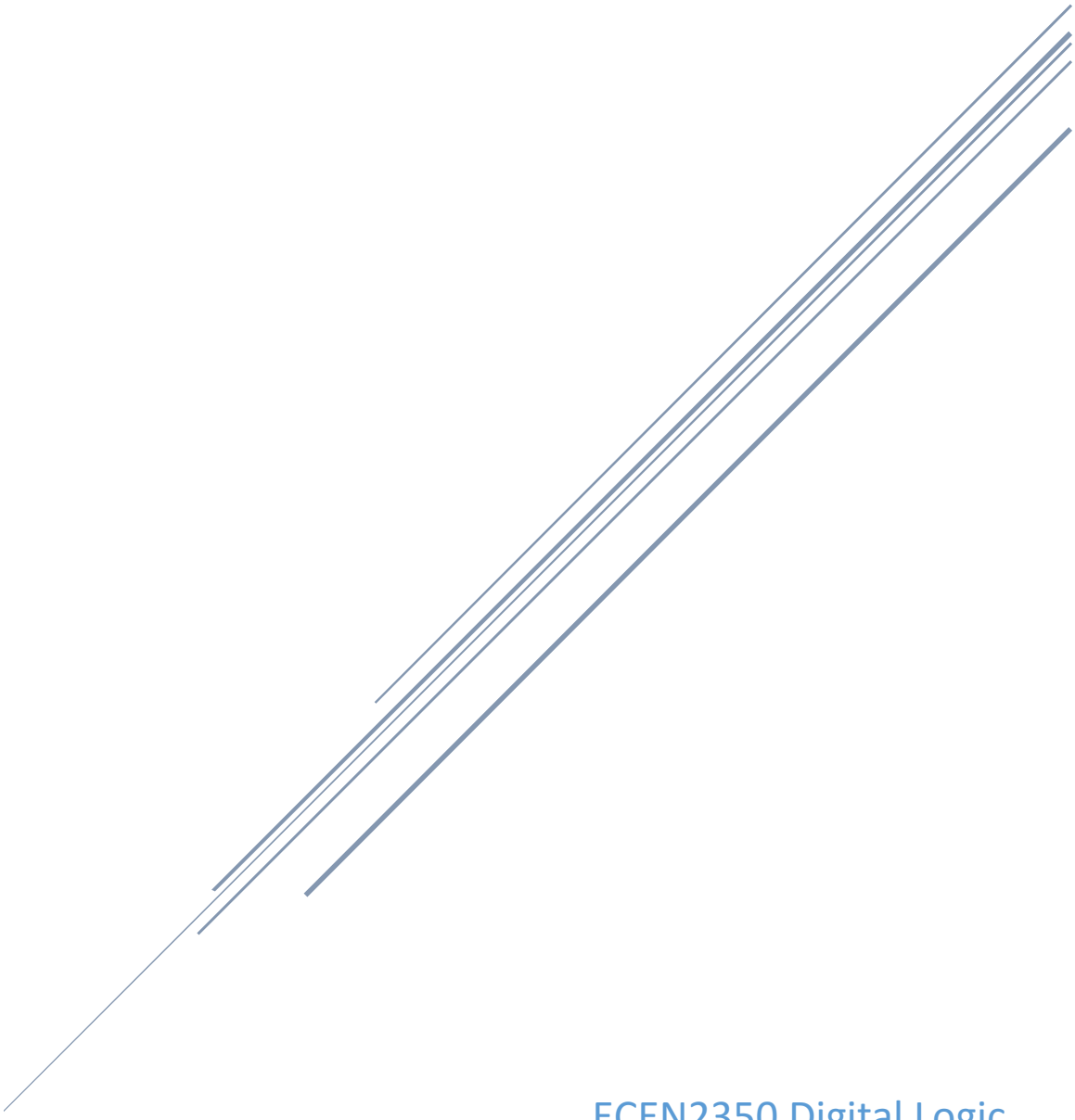
REACTION TIMER

Huan Nguyen, Jonathan Peterson



ECEN2350 Digital Logic
Spring 2016

Introduction

In this project, we made a reaction timer using the DE0 board. With two buttons and one of the switches on the board, a user can test their reaction time with accuracy up to one-hundredth of a second.

