

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

**Lecture – 51**

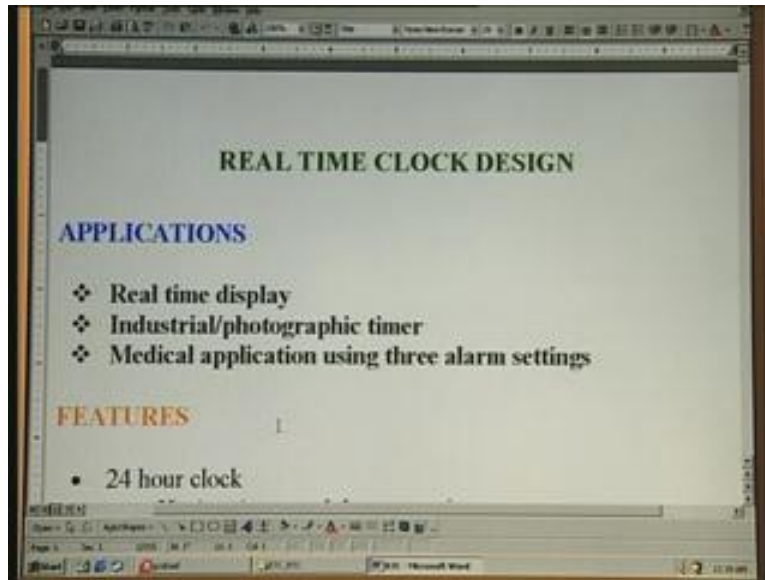
**System Design Examples using FPGA Board (Continued)**

(Refer Slide Time: 01:39)



We will take another design example for implementation on FPGA board, the real-time clock.

(Refer Slide Time: 02:19)



It has many applications right from real-time display, using which you can show 24 hours or 12 hours time range. What you see is the hardware here for the same.

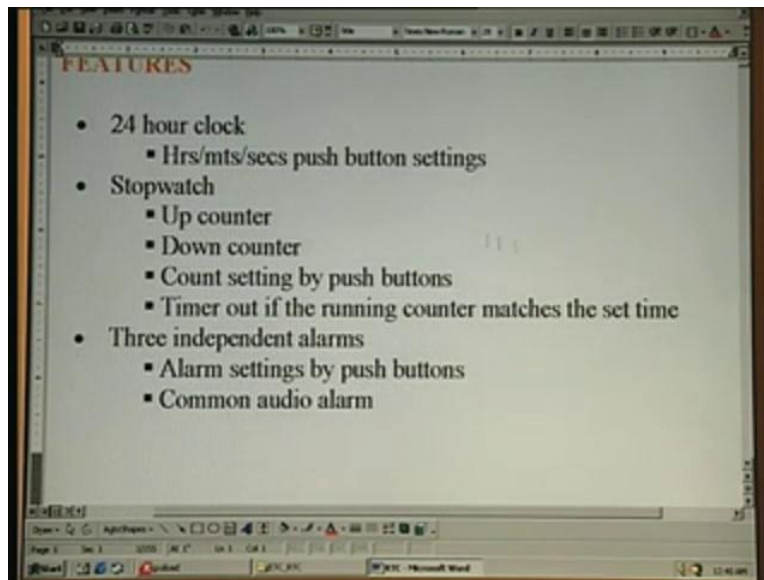
(Refer Slide Time: 02:38)



You notice that there is an FPGA board and also a digital I/O board and of course, there are power supplies, the details of which we will be seeing after a lecture or two. Let us get into the basic applications. Apart from the real-time display, we can have an up counter or down counter and we can use the same for an industrial or photographic timer. You can use the same timer with three different alarm settings for medical treatment, which demands the administration of medicine on a timely basis.

See, for example, patients with epilepsy are prone to fits and other attacks if they do not take medicine within 45 minutes to an hour. Needless to say, this will prove to be very useful for such patients.

(Refer Slide Time 03:44)

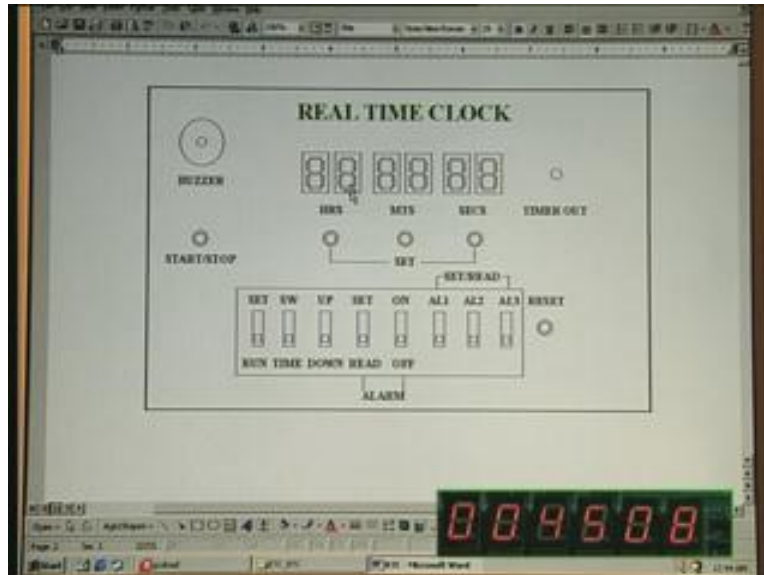


The features of the real-time clock are as follows. What we have designed is a 24-hour clock and you will be required to convert it into a 12-hour clock later on, when we deal with the assignments for you. The clock will feature hours, minutes, seconds and push button settings. You can have three independent push button switches to set the same. You have a stopwatch that can count up or down and count setting by push buttons. You can set the counter up or down by using the very same push button that you have used for this.

There is also a timer out. After having **run out of the ... timed out**, that is after running the set time, be it in up counter mode or down counter mode, when it finds the match with the set time, this timer out signal (a single-bit output) will be turned on. So you can use this, for example, for firing a rocket if you wish to. You can have a down counter and when the time is all 0s, you can fire the rocket and timer out will go to 1 and remain as such. If you do not like this mode, you can have a real-time counting in the up counter mode, wherein you can actually see the real time and when the set time matches with what is running (which is the same as the running time), once again, a timer out will be issued and you can use any of these two as per your real application.

We have three independent alarms that can be set using DAP switch and the alarm settings can be done by push buttons as you have done for the up counter, down counter or time mode setting. There is a common audio alarm. An alarm is made using a piezoelectric buzzer.

(Refer Slide Time 05:44)



When you design a system, what is important is the construction of the product. What will be your end product must be borne in mind. To start with, as a layman, if you are in a shop to buy a clock, you will be looking at only the display and not the inner details as to, whether you have used a controller or any other embedded system or microprocessor or

FPGA – it does not really matter as far as the customer or user is concerned. He is a common man or layman who does not know the intricacies of the design involved.

So, as a designer, you should always think from that point of view when you design a system. What appeals to our mind is you want to use a real-time clock and you want to keep it on your bench. Let us say we wish to put the real-time clock right on the tabletop. Naturally, you need a six-digit display for the same for hours, minutes and seconds. When you have this, there must be a provision to set them. You can locate the push buttons right beneath this and name it like this. The layout is quite important because it will be very easy to learn how to operate this from the common man's point of view.

There is a buzzer, which is a piezoelectric buzzer and there is a timer out, LED here. In addition to this, you will also have an output. We have already made use of ULN2003 to trigger that. The timer out can be connected to one of the outputs of ULN2003 and the other end of ULN2003, which is Darlington pair open collector, can be connected to a solenoid or a relay or a solid-state relay. You can connect any real-time output. For example, for photographic exposure, you can connect this timer out to the actual lamp that really exposes the film and you can use for the same thing as well.

You also need other controls. There are so many modes such as the time mode. If you want the time mode, you can set using this here. The down position is the time and the up position is the stopwatch. Once you have this, you also need to decide whether the time must run or stopwatch must run. If you put it in run stopwatch mode, naturally it runs the stopwatch. If you want to set the time, you put it to set here and then let it be in time. If you want the stopwatch setting, set stopwatch is all that you have to do. As you can see, so it is. It will be a simple thing, which even a child can handle if you design bearing in mind the ease of handling as far as the common user is concerned.

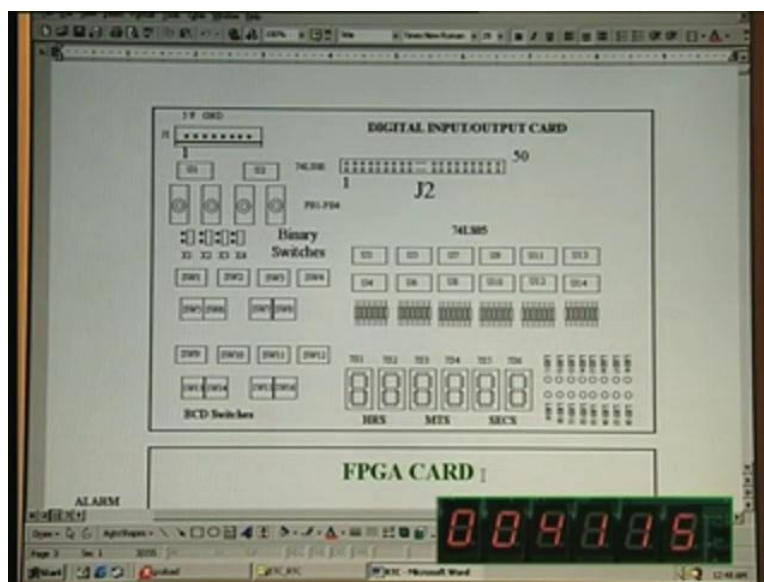
You have an up counter as well as a down counter, you have a switch for that, which you can set to up or down. Similarly, you need an alarm. These two are earmarked for alarm. You can switch off the alarm on switch on the alarm. This is only to switch off the actual audio alarm. This does not really reset whatever you have set. There are three different

alarms that you can set, alarm1 through 3. Of these, the topmost priority is AL1 (alarm1) and AL3 has the lowest or least priority.

If you take to this position, you would be setting or reading the respective alarm. You also have one set and read. This appears to be redundant but it is not really so because we also need to have the same controls for the stopwatch also. Once again, in a stopwatch, we have an up counter as well as down counter. Perhaps if you give a little more thought, you can save some of the switches, but I did not make much attempt in reducing because these switches are already available on the hardware that we have already used for the traffic controller earlier.

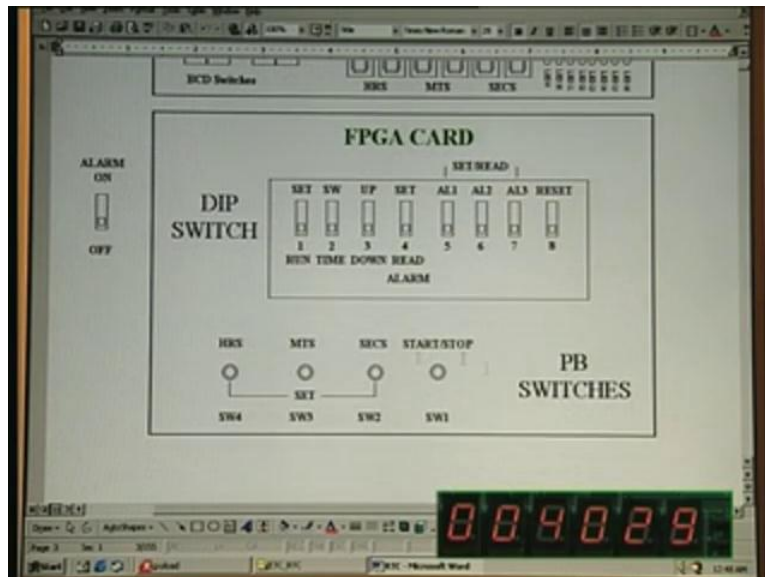
In addition to this, if you want to **reset...** Normally, this will not be provided for the user. This will be inside the unit, but just for the sake of completeness, I have put it here. In fact, you should conceal such resets from being tampered by the normal customer. This is the goal in mind. With this in mind, we should go for what hardware is available, especially at the R&D stage, just as we are right now. We are trying to learn the design, not the actual product that is going directly to the market – we are in the development stage. Nevertheless, we should start with this and then go to the actual hardware that we already have.

(Refer Slide Time 10:55)



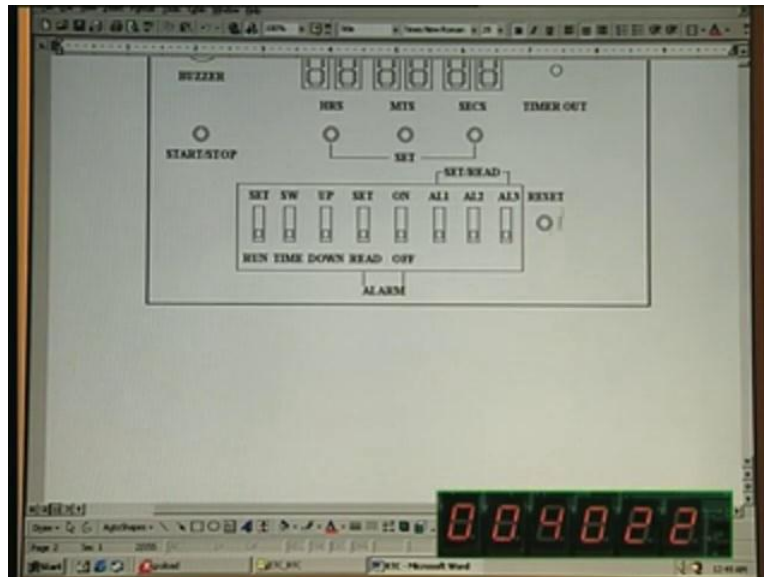
We have already seen the use of the digital input/output card and if you notice, we had six digits there. I am not going into the details of this because I have already explained to you how to use this. They are all socketed totally so that we can configure for any other application. The very same board such as digital I/O or FPGA board that is going to follow can be reconfigured for any number of applications, of course, subject to the limitation of the hardware that you have. What all you need in this digital I/O card is only this six-digit display. Note that I have just marked hours, minutes and seconds, which was not there earlier when you had seen it the last time. It is only to earmark hours, minutes, seconds and it uses only six digits. There is nothing more on that board.

(Refer Slide Time: 11:43)



On the FPGA card, we have an eight-bit DIP switch and we have already seen the setting earlier.

(Refer Slide Time: 11:50)

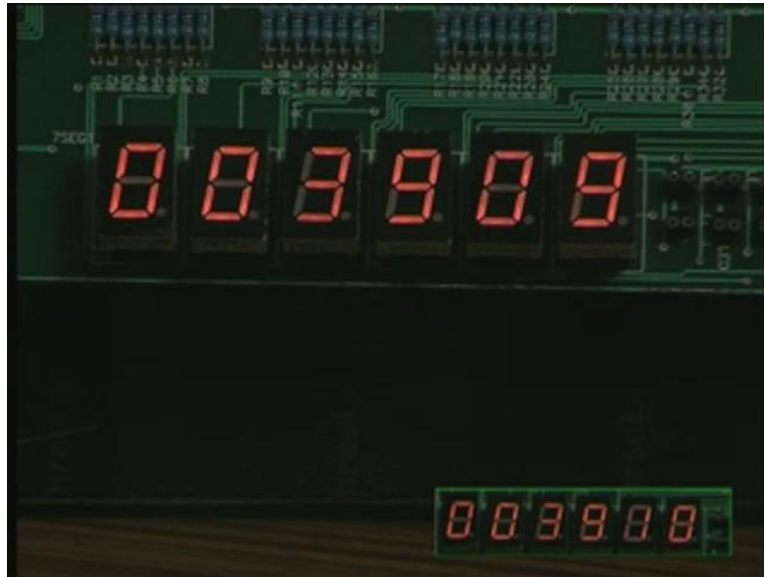


Our goal was this and naturally, you cannot get a one-to-one correspondence. There may have to be a slight difference when you configure the same thing because the hardware is already done and you have no choice over this. As far as possible, I have retained the very same arrangement for all the switches, except that reset is a part of the DIP switch here, which was shown as a push button switch in the previous layout. You have hours, minutes and seconds right at the FPGA board left hand side corner. They are all numbered SW4 through SW1. There is one more switch here, so that we may start or stop the up counter or down counter. This was also depicted in the earlier layout that we have seen.

