ECE 271, Dice Roller, Group 4

Casey Colley, Estuardo Ramos Alvarez, Rowan Lester, and Daniel Mendes

August 13th, 2021

Contents

1 Project Description

Inputs: Six switches determine which 7-segment displays are enabled. Two buttons are used: one acts as a reset for the stored/displayed value. The second button triggers the system to roll and display a new value.

Outputs: After rolling/resetting, he stored value is displayed in every enabled 7-segment display. When rolling, LEDs light up in the enabled 7-segment displays and above the switches.

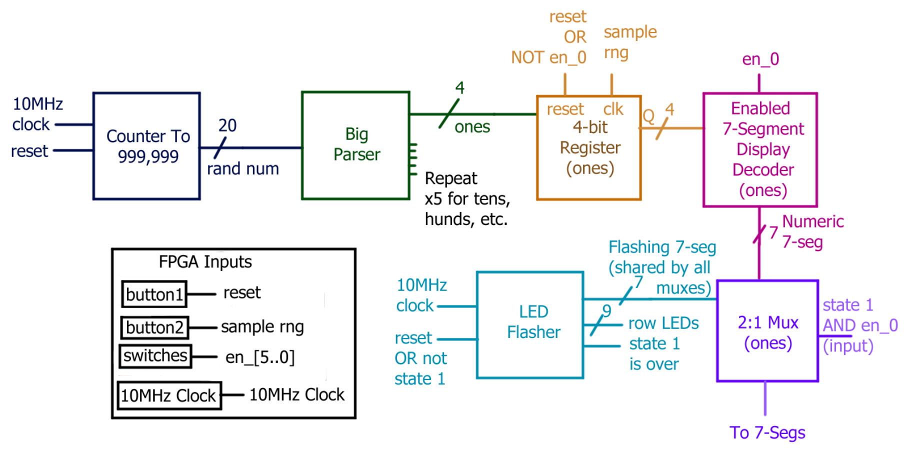


Figure 1: Top-level block diagram

The hardware diagram can be seen in figure 2. The entire project is confined to the FPGA itself, as there are no outside hardware elements required. While exact pin numbers are left out (it would be unfeasible to display all output pins in the diagram), the physical location of every input/output on the FPGA is mentioned.

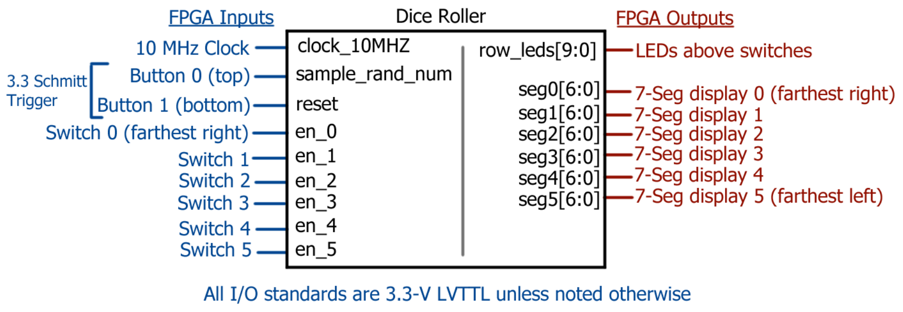

FPGA Inputs
Dice Roller
FPGA Outputs
10 MHz Clock
clock_10MHZ
row_leds[9:0]
LEDs above switches
3.3 Schmitt Trigger
Button 0 (top)
sample_rand_num
Button 1 (bottom)
reset
Switch 0 (farthest right)
en_0
Switch 1
en_1
Switch 2
en_2
Switch 3
en_3
Switch 4
en_4
Switch 5
en_5
seg0[6:0] 7-Seg display 0 (farthest right)
seg1[6:0] 7-Seg display 1
seg2[6:0] 7-Seg display 2
seg3[6:0] 7-Seg display 3
seg4[6:0] 7-Seg display 4
seg5[6:0] 7-Seg display 5 (farthest left)
All I/O standards are 3.3-V LVTTL unless noted otherwise

Figure 2: Hardware diagram.

The project was implemented on the FPGA as well. This is a video of the functionality: https://www.youtube.com/watch?v=bj3U9wBpOE8

2 High Level Description

Figure 1 contains the top-level block diagram. The "counter to 999999" block counts from 0 to 999,999, incrementing on every positive edge of a 10MHz clock. This number passes though a parser that splits it into 6 different signals: ones, tens, hundreds, thousands, ten-thousands, and hundred-thousands. Each of these signals are passed into a resettable register. All registers share a clock signal called "sample rng" that is tied to a button on the FPGA. Thus, a parsed, pseudo-random number is stored in the registers on every push of this button. This number then goes to the one of six enabled 7-segment display decoders (one for each digit). When the enable for a decoder is ON, it behaves as normal. When the enable is off, all display LEDs shut down. This enable is also tied to the reset of the resettable registers, forcing the registers to reset to 0 when the enable is off regardless of the reset input. There are 6 enables (one for each register/7-segment display), and they each are controlled by a switch on the FPGA. Lastly, a button, aptly named reset, is used to reset the initial counter and the register.