They do so by pressing a start button, which activates a pseudo-random delay before an LED lights up and the timer starts counting. Once the LED lights, the user presses the stop button, which turns off the LED and freezes the timer, allowing them to see how much time passed between the LED lighting up and their button press. The user then actuates the switch (switch zero) high then low, to reset all values, including the timer, and put the board back into an idle state where it waits for the start button to be pushed again.

The major stages of this project were: figuring out how to make a 100Hz clock signal (to get the hundredth-second accuracy) from the 50MHz clock signal that the DE0 includes; figuring out how to make a binary-coded decimal (BCD) counter that keeps track of the hundredth-seconds; displaying the counter; generating the random delay between the start button and the LED lighting; and implementing the actual timer from a mock state diagram.

Generating the 100Hz Signal

Generating the 100Hz signal from a 50MHz signal was a bit tough at first; we knew that, in a counter, each successive most significant bit counts at a rate half that of the bit before it (e.g. the leaset significant bit would count at 50MHz, the next bit at 25MHz, and so on).

The problem there was that, even using a counter with a large number of bits, dividing the frequency by powers of two still did not put us at 100Hz. Using a 19-bit counter (that counted to a max decimal value of 524,287), we would have a signal clocking in at 95.4Hz (from dividing 50,000,000Hz by 2^{19}). That was close, but we wanted and needed to have a signal at 100Hz with minimal error.

The solution that we eventually used, was to have an 18-bit counter that counted up to 249,999—which, starting from 0, is exactly 250,000 increments, or $\frac{1}{200}$ of 50,000,000 (we used an 18-bit counter because 18 bits can count up to a max decimal value of 262,143). We used an if-else block to check whether the counter was at 249,999, else to increment it.

Why did we use a counter of 250,000 increments? The 50MHz signal will make the full count from 0–249,999 two hundred times in a second. If we use that if-else block to toggle a register (which we did) each time the counter reached 249,999, the register would be logic high half the time and logic low the other half. We then used that register (called clk100Hz in our code, included in the appendix), as the 100Hz clock signal that we needed.

Once we figured that out, we had to make the BCD counter that kept track of the hundredth-seconds, using the 100Hz clock signal as an input.

The BCD Counter

For the BCD counter, we used the code given in the book (Figure 5.62, Section 5.14 of *Fundamentals of Digital Logic with Verilog Design*, our textbook for the course).

The code for the counter was easy enough to understand and then extend to a third digit.

The module, BCDcount, is given as inputs a clock signal, a clear signal, and an enable signal, and as outputs, three (two in the book's example) 4-bit registers that hold binary-coded decimal values (from 0000_2 to 1001_2, or 0-9 in decimal).

In an always block (triggered by the rising edge of whatever clock signal passed to the module—in this case, we passed the 100Hz signal to it), BCDcount will check if the clear signal is a logic high; if so, it sets all the BCD values in the 4-bit registers to 0. If not, an else if will check if the enable signal is high. If so, a nested if statement will check if the first digit (as represented by the BCD value in its respective 4-bit register) is a decimal '9'. If so, the digit will be set to 0, otherwise it will be incremented. If the first digit *is* a decimal '9', then a nested if conditional will check if the second digit is a decimal '9'. If not, the second digit will increment (and the first digit will have been reset to zero). If the second digit is 9, the process repeats once more for the third and final (for our project) digit: if the third digit is a '9' as well, it is set to 0 and the clock rolls over to 000. If not, the third digit is incremented by one.

There is a clear signal as well, being passed in from the top-level module, which is set by a switch. If the switch is high, the BCD values clear to 0.

Having gotten the BCD counter working with the help of the book, we turned our attention to having the DE0 visibly display the counter.

Displaying the BCD Counter

For this, we also used the code given in the textbook in Figure 5.63 of Section 5.14, the 'module seg7' BCD to 7-segment decoder example code.

This module takes as input a 4-bit register (the BCD value set by the BCDcount counter, in this case), and outputs 7 distinct bits to be used for setting a 7-segment display. The seven distinct bits in the output are assigned in nine different case statements (for 0-9 in decimal), and those case statements are selected between using the 4-bit BCD input to the module.

