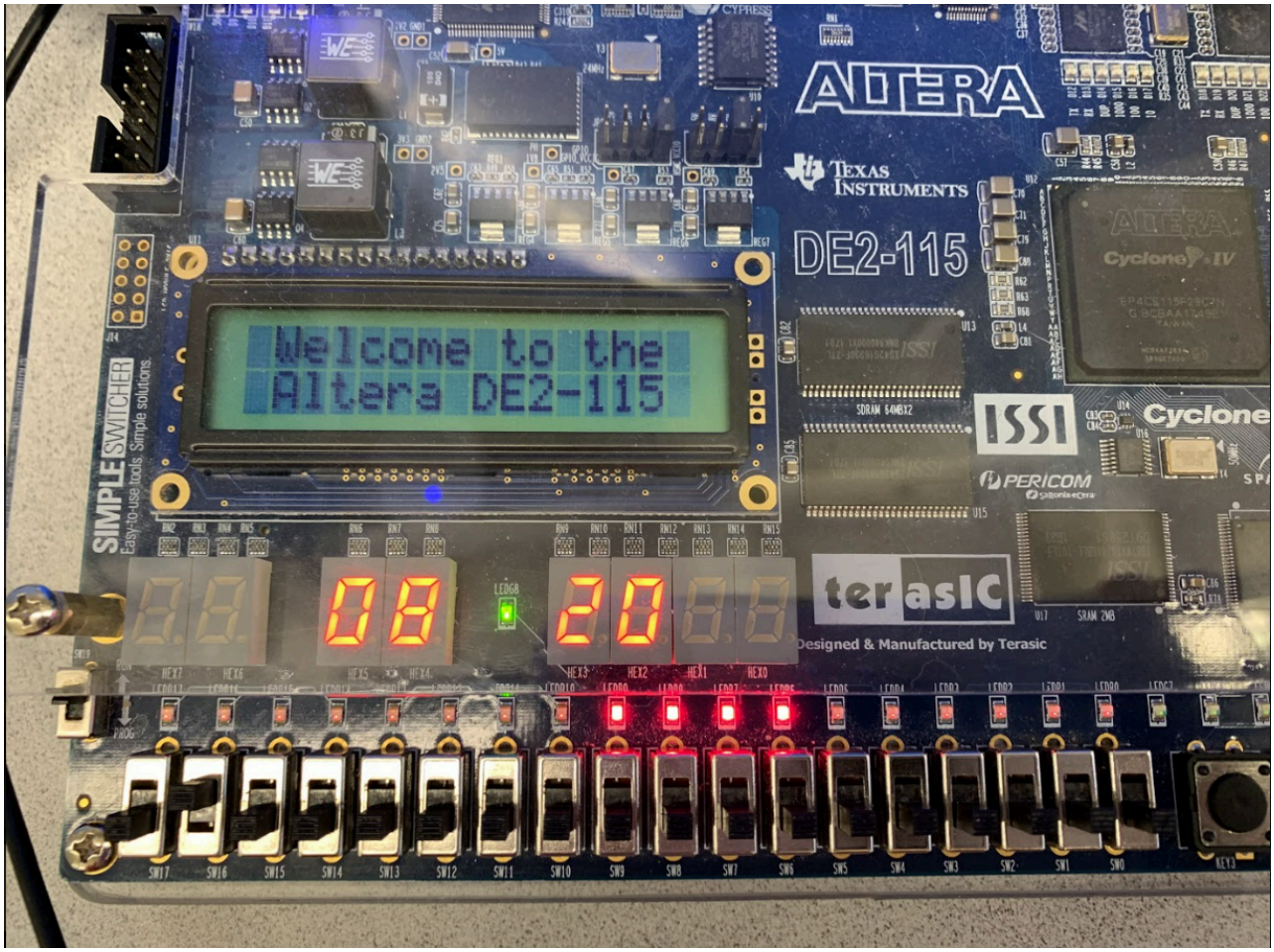
Svyatoslav Varnitskyy and Drew Kearns

Alarm Clock



Initial ideas:

When we were first brainstorming ideas for this project, we were thinking of a thermostat with an at-home and an away mode that would set the temperature to different degrees. We also considered doing a two-player guessing game that would have one player input a number in binary to guess, and the other would try to guess the number. Ultimately, we chose to do an alarm clock as it would be the right amount of challenge given our experience with the DE2-115 boards.

Chosen idea and implementation process:

Internal clock:

Our ideas for the clock included displaying the hours and minutes with a blinking light for seconds, setting an alarm, and flashing lights once the alarm goes off. A key part of a clock is its ability to count 60 seconds and then change the minute digit, which also counts to 60. The issue with the DE2-115 boards is that the internal clock is about 50 MHz, and a second is 1 Hz. S,o the priority was to create a new clock generator based on the one from lab 11. The clock generator in lab 11 only slows down the 50 MHz internal clock to 0.67 Hz, so we needed to create our clock dividers. To get the 50 MHz down to 1 Hz, we required a divide-by-5 divider and a divide-by-10 divider. The divide-by-10 divider was made using the divide-by-5 divider and a divide-by-2 divider. Using 1 divide-by-5 and 7 divide-by-10 dividers, we thought we had successfully slowed down the internal clock to 1 second. When we went to test it, however, we found that the clock was still high-speed, and through a lot of testing, we found that we had to add two more divide-by-2 dividers to slow down the internal board clock to a second.

Counters:

After we successfully slowed down the clock, we got started on the counters that would represent the hour and minute digits of a clock. Even though 60 is arguably the most critical number for a clock, we first made a counter to 24, representing the time in military hours. The counter was easy; the hard part was figuring out how to display a five-bit number with a 4 bit seven-segment display from lab 5. Then we remembered the BCD converter from lab 6, which uses a 5-bit input and outputs 8 bits to utilize two seven-segment displays. With the counter to 24 completed and displaying correctly, we moved on to arguably the most challenging part of the project, the counter to 60. This part of the project took the longest to get correct. When

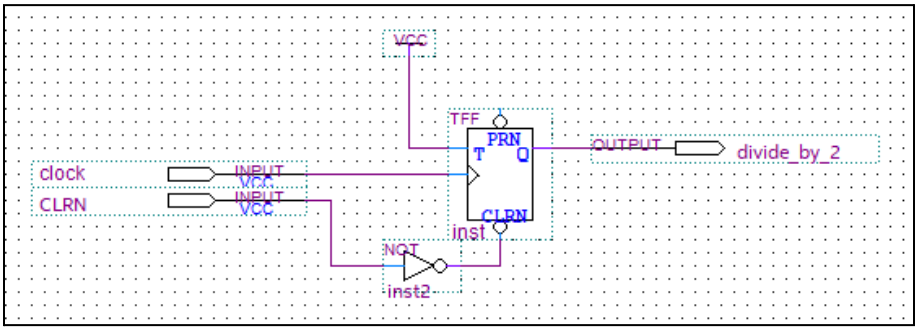
displaying the number, we could not use the BCD converter from lab 6, so we had to create a new one that went all the way to the number 63. This was excruciatingly painful as we had to make a 63-bit long truth table to find the equations for the required eight outputs. Once we got that figured out, we just had to put the counters and the clock generator together, and we had a working, displayable clock.

