5_reg == 5)&
                                        (cnt6_reg == 9)) ;

                        // cnt5 - cnt6 = 59 are
                        // the conditions for resetting this counter.

assign adv_cnt5 = ((set_time !=1)&(tbsec == 1)&(cnt5_reg < 5)&
                                        (cnt6_reg == 9)) |
                  ((adv_secs_time == 1)&(cnt5_reg < 5)&
                                        (cnt6_reg == 9) ) ;
```

Once again, res_ cnt5 is there and cnt5, cnt5 must be 59. Now, you see that it is much simpler. The very first thing was the worst among the lot – a lot of logic was involved.

(Refer Slide Time: 43:16)

```
                              (cnt6_reg == 9)) ;

                    // cnt5 - cnt6 = 59 are
                    // the conditions for resetting this counter.

assign adv_cnt5 = ((set_time != 1)&(tbsec == 1)&(cnt5_reg < 5)&
                              (cnt6_reg == 9)) |
               ((adv_secs_time == 1)&(cnt5_reg < 5)&
                              (cnt6_reg == 9)) ;

                 // cnt5 = 0 to 4 & cnt6 = 9 are the
                 // conditions for pre-incrementing this counter.

assign cnt5_next = cnt5_reg + 1 ;        // Pre-increment the counter.

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

Now, it is easing out. adv_cnt5 is here. Here, it is 0 to 4 and cnt6 alone needs to be 9. This is the realization for cnt5, then pre-incrementing.

(Refer Slide Time: 43:27)

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

begin

 if (reset_n == 1'b0)

        cnt5_reg <= 4'd0 ;      // Initialize when the system is reset.

 else if (res_cnt5 == 1'b1)     // Reset if terminal count is reached.
        cnt5_reg <= 4'd0 ;
```

```
        cnt5_reg <= 4'd0 ;        // Initialize when the system is reset.

    else if (res_cnt5 == 1'b1)    // Reset if terminal count is reached.
        cnt5_reg <= 4'd0 ;

    else if (adv_cnt5 == 1'b1)
        cnt5_reg <= cnt5_next ;   // Advance the count once if the
                                  // time watch is still running.
    else
                    ;             // Otherwise, don't disturb.
end
```

Then the block for cnt5, with next assigned here.

(Refer Slide Time: 43:35)

```
// cnt6_reg is the Time watche's least significant SECONDS digit.
// This is reset or advanced only in the RUN and TIME mode of
// operation every 1 sec.

assign res_cnt6 = ((set_time != 1'b1)&(tbsec == 1'b1)&
                                          (cnt6_reg == 9)) |
                         ((adv_secs_time == 1)&(cnt6_reg == 9)) ;
```

Finally for cnt6. This is the simplest of all.

(Refer Slide Time: 43:39)

```
                        // cnt6 = 9 is the condition
                        // for resetting this counter.

assign adv_cnt6 = ((set_time != 1'b1)&(tbsec == 1'b1)&
                                                (cnt6_reg < 9)) |
                        ((adv_secs_time == 1)&(cnt6_reg < 9)) ;

                        // cnt6 = 0 to 8 are the conditions
                        // for pre-incrementing this counter.

assign cnt6_next = cnt6_reg + 1 ;        // Pre-increment the counter.