Another section of the design, called the "LED flasher", activates when the system enters "state 1". This module is used to flash LEDs for a certain amount of time after the "sample rng" button is pushed. Both the 7-segment displays and the LEDs above the FPGA's switches are made to flash. A multiplexer based on the current state is used to determine whether to output the signal from the flasher or from the register into each 7-segment display.

As previously mentioned, several elements of the system operate only in certain states. The state transition diagram in figure 3 shows when transitions need to occur for the system to correctly function. First, the system must reset to state 0. When the "sample rng" button is pushed, the system transitions to state 1 and begins to light up LEDs. After the state register in the flasher is full, state 1 ends. This resets the counter in the flasher and displays the stored numbers.

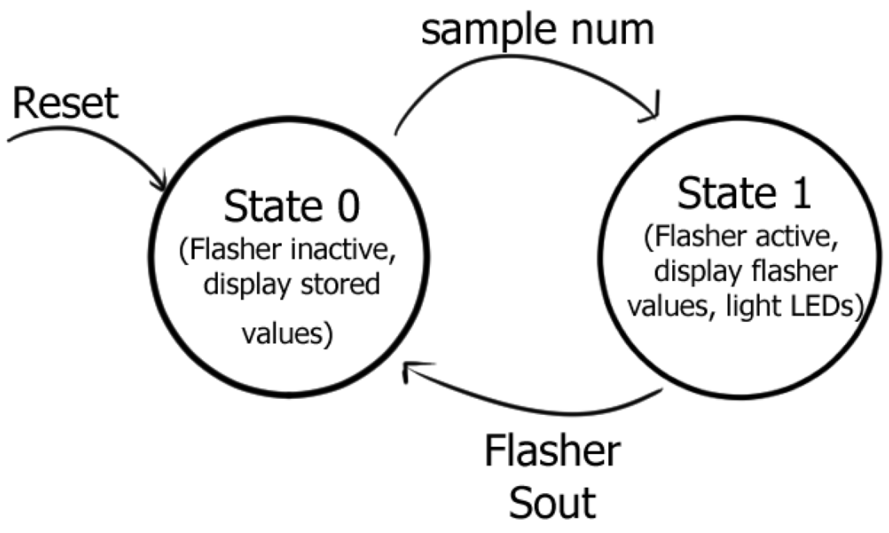

sample num
Reset
State 0
(Flasher inactive, display stored values)
State 1
(Flasher active, display flasher values, light LEDs)
Flasher Sout

Figure 3: State transition diagram

Figures 4 and 5 contain the simulation for the top-level file of the dice roller. Note that these simulations use a clock divider that is significantly smaller than in the final project. Also note that not all internal signals are shown, just the ones relevant to state transitions and the inputs/outputs. Figure 4 shows the transition from state 0 to state 1. This occurs when sample_rand_num is pushed. During state 1, the row LEDs light up one at a time while each of the enabled seven-segment displays displays the "loading" visual from the LED flasher module. Figure 5 shows the transition from state 1 to state 0, which occurs a delayed-clock cycle after every row LED has lit up. After the transition, the numbers from the shift registers are correctly displayed on the enabled 7-segment displays (disabled displays continue to stay off).

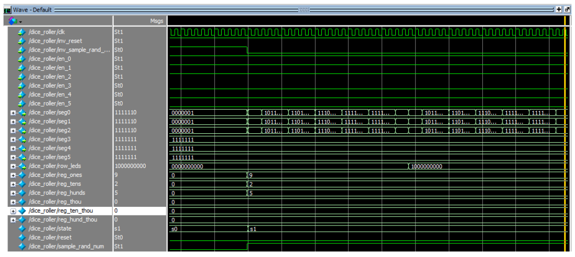


Figure 4: Simulation of the top-level file. Transition from state 0 to state 1.

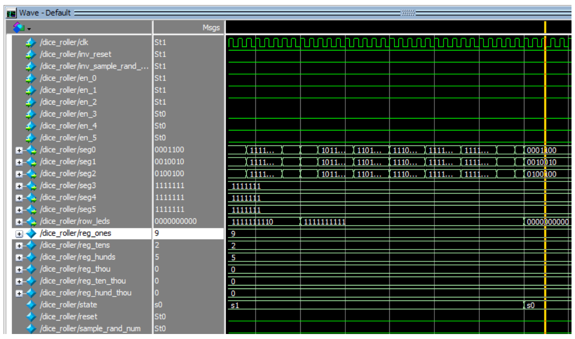


Figure 5: Simulation of the top-level file. Transition from state 1 to state 0

2.1 register

Figure 6 contains the block diagram for a register, a module included in the top-level design. The register merely holds the values that it is given and retains it until the next value is given. It does this in sync with the positive edge of the clock signal, and also reacts to an asynchronous reset signal.