These are naturally push button switches and this is a DIP switch. We have intentionally suppressed all other hardware on the FPGA card. In addition to this, we need one more switch – alarm on and off switch. This is a separate switch here, which is there on the board on your right end.

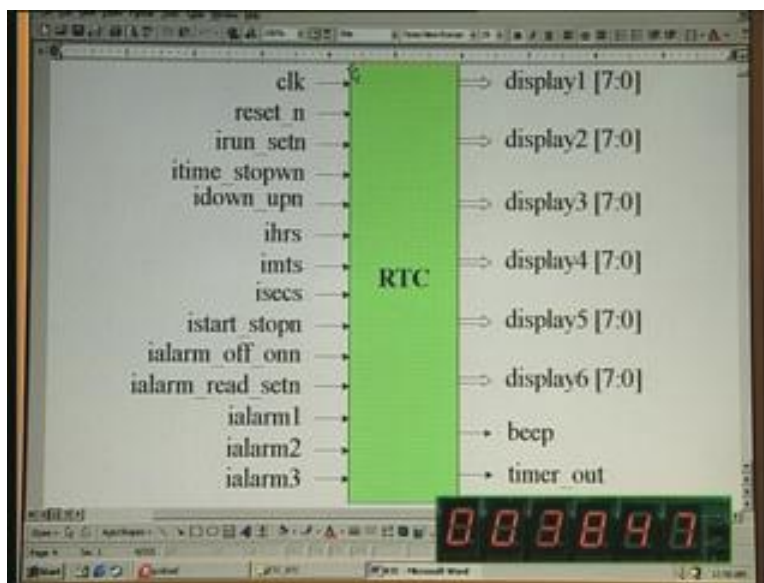


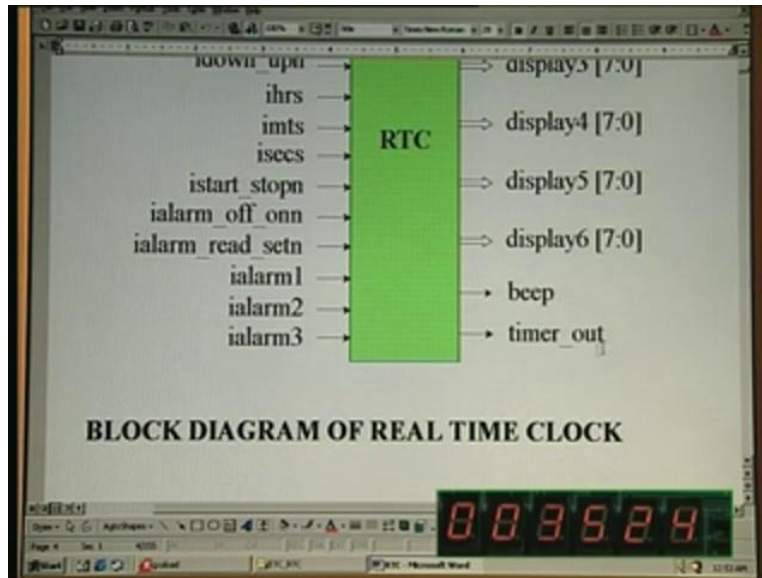
(Refer Slide Time: 13:02)



Now, let us go into more details. First, we have to formulate a specification. Before we specify anything, what we want to do is we are going to make a chip. We need a block diagram for the chip. For example, this chip will have to have different switches that you have already seen earlier.

(Refer Slide Time 13:25)





In addition, you need some more signals. For example, you need a system clock and you need a reset, which was already there. We need a run\_set switch, which was shown there. Now, notice that I have just added one i here, because this is the block diagram and the pin diagram. This is what we want to implement using Verilog. While using the Verilog code, we are going to use the very same nomenclature and nothing different from this.

For that, we have many such signals. For example, `irun` will be there, there will be `run`, `run_set` and so on. We will have three variants for each of these inputs. We will see the details later on. We need a time stopwatch and we already have seen all the switches, so I will quickly go through this. You need a down\_up and i stands for input. This is the physical input pin that you have to connect. You need three push button switches for hours, minutes, seconds setting and you need to start or stop if you are in the up or down counter mode.

If you want to turn on or turn off alarm, you have this switch. If you want to read or set the alarm, you have another switch. For three independent alarms, you need three switches here – alarm1 through 3. As far as the outputs are concerned, we have six digits of a seven-segment display that you have already seen and I hope it is running on a corner. What is running there on the corner is a display for down-counting. For example, the application for this studio demands a 50-minutes program and every episode is going to have 50 minutes. When we started, we have switched on the down counter and I hope

it is running in some corner. When the time is over, it will be 0, and it will stop as well as give the buzzer output. That is the purpose of this stopwatch. Using the very same hardware, we have done for this particular studio application.

Coming to the display, those seven-segment displays are six in number – display1 through display6, each of which is eight bits in width. We will go into more details, what each bit is doing. We need seven segments plus one decimal point also. It takes eight bits totally – that is the reason we have 7 through 0 (eight bits for each of the displays here). We also need a beeping sound alarm and so what we have to do is create a square pulse and that is what is output here in beep. This is connected straightaway to the ULN2003 or 74LS05 – I think it is the latter, the output of which is connected to the buzzer. The buzzer operates at around 8.5 Volts, which happens be the power supply for the FPGA board that we use. This is the naturally the pin diagram or you can regard it as the block diagram of the real-time clock. You have finally one more output for timer out, which we have already explained. Next is the actual signal description.

(Refer Slide Time: 16:54)

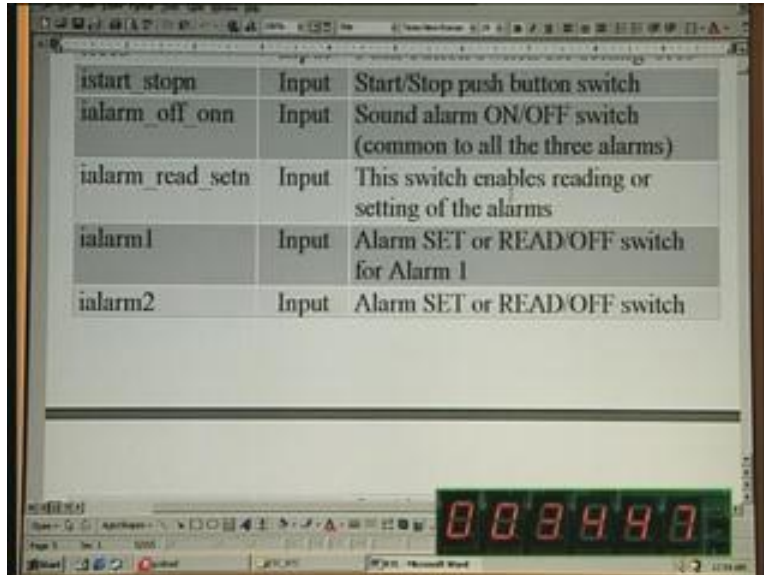
Signal	Input/Output	Description
clk	Input	System clock
reset_n	Input	Asynchronous, active low sys. reset
irun_setn	Input	RUN/SET mode switch
itime_stopwn	Input	TIME/STOP WATCH mode switch
idown_upn	Input	UP/DOWN mode switch
ihrs	Input	Push button switch for setting 'hrs'
imts	Input	Push button switch for setting 'mts'
isecs	Input	Push button switch for setting 'secs'
istart_stopn	Input	Start/Stop push button switch
ialarm_off_onn	Input	Sound alarm ON/OFF switch (common to all the three alarms)

003458

Some of this I have already described and so, I will quickly go through each of this here. **clk** is the system clock. reset is asynchronous and it is active low, as we have used in many design examples or applications that we have seen before. You have a RUN/SET

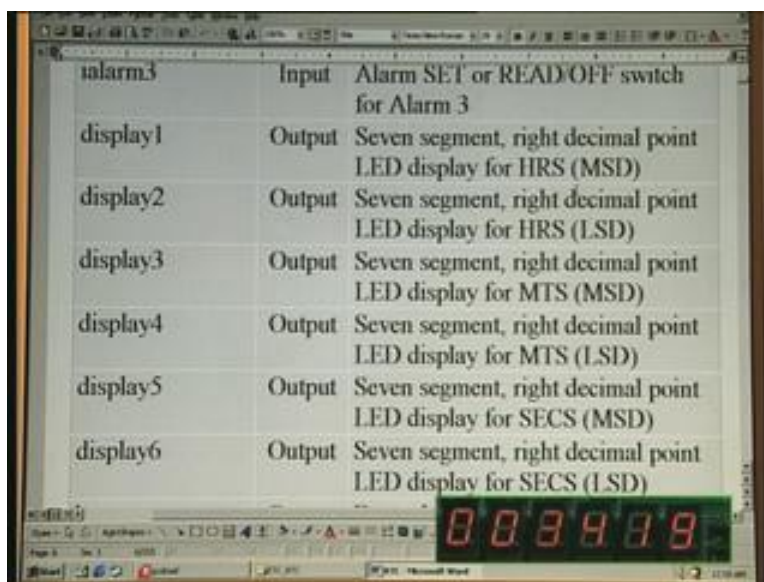
switch, then TIME/STOP switch, an UP/DOWN switch, then push button switches for hours, minutes, seconds, then a Start/ Stop push button switch, then sound alarm ON/OFF, which is common to all the three alarms.

(Refer Slide Time: 17:25)



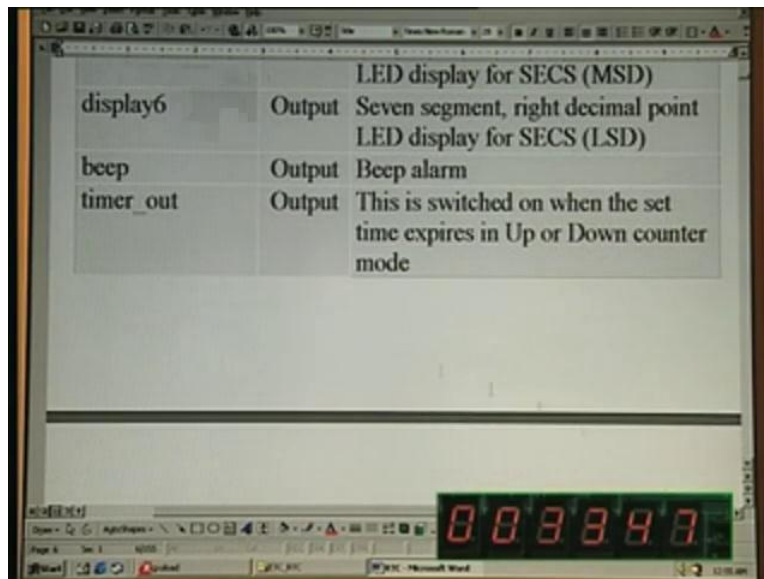
Then, this switch alarm\_read\_setn enables reading or setting of the alarms. You can set or read or even switch off the alarm1 or alarm2 or alarm3 as you see here, listed separately.

(Refer Slide Time: 17:40)



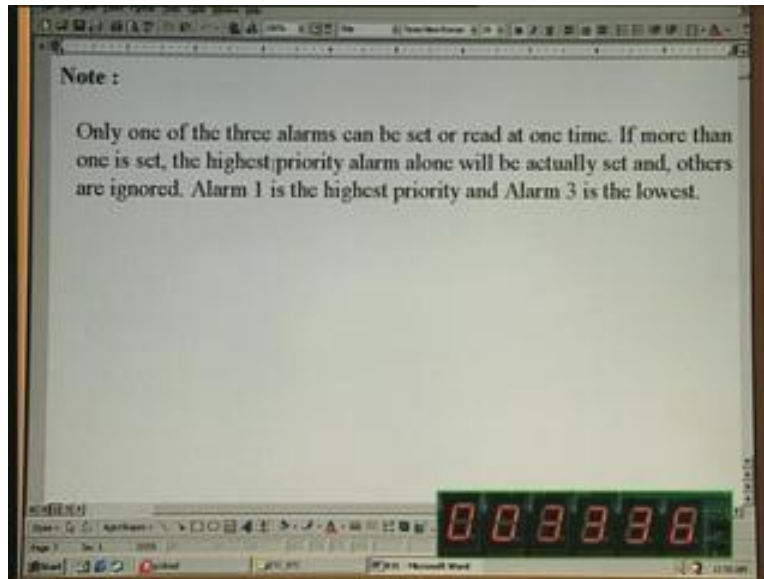
There are obviously six digit displays, which we have already seen. Here, what it means is display1 is the hours and so also is display2. Hours will be on your left side, which you are seeing on the stopwatch running there and that will be the most significant digit on the extreme left. That is display1 or counter1 that we are going to see when you look into the actual Verilog code. So is the case for display2, which happens to be just the LSD hours. So is the case for minutes and there are two digits once again – MSD and LSD, corresponding to display3 and display4. 5 and 6 are MSD and LSD, respectively, for seconds display.

(Refer Slide Time: 18:25)



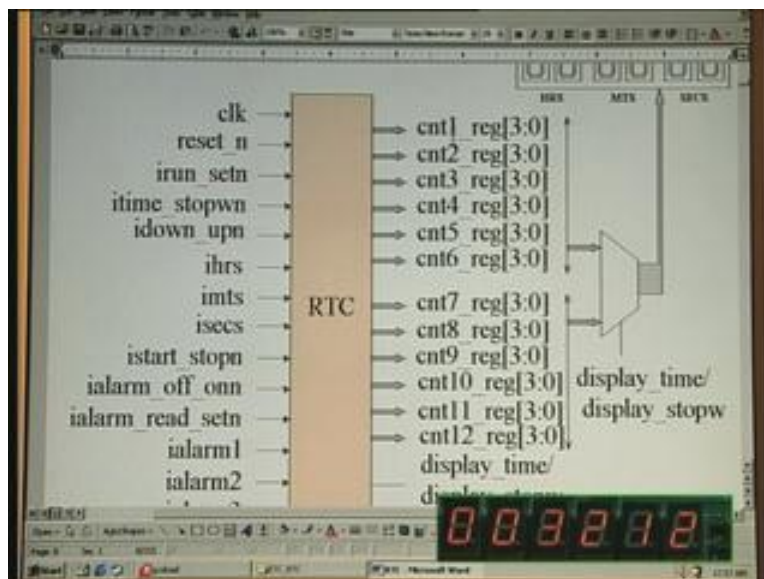
You have a beep alarm, which you have already seen the last time. The last is timer\_out and this is switched on when the set time expires in up or down counter mode.

(Refer Slide Time: 18:34)



There is a note here and I will read it out for you. Only one of the three alarms can be set or read at one time – you cannot do all the three simultaneously. You have to do one after another. If more than one is set, the highest priority alarm alone will be actually set and the others will be totally ignored. Alarm 1 is the highest priority, alarm 2 is of medium priority and alarm 3 is of lowest priority.