Putting it all together:

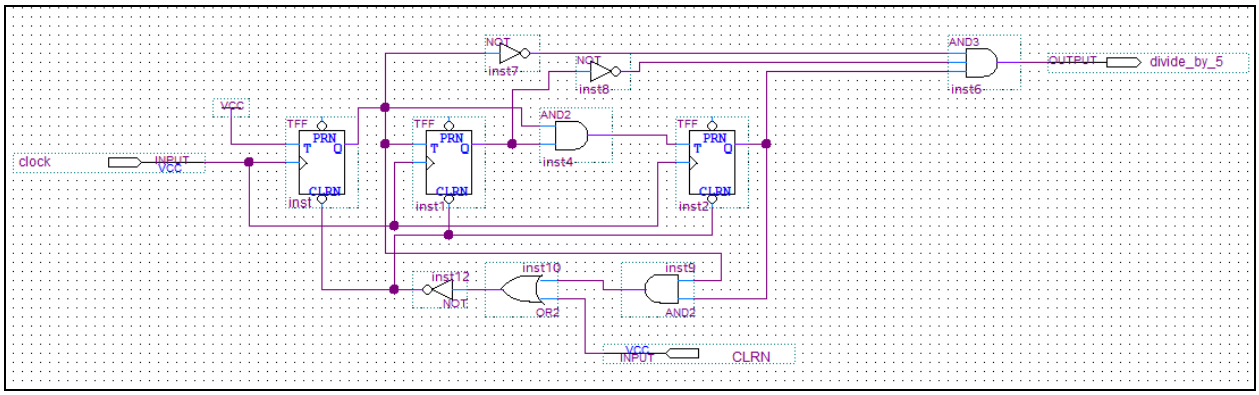
After the clock generator and counters were made, it was time to put it all together. This was pretty straightforward as we just had to connect the internal clock to the clock generator, then feed that through two counters to 60 and one counter to 24 to get our clock working. To make it display, we had to connect the counters to BCD converters and seven-segment decoders, and we had a displayable clock. Setting the clock and alarm took a couple of muxes and a few registers, respectively. Ultimately, this step was the easiest, and it was nice to see our work do what we wanted it to do.

Deeper look

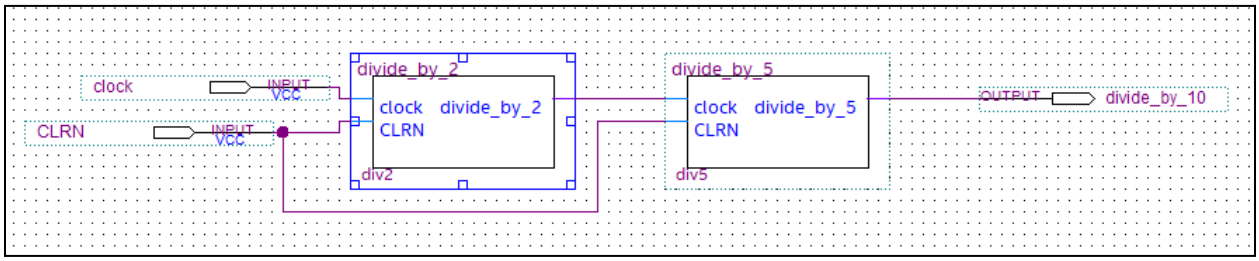
Clock dividers



Pictured above is the divide-by-2 divider, which was simple to implement. The T-flip-flop helps to delay the clock so that for every two inputs, only one output is given.

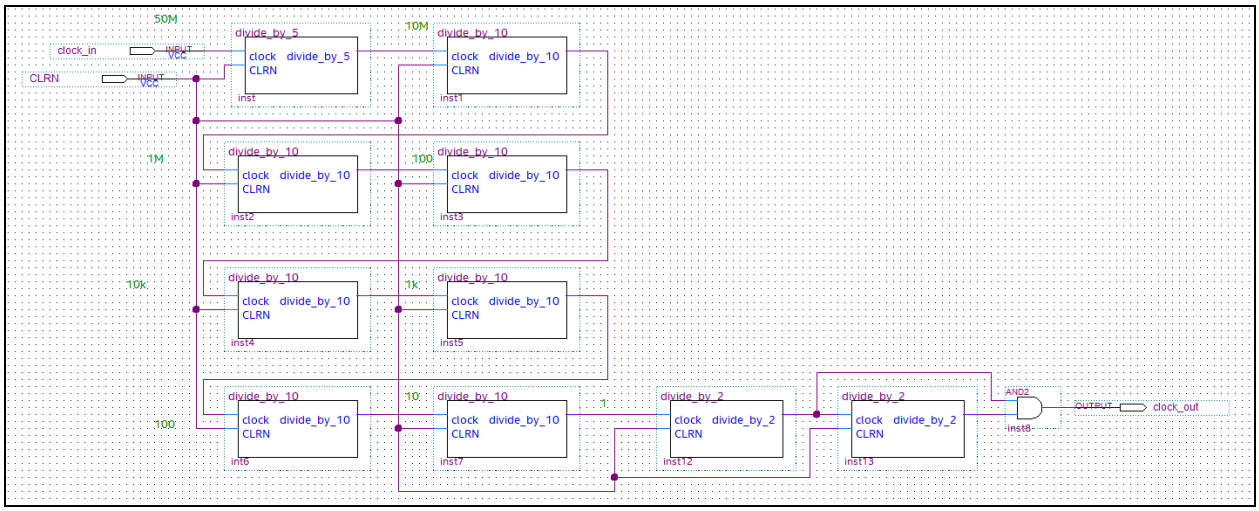


Above this is the divide-by-5 divider, a counter that resets once the first and last TFF outputs are 1, meaning the number 5 has been reached. An output is given once the last TFF reads 1, meaning there have been five clock cycles given that all the flip-flops start at 0.



The divide-by-10 divider is just a combination of the previous two dividers in succession. With the divide-by-2 divider only giving an output with every two clock inputs and the divide-by-5 divider giving an output with every five inputs, 2 times 5 equals 10.

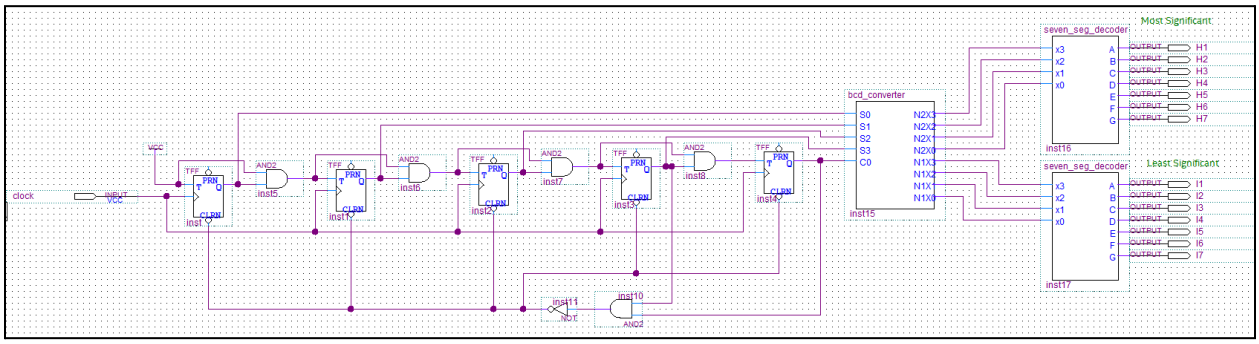
Clock generator



The clock generator was a little finicky. In one of the labs we did, we were told that the internal clock was about 50 million Hertz. To slow down the clock to 1 Hertz, we would need one

divide-by-5 divider and seven divide-by-10 dividers. When we tested that design, we noticed that the clock was still going way too fast, and even the TAs could not figure out why. One solution that was given to us was to add another divide-by-2 divider at the end. While it did slow the clock down a little, it still was not as close to 1 Hertz as we would like it to be. Through a little more testing, we found that the design pictured above gave us the nearest value to 1 Hertz.