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

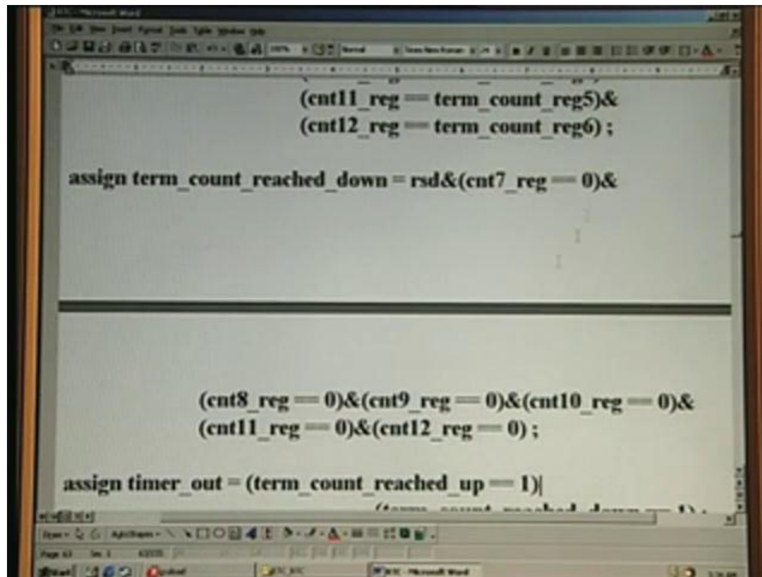
```
begin
  if (reset_n == 1'b0)
        cnt6_reg <= 4'd0 ;       // Initialize when the system is reset.

  else if (res_cnt6 == 1'b1)     // Reset if terminal count is reached.
        cnt6_reg <= 4'd0 ;

  else if (adv_cnt6 == 1'b1)
```

The condition is cnt6 is equal to 9 and advance is 0 to 8 condition. Again, the pre-increment is here and the actual realization of cnt6 is here.

(Refer Slide Time: 43:51)

```
            cnt6_reg <= cnt6_next ;    // Advance the count once if the
                                       // time watch is still running.
  else
                       ;               // Otherwise, don't disturb.
end


//                 Stop watch implementation starts here

// RUN, STOP WATCH, DOWN mode

assign term_count_reached_up = (set_stopw != 1)&
                         (down_upn == 0)&(start_stopn_reg == 1)&
                          (cnt7_reg == term_count_reg1)&
                          (cnt8_reg == term_count_reg2)&
```

```
//                 Stop watch implementation starts here

// RUN, STOP WATCH, DOWN mode

assign term_count_reached_up = (set_stopw != 1)&
                         (down_upn == 0)&(start_stopn_reg == 1)&
                          (cnt7_reg == term_count_reg1)&
                          (cnt8_reg == term_count_reg2)&
                          (cnt9_reg == term_count_reg3)&
                          (cnt10_reg == term_count_reg4)&
                          (cnt11_reg == term_count_reg5)&
                          (cnt12_reg == term_count_reg6) ;

assign term_count_reached_down = rsd&(cnt7_reg == 0)&
```



This completes the running time. Next is stopwatch implementation. We need to implement the stopwatch in precisely the same way, except that we have to do tamper with cnt7 through cnt12. That is what we see here. If you are in up counting mode, term_count_reached_up variable will be set for this condition. For example, when term_count_reg1 through term_count_reg6 are equal to the respective running counters and if start_stop is in start mode and down_up is in up mode and set_stopw is not equal to 1, whenever it is not in set stopwatch mode, then only the up counter should run. It must run only in all other modes other than the set stopwatch mode. It should also be in up mode, because what you want is up counter. When the terminal count match has been found is sensed only by this. Naturally, you should have started the button or pushed the button for starting – that is what is stored here. term_count_reached_down is the exact counter for down counter. rsd will be explained now. For cnt7 and for all these conditions, you can just have a look.

(Refer Slide Time: 45:23)

```
(cnt8_reg == 0)&(cnt9_reg == 0)&(cnt10_reg == 0)&
(cnt11_reg == 0)&(cnt12_reg == 0) ;

assign timer_out = (term_count_reached_up == 1)|
                   (term_count_reached_down == 1) ;

assign rsd = (set_stopw != 1)&(down_upn == 1)&
             (start_stopn_reg == 1) ;

// start_stopn_reg == 1 means START, otherwise STOP.

// cnt7_reg is the Stop watch's most significant HOUR digit.
// This is reset or advanced only in the RUN and STOP WATCH
```