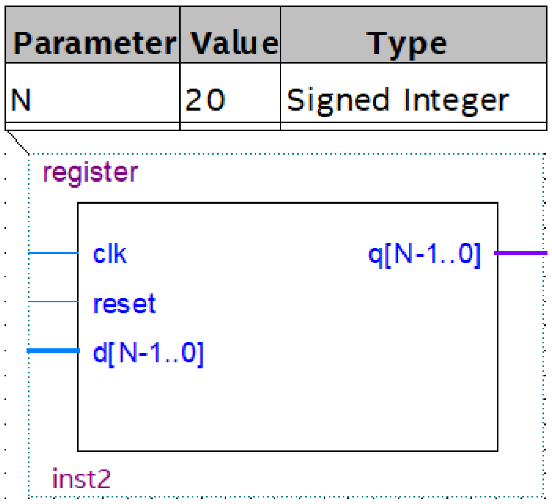


Figure 6: Block diagram of the register

Figure 7 contains the simulation for the register. In Figure 6, we see the register initially output 0 as a result of the reset signal. Then, we see the register start to output the values it is given, and finally 0 as the register is reset back to 0.

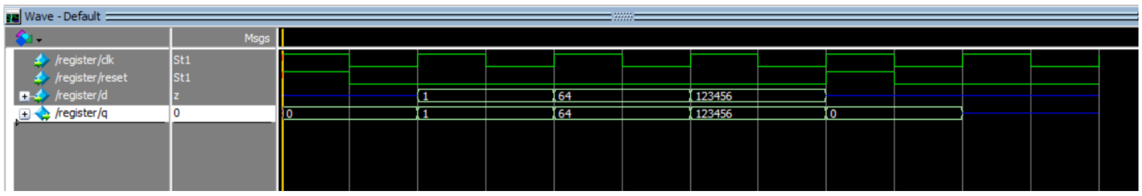


Figure 7: Simulation of the register

2.2 en_sevenseg

Figure 8 contains the block diagram for the enabled 7-segment display decoder included in the top level design. This module is similar to a standard 7-segment decoder, but with the added caveat that the decoders only output the number provided to it if the enable is on. Otherwise, it outputs a blank display (111_1111).

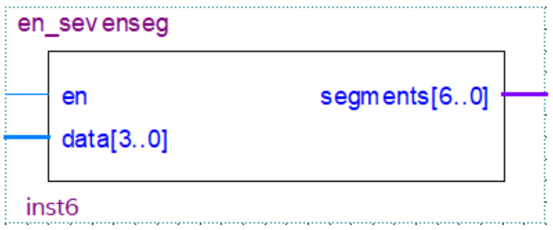


Figure 8: Block diagram of the enabled 7-segment display decoder

Figure 9 contains the simulation for the enabled 7-segment display decoder. Here we see that as the enable is turned on, the decoder translates the 4-bit numbers into their display. When it is turned off, the decoder ceases to output any of the data. There is a change in data to 0010 that is not displayed until enable turns on again.

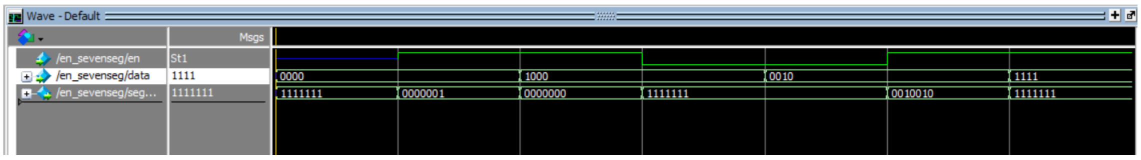


Figure 9: Simulation of the enabled 7-segment display decoder

2.3 mux2

Figure 10 contains the block diagram for the 2:1 multiplexer that is included in the top-level design. This multiplexer selects between two N-bit options based on the status of two enables (rather than one, as is standard). If both enables are on, then the mux chooses option a; otherwise, it chooses option b. In the project, these two variables are the corresponding 7-segment display's enable (en) and a boolean that is true when the system is in state 1.

Figure 11 contains the simulation for the 2:1 mux. Here we see the output change based on whether s, en, or both are on. When both are on, the output is a. When at least one is turned off, it reverts back to b.

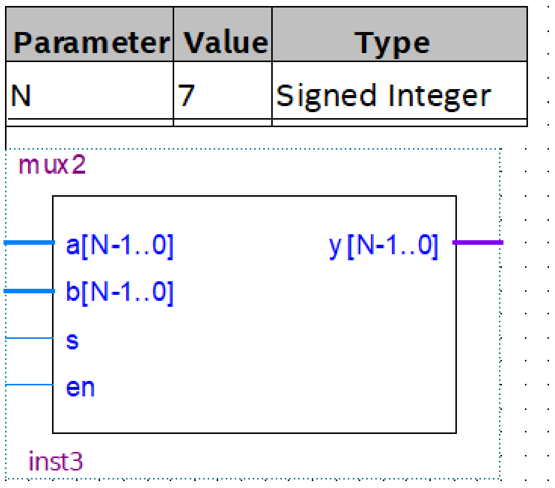


Figure 10: Block diagram of the 2:1 multiplexer

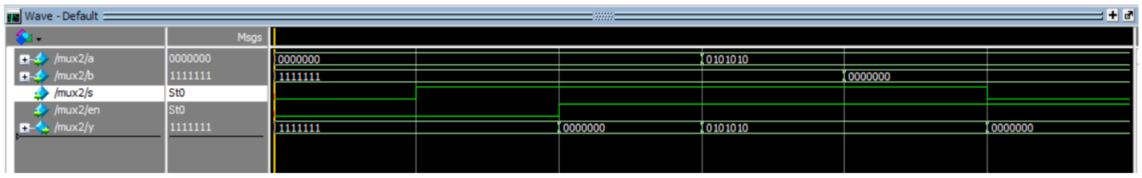


Figure 11: Simulation of the 2:1 multiplexer

2.4 big_parser

Figure 12 contains the block diagram for the big_parser module, which is included in the top-level design. This parser is a modified version of the parser from lab 5, so that it separates a number in the hundreds of thousands into its constituent digits.