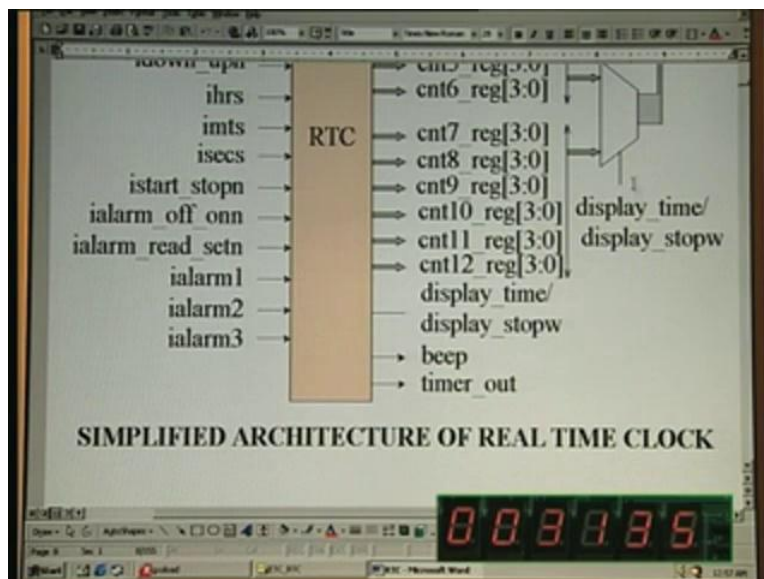
(Refer Slide Time: 20:00)



What we have already seen we are seeing once again, and this time it is going to be a little different. Now, we are going to start the Verilog coding. Prior to that, you should grasp the fundamental principle of the design. This is basically the same pin out that we have already seen and I am not going into these details. Instead of display1 through display6, that is actually here. This is what is going to power the seven-segment display that is running right now there.

If it is in stopwatch mode, we are going to use cnt7 through cnt12, which is the case for this particular example that you are seeing. The running stopwatch is using cnt7 through cnt12. cnt7 corresponds to display1 in the same order hours, minutes and seconds. It is exactly the same order that we have seen for the display. So is the case for cnt1 through 6, except that this is going to show the actual real time. The real time that you have in your watch display will be through this cnt1 through 6 and one of this will have to be routed to this display and that is done by a MUX here. So many bits of information are going here or here. The MUX is the selector and it will select either this or this depending upon this control. This is not as simple as just a single signal – there are multiple signals and there may be six signals or so. When we go into the details, we will see more about this.

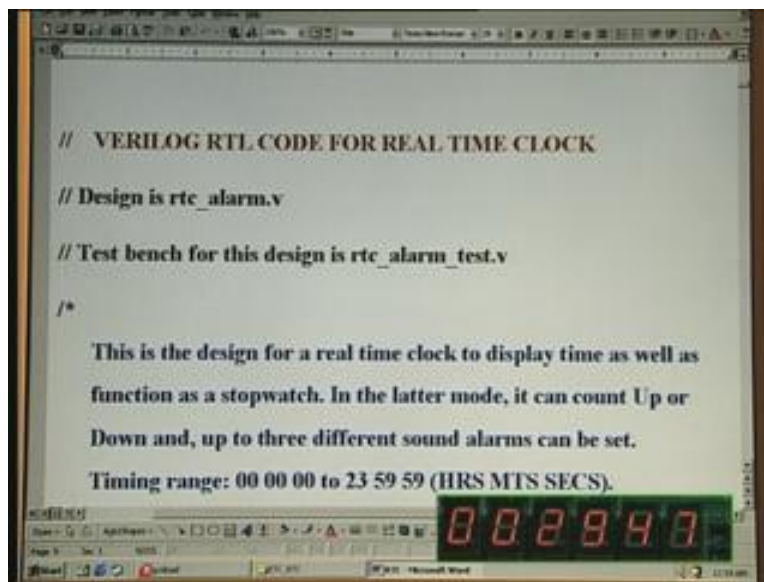
(Refer Slide Time: 20:37)



It is enough to say primarily they are in either time mode or in stopwatch mode. If it is in time mode, let us say if it is 1, so cnt1 through cnt6 is connected to the display that you see there. Otherwise, when stopwatch is on; when it is 0, cnt7 through cnt12 will be routed and 1 through 6 will be off.

This is the simplified architecture of the real-time clock. We have seen in any design how a customer will have to look. We have to look from his point of view and fashion it and then go on to the architecture. If there is an algorithm, we have to do as we have done in detail for MPEG, JPEG, making use of DCT quantization earlier, which we have seen elaborately. The same is the approach for this design as well. For any other design, I strongly recommend, from my own experience, that you adopt a similar strategy. Now we are going into the Verilog code.

(Refer Slide Time: 21:34)



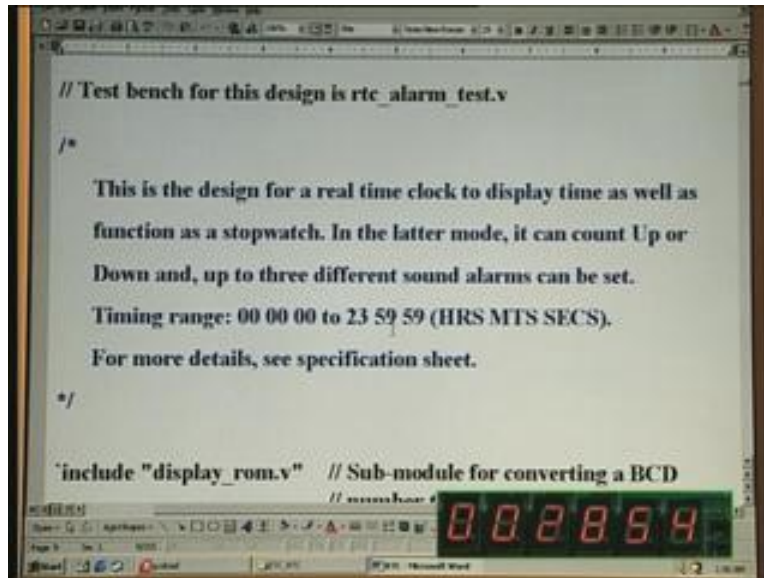
```
// VERILOG RTL CODE FOR REAL TIME CLOCK
// Design is rtc_alarm.v
// Test bench for this design is rtc_alarm_test.v
/*
This is the design for a real time clock to display time as well as
function as a stopwatch. In the latter mode, it can count Up or
Down and, up to three different sound alarms can be set.
Timing range: 00 00 00 to 23 59 59 (HRS MTS SECS).
```

This is the Verilog code for a real-time clock. The design file you have to put this particular design is going to be more than two hours lecture for this, the code that is going to run. Please stop me at any point of time if you are not clear, because we are going into more and more intricate details. It is like entering a labyrinth, let me warn you, so follow it very carefully. This particular file is a design file, so we will use nomenclature by the common names. For example, I want an alarm also. There are three alarms. The nomenclature has rtc, standing for real-time clock, with alarm facility. The .v means it is



a Verilog file. The file that you are right now seeing is .v. You have to locate all these Verilog codes, which is the same as writing any other C program, except that this is hardware design language and not a software design language. We have seen many times the difference between C and this.

(Refer Slide Time: 22:37)



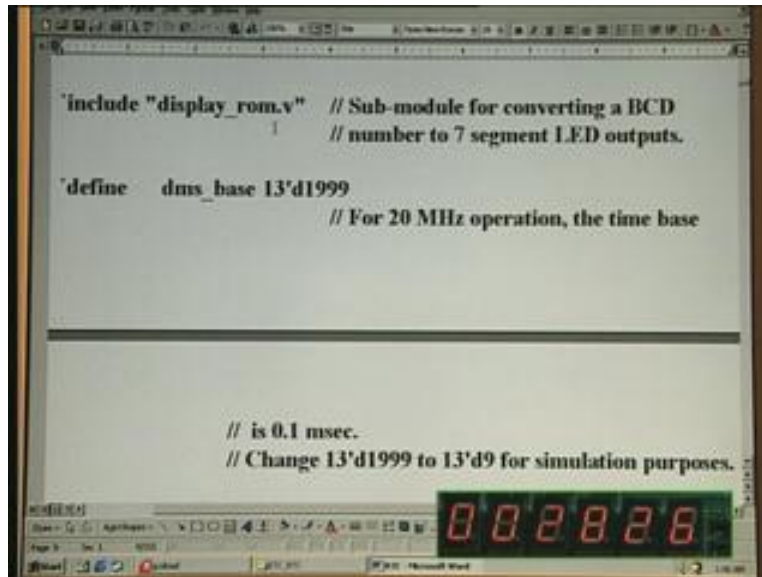
```
// Test bench for this design is rtc_alarm_test.v
/*
  This is the design for a real time clock to display time as well as
  function as a stopwatch. In the latter mode, it can count Up or
  Down and, up to three different sound alarms can be set.
  Timing range: 00 00 00 to 23 59 59 (HRS MTS SECS).
  For more details, see specification sheet.
*/

#include "display_rom.v" // Sub-module for converting a BCD
```

I will read out the comments here. You also need a test bench to test this. I have not written a very elaborate test bench. I will explain the further details later on. That will be `rtc_alarm_test.v`. This is the test bench in order that we may test the design. This is the design for real-time clock to display time as well as function as a stopwatch. In the latter mode, it can count up or down and up to three different sound alarms can be set.

I have separated out hours and minutes for each of the readings, although it is not so in the hardware that we have. It is a 24-hours timer and it will go right up to 23, 59, 59, after which it will wrap around this and keep going on so long as there is power available to the system. It is in hours, minutes and seconds. For more details, see the specification sheet. We have actually come through the specification sheet. If you have a doubt, go back to the specification sheet.

(Refer Slide Time: 23:40)



```
include "display_rom.v" // Sub-module for converting a BCD
                        // number to 7 segment LED outputs.

define    dms_base 13'd1999
          // For 20 MHz operation, the time base

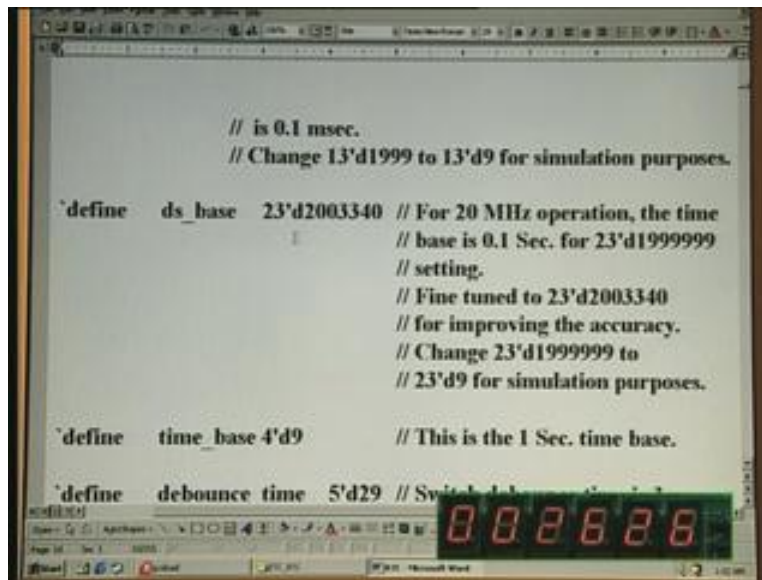
// is 0.1 msec.
// Change 13'd1999 to 13'd9 for simulation purposes.
```

The real code starts here and here we are going to use only one submodule, which is display\_rom. It is a code conversion from BCD to seven-segment LED outputs and basically what is going to run as the timer or counter is counter1 or counter7 as we have seen before. It will be four bits in width and that will have to be converted into seven bits. There is one more bit also earmarked for the decimal point. We are not going to use it in this particular project. It might probably come as an assignment, I am not very sure.

We have to include that submodule here and that is why we say include. We also need some variables. For example, I want to keep track of how many milliseconds have passed or how many seconds have passed or how many deci-milliseconds, which means 0.1 millisecond, have passed. This is for 0.1 millisecond and we have to keep track of how much time has elapsed. For that, we need to define. It simply means dms\_base is equal to 1999. We are going to operate a 20-Megahertz operation and the time base is 0.1 millisecond. If you want to change it for simulation purposes, you can make it 9 instead of 1999 so that we can run it much faster. Otherwise, it will be an impractical task for you to run the test bench because we are no longer dealing with a small number but almost astronomical numbers. We are dealing with hours, minutes and seconds. Again, from 20 Megahertz onwards, we have to start counting. That means 20 million count, followed by

60 for seconds, 60 for minutes and then 24 hours. You see it has already skyrocketed to astronomical heights. Writing an elaborate test bench will be a very difficult affair.

(Refer Slide Time: 25:44)



```
// is 0.1 msec.  
// Change 13'd1999 to 13'd9 for simulation purposes.  
  
'define ds_base 23'd2003340 // For 20 MHz operation, the time  
// base is 0.1 Sec. for 23'd1999999  
// setting.  
// Fine tuned to 23'd2003340  
// for improving the accuracy.  
// Change 23'd1999999 to  
// 23'd9 for simulation purposes.  
  
'define time_base 4'd9 // This is the 1 Sec. time base.  
  
'define debounce_time 5'd29 // Switch debounce time
```

The screenshot shows a code editor window with the above Verilog code. At the bottom right of the editor, there is a digital display showing the number '000000' in red LEDs.

We need another definition. For example, we need decisecond, for which we need a 23-bit counter later on. For that, we need to set it here. In this setting here, for 20-Megahertz operation, the time basis is 0.1 second because decisecond implies 0.1 second. For 23, all 1 9s setting, this is what is required for 20 Megahertz. It counts right from 0 to...

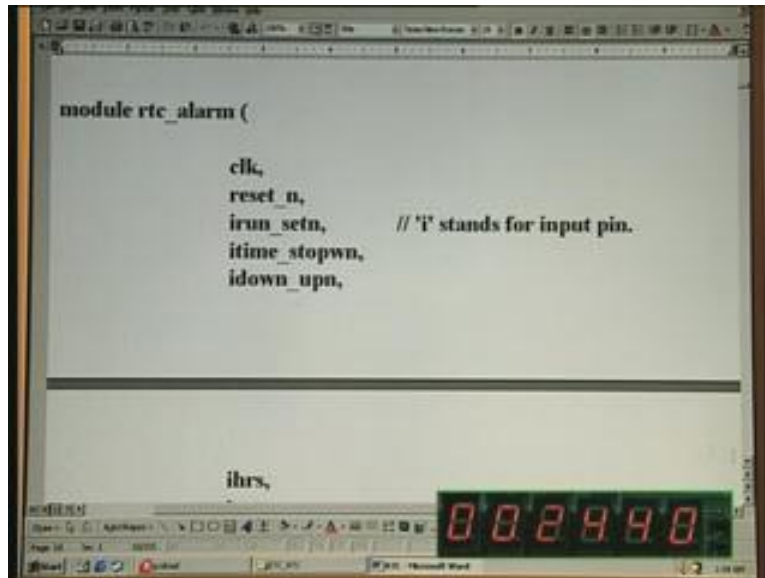
As we have done for non-retriggerable mono earlier, here also it demands going from 0 through whatever is the last count here – 1 followed by 999999. Normally, the FPGA card will have a crystal oscillator, which may not run very accurately. In fact, this particular card has some inaccuracy and so we have to fine-tune it. Fortunately, we have the software way of fine-tuning and that is simply done by changing this value.

Theoretically, you should have put 1 followed by all 9s here, but I have put it here so that I get an accuracy within less than a second for this 50 minutes.