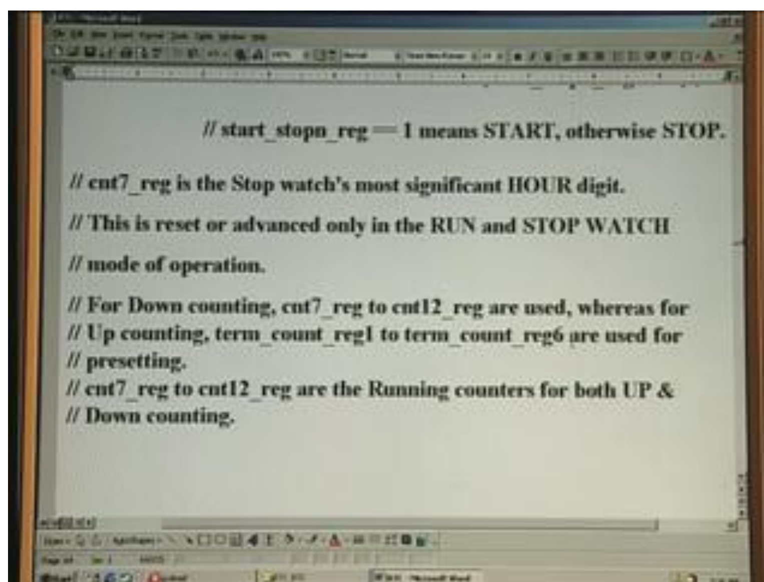
All must be 0, because it is down counting. Notice that all the counters are 0. Then, it will stop as you have witnessed in the demo already – when it is 000 after 50 minutes of the stopwatch. You have already seen this. When it becomes 00, it rang a buzzer and stopped. It forced the display to all 0s. That is precisely being done because of this statement.

(Refer Slide Time: 45:55)

```
(cnt8_reg == 0)&(cnt9_reg == 0)&(cnt10_reg == 0)&
(cnt11_reg == 0)&(cnt12_reg == 0) ;

assign timer_out = (term_count_reached_up == 1)|
                   (term_count_reached_down == 1) ;

assign rsd = (set_stopw != 1)&(down_upn == 1)&
             (start_stopn_reg == 1) ;

// start_stopn_reg == 1 means START, otherwise STOP.

// cnt7_reg is the Stop watch's most significant HOUR digit.
// This is reset or advanced only in the RUN and STOP WATCH
// mode of operation.
```

timer_out is also displayed. Had you noticed in a bar graph there on the display earlier, one central bar will be switched on. That is also available for the outside world. We can fire a rocket by using that output. That is precisely here and timer out. When timer count reaches up and a match is found, either that is found or down count match is found, only for these two conditions, you need to set the timer_out. That will give you the timed output. We have already seen rsd, which means that it is not in set stopwatch mode and it is in down mode. The abbreviation rsd means run stopwatch down. It must also be in start mode.

(Refer Slide Time: 46:47)



cnt7_reg is the stopwatch's most significant digit. This is reset or advanced only in the RUN and STOPWATCH mode o operation. For down counting, cnt7_reg through cnt12_reg are used, whereas for up counting, term_count_reg1 to term_count_reg6 are used for presetting. Only for presetting, we use this – term_count_reg1 through term_count_reg6. cnt7 through cnt12 are running counters for both up and down counting. If this is clear, everything else will be understood.