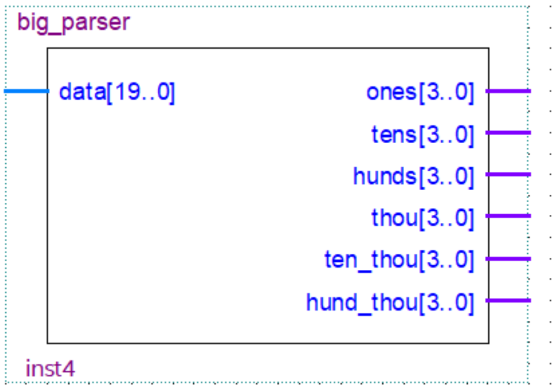


Figure 12: Block diagram of the big parser

Figure 13 contains the simulation for the big_parser module. Here we can see how as the digit 2 is moved around the various 10-places, the parser correctly isolates the digits from the input number.

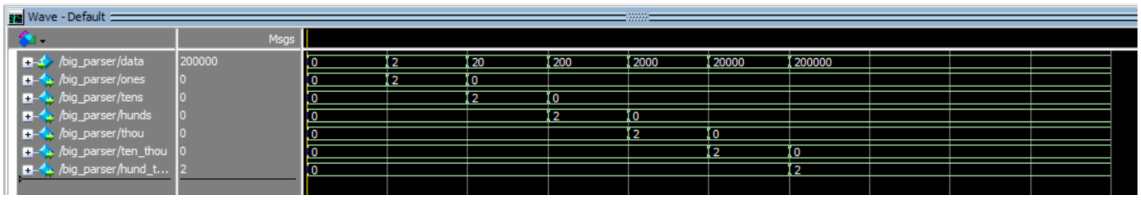
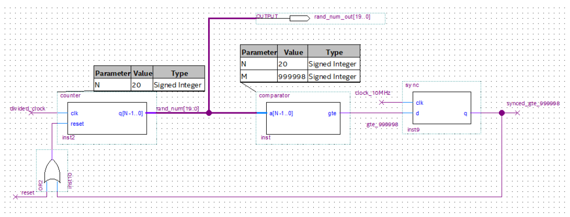


Figure 13: Simulation of the big_parser module

2.5 counter_to_999999

Figure 14 contains the block diagram for the counter_to_999999 module that is found in the top-level design. This module contains a counter, a comparator, and a synchronizer (each explained with greater detail below). This is the same design as the Clock from lab 5, with the exception that this module counts up from 0 to 999,999, resetting every time it hits 999,999. In the context of the project, this counter counter produces pseudo-random numbers by counting much faster than a human could reasonably "predict" and choose from the counter.



Parameter	Value	Type
N	20	Signed Integer

Parameter	Value	Type
N	20	Signed Integer
M	999998	Signed Integer

Figure 14: Block diagram of the "counter to 999999" block

Figure 15 contains the simulation for the "counter to 999999" block. Here we see the counter incrementing every clock cycle while the signal coming from the comparator is 0 (the counter is less than 999,999).

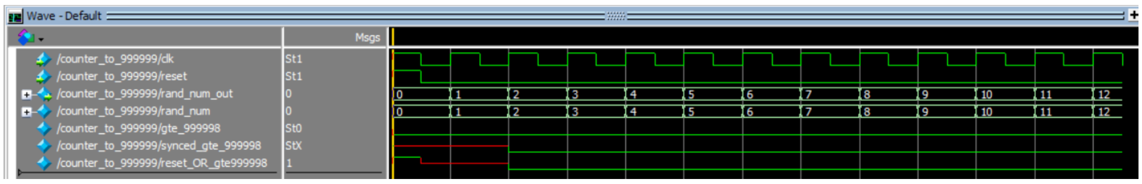
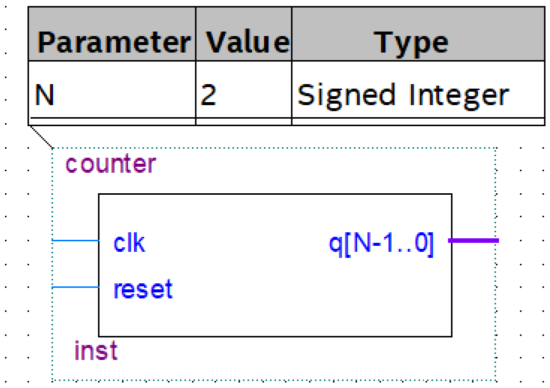

Wave - Default
Msgs
/counter_to_999999/clk St1
/counter_to_999999/reset St1
/counter_to_999999/rand_num_out 0
/counter_to_999999/rand_num 0
/counter_to_999999/gte_999998 St0
/counter_to_999999/synced_gte_999998 StX
/counter_to_999999/reset_OR_gte999998 1
0 1 2 3 4 5 6 7 8 9 10 11 12

Figure 15: Simulation of the "counter to 999999" block

2.5.1 counter

Figure 16 contains the block diagram for the counter module. This is the same counter that was synthesized in lab 5. A counter is a fundamental building block that is classified as a state machine. Generally counters take in a pulse from something like a clock and increment the state by one on the positive edge.