I hope you are timing in your watch and crosschecking whether it is really true. I have checked couple of times and I am satisfied with that. I think it is 0.5 second or even less – that is the accuracy you get. We need to define the time base, which is seconds. We put only 9 there because we are going to count only 0.1 second here. If you count 0.1 second

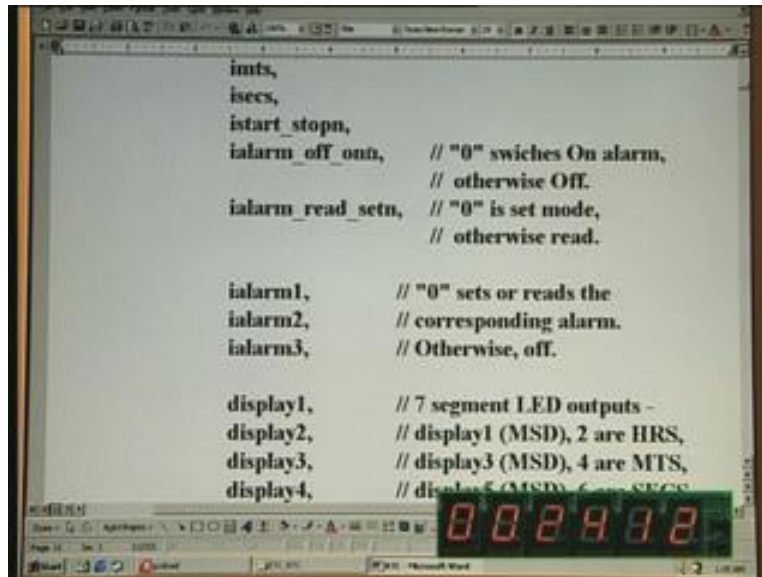
10 times, you get 1 second. That is how you get a time base of 1 second. Similarly, for debouncing the switch, we need 3 milliseconds and we are going to derive this from DMS, which we have already seen – deci-millisecond. That will serve as the time base. If you count 30 for that, 30 into 0.1 millisecond is 3 milliseconds. You would have made 3 millisecond debounce time and the actual module starts here.

(Refer Slide Time: 27:34)



We declare the module name as `rtc_alarm`. This is the actual design module. It lists all the I/Os that we have used, which you have already seen in the architecture: clock, reset, run and so on.

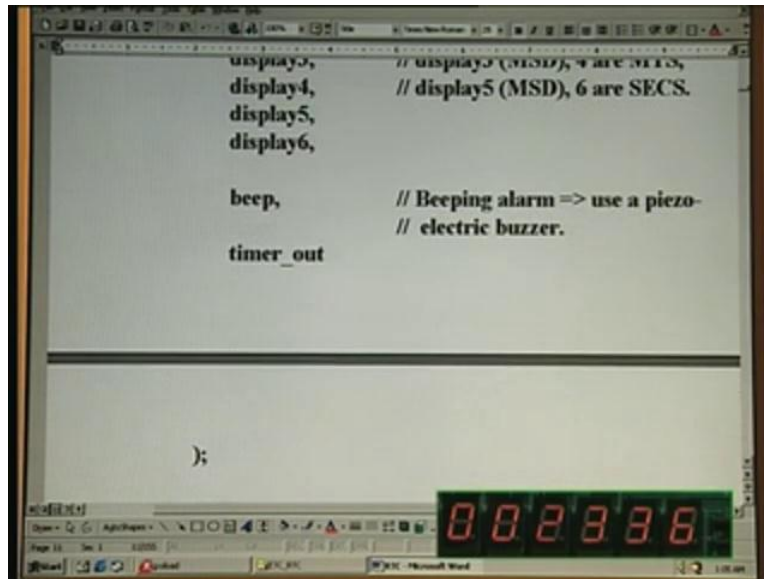
(Refer Slide Time: 27:47)



I am not going through these once again. You can just have a look. I have also put a comment here to denote what it is. For example, if you take alarm\_off\_on, one nomenclature I have given as I had done before for all other design examples earlier, here n stands for negative, rather it is low. That means this is a single-bit signal and if it is 0, it means on and if it is 1, it means off. That is what these comments say. It is exactly the same for all of them.

So is the case for alarm. Then display1 through display6 are there and they are seven-segment LED displays. display1 and display2 are for hours, display3 and display4 for minutes and display5 and display6 for seconds; 1, 3 and 5 are MSD, while all other displays are LSD.

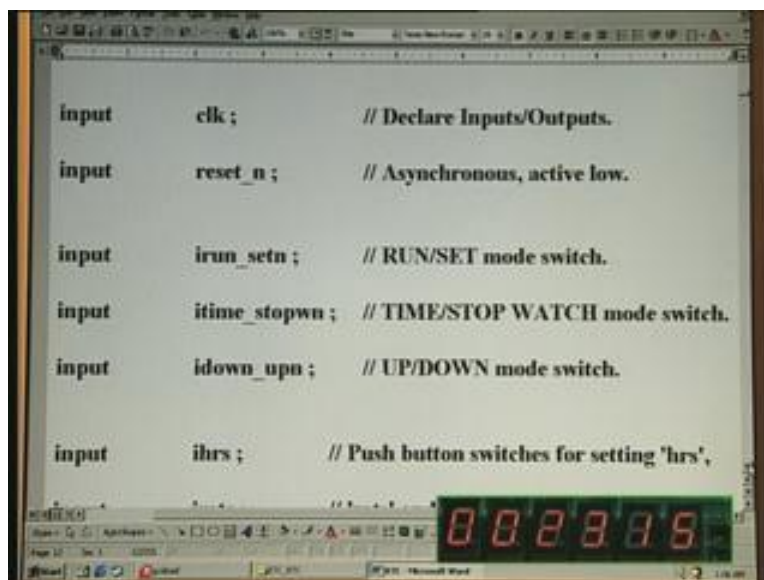
(Refer Slide Time: 28:36)



```
display3, // display3 (MSD), 4 are SECS,  
display4, // display5 (MSD), 6 are SECS.  
display5,  
display6,  
  
beep, // Beeping alarm => use a piezo-  
// electric buzzer.  
  
timer_out  
  
);
```

You need a beeping alarm and the highly recommended is the piezoelectric buzzer. It is a miniature thing, which you see on the hardware there. You also need a timer out and this completes the module declaration. We are just listing here all that we have put inside as I/Os.

(Refer Slide Time: 28:57)



```
input clk; // Declare Inputs/Outputs.  
  
input reset_n; // Asynchronous, active low.  
  
input irun_setn; // RUN/SET mode switch.  
  
input itime_stopwn; // TIME/STOP WATCH mode switch.  
  
input idown_upn; // UP/DOWN mode switch.  
  
input ihrs; // Push button switches for setting 'hrs',
```

I will quickly go through these and it is for you to grasp it totally. These have been already described to you and so I do not have to go through it. Hours, minutes, seconds, start/stop push button.

(Refer Slide Time: 29:08)

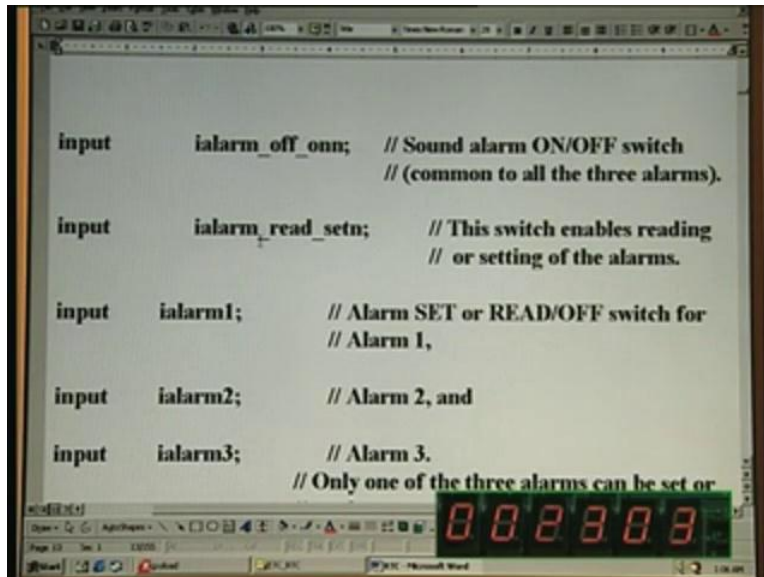
```
input    ialarm_off_onn; // Sound alarm ON/OFF switch
                        // (common to all the three alarms).

input    ialarm_read_setn; // This switch enables reading
                        // or setting of the alarms.

input    ialarm1; // Alarm SET or READ/OFF switch for
                // Alarm 1,

input    ialarm2; // Alarm 2, and

input    ialarm3; // Alarm 3.
                // Only one of the three alarms can be set or
```



This is alarm\_off\_on and then the three alarms.

(Refer Slide Time: 29:16)

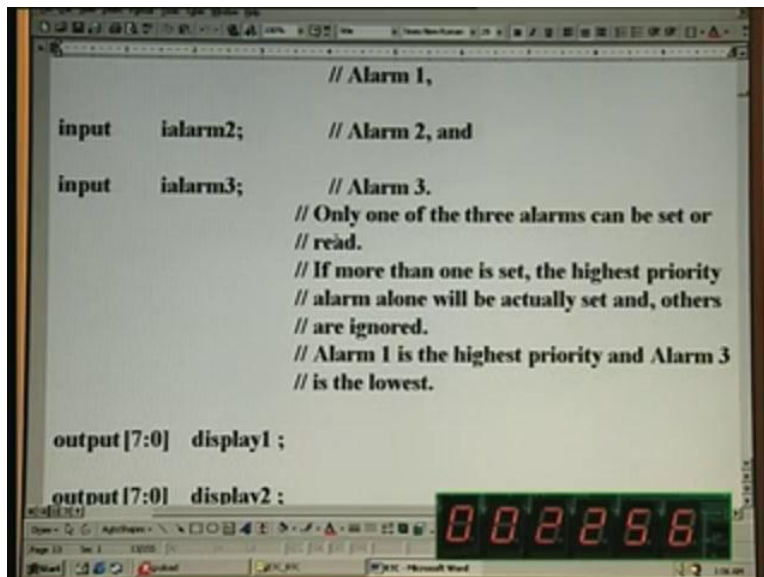
```
// Alarm 1,

input    ialarm2; // Alarm 2, and

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

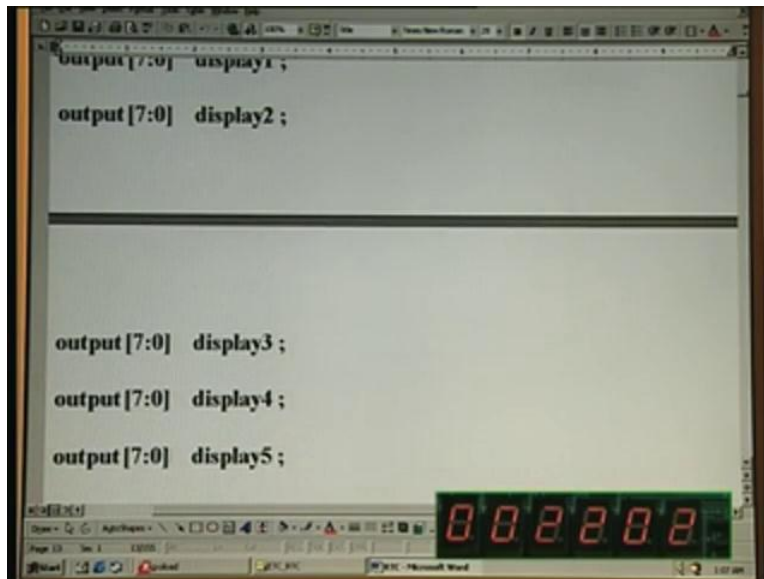
output [7:0] display1 ;

output [7:0] displav2 ;
```



Only one of the three alarms can be set or read. If more than one is set, the highest priority alarm alone will be actually set and others are ignored. Alarm 1 is the highest priority whereas alarm 3 is the lowest. In specification we have seen exactly the same thing, but there is no harm in putting the same thing in the text also. When you write the code, you are not merely writing for your own sake. You do not want to just pass your examination and go away. It is to serve for all others who are going to use this subsequently. Some other designer may come and would like to change your code for some other application. Do you not think you will be doing a great service if you put the right comment? That is why I have been emphasizing that you put the proper comments all through – if possible line-by-line comment. I think I have adequately put comments but if it is lacking, let me know, so that I can improvise on the existing ones.

(Refer Slide Time: 30:11)





```
output    beep;


output    timer_out;

wire      [7:0] display1 ; // Declare outputs as nets
           // (combinational circuit outputs).

wire      [7:0] display2 ;

wire      [7:0] display3 ;

wire      [7:0] display4 ;
```




```
wire      [7:0] display3 ;

wire      [7:0] display4 ;

-----

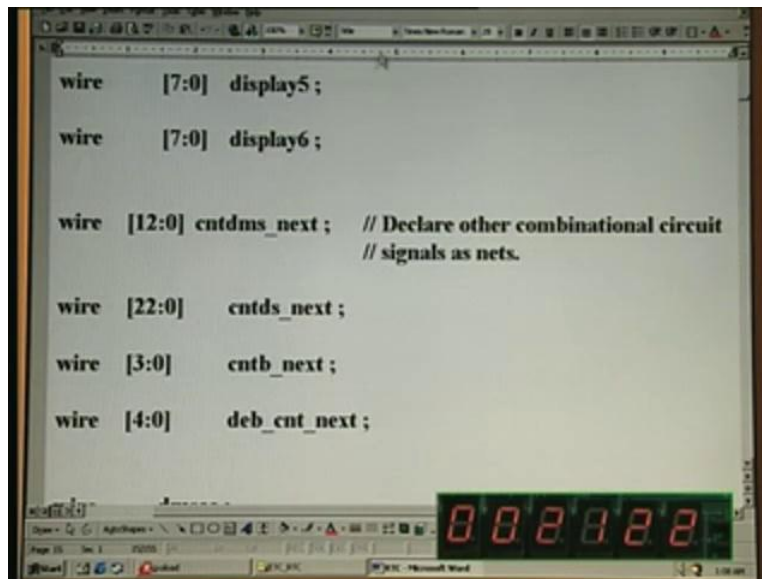
wire      [7:0] display5 ;

wire      [7:0] display6 ;
```



These are all the output declarations. There are eight bits in total. You see display1 through display6. We have a beep – single bit. Therefore, there is nothing here, whereas eight bits are described here in this fashion. Beep and timer out are the two single-bit outputs. Wherever you have used signals in assign statements, it will be a wire in the design, which we have already seen earlier in several examples. The width is declared here. display1 through display6 is declared as wire, because finally you are going to use an assign statement for the same.

(Refer Slide Time: 30:52)

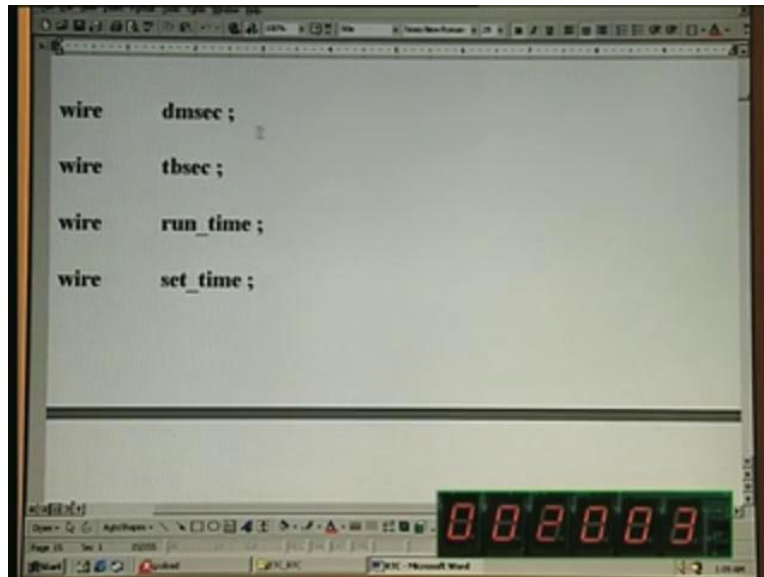


```
wire [7:0] display5;  
wire [7:0] display6;  
  
wire [12:0] cntdms_next; // Declare other combinational circuit  
// signals as nets.  
  
wire [22:0] cntds_next;  
wire [3:0] cntb_next;  
wire [4:0] deb_cnt_next;
```

The screenshot shows a code editor window with the above Verilog code. Below the code, there is a digital display showing the number '000000' in red LEDs. The display is part of a larger circuit board or simulation environment.