Counters

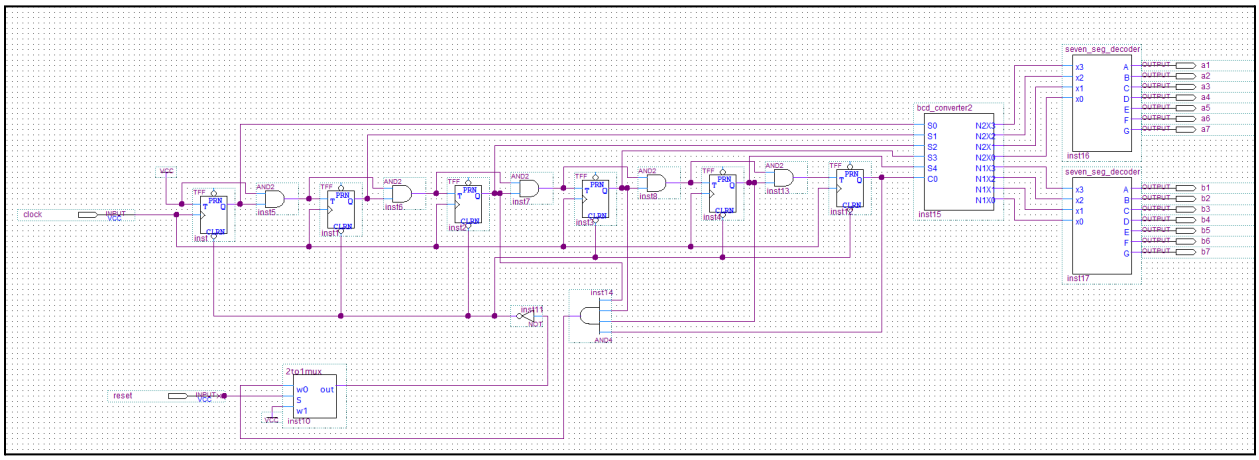


Above is the counter to 24. This asynchronous counter counts from 0 to 23 and resets once the most significant TFFs equal 1, meaning the binary number for 24 has been reached. This design utilizes the BCD converter from lab 6, which allows us to change our five-bit number into eight outputs that feed into the seven-segment decoder we made in lab 5. Thankfully, as I mentioned earlier, the BCD converter we made for the lab was already compatible with a 5-bit input since it was initially designed for use with an adder, and including this in our design made it easier to display what will be the hour digits.

```
module bcd_converter(S0,S1,S2,S3,C0,N2X3,N2X2,N2X1,N2X0,N1X3,N1X2,N1X1,N1X0);

   input C0, S3, S2, S1, S0;
   output N2X3, N2X2, N2X1, N2X0, N1X3, N1X2, N1X1, N1X0;

   assign N2X3 = 0;
   assign N2X2 = 0;
   assign N2X1 = ((C0 & S2) | (C0 & S3));
   assign N2X0 = ((~C0 & S3 & S1) | (~C0 & S3 & S2) | (S3 & S2 & S1) | (C0 & ~S3 & ~S2));

   assign N1X3 = ((~C0 & S3 & ~S2 & ~S1) | (C0 & ~S3 & ~S2 & S1) | (C0 & S3 & S2 & ~S1));
   assign N1X2 = ((~C0 & ~S3 & S2) | (~C0 & S2 & S1) | (C0 & ~S2 & ~S1) | (C0 & S3 & ~S2));
   assign N1X1 = ((~C0 & ~S3 & S1) | (~S3 & S2 & S1) | (~C0 & S3 & S2 & ~S1) | (C0 & ~S3 & ~S2 & ~S1) | (C0 & S3 & ~S2 & S1));
   assign N1X0 = S0;

endmodule
```

This is the BCD converter Verilog code made for lab 6, but has been utilized in our project. Our project “C0,” initially assigned the carry out the bit in lab 6, is now assigned the most significant bit in our binary number. The idea behind the BCD converter is to take the given five inputs and turn them into eight outputs. The seven-segment decoder will read these eight outputs to be transformed from binary to a decimal depiction on the seven-segment displays. The most significant of the eight outputs is N2X3-N2X0, which place when put on the seven-segment

display, will be the 10's. The least significant outputs N1X3-N1X0 will represent the 1's place when put on the seven-segment display.



The hardest part of this project was the counter to 60 pictured above. We naively started creating this with the counter to 24 as a model. As we tested the design by making each TFF output light on the DE2-115 board, we noticed that the clock reset at 48, not 60. We couldn't figure out why this was happening and couldn't fix it with help from Professor Stoytchev. When we connected the counter to the BCD converter two and the seven-segment decoder, we noticed that it was counting correctly and resetting at 60, even though it was resetting at 48 by itself.

```
module bcd_converter2(S0,S1,S2,S3,S4,C0,N2X3,N2X2,N2X1,N2X0,N1X3,N1X2,N1X1,N1X0);

   input C0, S4, S3, S2, S1, S0;
   output N2X3, N2X2, N2X1, N2X0, N1X3, N1X2, N1X1, N1X0;

   assign N2X3 = 0;
   assign N2X2 = (C0 & (S4 | S3));
   assign N2X1 = ((~C0 & ((S4 & S2) | (S4 & S3))) | (C0 & ~S4 & ~S3) | (S4 & S3 & S2));
   assign N2X0 = ((C0 & ((~S4 & ~S3) | (~S3 & S1) | (~S3 & S2) | (S4 & S3 & ~S2)))
               | (~C0 & ((~S4 & S3 & S1) | (~S4 & S3 & S2) | (S3 & S2 & S1) | (S4 & ~S3 & ~S2))));

   assign N1X3 = ((C0 & ((~S4 & ~S3 & S2 & S1) | (S4 & ~S3 & ~S2 & ~S1) | (S4 & S3 & ~S2 & S1)))
               | (~C0 & ((~S4 & S3 & ~S2 & ~S1) | (S4 & ~S3 & ~S2 & S1) | (S4 & S3 & S2 & ~S1))));

   assign N1X2 = ((C0 & ((~S4 & S3 & S2) | (~S4 & ~S3 & ~S2 & S1) | (~S4 & ~S3 & S2 & ~S1) | (S4 & ~S3 & S2 & S1) | (S4 & S3 & ~S2 & ~S1)))
               | (~C0 & ((~S4 & ~S3 & S2) | (~S4 & S3 & S2 & S1) | (S4 & ~S2 & ~S1) | (S4 & S3 & ~S2))));

   assign N1X1 = ((~C0 & ((~S4 & ~S3 & S1) | (~S3 & S2 & S1) | (~S4 & S3 & S2 & ~S1) | (S4 & ~S3 & ~S2 & ~S1) | (S4 & S3 & ~S2 & S1)))
               | (C0 & ((~S4 & ~S3 & ~S1) | (~S3 & S2 & ~S1) | (~S4 & S3 & S1) | (S3 & S2 & S1) | (S4 & S3 & ~S2 & ~S1))));

   assign N1X0 = S0;

endmodule
```