Figure 17 contains the simulation for the counter. Here we see the counter count up to 2-bits before restarting, incrementing every clock edge. This simulation of the counter is in 2-bits as that is the default parameter number of bits, however it is used in the project with many more bits.



Parameter	Value	Type
N	2	Signed Integer

Figure 16: Block diagram of the counter

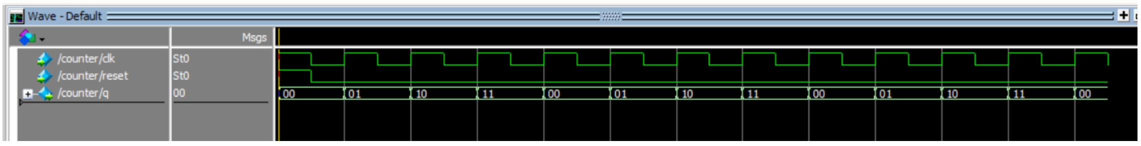
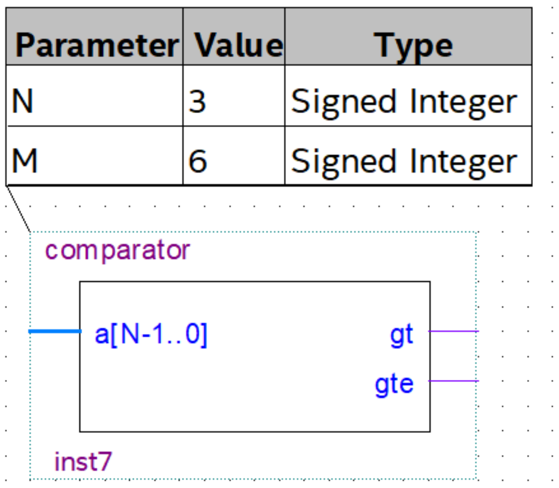


Figure 17: Simulation of the counter block

2.5.2 comparator

Figure 18 contains the block diagram for the comparator. This is almost the same design as the comparator from lab 5, except instead of comparing input a with input b, the module compares input a to a parameter value M. By default, this parameter is set to 800. When input a is greater than or equal to M (800), the module outputs 1 appropriately; otherwise it outputs 0.



Parameter	Value	Type
N	3	Signed Integer
M	6	Signed Integer

Figure 18: Block diagram of the comparator

Figure 19 contains the simulation for the comparator. Here we can see that a can be all the way up to value 799 without triggering the output. At 800, the signal GTE is driven high, and at 801, the signal GT is also driven high.

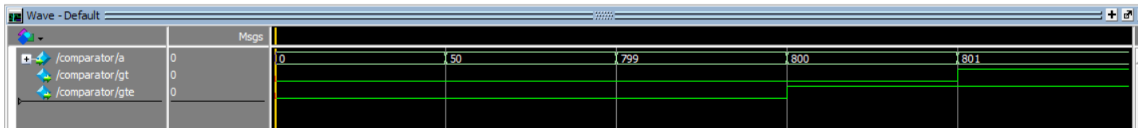


Figure 19: Simulation of the comparator

2.5.3 sync

Figure 20 contains the block diagram for the synchronizer. A synchronizer is a device that converts an asynchronous signal into a synchronous signal. Synchronizers are often placed in-between two sequential logic blocks to ensure that any one block does not fall out of step with the clock. Synchronizers can allow for parallel logic by breaking the machine into multiple segments that can move asynchronously until ready to be synchronized back into the clock.

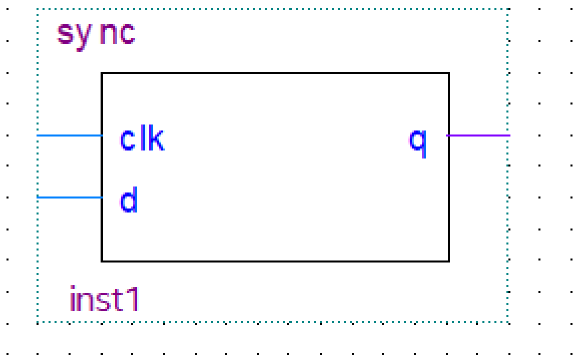


Figure 20: Block diagram of the synchronizer

Figure 21 contains the simulation for the synchronizer. Here we set d to a cycle asynchronous to the main clock. Each clock edge after the change in input, the input is transmitted as output. This ensures that the value keeps in time with the clock.