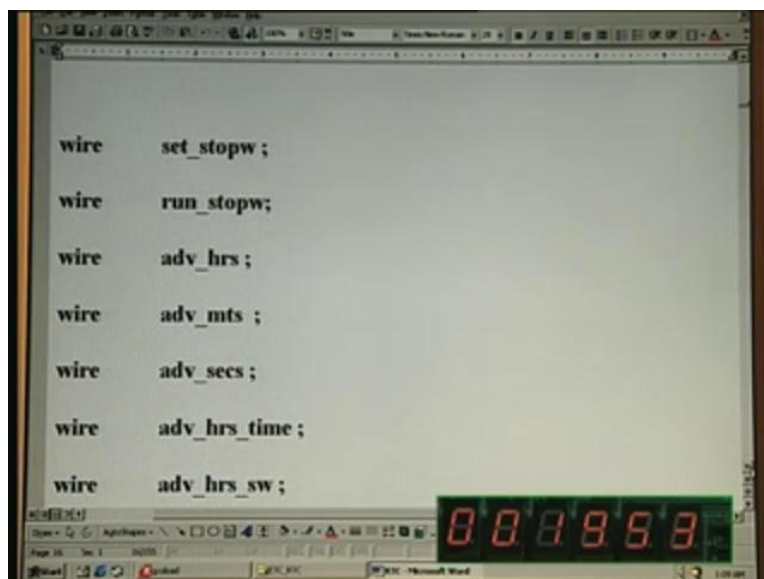
Similarly, there are so many other wires. For example, I want a counter for deci-millisecond, I want to keep track of 0.1 second. I can do this by running a counter that has 13 bits. You can arrive at the number of bits by simply using the calculator that you have on your desktop on the computer. That is how I have arrived at this. You should know the maximum that you need to [31:16]. Accordingly, you have to set the number of bits. So is the case for decisecond. Decisecond is a huge figure. You need 23 bits because we need to track 20 million totally. If you want a counter base that is in seconds, you need just four bits because we need to keep a track of 0 through 9 only. All other timing is already kept by this. This is going to refer to this. This cnt\_dms is used to keep track of 3 millisecond debounce time, for which you need just five bits here. Since it is going from 0 to 31, you need five bits. That is how you arrive at the accuracy that you need.

(Refer Slide Time: 32:09)



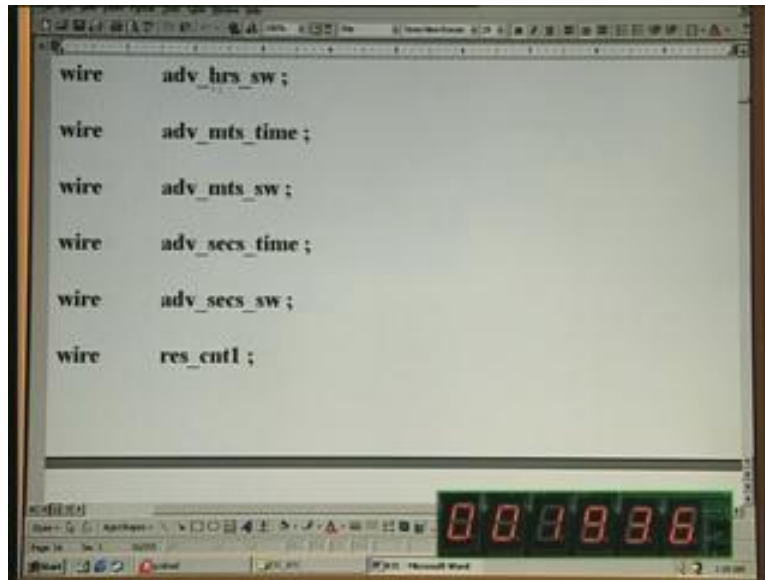
Then, you declare other wires, for example, deci-millisecond, time base in second, run\_time and set\_time. I will quickly go through all this because there are many of them.

(Refer Slide Time: 32:20)



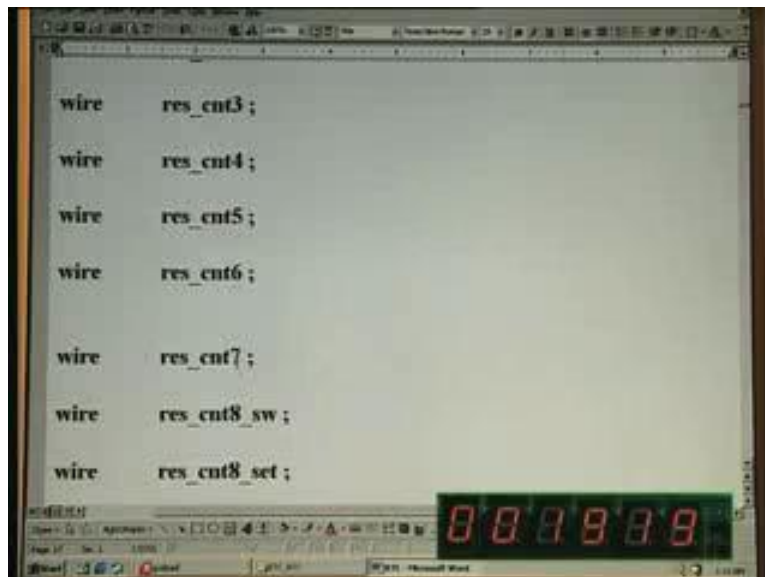
Set\_stop. I will just read it out and when we come to a particular signal, I will describe what it is. Run\_stop, then adv\_hours, adv\_mts, adv\_seconds, adv\_hrs\_time, adv\_hrs\_sw.

(Refer Slide Time: 32:30)



Then adv\_mts\_time, adv\_mts\_sw, then seconds similarly, then time and stopwatch. You need to reset the counter for certain conditions.

(Refer Slide Time: 32:47)

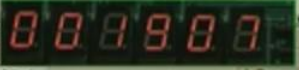


```
wire    res_cnt6 ;

wire    res_cnt7 ;

wire    res_cnt8_sw ;
wire    res_cnt8_set ;

wire    res_cnt9 ;
wire    res_cnt10 ;
wire    res_cnt11 ;
```




Then you need this wire for reset counter1 through counter6 here. 1 through 6 are for the running timer. Counter 7 to 12 is for the stopwatch up down counter. They also need resetting when the occasion demands. You also have to discriminate whether it is in stopwatch mode or in set mode. These are all the intermediate signals that will be needed in order to fulfill your total design. So is the case for 9, 10, 11 and 12.

(Refer Slide Time: 33:27)

```
wire    res_cnt12 ;

wire    pres_cnt8 ;    // Signal to indicate preset condition for
                      // the counter.

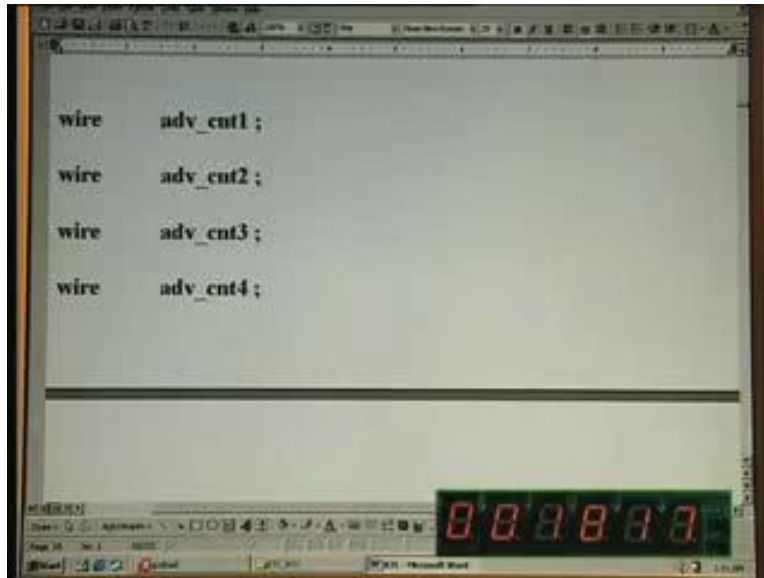
wire    pres_cnt9 ;
wire    pres_cnt10 ;
wire    pres_cnt11 ;
wire    pres_cnt12 ;
```



You also need presetting. For example, after you go to the flag end, let us say 9 is the last and after that, it should become 0. If it is in down count mode, you may have to go from

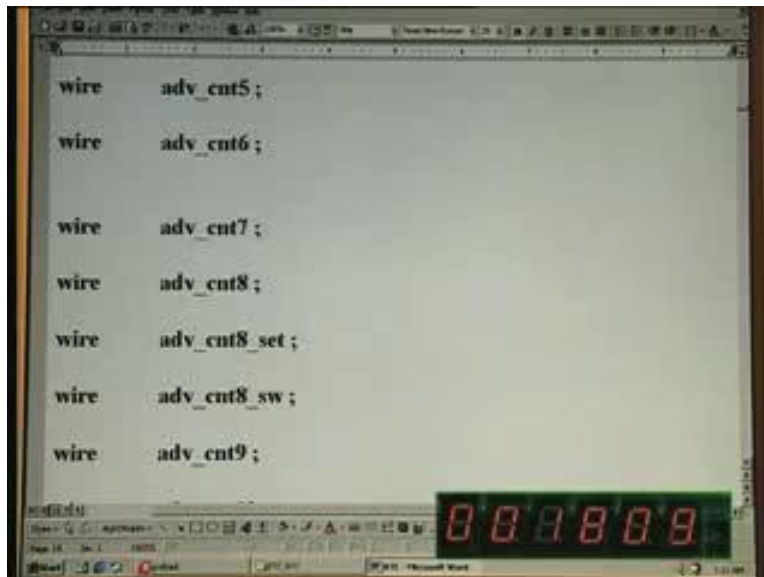
9, 8, right up to 0, and after 0, you have to wrap to 9. This is called preset. Normally, this is used for the down counter mode and 8 through 12 is for presetting the same.

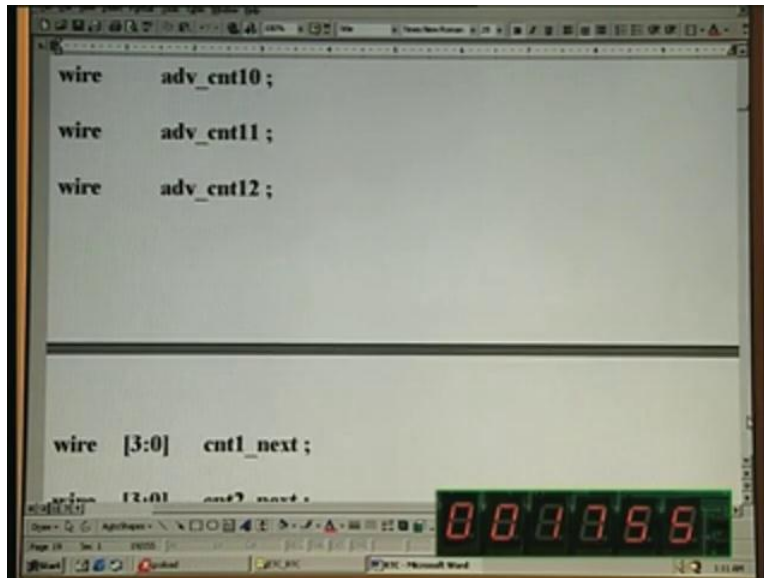
(Refer Slide Time: 33:54)



Then you have counter 1 through 6 for advancing. When should you advance the count by 1? That is what it means here.

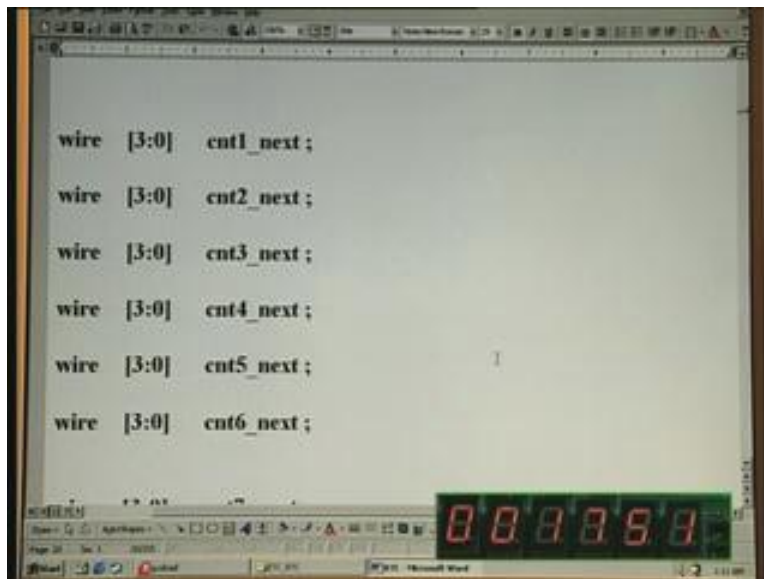
(Refer Slide Time: 34:02)

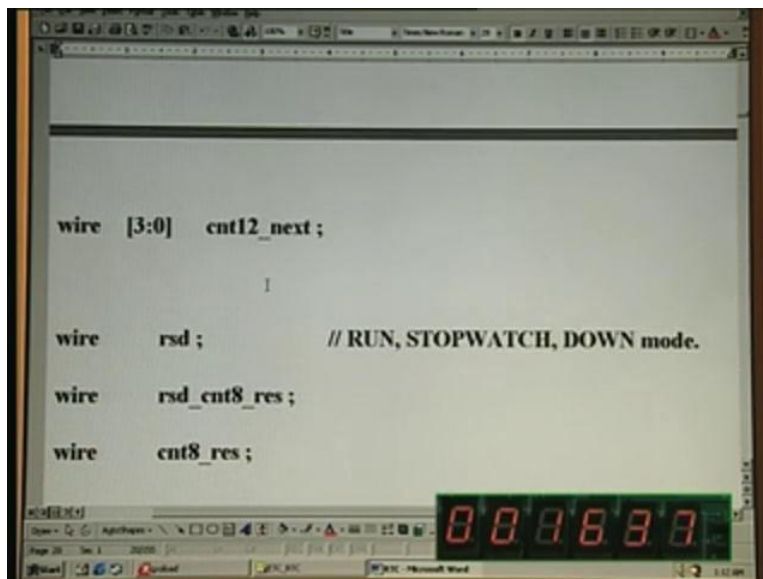
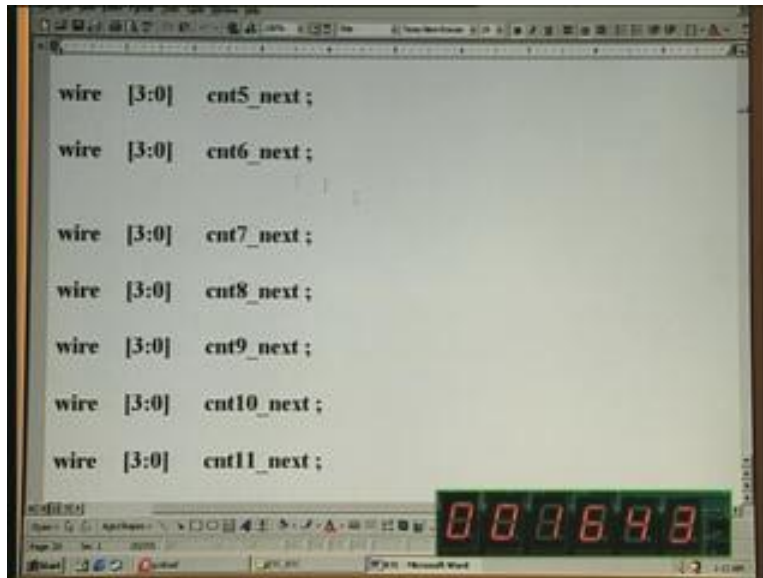




Once again, for counter 7 through 9, you have a similar thing as you have seen for preset and reset. This time, it is going to be advanced by 1. Once again, it depends upon whether it is in set mode or stopwatch mode. Right up to counter 12, you need advance.