(Refer Slide Time: 47:19)

```
assign res_cnt7 = ( ((set_stopw == 1)&(down_upn == 0))|
                    ((adv_hrs_sw == 1)&(cnt7_reg == 2)&
                                        (cnt8_reg == 3)) );

                    // Counter must be reset in SET UP mode
                    // cnt7 – cnt8 = 23 are the conditions for
                    // resetting the counter in SET DOWN COUNTER
                    // mode.

assign adv_cnt7 = ((set_stopw != 1)&(down_upn == 1'b0)&
                   (term_count_reached_up == 0)&
                   (tbsec == 1)&(cnt7_reg < 2)&(cnt8_reg == 9)&
              (cnt9_reg == 5)&(cnt10_reg == 9)&(cnt11_reg == 5)&
```

Based on that, we see some more here. Once again, resetting, advancing counters, and so on. For example, cnt7 is the hours MSD. I am going to read only the comment hereafter, because all the other nomenclature is exactly the same as we have followed earlier. The counter must be reset in SET UP mode and cnt7, cnt8 must be 23. These are the conditions for resetting the counter in SET DOWN COUNTER mode. This is the down counting mode and we are in set. When that happens, when you want to the reset cnt7 is put here. This is the condition.

(Refer Slide Time: 48:04)

```
                    // mode.

assign adv_cnt7 = ((set_stopw != 1)&(down_upn == 1'b0)&
                   (term_count_reached_up == 0)&
                   (tbsec == 1)&(cnt7_reg < 2)&(cnt8_reg == 9)&
              (cnt9_reg == 5)&(cnt10_reg == 9)&(cnt11_reg == 5)&
                   (cnt12_reg == 9)&(start_stopn_reg == 1'b1)) |
              ((adv_hrs_sw == 1)&(cnt7_reg < 2)&(cnt8_reg == 9)) ;

                    // cnt7 - cnt12 = 09 59 59 or 19 59 59 are the
                    // conditions for pre-incrementing the counter.

assign cnt7_next = cnt7_reg + 1 ;        // Pre-increment the counter.
```

```
// mode.

assign adv_cnt7 = ((set_stopw != 1)&(down_upn == 1'b0)&
          (term_count_reached_up == 0)&
          (tbsec == 1)&(cnt7_reg < 2)&(cnt8_reg == 9)&
     (cnt9_reg == 5)&(cnt10_reg == 9)&(cnt11_reg == 5)&
          (cnt12_reg == 9)&(start_stopn_reg == 1'b1)) |
     ((adv_hrs_sw == 1)&(cnt7_reg < 2)&(cnt8_reg == 9)) ;

          // cnt7 - cnt12 = 09 59 59 or 19 59 59 are the
          // conditions for pre-incrementing the counter.

assign cnt7_next = cnt7_reg + 1 ;       // Pre-increment the counter.
```

The condition for advancing the same counter is here. cnt7 through cnt12 must be 09, 59, 59 or 19, 59, 59. These are the conditions for pre-incrementing the counter. You can see that we have followed exactly the same argument for counter 1 earlier for time. Now, we are talking with respect to stopwatch. We have pre-increment here.

(Refer Slide Time: 48:28)

```
assign decr_cnt7 = rsd & (cnt8_reg == 0)&(cnt9_reg == 0)&
          (cnt10_reg == 0)&(cnt11_reg == 0)&
               (cnt12_reg == 0)&(cnt7_reg > 0)&
                    (cnt7_reg <= 2)&(tbsec == 1) ;

assign cnt7_nextd = cnt7_reg - 1 ;      // Pre-decrement the counter.

always @ (posedge clk or negedge reset_n)

begin
```

In addition to that, we also need decr_cnt7, because it may go into down count mode. We have already seen that this is for down count. When all the counters cnt8, cnt9, cnt10, cnt11 and cnt12 are 0s, it is decrementing. When this happens, what should you do? cnt7 must be greater than 0,

cnt7 must be less or equal to 2 and it should also be 1 second. Only then, you take this decision – only then, you decrement cnt7. That is the condition when you should decrement.