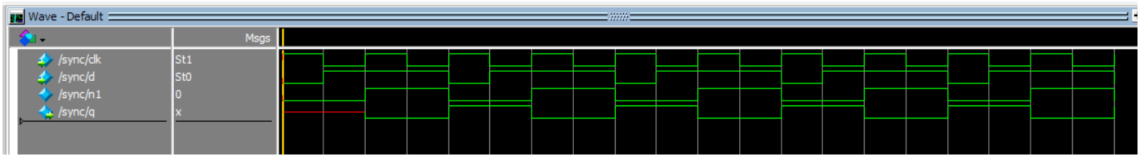


Figure 21: simulation of the synchronizer

2.6 led_flasher

This module contains a counter/comparator/synchronizer (as constructed in lab 5) and constantly counts to 6 before resetting. The counter uses a clock divider so that each delay is sufficiently long to visually identify on the FPGA (divided by 2^{20}). The value from the counter is fed into a 7-segment display decoder that outputs a specific coding for every number 0-6. When these values are displayed on a 7-segment display, they appear to spin in a circle. Every time the counter resets, an 11-bit shift register (that starts entirely full of 0s) adds a 1 to itself. This shift register contains the state of the LEDs above the switches, and thus will light up an additional LED every time the counter resets. When the shift register is full (I.E. when Sout is 1), state 1 ends and the system transitions back into state 0. The block diagram for this module is shown in Figure 22.

Figure 23 contains the simulation for the flasher module. Here we see a counter incrementing up to 6 before restarting at 0. A large clock counter (dividedClock) increments to 2^{20} before triggering

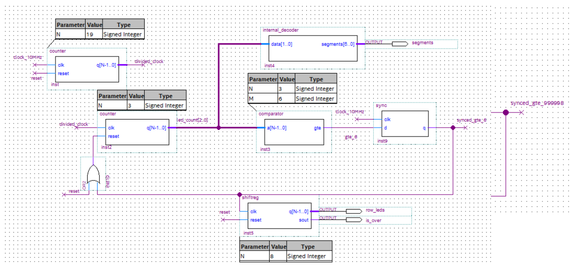


Figure 22: Block diagram of the flasher module

an increment for the flasher counter. In the simulation, these values are artificially forced into the counter. Once the flasher counter resets back to 0, we see counter row_led_status increment by one. Additionally, as the flasher counter changes, the 7-segment output changes as well so that the display appears to show a spinning circle.

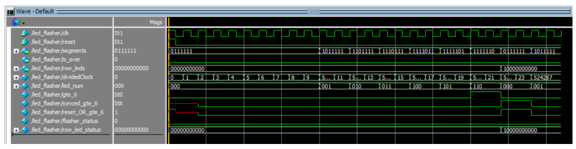


Figure 23: Simulation of the flasher module

2.6.1 counter

See the subsections under counter_to_999999.

2.6.2 comparator

See the subsections under counter_to_999999.

2.6.3 sync

See the subsections under counter_to_999999.

2.6.4 shiftreg

This shift register is a slightly modified version of the shift register from the textbook, and its block diagram is shown in Figure 24. The shiftreg stores a 1 to its most significant bit on every rising edge of the clock. The least significant bit is output. When this value becomes 1, it triggers the transition from state 1 to state 0 at the top-level.

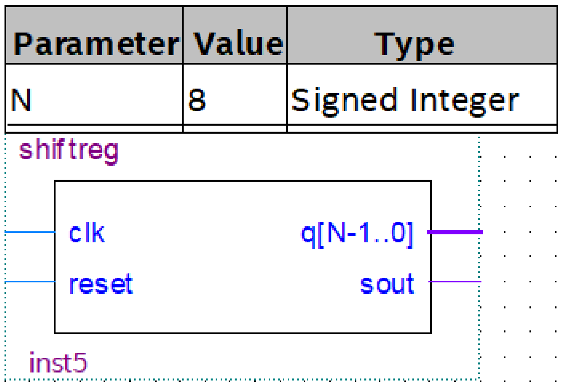


Figure 24: Block diagram of the shift register

Figure 25 contains the simulation for the synchronizer. Here we see the shift register push a 1 for every positive edge of the clock, until it is entirely full. When the shift register is full, the shift register outputs a 1 to show this and the shift register is reset.