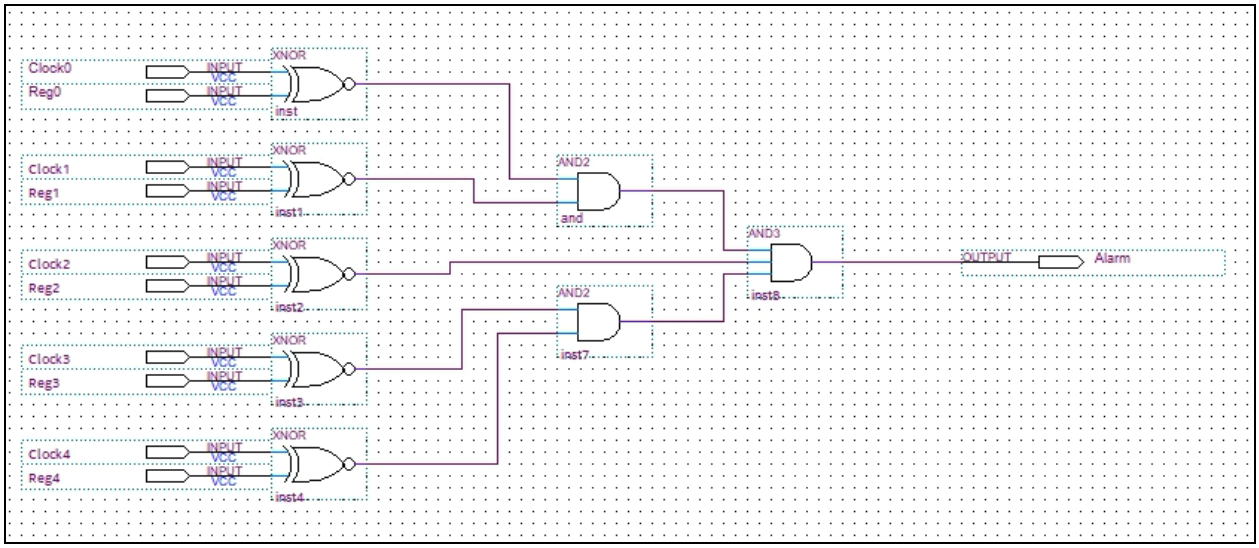
The bcd converter2 was also not fun to do. At first, we thought that we could just rename "C0" from the first BCD converter as "S4" and have the new significant bit be "CO," then just through a not in front of the new "C0" to keep the old equations and just work on the new ones. This, however, did not work. So we resorted to making a 63-bit truth table for the BCD converter two and coming up with entirely new equations. When all that was done, we tested it only to find that once we got to the number 56, it didn't work and would go down to 16-19 before resetting. This

was because we had accidentally used a “+” instead of “|” for N2X2, which took us 3 hours to find.

```
module seven_seg_decoder(x3,x2,x1,x0,A,B,C,D,E,F,G);
   input x3,x2,x1,x0;
   output A,B,C,D,E,F,G;

   reg [6:0] H;

   always @(x3 or x2 or x1 or x0)
   begin
      case({x3,x2,x1,x0})

         4'b0000: H = 7'b0000001; //1
         4'b0001: H = 7'b1001111;
         4'b0010: H = 7'b0010010; //3
         4'b0011: H = 7'b0000110;
         4'b0100: H = 7'b1001100; //5
         4'b0101: H = 7'b0100100;
         4'b0110: H = 7'b0100000; //7
         4'b0111: H = 7'b0001111;
         4'b1000: H = 7'b0000000; //9
         4'b1001: H = 7'b0000100;
         4'b1010: H = 7'b0001000; //11
         4'b1011: H = 7'b1100000;
         4'b1100: H = 7'b0110001; //13
         4'b1101: H = 7'b1000010;
         4'b1110: H = 7'b0110000; //15
         4'b1111: H = 7'b0111000;
      endcase
   end

   assign{A,B,C,D,E,F,G} = H;


endmodule
```

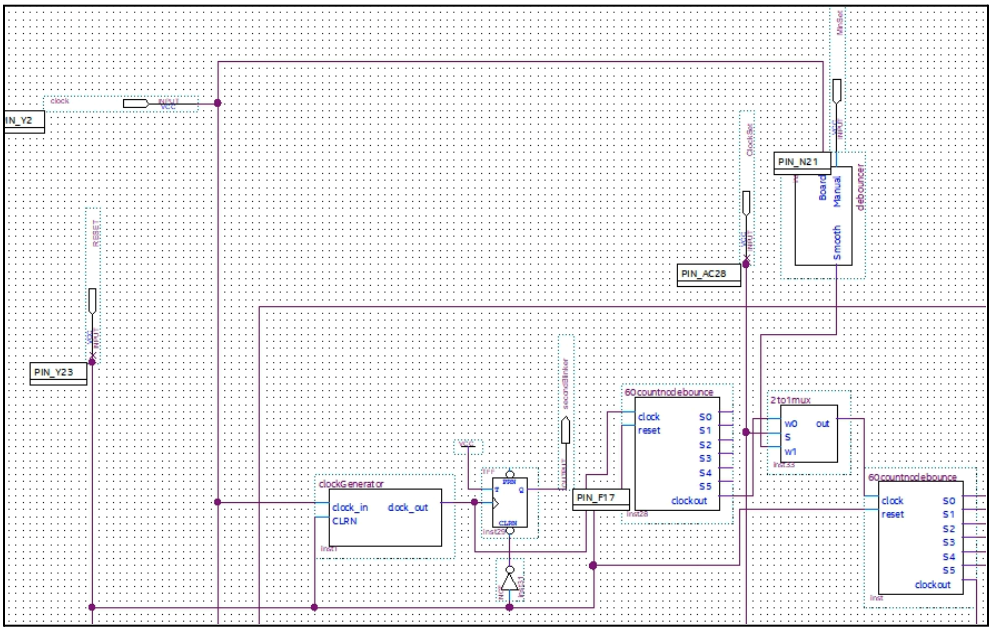
The seven-segment decoder from lab 5 takes in four inputs and creates four outputs. The most significant outputs are the 10’s digit, and the least significant outputs are the 1’s. Although this code allows us to go all the way to the binary number 15, we made our circuit with BCD converters that make the highest possible number 9. The combination of 1’s and 0’s that comprise the output is linked to the seven-segment display, with “A, B, C, D, E, F, G” assigned a specific part of the display. When the output is 0, the display turns on; otherwise, it remains off.