(Refer Slide Time: 49:09)

```
assign decr_cnt7 = rsd & (cnt8_reg == 0)&(cnt9_reg == 0)&
                              (cnt10_reg == 0)&(cnt11_reg == 0)&
                                    (cnt12_reg == 0)&(cnt7_reg > 0)&
                                         (cnt7_reg <= 2)&(thsec == 1) ;

assign cnt7_nextd = cnt7_reg - 1 ;        // Pre-decrement the counter.

always @ (posedge clk or negedge reset_n)

begin

 if (reset_n == 1'b0)
      cnt7_reg <= 4'd0 ;       // Initialize when the system is reset.
```

When you should preset, advance, reset or decrement, etc., you will have to do appropriately. This is the core of the working. You need to pre-decrement here.

(Refer Slide Time: 49:22)

```
begin

 if (reset_n == 1'b0)
      cnt7_reg <= 4'd0 ;       // Initialize when the system is reset.

 else if (res_cnt7 == 1'b1)    // Reset if terminal count is reached.
      cnt7_reg <= 4'd0 ;

 else if (adv_cnt7 == 1'b1)
      cnt7_reg <= cnt7_next ;   // Advance the count once if the
                                // time watch is still running.
 else if (decr_cnt7 == 1'b1)
      cnt7_reg <= cnt7_nextd ;  // Decrement the count once if
                                // the stop watch is still running.
```

That is what we are doing here and this is the cnt7 block. It is exactly the same, except that reset has been taken here, then advance and then decrement here.

(Refer Slide Time: 49:35)

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

// cnt8_reg is the Stop watche's least significant HOUR digit.
// This is reset or advanced/decremented only in the RUN and
// STOPWATCH mode of operation.

assign rsd_cnt8_res = rsd&((cnt7_reg > 2)|((cnt7_reg == 2)&
                      (cnt8_reg > 3))|((cnt7_reg < 2)&(cnt8_reg > 9))) ;
```

```
// STOPWATCH mode of operation.

assign rsd_cnt8_res = rsd&((cnt7_reg > 2)|((cnt7_reg == 2)&
                      (cnt8_reg > 3))|((cnt7_reg < 2)&(cnt8_reg > 9))) ;
                      // Illegal values

assign res_cnt8_set = ((adv_hrs_sw == 1)&(cnt7_reg < 2)&
                                         (cnt8_reg == 9))|
           ((adv_hrs_sw == 1)&(cnt7_reg == 2)&(cnt8_reg == 3)) ;

assign res_cnt8_sw = (set_stopw != 1)&(down_upn == 1'b0)&
           (term_count_reached_up == 0)&(tbsec == 1)&
             ( ((cnt7_reg == 2)&(cnt8_reg == 3))|
               ((cnt7_reg < 2)&(cnt8_reg == 9)) )&
               (cnt9_reg == 5)&(cnt10_reg == 9)&
               (cnt11_reg == 5)&(cnt12_reg == 9) ;
```

That is for cnt7. cnt8 is exactly similar, except that resetting will be a different condition.

(Refer Slide Time: 49:44)

```
assign res_cnt8_set = ((adv_hrs_sw == 1)&(cnt7_reg < 2)&
                        (cnt8_reg == 9))|
          ((adv_hrs_sw == 1)&(cnt7_reg == 2)&(cnt8_reg == 3)) ;

assign res_cnt8_sw = (set_stopw != 1)&(down_upn == 1'b0)&
               (term_count_reached_up == 0)&(tbsec == 1)&
                  ( ((cnt7_reg == 2)&(cnt8_reg == 3))|
                   ((cnt7_reg < 2)&(cnt8_reg == 9)) )&
                   (cnt9_reg == 5)&(cnt10_reg == 9)&
                   (cnt11_reg == 5)&(cnt12_reg == 9) ;
```

You have to take into account stopwatch also. This is the condition for that.

(Refer Slide Time: 49:52)

```
               // are the conditions for resetting this counter.