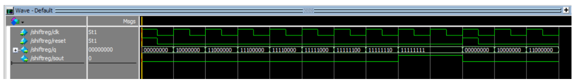


Figure 25: Simulation of the shift register

A SystemVerilog Files

```
/*
Module Name: dice_roller
Description: Top level document for the project.
Flashes LEDs on the press of a button for a certain amount of time, then displays numbers on 7-segment displays.
LEDs are controlled by switches, and numbers will only be displayed on enabled segments.
*/
module dice_roller (input logic clk, inv_reset, inv_sample_rand_num,
                    input logic en_0, en_1, en_2, en_3, en_4, en_5,
                    output logic [6:0] seg0, seg1, seg2, seg3, seg4, seg5,
                    output logic [9:0] row_leds);

    //Inverting inputs from active low buttons
    wire reset; //Resets state to S0
    assign reset = !(inv_reset);
    wire sample_rand_num; //This is the trigger for transitioning from state 0 to state 1
    assign sample_rand_num = !(inv_sample_rand_num);

    //Creating reset/enable signals for the registers (so they only store numbers when enabled AND still are able to reset on button push)
    wire reset_OR_not_en0, reset_OR_not_en1, reset_OR_not_en2, reset_OR_not_en3, reset_OR_not_en4, reset_OR_not_en5;
    assign reset_OR_not_en0 = !(en_0) | reset;
    assign reset_OR_not_en1 = !(en_1) | reset;
    assign reset_OR_not_en2 = !(en_2) | reset;
    assign reset_OR_not_en3 = !(en_3) | reset;
    assign reset_OR_not_en4 = !(en_4) | reset;
    assign reset_OR_not_en5 = !(en_5) | reset;

    //Creating signals based on the states
    //      reset_OR_not_s1 is used to keep the LED Flasher disabled during state 0 AND allow the flasher to reset on button push
    logic reset_OR_not_s1;
    assign reset_OR_not_s1 = reset | !(state == s1);
    logic is_state_s1;
    assign is_state_s1 = (state == s1);


    //Signal that takes output from the counter_to_999999; Goes to the parser
    wire [19:0] pseudo_rand_num;

    //Wires that take the parsed number to the different modules. One for each digit
    wire [3:0] rand_ones, rand_tens, rand_hunds, rand_thou, rand_ten_thou, rand_hund_thou;
    wire [3:0] reg_ones, reg_tens, reg_hunds, reg_thou, reg_ten_thou, reg_hund_thou;
    wire [6:0] seg_ones, seg_tens, seg_hunds, seg_thou, seg_ten_thou, seg_hund_thou;

    //Wires that control the numeric output to the 7-segment displays. Feeds into the Mux
    wire [6:0] ones_out, tens_out, hunds_out, thou_out, ten_thou_out, hund_thou_out;

    //Creating signals that are controlled by outputs from the LED flasher
    wire [6:0] flasher_seg_leds;
    wire [10:0] flasher_row_leds;
    wire flasher_is_over; //This is the trigger for transitioning from S1 to S0


    //From the textbook, creates a type for states
    typedef enum logic {s0, s1} statetype;
    statetype state, nextstate;

    //State register
    always_ff @(posedge clk, posedge reset)
        if (reset)
            state <= s0;
        else
            state <= nextstate;

    //Next-state logic as outlined in the state transition diagram. Starts in and resets to S0.
    always_comb
        case (state)
            s0: if (sample_rand_num) nextstate = s1;
                else nextstate = s0;
            s1: if (flasher_is_over) nextstate = s0;
                else nextstate = s1;
            default: nextstate = s0;
        endcase



    //This flashes the LEDs when state 1 is entered,
    //      Outside of state 1 this is constantly being reset with the signal "reset_OR_not_s1"
    led_flasher flasher (.clk(clk), .reset(reset_OR_not_s1), .segments(flasher_seg_leds), .is_over(flasher_is_over), .row_leds(flasher_row_leds));

    //This generates the pseudo-random number through counting
    counter_to_999999 rng (.clk(clk), .reset(reset), .rand_num_out(pseudo_rand_num));

    //Parses the number from the counter_to_999999 into 6 separate digits
```

```
        big_parser num_parser (.data(pseudo_rand_num), .ones(rand_ones), .tens(rand_tens), .hunds(rand_hunds), .thou(rand_thou), .ten_thou(rand_ten_thou), .hund_thou(rand_hund_thou));

        //Stores the parsed numbers into 6 registers. Clock is tied to sample_rand_num, a button,
        register #(.N(4)) ones_reg (.clk(sample_rand_num), .reset(reset_OR_not_en0), .d(rand_ones), .q(reg_ones));
        register #(.N(4)) tens_reg (.clk(sample_rand_num), .reset(reset_OR_not_en1), .d(rand_tens), .q(reg_tens));
        register #(.N(4)) hunds_reg (.clk(sample_rand_num), .reset(reset_OR_not_en2), .d(rand_hunds), .q(reg_hunds));
        register #(.N(4)) thou_reg (.clk(sample_rand_num), .reset(reset_OR_not_en3), .d(rand_thou), .q(reg_thou));
        register #(.N(4)) ten_thou_reg (.clk(sample_rand_num), .reset(reset_OR_not_en4), .d(rand_ten_thou), .q(reg_ten_thou));
        register #(.N(4)) hund_thou_reg (.clk(sample_rand_num), .reset(reset_OR_not_en5), .d(rand_hund_thou), .q(reg_hund_thou));

        //Decodes the number inside the register when enabled. Otherwise sends a signal that turns all LEDs off.
        en_sevenseg ones_seg (.data(reg_ones), .segments(seg_ones), .en(en_0));
        en_sevenseg tens_seg (.data(reg_tens), .segments(seg_tens), .en(en_1));
        en_sevenseg hunds_seg (.data(reg_hunds), .segments(seg_hunds), .en(en_2));
        en_sevenseg thou_seg (.data(reg_thou), .segments(seg_thou), .en(en_3));
        en_sevenseg ten_thou_seg (.data(reg_ten_thou), .segments(seg_ten_thou), .en(en_4));
        en_sevenseg hund_thou_seg (.data(reg_hund_thou), .segments(seg_hund_thou), .en(en_5));


        //Set of multiplexers
        //      Selects the flasher outputs when the segment is enabled and it is currently state 1
        //      Otherwise, select the decoded outputs from the register.
        mux2 #(.N(7)) seg_0_mux (.a(flasher_seg_leds), .b(seg_ones), .en(en_0), .s(is_state_s1), .y(ones_out));
        mux2 #(.N(7)) seg_1_mux (.a(flasher_seg_leds), .b(seg_tens), .en(en_1), .s(is_state_s1), .y(tens_out));
        mux2 #(.N(7)) seg_2_mux (.a(flasher_seg_leds), .b(seg_hunds), .en(en_2), .s(is_state_s1), .y(hunds_out));
        mux2 #(.N(7)) seg_3_mux (.a(flasher_seg_leds), .b(seg_thou), .en(en_3), .s(is_state_s1), .y(thou_out));
        mux2 #(.N(7)) seg_4_mux (.a(flasher_seg_leds), .b(seg_ten_thou), .en(en_4), .s(is_state_s1), .y(ten_thou_out));
        mux2 #(.N(7)) seg_5_mux (.a(flasher_seg_leds), .b(seg_hund_thou), .en(en_5), .s(is_state_s1), .y(hund_thou_out));

        //7-segment display outputs. From Muxes
        assign seg0 = ones_out;
        assign seg1 = tens_out;
        assign seg2 = hunds_out;
        assign seg3 = thou_out;
        assign seg4 = ten_thou_out;
        assign seg5 = hund_thou_out;

        //LEDs above the switches output. From LED flasher.
        //Uses only the 10 most significant bits, as the last bit is exclusively used to change states
        // This lets all 10 LEDs light up rather than just the initial 9.
        assign row_leds = flasher_row_leds[10:1];



endmodule
```