Alarm Clock Compare



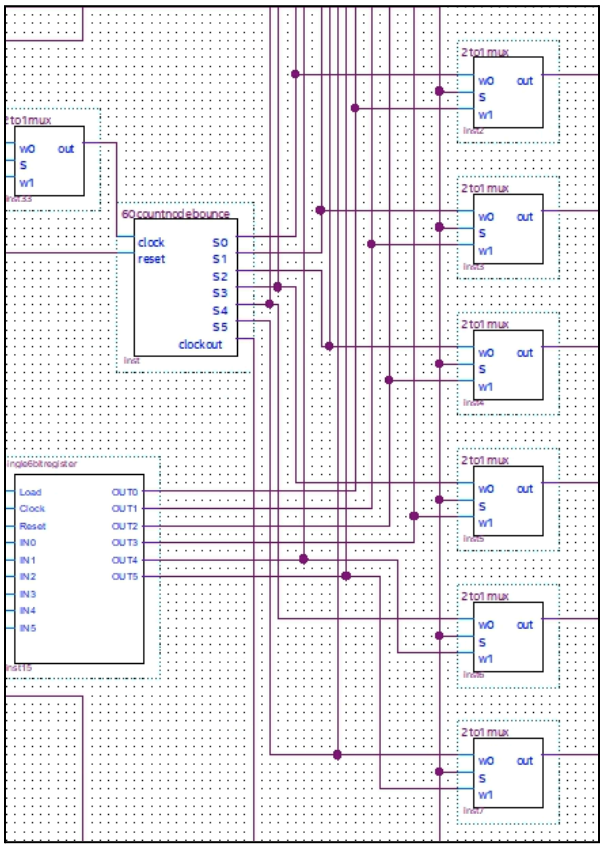
This is an alarm clock comparator. This is the 5-bit version, but we also made a 6-bit version for the minutes part of the circuit, which has an additional XNOR gate and two more inputs. We used XNOR gates since we needed a one output when either of the inputs was 0 or 1. These feed into some gates, and the module outputs a 1 when the alarm and clock times match. We used one module for the minute side and one for the hour side since we needed the alarm to go off when the time matched the alarm time stored in the registers.

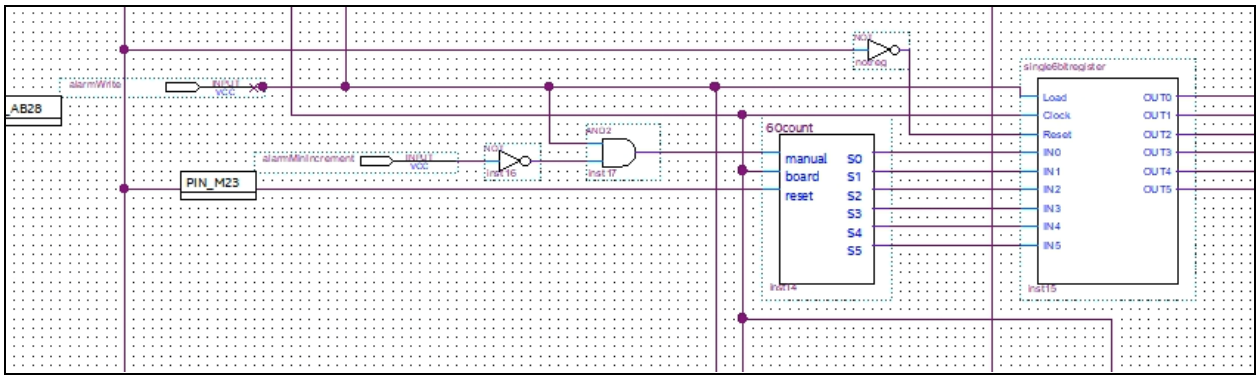
Final Assembly



Once we had all the pieces, we needed to assemble the final product in our main alarm clock file. We started using the clock generator, which slowed our clock to one cycle a second. This fed into

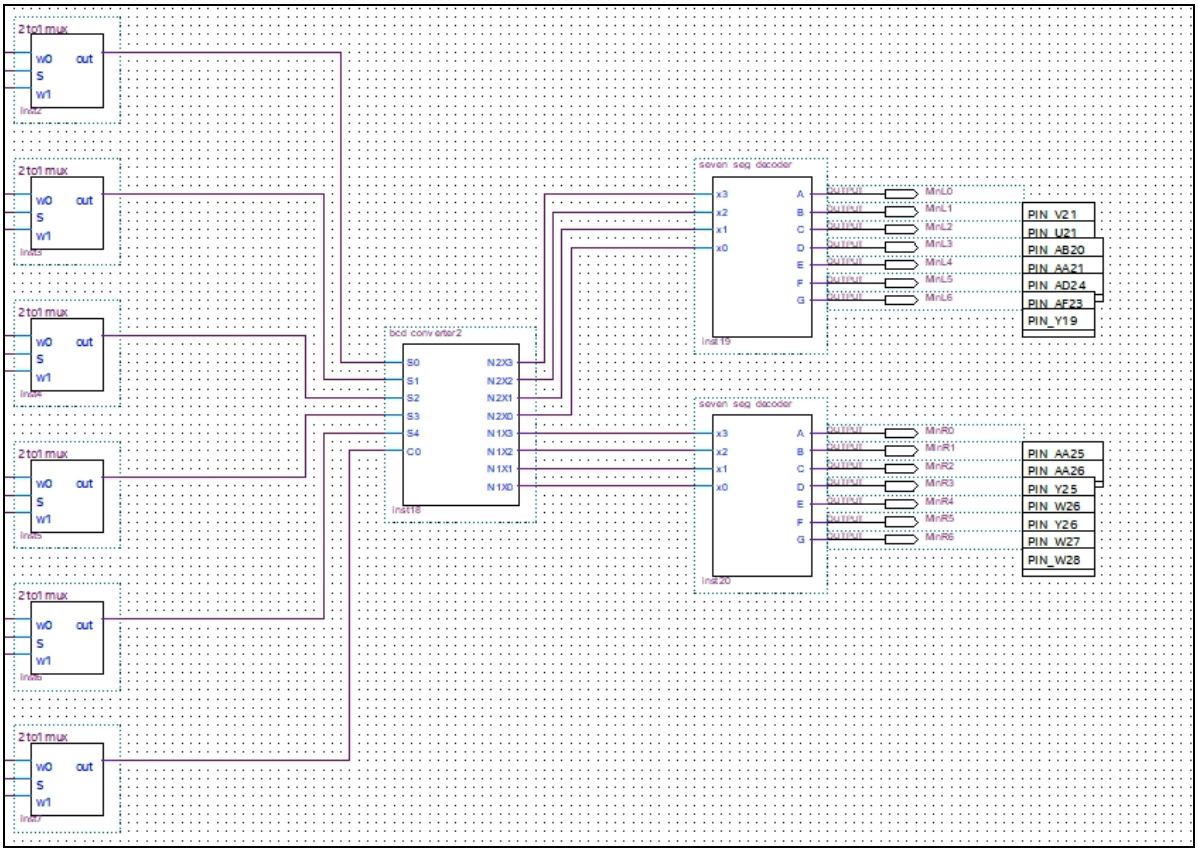
a TFF, which would change the clock to be on for a second and off for a second, which we used to create the blinking LED between our hour and minute seven-segment displays. The signal that avoided this TFF goes into the first 60-second counter, which counts up to 60 seconds. Since we didn't plan to display the seconds part of the clock, we didn't connect any other outputs than the clock out. The clock-out signal goes into a mux because we needed a way to set the clock. By default, the initial clock is displayed and works as usual; however, it will always start at 00:00. However, Once the select line is flipped and the reset is flipped up for a brief moment, the hour and minute set buttons are active. Here, we implemented a debouncer to accurately set the clock without overstepping. Once the set clock switch is flipped off, the clock functions normally. The mux output goes into our second 60 counter, which counts minutes here, but then we needed to display the output for S0-S5. We also needed to send the clock out to our hour counter which will get to later. But before moving on to the production, we also need to mention the alarm part of our design, as it works hand in hand with our clock output.