In our main 'reaction' module, we called seg7 three times, and passed it each BCD digit that the BCDcount module provided at the 100Hz rate. In the main module, we had three 7-bit output wires that the seg7 modules each output to, and assigned the output of each of those 7-bit wires to the 7-segment digits on the DE0.

The problem with the code that we fixed: in the example, the outputs are given so that each bit, of the 7-bit output that is intended for a segment of a 7-segment digit, is set to logic high to light up its respective LED. However, on the DE0, the individual LEDs of the 7-segment digits are active low.

This meant that the example code was setting each bit exactly the opposite of what it needed to be set to display each unique digit. Since the error was so exact and opposite, the fix was easy: we put a tilde in front of each of the values being assigned to the 7-bit output to invert the bits.

Now, we had a BCD counter with an input signal of 100Hz, and a way to display the counter. The next step we attacked was that of generating a random delay.

Generating Random Delay with a Linear Feedback Shift Register

At this point, we had a working implementation of something that was not quite a reaction timer. With the proper pin assignments, we could press a button, have an LED light and the BCD counter start at 100Hz and display, and turn off the LED and freeze the counter with another button (and reset the values with a switch), but we had not yet made the start delay. The moment the start button was pressed, the LED would light and the counter would begin.

To truly be a reaction timer, we had to figure out a way of implementing a random delay between the push of the start button and the lighting up of the LED.

The hint we were given in the project description was to use a linear feedback shift register (referred to as LFSR) generator.

First, we had to figure out exactly what a LFSR did. It turns out that LFSR generators generate pseudo-random sequences of binary values using a shift register of n-bits (where n=2, 3, 4, 6, 7, 15, 22, 60, 63, 127—we used 7, as 15 gave too long of a delay, and 7 was the next least number) and an XOR gate. In this case, in a reversal of the example given in the project description, we connected the inputs of the XOR to the two least significant bits, and the output to be assigned to the most significant bit of the 7-bit of the shift register (which does right shifting, from the most significant to the least significant bit).

As bits shift over, the XOR introduces enough of a variance into the input of the most significant D flip-flop that the sequence of values 'feels' random enough to the human perception (though calculating the sequence of values is well within the realm of possibility) to use as a random delay between the start button and the LED lighting up.

All we had to do then was write the module (which is included in the appendix) and output the 7-bits to a 7-bit register in the main 'reaction' module, which was the next challenge we faced: putting all our previous steps together.

Implementation of the Reaction Timer

To start building our mental model of the overall project, we drew a simple state diagram:

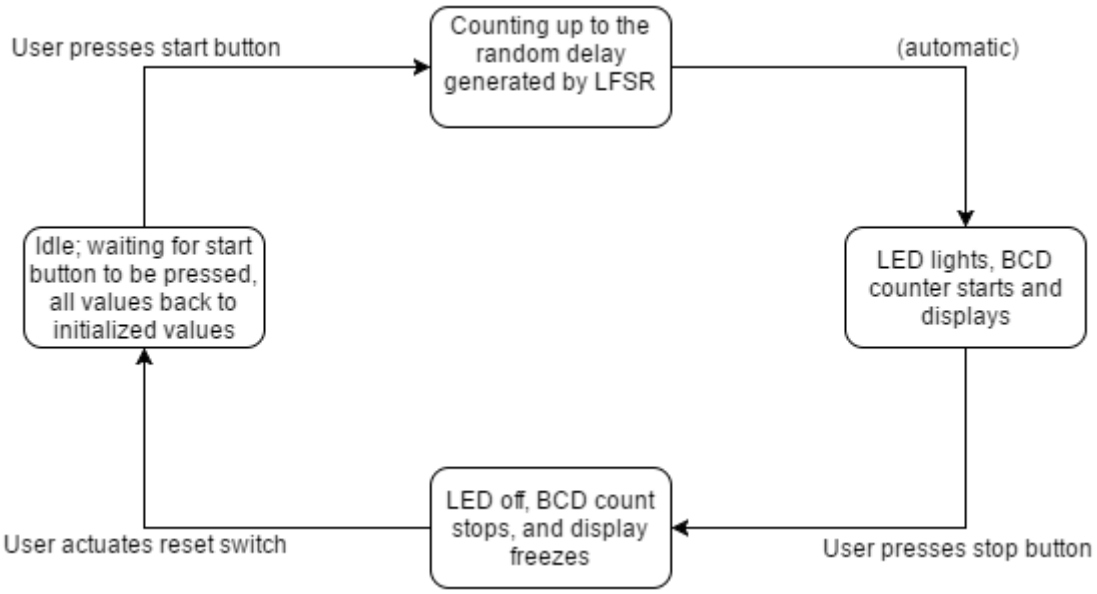


Figure 1: state diagram for reaction timer

Our top-level 'reaction' module was where we tied everything together. We implemented the clock divider inside this module, and called the rest (BCDcount, seg7, and LFSR), tying the outputs, of the modules being called, to output wires and registers in the main 'reaction' module.

After figuring out the first few puzzle pieces like the BCD counter and the LFSR generator, writing the top-level module became really only about implementing the state machine and having everything be synchronous to the 100Hz signal, as well as having the proper inputs and outputs.

We set as inputs the 50MHz clock signal, a start and stop button—the 'w' and 'push_n' buttons, respectively—, a reset switch to put all values back to their initialized value, wires in from the sub-modules and out to light the LEDs and 7-segment displays, and registers to hold state variables ('reg LED') and the random value generated by the LFSR generator.

On every clock edge of the 100Hz signal, we check to see if the start button 'w' has been pushed. If so, the current value in the LFSR will be assigned to a register for another counter to check against, and starts the counter.

Another if checks that the register holding that counter is equal to the register holding the value taken from the LFSR generator; if not, the counter is incremented until the two values are equal. Once this is so, signal 'LED' is set high, turning on the LED and starting the BCD count.

When that happens, the user presses the stop button, and the register which holds the value given by the LFSR is set to 0 (to stop the random delay and stop the LED from lighting up again after the user presses stop), and the register holding the 'LED' signal is set to low (which is also the enable to the BCDcount module, which stops, and the display is stopped at whatever reaction time the user had).

As mentioned in the section about the BCDcount, the user then sets a switch high and low, which clears values and sets the DE0 back to its idle state of waiting for the start button to be pressed.

Testing the Reaction Timer

To test that the initial delay is random, we performed ten trials where we pushed the start button on the DE0 and a stopwatch at the same time, and stopped the watch when the LED lit. This obviously introduced some human error, but we are confident in our perception that the delay time indeed varied between trials, and our results supported our conclusion.

Trial #	Time (seconds)
1	1.201
2	0.221
3	1.443
4	0.736
5	1.394
6	1.301
7	0.645
8	0.787
9	0.606
10	1.196