(Refer Slide Time: 34:22)





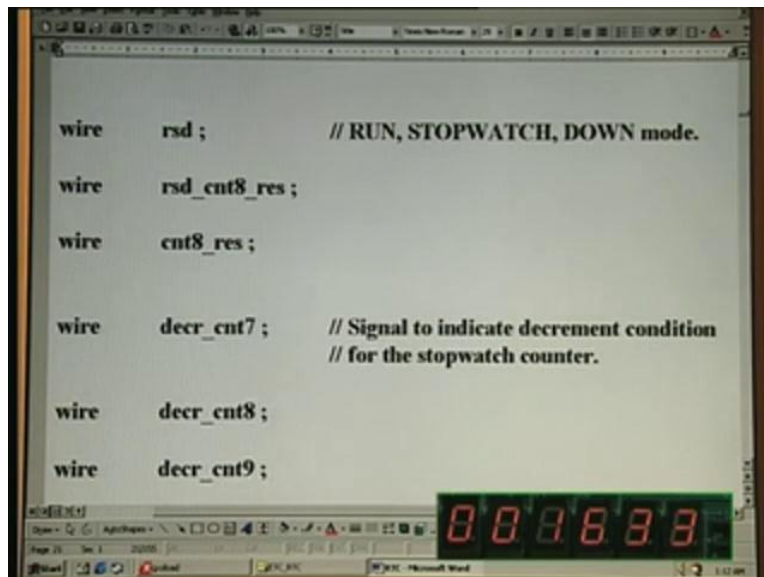
We have always been using statements for realizing registers. A counter can be realized by that. We have seen that also in many number of applications. We are going to continue the same trend even in this design. Once again, this is a very good example for counter-based design. It is full of counters and nothing more. Simply by using assign statements and always statements, the entire Verilog can be contained. You can be fully RTL-compliant just by using this. The learning time for Verilog is dramatically reduced if you just stick to this. You will also conform to RTL coding guidelines, which will certainly



work in the hardware. Otherwise, you are not guaranteed of its working on the hardware – I emphasize that.

You need to keep track of the counters and you need to do the pre-incrementing. We do that by having a separate counter running. It is also four bits in width. They are still wires using assign statements – counter 1 through 6 here. This will be used for pre-incrementing the actual counter6\_reg, which is the real counter. Counters 7 through 12 are serving a similar purpose for the stopwatch.

(Refer Slide Time: 35:35)



There are plenty of wires. We have to wade through these signals unfortunately. We are probably halfway through declaring. We have another signal for run, stopwatch, down mode. When it is in such a mode, we activate another signal so that once you give different meaningful signals, you can use them when the occasion demands – you can straightaway borrow this signal there and do the logic, rather than write a long code each time. That is the reason we are using so many intermediate signals. In this mode, if you want to reset the counter8, we need one more signal. Similarly, you want to reset the counter, I think this is while the stopwatch is running. Is it correct?

(Refer Slide Time: 36:43)

```
wire cnt8_res ;

wire decr_cnt7 ; // Signal to indicate decrement condition
// for the stopwatch counter.


wire decr_cnt8 ;

wire decr_cnt9 ;

wire decr_cnt10 ;

wire decr_cnt11 ;

wire decr_cnt12 ;
```



If you want to decrement the counters 7 through 12, you need these signals here.

(Refer Slide Time: 36:47)

```
wire [3:0] cnt7_nextd ; // 'd' stands for decrement.


wire [3:0] cnt8_nextd ;

wire [3:0] cnt9_nextd ;

wire [3:0] cnt10_nextd ;

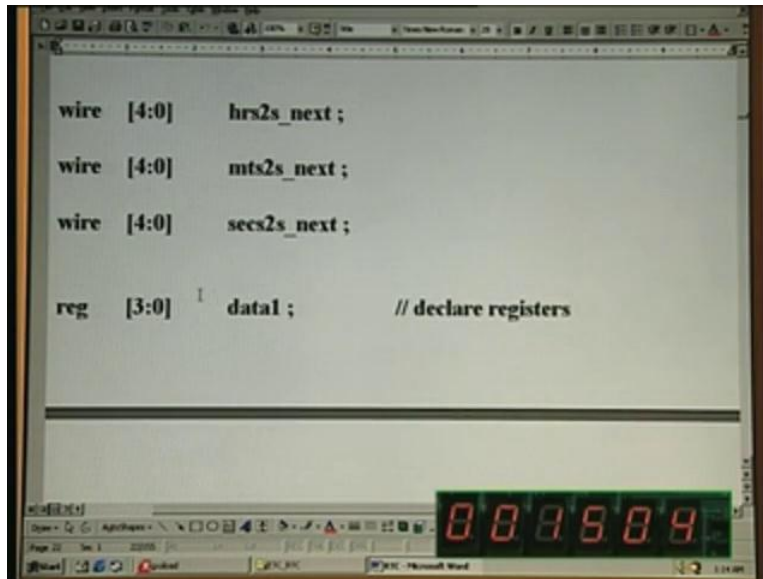
wire [3:0] cnt11_nextd ;

wire [3:0] cnt12_nextd ;
```



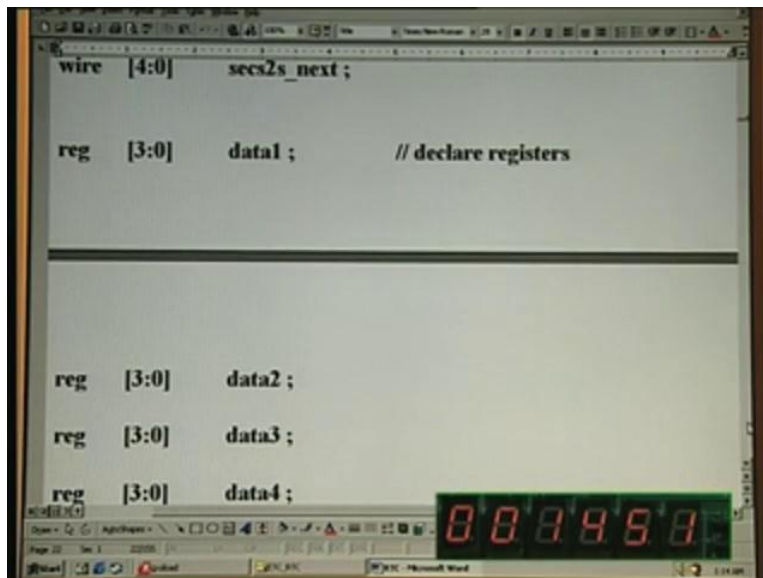
If you want to decrement 7 through 12, it is just like we used increment, we are going to use pre-decrement counters before we assign to the actual register, which is cnt7\_reg. d stands for decrement here. Once again, the bits are same.

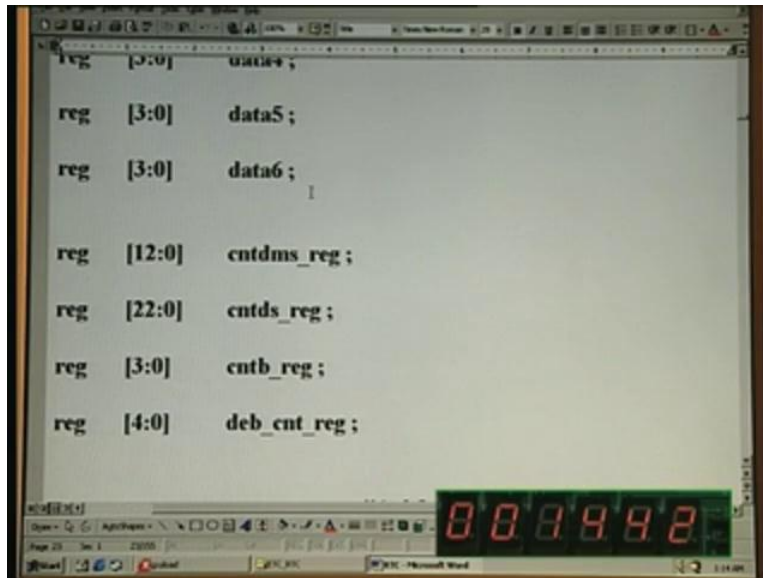
(Refer Slide Time: 37:08)



Now comes the declaration of registers. Registers, as you know, are the signals that we use in the always block, whether it is a combinational logic or sequential logic.

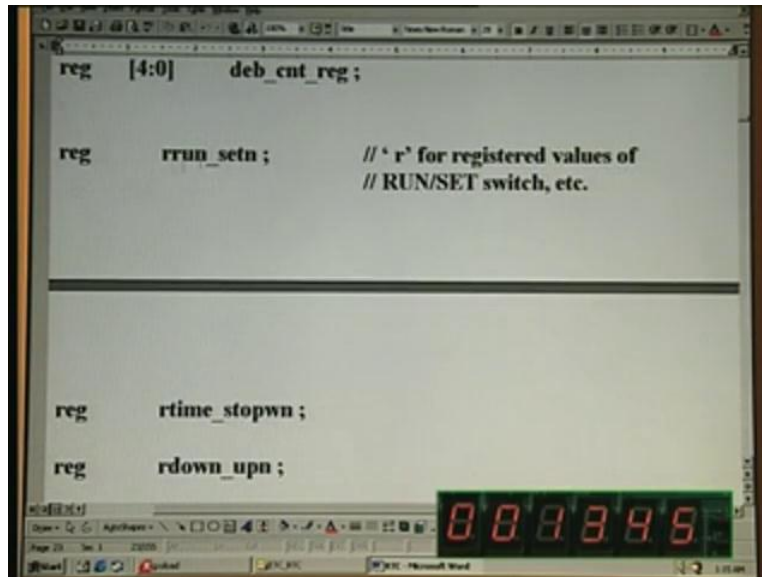
(Refer Slide Time: 37:21)





Whatever is appearing as signals within the always block will have to be declared as registers and we need data1 through data6 here, which is a signal to mean either counter1 through counter6 or counter7 through counter12. We have seen right in the simplified architecture routing one of the two. This is precisely that signal which has that single output directly fed to the display. We also need a 13-bit counter deci-millisecond as a register. Similar names might have appeared but they are all slightly different signals. It might be a next signal and I hope there are no duplicates. If there were any duplicates, it would have shown up as some error or warning. I did not see anything. If you happen to see any of them, you can correct it. Then for decisecond, you need 23 bits. This is the actual register that is going to run. For time base one second, this is the register and then this is for debounce counter.

(Refer Slide Time: 38:28)



```
reg [4:0] deb_cnt_reg ;

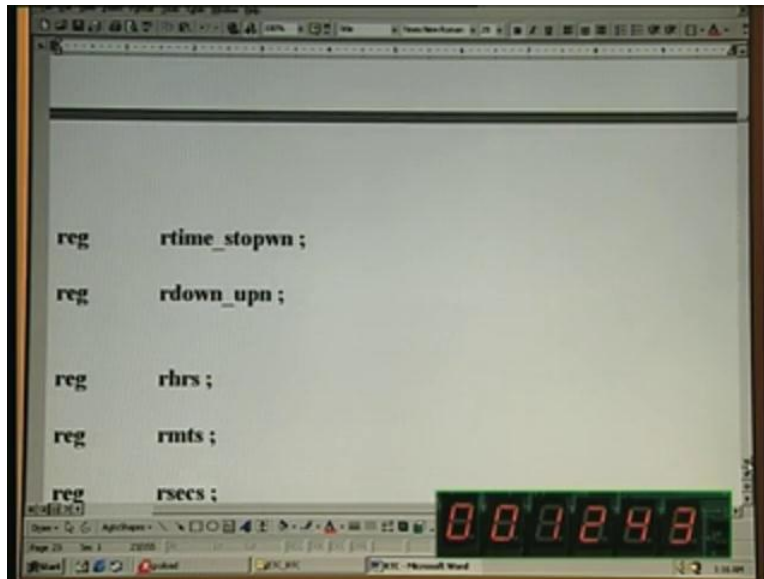
reg rrun_setn ; // 'r' for registered values of
                // RUN/SET switch, etc.

reg rtime_stopwn ;
reg rdown_upn ;
```

The slide also features a 7-segment display at the bottom right showing the number 001345.

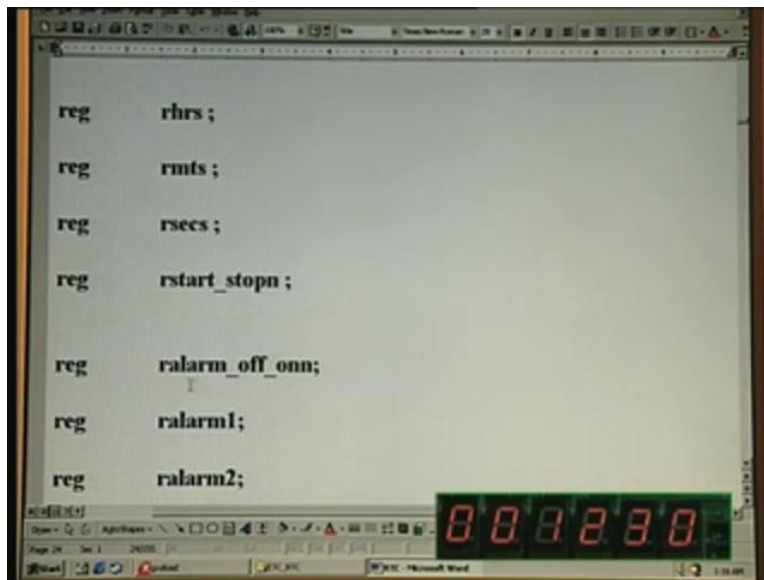
We have already used `irun_set`. `run_set` is the actual switch and `irun_set` is the actual physical signal that goes into the chip input. Within that, we need a register so that we can know the current value as well as compare it with the previous value, so that we can know when the transition has been made. For example, if you push a button; how do you recognize the button being pushed? Only if you keep track of and continuously monitor its status. At any point of time if you read 0 for that push button switch and later on if you read 1 for the same thing, you recognize that the push button has been pressed. That is how you recognize. For that, you need to store in a register. This is nothing but the `irun_set` but with a delay. This has been registered when the clock strikes. That is the nomenclature adapted all through and `r` stands for the register. It is nothing other than the actual input signal that is registered inside so that we may keep track of any transition that you make. Otherwise, you cannot recognize the push button being pressed.

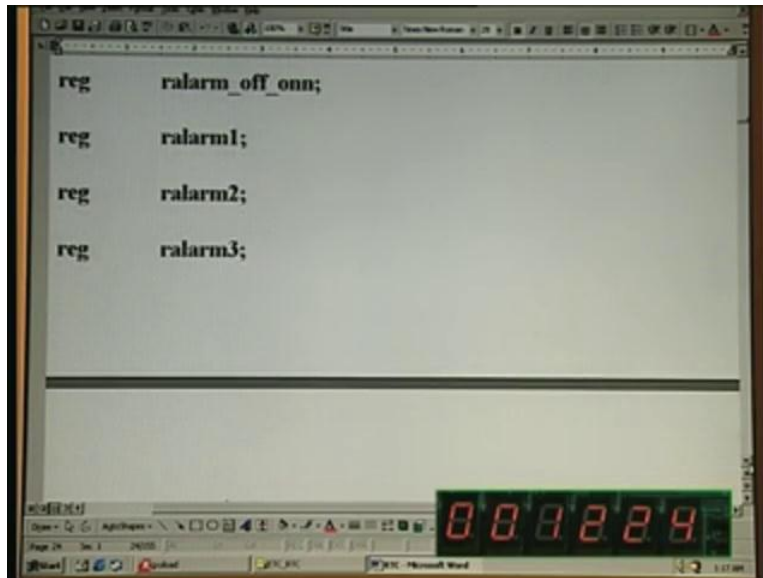
(Refer Slide Time: 39:29)



For time\_stopwatch, down\_up, these are the registers. What you seen down here we have already seen as inputs.