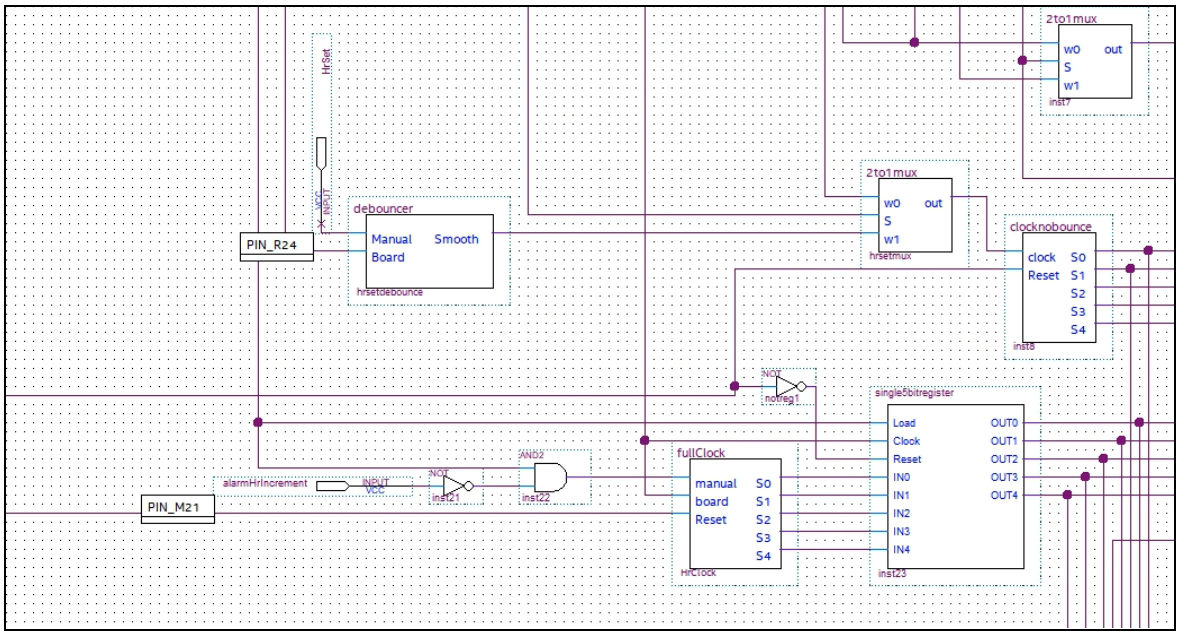


Pictured above is the part of our clock where you can set an alarm. To set an alarm, the set alarm switch must be up, and then the alarm can be set. So we added an AND gate with the load line as one input and the other as the button you press to increment the time. The AND gate feeds into a 60 counter, which is almost identical to our other 60 counter, just with the addition of a debouncer. The outputs for the 60 counter go into a 6-bit register to store the values set for the alarm to know when to make it go off.

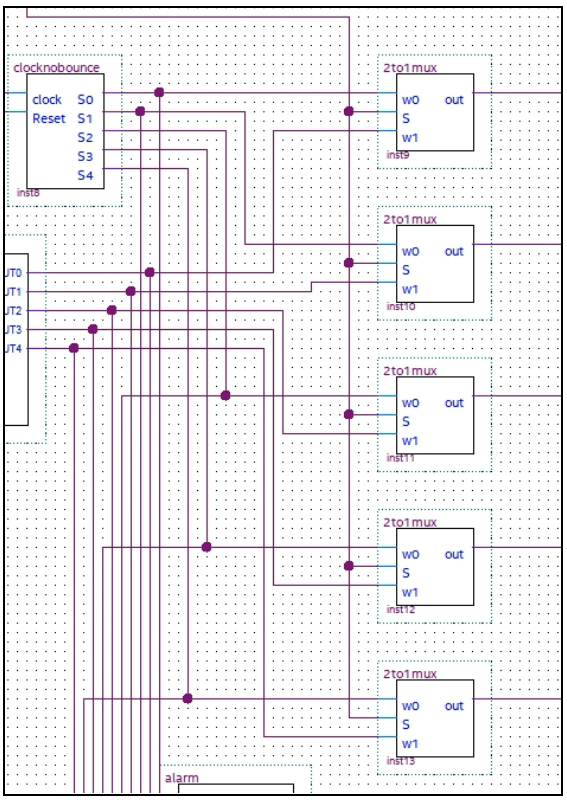
Now that we have both outputs, one from the clock counter and one from our register, which stores the minute part of the alarm, both go into the muxes. By default, the muxs display the time that is being output by the clock. However, once we flip the load/set alarm switch, these muxes switch to the register output, so the time set for the alarm to go off is displayed. Ignore all the other complicated-looking wiring for now; we will get to that later.



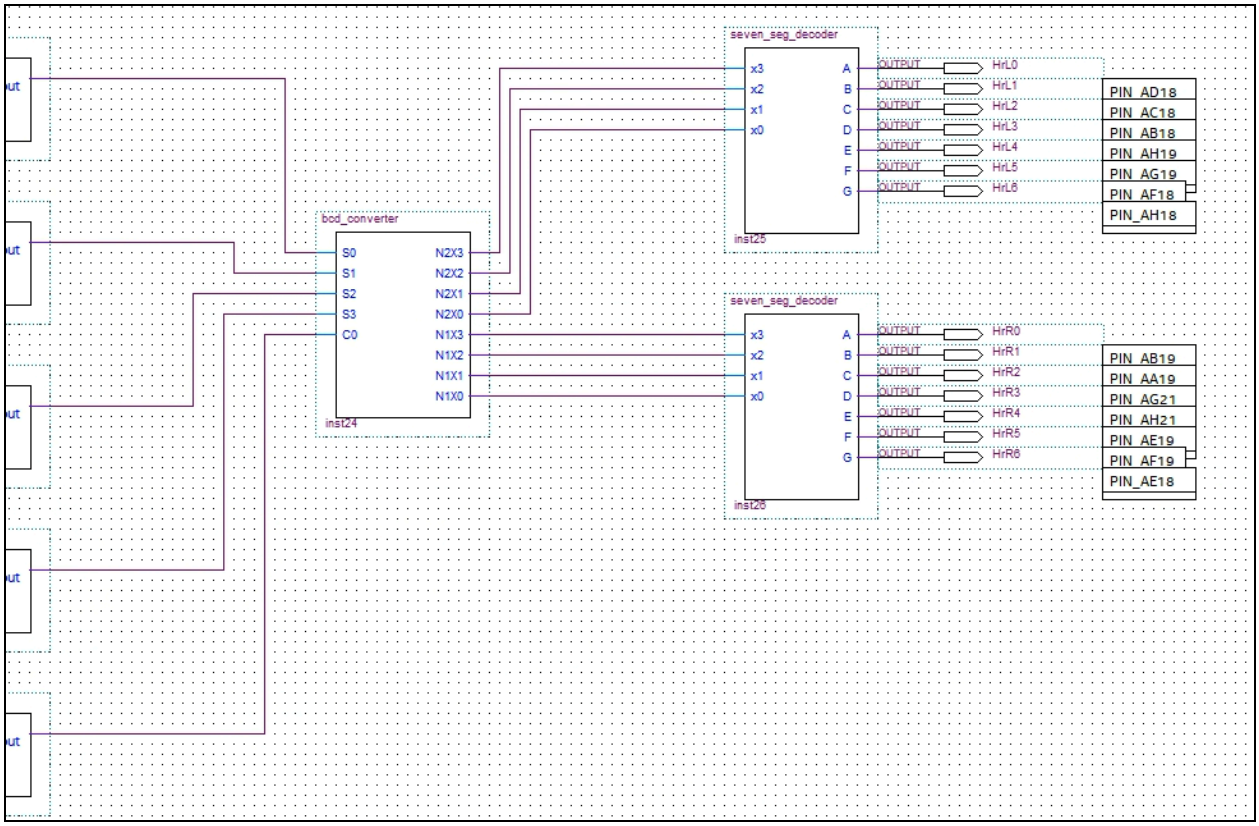
Now that we have the outputs in binary, we need to convert them to a seven-segment display signal, which can be used to show decimal numbers. So we have all our mux outputs going into a BCD converter, which can correctly display a number up to 59. The BCD converter then outputs to 2 seven seven-segmentegment display decoders, of which outputsuts to one seven-segment displatputs the number. The top is for the left part of the number, and the bottom is for the right half.