Figure 2: results from LFSR time trials

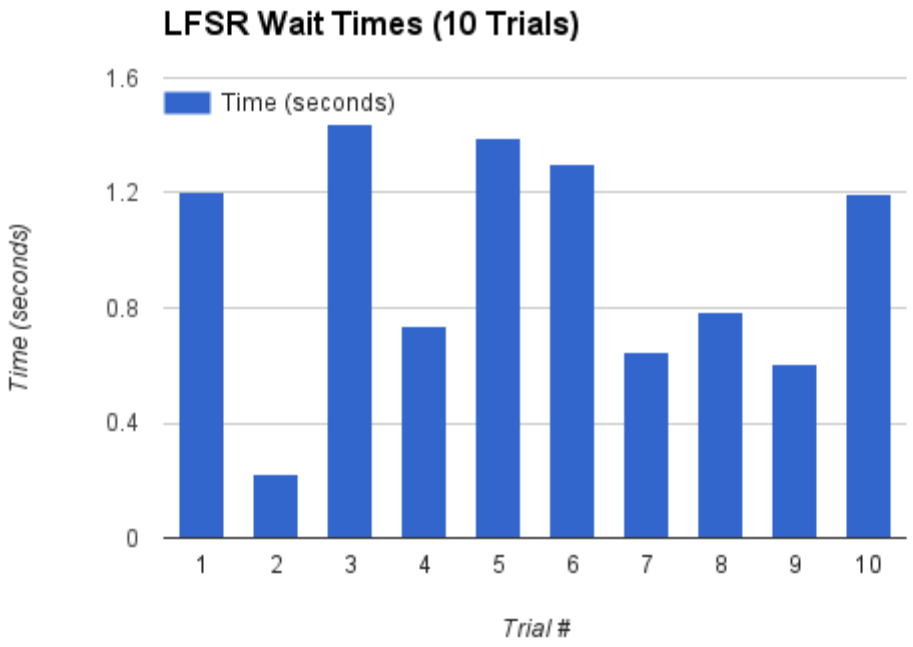


Figure 3: results in bar format

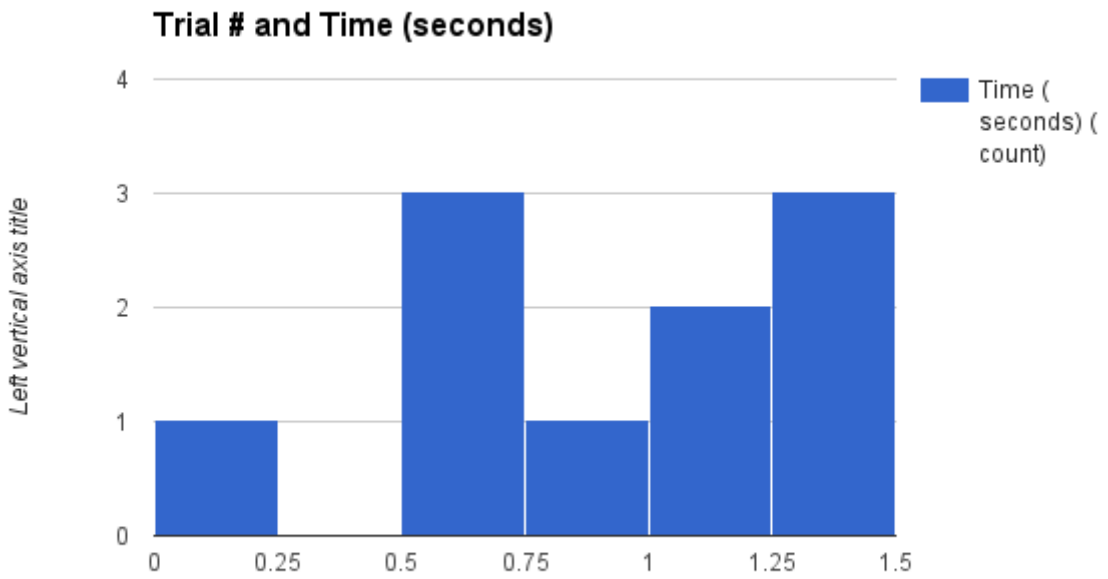


Figure 4: results in **histogram** format

We actually found that some math supported our perception: with a register of 7-bits, the max decimal value is 127. A 100Hz signal would count to 127 in 1.27 seconds, and the variance introduced by the linear feedback (the XOR gate) would make that time fluctuate.

Our results are well within that value; the values that are slightly over, we believe are due to the human error inherent with the delay from when the LED lights, our brains process it, and we stop the stopwatch. If the textbook is to be believed (at the end of Section 5.14), the average human reaction time is around a fifth of a second. If we subtract twenty-hundredths of a second from each of those values, they all fall under the calculated limit of 1.27 seconds.

After testing the LFSR, we tested the reaction timer, and asked some friends to do so as well. Each person did five trials on the reaction timer.