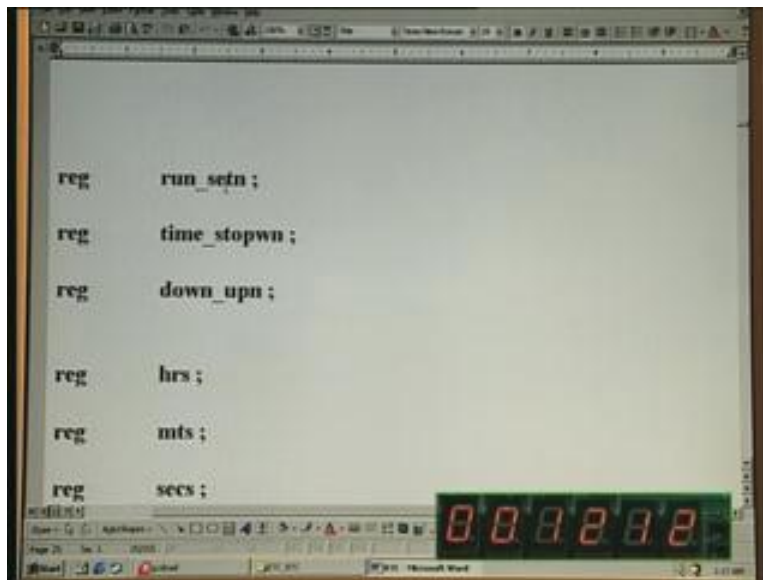
(Refer Slide Time: 39:40)





Hours, minutes, seconds, start\_stop, etc., are all registered. So is the case for alarm\_off\_on and the three alarms are also registered internally here.

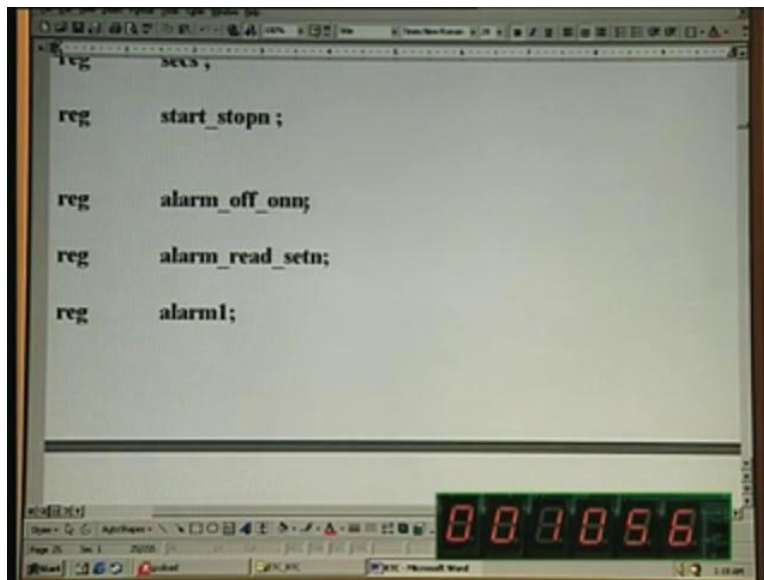
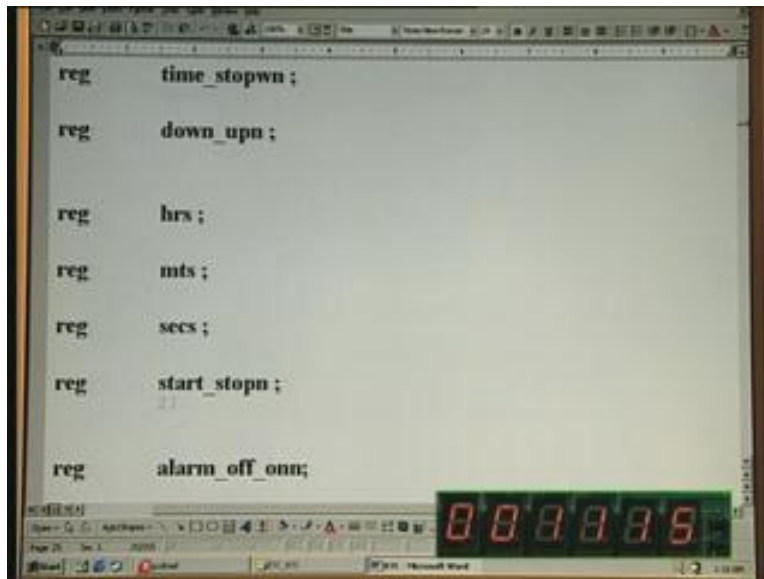
(Refer Slide Time: 39:50)



In addition to this, in the layout that we have already seen, we used only run\_set and we are going to precisely use the very same variable name. Rather, we should not use 'variable' because it is C domain. Coming back to hardware domain, we speak in terms of signals. **Habits die very hard. It is reflected in my own sentence, so just beware of that.** Refer to it as a signal. Here, what is rrun? It is actually run\_set after doing the

debouncing. It has to be debounced and you have to write appropriate code for debouncing, which we will see later on. We see that all this is precisely the very same input or reg with the same name, but now it implies that the debounced switch condition. It is actually the same as what we have started with earlier in the layout for the common man. We are going to use the very same symbol hereafter – this will be used extensively hereafter.

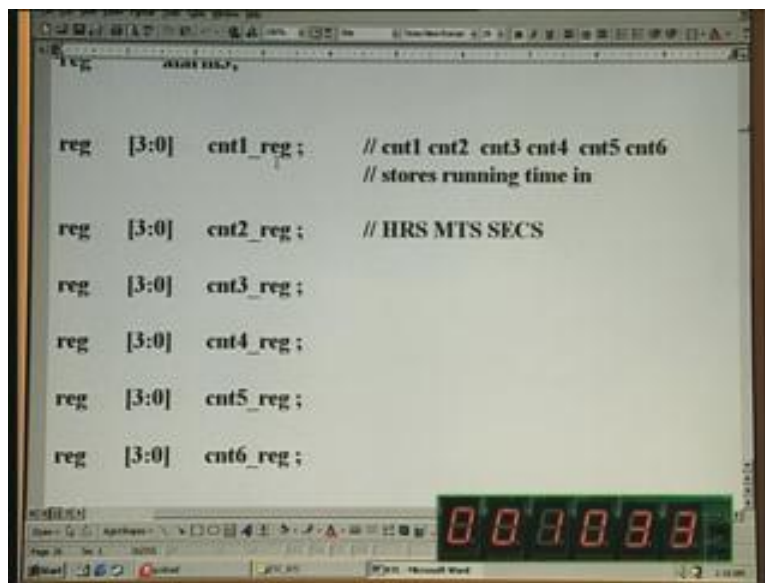
(Refer Slide Time: 40:52)





So is the case for the hours, minutes and seconds push button switches. This is also a push button switch that says start\_stop. If you push the button once, it will be in start mode and if you push the same button again, it will be in stop mode. One push button switch plays a dual role. You have an alarm\_off\_on and once again off and notice that on is low. This is a single bit. All of them are single bits. So is the case for alarm\_read\_set. You can either read or set the alarm. There are three different alarms. This is precisely the same thing.

(Refer Slide Time: 41:38)



Now we also have four-bit width counters. This is the actual counter that is really running for your time. Notice that we have used here reg. Earlier, we have used next. That was declared as wire whereas here it is declared as reg because we are going to use this in the always block. There are 4 bits here and counter 1 through 6 are used for hours, minutes, seconds.


(Refer Slide Time: 41:55)

```
reg [3:0] cnt6_reg ;

reg [3:0] cnt7_reg ; // cnt7 cnt8 cnt9 cnt10 cnt11 cnt12
                    // stores the time count (counter) in

reg [3:0] cnt8_reg ; // HRS MTS SECS

reg [3:0] cnt9_reg ;
```



If it is in stopwatch mode, exactly the same hours, minutes, seconds counter 7 through 12. They are also registers.

(Refer Slide Time: 42:04)

```
reg [3:0] cnt11_reg ;


reg [3:0] cnt12_reg ;

reg start_stopn_reg ; // START/STOP mode register ==>
                    // start_stopn_reg == 1 means
                    // START, otherwise STOP.

reg start_stopnp_reg ; // Previous value of start/stop.

reg hrsp_reg ; // Previous value of HRS PB.

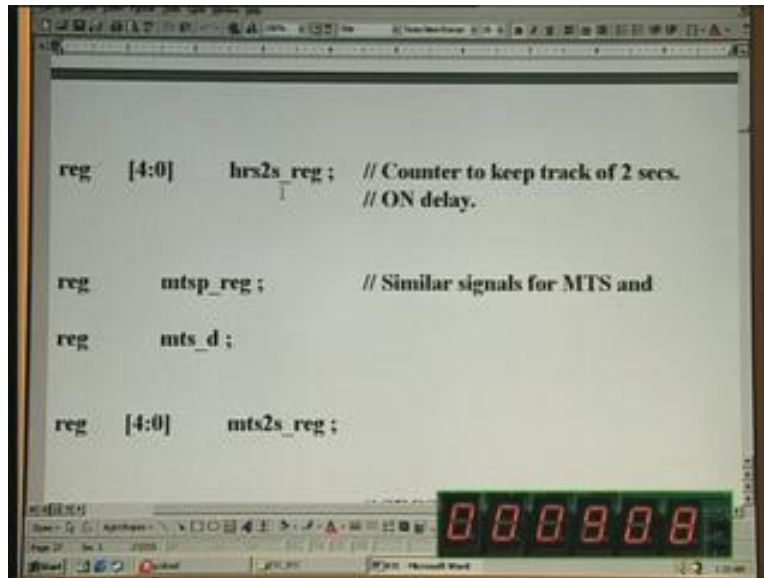
reg hrs_d ; // ON delay output of HRS PB.
```



We have start\_stopn\_reg and start\_stopnp\_reg. We have some more hours here. You are already familiar with start and stop. When you press, you should know what was the previous value and that previous value will have to be a different register, so we use the

exactly same name with p here – this implies it is previous and only then, by comparison of the two, you can know when it has been pressed. Similarly, for hours previous value and hrs\_d is what we are going to explain later on and this is in the setting mode – on delay output hours. We have what is called an on delay timer and that output is this, which will be needed at the time of setting the hours display.

(Refer Slide Time: 42:52)



Similarly, you need a delay of 2 seconds so that when you want to advance, if you press the push button once, it will advance by one. If you keep it pressed for a minimum of 2 seconds, it will automatically run 10 times faster. If you want to display for a long time, you can set it fast. It runs at 10 times the normal speed. Otherwise, it will run at 1 second and every time you push, it will run. When you near the targeted value, you can stop before releasing the button, and then advance it by pushing it once so that you advance step by step – this is what is available in the market. In a Philips two-in-one, you would have seen this – a real-time display is also available. They have used the same strategy there in order to set. So is the case for minutes and seconds.


(Refer Slide Time: 43:48)

```
reg    sefsp_reg ;    // SECS.

reg    secs_d ;

reg    [4:0]    secs2s_reg ;

wire    adv_res_cnt2 ;    // adv, res mean advance and reset
                                // counters respectively.
```



Again, 2 seconds and all are exactly the same. You also need `adv_res_cnt2` and this we have not covered before. So we need to declare it as a wire because we are going to use it in the assign statement.

(Refer Slide Time: 44:01)


```
wire    adv_res_cnt2 ;    // adv, res mean advance and reset
                                // counters respectively.
```

---

```
wire    res_cnt2_time ;

wire    res_cnt2_set ;


reg    [3:0]    temp_alarm_reg1 ;    // Individual alarms are set
                                        // via temporary registers.
```



Then, `res_cnt2_time` and `res_cnt2_set`, then reg temporary. We need temporary alarm registers.


(Refer Slide Time: 44:12)

```
reg [3:0] temp_alarm_reg1; // Individual alarms are set
// via temporary registers.
reg [3:0] temp_alarm_reg2;
reg [3:0] temp_alarm_reg3;
reg [3:0] temp_alarm_reg4;
reg [3:0] temp_alarm_reg5;
reg [3:0] temp_alarm_reg6;
```



```
wire [3:0] temp_alarm_reg1_next;
wire [3:0] temp_alarm_reg2_next;

wire [3:0] temp_alarm_reg3_next;
wire [3:0] temp_alarm_reg4_next;
wire [3:0] temp_alarm_reg5_next;
```



For six different alarm displays, we need temporary registers. It is exactly the same as counter1, counter 2 and so on. For each, you need a temporary register so that we can do the alarm setting. For each, there are three different alarms that will be set using alarm reg1, reg2 and so on. This is the pre-increment as you have seen next earlier also.

(Refer Slide Time: 44:54)

```
wire [3:0] temp_alarm_reg6_next ;

wire adv_temp_alarm_reg1 ;


wire adv_temp_alarm_reg2 ;

wire adv_temp_alarm_reg3 ;

wire adv_temp_alarm_reg4 ;

wire adv_temp_alarm_reg5 ;

wire adv_temp_alarm_reg6 ;
```



We have 1 through 6 here for temporary alarm setting and we need adv\_temp\_alarm\_reg1. This is advancing the very same particular register. If you want to advance, you need another signal and you need six such registers here.

(Refer Slide Time: 45:09)

```
wire adv_temp_alarm_reg6 ;

wire res_temp_alarm_reg1 ;




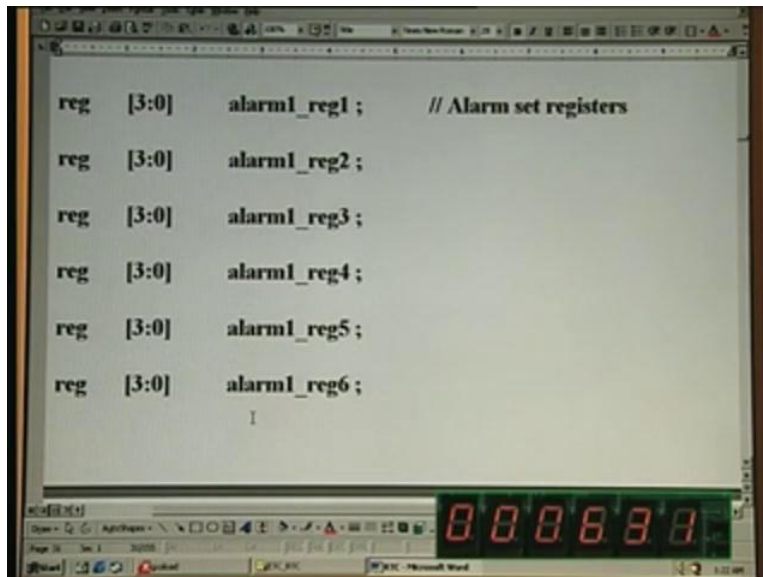
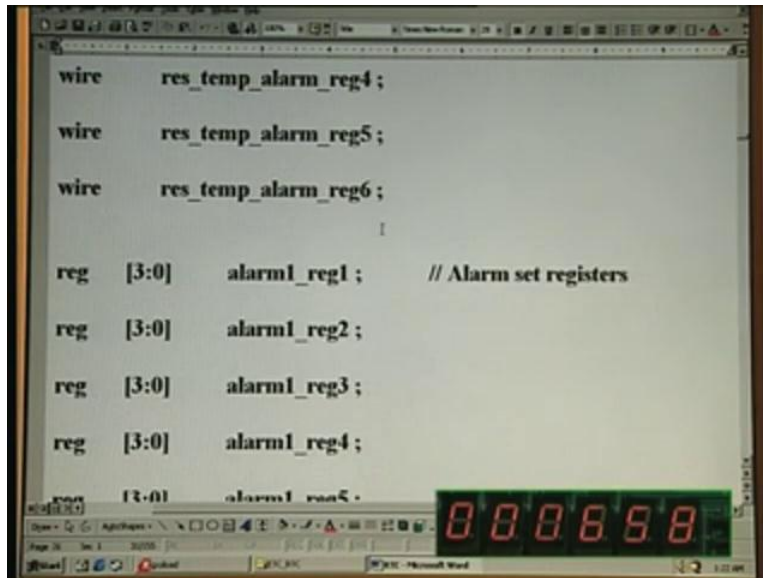
---



wire res_temp_alarm_reg2 ;

wire res_temp_alarm_reg3 ;
```





We have not seen reset. For the same registers, we need reset here and that is up to this point. Now come the independent alarms, for example, alarm1\_reg1 through 6. This corresponds to the counter1 through counter6 or counter7 through counter12. Which is the one? Second one? counter7 through 12, right? This corresponds to that. It is not the same – it is different from counter 7 but it has a similar meaning. This is exclusively for setting the alarm. You have three different alarms that can be set.