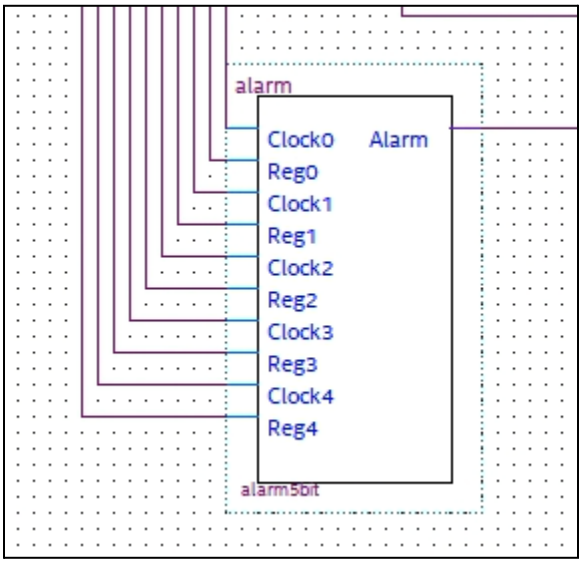
Now we repeat the whole thing for the hour part of the circuit. So we have another mux with the default input being the clock out from the minute counter and the other input being the manual clock used for setting the clock going into the clocknobounce module, which is the clock part of the circuit, which doesn't have the debouncer in the circuit. Then we have the alarm part on the bottom, which uses a 24 counter but with the addition of a debounce since the alarm will be set using buttons, and we didn't want any misinputs. The counter then goes into a 5-bit register since the most significant value we need to store is 31, even though our counter never makes it past that.

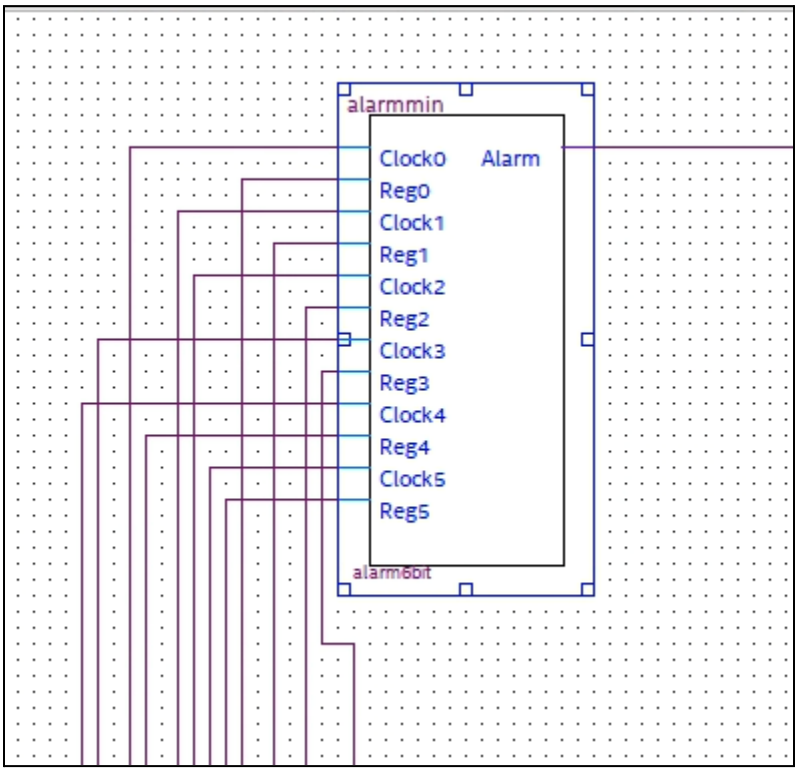


Same concept here as for the 60-minute counter, we will use muxes for each output from the clock and the register. The default output is the clock, and once the load/set alarm switch is flipped up, the muxes switch to the register and display the output. (Left)