Trial #	Tyler	Milica	Joey	Huan	Jonathan
1	27	17	22	16	19
2	27	22	21	17	19
3	29	20	26	25	22
4	25	24	21	18	20
5	26	21	19	22	18

Figure 5: results of the reaction timer trials, with reaction time measured in hundredth-seconds

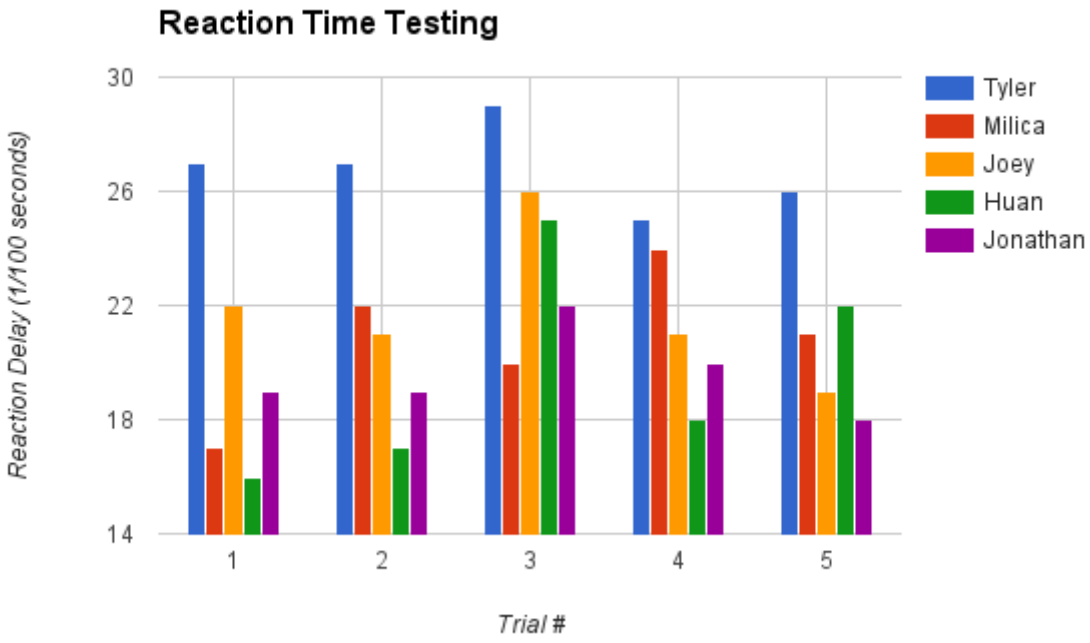


Figure 6: results of reaction timer trials, in bar format

Conclusion

Overall this project was quite fun to work on, and really challenged our logic design skills.

Each component of the project took some thought, and putting them together according to the plan and state diagram we had outlined previously also took some redesigning. Part of the biggest challenge there, we believe, is the fact we locked ourselves into one way of thinking by looking at the code in the book. By using that code, we primed our brains to think in the same way as the book, which worked out in our favor this time, since that code was written well and easy for us to read.

Putting the puzzle pieces together required thought in that the modules all worked well on their own, but tying them together synchronously using always blocks was challenging in a fun way. Pride and satisfaction abounded after we figured out how to do so.

This project helped to cement the logic concepts and Verilog we covered throughout the semester, and we were very happy with the final product; friends had fun with it as well.

Appendix

Bibliography

Brown, Stephen, and Zvonko Vranesic. *Fundamentals of Digital Logic with Verilog Design*. New York, NY: McGraw-Hill, 2014. Print.