As I mentioned earlier for epilepsy patients, we would need to set three times, say at morning 7 o'clock, then in the afternoon at 1 o'clock and then again at dinnertime, say 8

o'clock. If they miss taking the tablets – if they do not take within half an hour, they will get fits. In order to overcome that problem, you can have three different settings and so, we have alarm1 through alarm3 here.

(Refer Slide Time: 46:17)

```
reg [3:0] alarm3_reg5 ;
reg [3:0] alarm3_reg6 ;


wire adv_hrs_temp_alarm ;
wire adv_mts_temp_alarm ;
wire adv_secs_temp_alarm ;
wire set_alarm ;
```

A screenshot of a digital display showing the number 000548 in red LEDs on a green background. The display is part of a larger interface, likely a simulation or a physical device's output.

```
reg [3:0] alarm3_reg6 ;

wire adv_hrs_temp_alarm ;
wire adv_mts_temp_alarm ;
wire adv_secs_temp_alarm ;
wire set_alarm ;

reg ralarm_read_setn ;
```

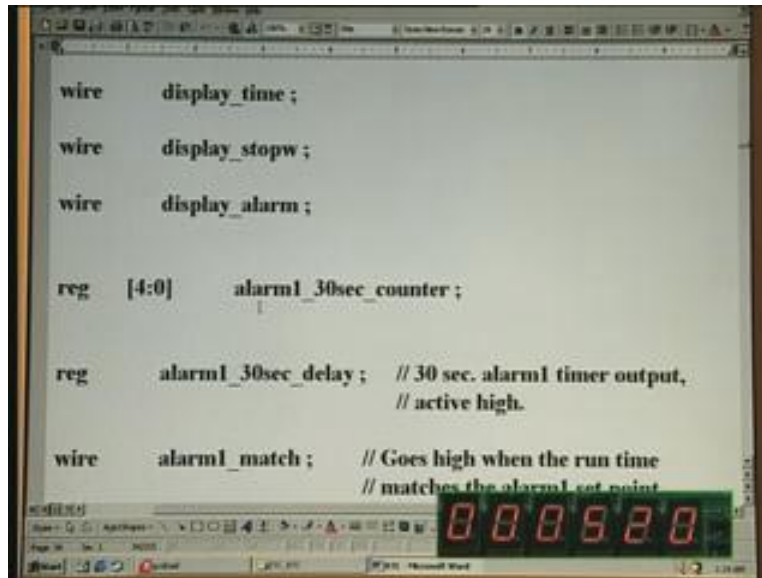
A screenshot of a digital display showing the number 000535 in red LEDs on a green background. The display is part of a larger interface, likely a simulation or a physical device's output.

You also need adv\_hrs\_temp\_alarm – you have to advance them so. For hours, minutes and seconds, you have to advance. Two digits will run independently. You have to set hours separately, apart from minutes and seconds. You also have set\_alarm and that is



declared as wire and 1 alarm is there, r stands for register here. You also need alarm\_read\_set.

(Refer Slide Time: 46:47)



```
wire    display_time ;  
  
wire    display_stopw ;  
  
wire    display_alarm ;  
  
  
reg     [4:0]    alarm1_30sec_counter ;  
  
reg     alarm1_30sec_delay ; // 30 sec. alarm1 timer output,  
                             // active high.  
  
wire    alarm1_match ; // Goes high when the run time  
                       // matches the alarm1 set point
```

You have display\_time, display\_stopw and display\_alarm. When we go into the details, we will see more about this. We also need to sound the buzzer and for that, we need some 30 seconds counter, at the end of which the buzzer will automatically stop. You do not have to do any resetting, etc., – everything is automatic. If you want to introduce more automation, you can – you are free to amend the codes after you learn this. You also need another register for delay. Another signal is required to match for independent alarms. This goes high when the run time matches the alarm1 set point. We will later see how to set, etc.


(Refer Slide Time: 47:29)

```
wire    alarm1_match;    // Goes high when the run time
                        // matches the alarm1 set point.

wire [4:0]    alarm1_30sec_counter_next;

-----

wire    adv_alarm1_30sec_counter;
```



We also need alarm1\_30sec\_counter\_next. This is for pre-incrementing the counter and this is if you want to advance alarm1\_30sec\_counter – this is the real counter. This is the pre-increment for this.

(Refer Slide Time: 47:44)


```
reg [4:0]    alarm2_30sec_counter;

reg    alarm2_30sec_delay;

wire    alarm2_match;

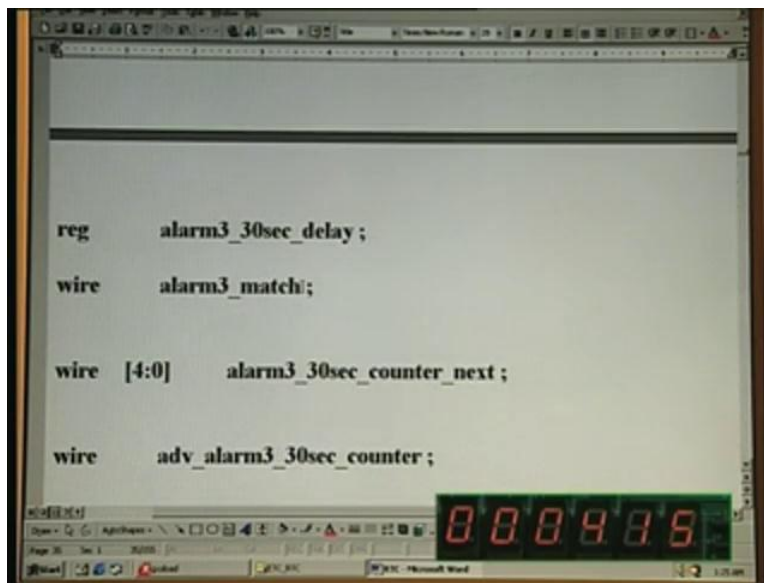
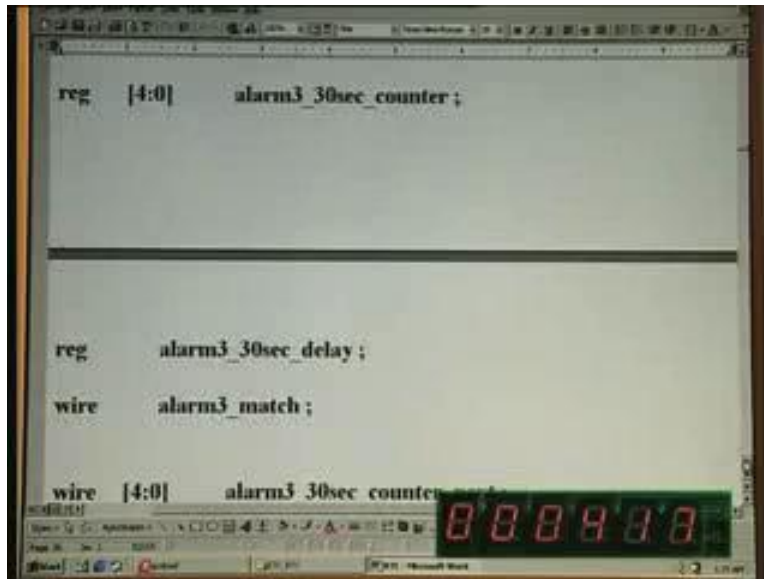
wire [4:0]    alarm2_30sec_counter_next;

wire    adv_alarm2_30sec_counter;
```



You have similarly alarm2 and exactly the same signals for alarm2.

(Refer slide: 47:54)




So is the case for alarm3 here, right up to the match, counter and so on.

(Refer Slide Time: 48:00)

```
reg    beep ;           // Generates square pulse for
                        // beeping of a buzzer

wire   ring ;          // if this signal is active.


reg    [2:0]    beep_counter ; // Counter for generating square
                        // pulse and its
wire   [2:0]    beep_counter_next ; // advanced counter.
```



You also need registers for beep and ring. They are all single bit and this is a buzzer we are going to connect. In order to do the beeping, we need to have another counter. Three bits are adequate for that counter. You need to pre-increment, so you need this as well.

(Refer Slide Time: 48:19)


```
reg    [3:0]    term_count_reg1 ; // For use with Up counter.
reg    [3:0]    term_count_reg2 ;
reg    [3:0]    term_count_reg3 ;
reg    [3:0]    term_count_reg4 ;
reg    [3:0]    term_count_reg5 ;
reg    [3:0]    term_count_reg6 ;
```




We have some more registers term\_count\_reg1 through reg6 for use with an up counter. When you set the up counter terminal count, you need this.

(Refer Slide Time: 48:32)

```
reg [3:0] term_count_reg5;  
reg [3:0] term_count_reg6;  
  
wire [3:0] term_count_reg1_next;  
wire [3:0] term_count_reg2_next;  
wire [3:0] term_count_reg3_next;  
wire [3:0] term_count_reg4_next;  
wire [3:0] term_count_reg5_next;
```



```
wire [3:0] term_count_reg6_next;  
  
wire adv_term_count_reg1;  
wire res_term_count_reg1;  
wire adv_term_count_reg2;  
wire res_term_count_reg2;
```




You also have another terminal count for next till up to this here.

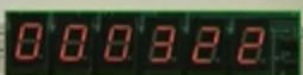
(Refer Slide Time: 48:42)

```
wire [3:0] term_count_rego_next;

wire  adv_term_count_reg1;
wire  res_term_count_reg1;
wire  adv_term_count_reg2;
wire  res_term_count_reg2;
wire  adv_term_count_reg3;
wire  res_term_count_reg3;
wire  adv_term_count_reg4;
```




```
wire  res_term_count_reg3;
wire  adv_term_count_reg4;
wire  res_term_count_reg4;
wire  adv_term_count_reg5;
wire  res_term_count_reg5;
```



```
wire    adv_term_count_reg6 ;
wire    res_term_count_reg6 ;

wire    adv_hrs_tcr ; // 'ter' means terminal count register.
wire    adv_mts_tcr ; // Applicable while setting.
wire    adv_secs_tcr ;
wire    term_count_reached_up ;
wire    term_count_reached_down ;
```




Then, you need to advance the terminal count register as well as reset. Once again, advance, reset, advance, reset for each of the six registers that you have here. You also need adv\_hrs. tcr here means terminal count register. You need some more signals. terminal count reached. For up counting if it is reached, this will be indicated. If it is down counting, the terminal count is reached when the set delay matches, that is what we mean, this will be done. They are all wires there.

(Refer Slide Time: 49:15)

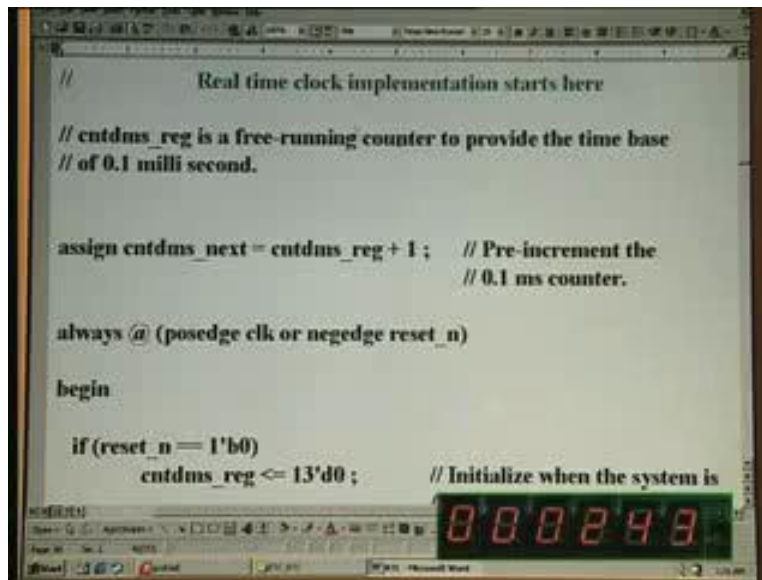
```
reg [4:0] timer_out_alarm_counter ;
           // 30 secs. counter for audio alarm.
wire [4:0] timer_out_alarm_counter_next ;

wire timer_out_alarm ;
```



You need a timer\_out\_alarm\_counter, which is a 30-seconds counter for audio alarm. Once again, pre-increment, then timer\_out\_alarm.

(Refer Slide Time: 49:25)



```
// Real time clock implementation starts here

// cntdms_reg is a free-running counter to provide the time base
// of 0.1 milli second.

assign cntdms_next = cntdms_reg + 1 ; // Pre-increment the
// 0.1 ms counter.

always @ (posedge clk or negedge reset_n)

begin

if (reset_n == 1'b0)
cntdms_reg <= 13'd0 ; // Initialize when the system is
```

The real-time clock implementation starts here. Do you have any specific question? Let us clear that first because we have just finished the declaration. I think we should take it up only in the next lecture so that we are fresh – the actual core is starting from here. So far, what we have seen is only the declaration.

Question (by Student): You said it is not required to press the buttons independently to set the time for up/down counting. If the button is kept pressed for a long time, you can advance the count. Can you please explain how this is exactly done in the code again?

Not right in the code but let us go back to the actual layout that we have started with. That was here (Refer Slide Time: 50:16). I said that we need to have this for the layman. Let us understand this here. If you want to set, let us say, hours, minutes, seconds. How do you go and set it up? What do you want to set? Do you want stopwatch to be set or timer to be set? Please come out with what you want. Stopwatch. Stopwatch, okay.

You just set it in set here, then stopwatch. Make sure that these things are all off. This is preferably in read, this can be off and this is also preferably off. What is going to happen



is we want to set this stopwatch. How much time do you want to set? 20 seconds. 20 seconds, okay. All we need to do is we just press this. Each time you press and release the button, this will advance by 1. For example, 01, next time you press it will be 02, 03 and so on. This is a very tedious thing. If you want to set 59, it will be a tedious process – you have to press 59 times. In order to relieve yourself of the burden, what you do is you just press once and hold it.

Having pressed once, the code will recognize and advance by 1 and then, it will continue to see whether you are continuing to press for 2 seconds and more. If it is beyond 2 seconds, it will start running at a very rapid pace and automatically it will roll. It will advance 10 times faster. For example, this will become 1, 2, 3 every second instead of this one. So you have a provision of setting it fast. Here, 1, 2, 3, 4 and when it comes to 5, you release it. Then you would have got almost 51, 52 – you have no control over that. After that, suppose you want to set 59 seconds instead of 20. You are now at 52, it has stopped and you keep on advancing only six times and that is precisely what you have here in the timer also – the buzzer has gone because it is the right time. Does it answer your question? Thank you.

(Refer Slide Time: 52:24)





Real Time Clock Design  
Features and Specification  
Block Diagram and Signal  
Descriptions of Real Time Clock  
Simplified Architecture of Real  
Time Clock  
Verilog RTL Code for Real Time  
Clock