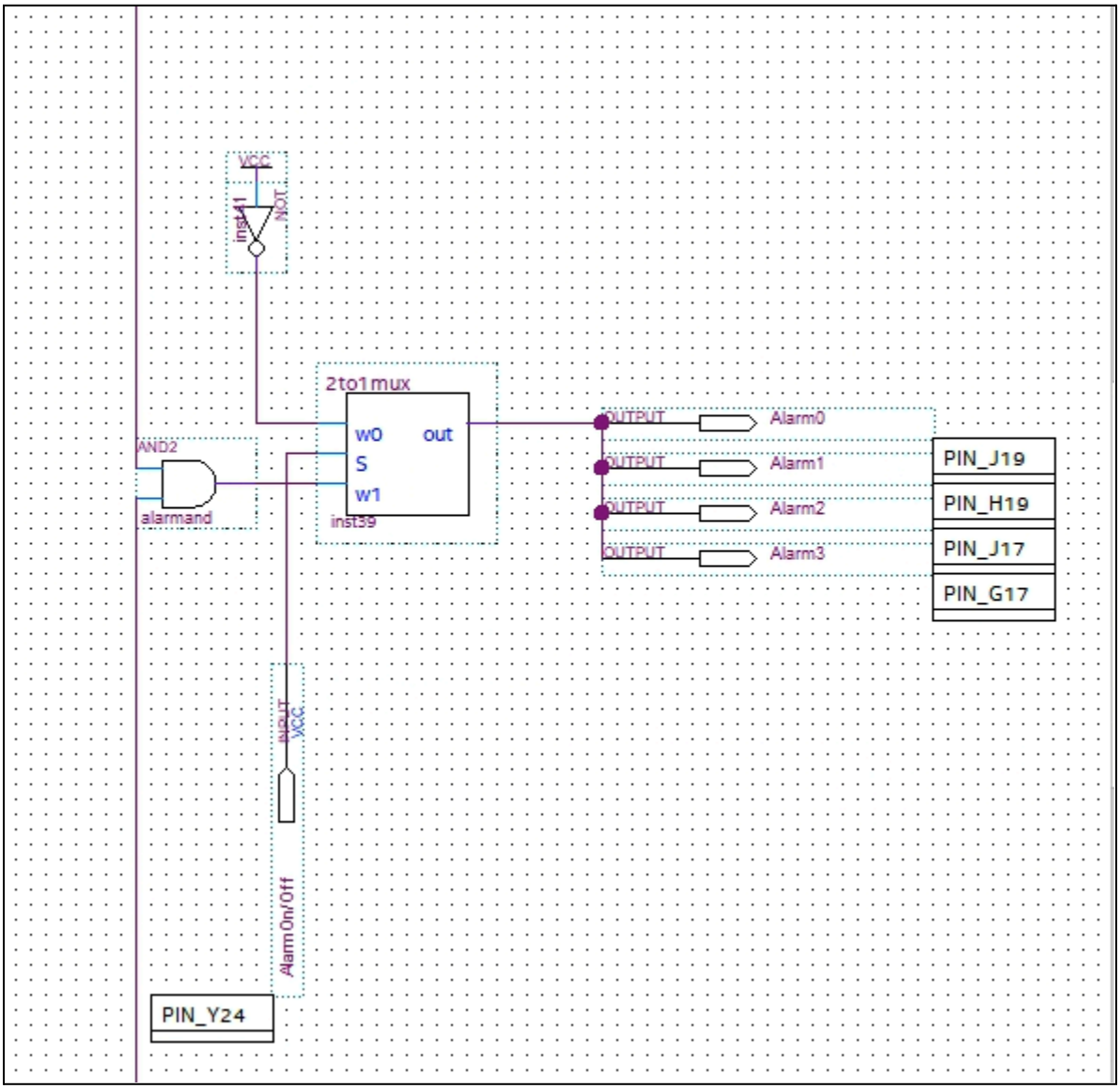


This is the same as the minute part, with the only change being the BCD converter. We just used the BCD converter from our lab since it can output numbers up to 31, and our highest value was 23. Then we connected all the BCD converter outputs to seven seven-segment display decoder inputs, with the top being the 10's digit and the bottom being the 1's digit. (Above)



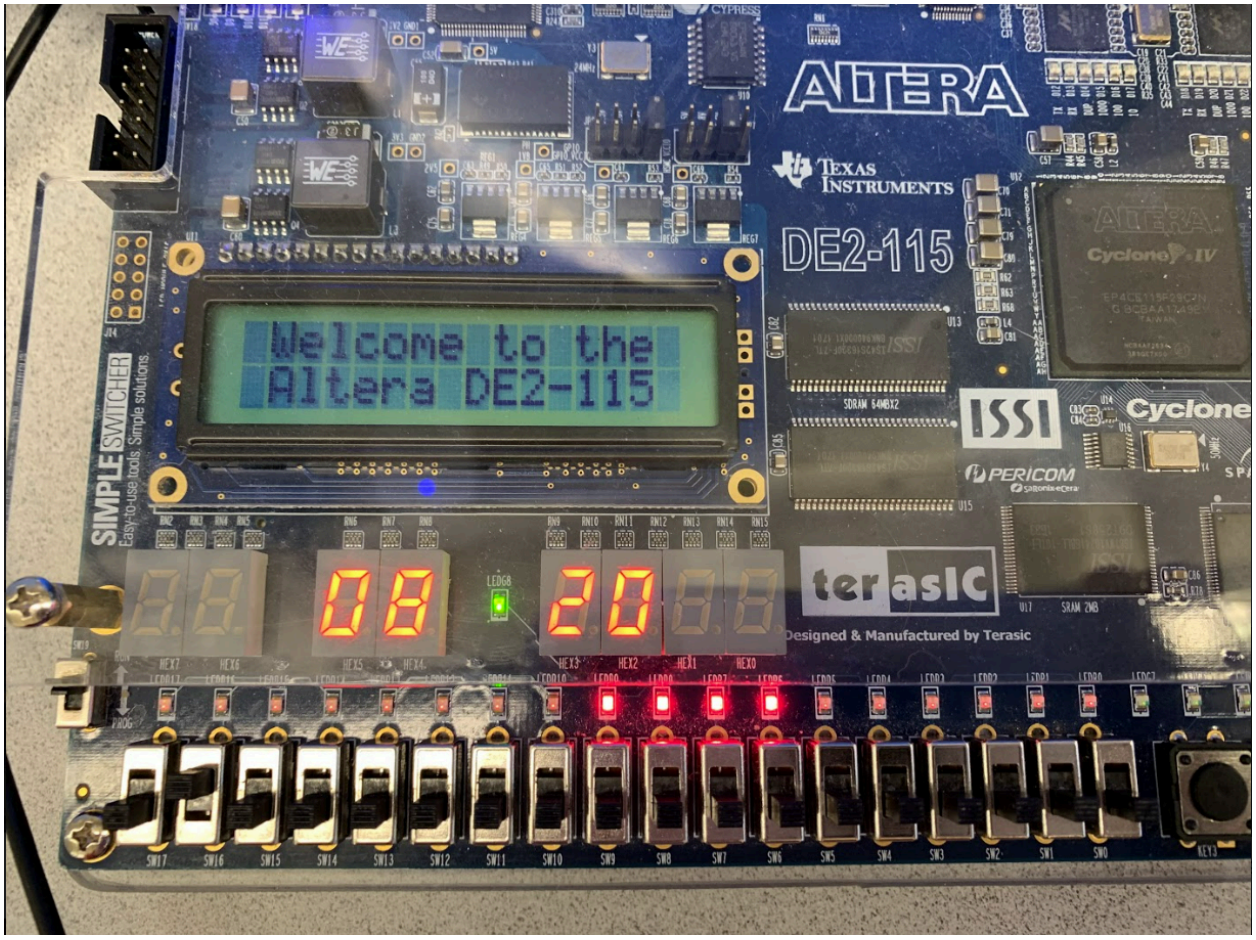


Finally, for our alarm to go off, we needed a way of comparing the time of the clock to the time of the register. So we created these alarm modules (pictured above), one for an hour and one for a minute, with XNOR gates to compare the inputs. When all the times match, an output is sent out.



These two outputs meet up at this AND gate, which will output a 1 when the alarm and clock times match up. As long as the alarm is armed there will be 4 LEDs to display that the alarm is going off. In the default state, when the alarm is off, nothing happens.

How to operate the clock



For ease of reference, we will use this picture of the DE2-115 board. From left to right, the keys/buttons needed are switches: SW17, SW16, SW1, SW0, and buttons: KEY3, KEY2, KEY1, KEY0. Loading the clock will start at 00:00 (note that this clock is in military time). To set the clock, you will need to switch SW1 up. You will see that this gives you a random clock value. We did not know why this was happening or how to fix it, so we just reset the clock every time, which can be done with SW17. If switch SW17 is up (meaning you reset the clock), you must switch it back down to set it. To set the clock to the desired time, use KEY3 to change the hour and KEY2 to change the minute. Now that the clock is set, you should see your preferred time. To set an alarm, you must switch up SW0, as you do this, the seven-segment displays will all read 00:00 until you set the alarm. This is a good thing, and you should not reset the clock unless you want to change the time again.

With SW0 flipped up, use KEY1 to set the alarm hour time and KEY0 to set the alarm minute time. Once you have set the alarm, flip switch SW0 back down. To see that the alarm has gone

off, switch SW16 must be flipped up. You will know the alarm has gone off when the four red LEDs pictured above are turned on. To shut the alarm off, flip SW16 back down (note that if you set another alarm, it will not go off until switch SW16 is flipped back up). If you want to change the time again and decide to reset the clock when you do so, you will find that your alarms have also been reset. We advise that you set the clock before setting the alarm. That way, you do not have to set the same alarm twice. That's all there is to it; now enjoy messing with our clock.