Code

```
module reaction(clk50MHz, reset, clear, push_n, LED_out, w, digit2,
digit1, digit0);

	// Inputs
	input clk50MHz, // 50MHz signal from DE0
	reset, // intended to be a reset button, but changed to switch
	w, // input/start button
	clear, // reset switch
	push_n; // user push button when LED lights

	// Outputs
	output wire LED_out; // negative of LED reg, to light LED
	output wire [1:7] digit2, digit1, digit0; // 7-bit output
	// to link to 7seg displays

	// Registers and Wires
	reg LED; // variable to light LED and enable BCDcount
	wire [3:0] bcd2, bcd1, bcd0; // 4-bit encoded decimal
	// coming from BCDcount
	wire [7:0] rand_num;
	reg [6:0] lfsr_num = {7{1'b0}};
	reg [6:0] lfsr_count = {7{1'b0}};


	/*************
	Generate 100 Hz signal using a 50MHz signal
	*************/
	reg clk100Hz = 0;
	reg [17:0] count250k = 0;

	always @ (posedge clk50MHz or negedge reset)
	begin
		if (!reset)
		begin
			clk100Hz <= 0;
			count250k <= 0;
		end
		else if (count250k < 249999)
			count250k <= count250k + 1'b1;
		else
		begin
			count250k <= 0;
			clk100Hz <= ~clk100Hz;
		end
	end
	// end of generating 100 Hz signal

	assign LED_out = LED;

	always @ (posedge clk100Hz) // On every +ve 100Hz edge
	begin
		if (!push_n)
		begin
```

```
                LED <= 0; // set LED low, LED_n low, LED off and
                // BCDcount off
                lfsr_num <= 0;
          end
          else if (!w)
                lfsr_num <= rand_num;
          if( (lfsr_num !== 0) && (lfsr_count == lfsr_num) )
                LED <= 1; // set LED high, LED_n high, LED lights and
                // BCDcount starts
          else
                lfsr_count <= lfsr_count + 1'b1;
    end

    BCDcount count0 (clk100Hz, clear, LED, bcd2, bcd1, bcd0);

    seg7 dig0 (bcd0, digit0);
    seg7 dig1 (bcd1, digit1);
    seg7 dig2 (bcd2, digit2);

    lfsr(rand_num,clk100Hz);


endmodule

module BCDcount (clock, clear, enable, bcd2, bcd1, bcd0);

    // Inputs are clock signal (100Hz), clear, enable, reset.
    input clock, enable, clear;

    // Outputs are 3 4 bit registers to be passed to 7-seg
    // decoder.
    output reg [3:0] bcd2, bcd1, bcd0;

    // On the posedge of the 100 Hz signal:
    always @ (posedge clock)
          begin
          if (clear)
                      begin
                            // If (clear) set all digits to 0.
                            bcd2 <= 0;
                            bcd1 <= 0;
                            bcd0 <= 0;
                      end
                else if (enable)
                      begin
                            // Check if digit 0 (least sig.) is 9
                            if (bcd0 == 4'b1001)
                                  begin
                                        // If so, set it to 0, check if
digit 1 == 9
                                        bcd0 <= 0;
                                        if (bcd1 == 4'b1001)
                                              begin
```

```
                                                      // If so, set d1 to 0,
check digit 2 == 9
                                                      bcd1 <= 0;
                                                      if (bcd2 == 4'b1001)
                                                            // If so, set to
0 (this will restart whole count)
                                                            bcd2 <= 0;
                                                      else
                                                            // If not,
increment digit 2
                                                            bcd2 <= bcd2 +
1'b1;
                                                end
                                          else
                                                // If digit 1 is not 9,
increment it
                                                bcd1 <= bcd1 + 1'b1;
                                    end
                              else
                                    // If digit 0 is not 9, increment it
                                    bcd0 <= bcd0 + 1'b1;
                        end
            end
endmodule

module seg7 (bcd, leds);
      input [3:0] bcd;
      output reg [1:7] leds;

      always @(bcd)
            case (bcd) //abcdefg
                  0: leds = ~7'b1111110;
                  1: leds = ~7'b0110000;
                  2: leds = ~7'b1101101;
                  3: leds = ~7'b1111001;
                  4: leds = ~7'b0110011;
                  5: leds = ~7'b1011011;
                  6: leds = ~7'b1011111;
                  7: leds = ~7'b1110000;
                  8: leds = ~7'b1111111;
                  9: leds = ~7'b1111011;
            default: leds = 7'b0000000;
            endcase

endmodule

module lfsr (rand_num, clk);
      output reg [6:0] rand_num = {7{1'b1}}; // 8 bit output register
      // Initialize it to 1
      input clk; // inputs enable, any clock signal

      /*
      On the posedge of clk do the right shift (with XOR of 2 MSb's to go
into LSb).
```

```
      This will output pseudo-random 8-bit binary encoded number.
      */

      always @ (posedge clk)
            begin
                  rand_num <= {rand_num[6] ^ rand_num[5], rand_num[5],
rand_num[4], rand_num[3], rand_num[2], rand_num[1], rand_num[0]};
            end
endmodule
```