assign cnt8_res = (rsd_cnt8_res | res_cnt8_sw | res_cnt8_set) |
                  ((set_stopw == 1)&(down_upn == 0)) ;

assign adv_cnt8_set = ((adv_hrs_sw == 1)&(cnt7_reg < 2)&
                        (cnt8_reg < 9))|
                   ( (adv_hrs_sw == 1)&(cnt7_reg == 2)&
                        (cnt8_reg < 3) ) ;

assign adv_cnt8_sw = (set_stopw != 1)&(down_upn == 1'b0)&
                  (term_count_reached_up == 0)&(tbsec == 1)&
                     ( ((cnt7_reg < 2)&(cnt8_reg < 9))|
                    ((cnt7_reg == 2)&(cnt8_reg < 3)) )&
                    (cnt9_reg == 5)&(cnt10_reg == 9)&
```

cnt8 reset is required. These are all the conditions. This is advancing cnt8 when it is in set mode and this is when it is in stopwatch mode. All these are exactly the same. I am not going into the details. I will read out only the comments, if any. If it is exactly same as before, I do not have to do that. This is also exactly the same. cnt7, cnt8 is 00 to 18 or 20 to 22 and other counters must be 59, 59. These are all the conditions for pre-incrementing the counter.

(Refer Slide Time: 50:23)

cnt8 is incremented and the decrement conditions are here: 01 to 09, then 11 to 19 or 21 to 23. All other counters must be 0. This is the condition for decrementing the counter and the pre-decrement is here.

(Refer Slide Time: 50:40)

```
assign pres_cnt8 = rsd & (tbsec == 1)&(cnt8_reg == 0)&
                    (cnt9_reg == 0)&(cnt10_reg == 0)&(cnt11_reg == 0)&
                     (cnt12_reg == 0)&(cnt7_reg > 0)&(cnt7_reg <= 2) ;

                              // Preset if cnt7-cnt12 = 10 00 00 or 20 00 00.

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

```
begin

 if (reset_n == 1'b0)
         cnt8_reg <= 4'd0 ;      // Initialize when the system is reset.

 else if (cnt8_res == 1'b1)      // Reset if terminal count is reached.
         cnt8_reg <= 4'd0 ;

 else if (adv_cnt8 == 1'b1)
         cnt8_reg <= cnt8_next ;   // Advance the count once if the
                                   // stop watch is still running.
 else if (decr_cnt8 == 1'b1)
         cnt8_reg <= cnt8_nextd ;  // Decrement the count once if the
                                   // stop watch is still running.
 else if (pres_cnt8 == 1'b1)       // Preset if count down terminal
```

This is the block for actual decrement here.

(Refer Slide Time: 50:48)

```
else if (pres_cnt8 == 1'b1)        // Preset if count down terminal
                                   // count is reached.
        cnt8_reg <= 4'd9 ;
    else

              ;                    // Otherwise, don't disturb.
end

// cnt9_reg is the Stop watch's most significant MINUTES digit.
// This is reset or advanced only in the RUN and STOP WATCH
```

It is exactly the same. Preset happens here, when it is 9. 7, 8, this is the second digit. 7 and 8, right? What should be this preset condition?

(Refer Slide Time: 51:16)

```
assign cnt8_nextd = cnt8_reg - 1 ;     // Pre-decrement the counter.

assign pres_cnt8 = rsd & (tbsec == 1)&(cnt8_reg == 0)&
            (cnt9_reg == 0)&(cnt10_reg == 0)&(cnt11_reg == 0)&
              (cnt12_reg == 0)&(cnt7_reg > 0)&(cnt7_reg <= 2) ;

                      // Preset if cnt7-cnt12 = 10 00 00 or 20 00 00.
```

This is the preset condition. If it is 10 or 20, what should you do for cnt8? This is 7 and this is 8. What should you do? We are in decrement mode. This means decrement mode. Automatically, 0 must go to 9. You cannot reset it. You will have to preset. That is why preset has been used. We

will continue from this point onwards. We have seen up to cnt8 in the stopwatch mode. We will have to continue from cnt9, which we will do in the next lecture. Thank you.

(Refer Slide Time: 51:51)