A.1 register

```
/*
Module Name: register
Description: A simple register.
Stores an N-bit value (d) on the rising edge of a clock. Then, outputs that value to an N-bit output (q).
Resets on the rising edge of the reset input signal.
*/
module register #(parameter N = 20) (input logic clk, reset, input logic [N-1:0] d, output logic [N-1:0] q);

        always_ff @(posedge clk, posedge reset)
                if (reset)
                        q <= 0;
                else
                        q <= d;

endmodule
```

A.2 en_sevenseg

```
/*
Module Name: en_sevenseg
Description: A slightly modified version of the 7-segment display decoder from lab 3.

When the enable (en) is true, the display acts as expected (for decimal values only).
When the enable is false, then a value that would disable all LEDs on the display is output.
*/
module en_sevenseg(input logic en,
                                                input logic [3:0] data,
                                                output logic[6:0] segments);

                always_comb begin
                                if (en)
                                        case(data)
                                                // segments      abc_defg
                                                0: segments = 7'b000_0001;
                                                1: segments = 7'b100_1111;
                                                2: segments = 7'b001_0010;
                                                3: segments = 7'b000_0110;
                                                4: segments = 7'b100_1100;
                                                5: segments = 7'b010_0100;
                                                6: segments = 7'b010_0000;
                                                7: segments = 7'b000_1111;
                                                8: segments = 7'b000_0000;
                                                9: segments = 7'b000_1100;
                                                default: segments = 7'b111_1111; //blank for any unexpected value
                                        endcase
                                else
```

```
                                        segments = 7'b111_1111; //blank when enable is false
                end
endmodule
```

A.3 mux2

```
/*
Module Name: mux2
Description: Selects between one of two N-bit options.

Selection is based on two selection variables (s and en) AND gated together.
        In the project, these two variables are the corresponding 7-segment display's enable (en) and a boolean that is true when the system is in state 1 (is_state_s1)
*/
module mux2 #(parameter N = 7) (input logic [N-1:0] a, b, input logic s, en, output logic [N-1:0] y);
        assign y = (s & en) ? a : b;
endmodule
```

A.4 big_parser

```
/*
Module Name: big_parser
Description: A modified version of the parser from lab 5.
Parses a 20-bit values (in the project this value is <=999,999) into 6 digits.
*/module big_parser (input logic [19:0] data,
                                            output logic [3:0] ones, tens, hunds, thou, ten_thou, hund_thou);

                assign ones = data % 10;
                assign tens = (data / 10) % 10;
                assign hunds = (data / 100) % 10;
                assign thou = (data / 1000) % 10;
                assign ten_thou = (data / 10000) % 10;
                assign hund_thou = (data / 100000) % 10;
endmodule
```

A.5 counter_to_999999

```
/*
Module Name: counter_to_999999
Description: Contains a counter/comparator/syncronizer setup.
The module counts up from from 0 to 999,999, resetting every time it hits 999,999.
The current output from the counter is output outside the module.
*/
module counter_to_999999 (input logic clk, reset, output logic [19:0] rand_num_out);

                //Contains the output value
                wire [19:0] rand_num;

                //Used for resetting
                wire gte_999998;
                wire synced_gte_999998;
                logic reset_OR_gte999998;


                //20 bit counter
                counter #(.N(20)) rand_num_clock (.clk(clk), .reset(reset_OR_gte999998), .q(rand_num));

                //Comparator; Sends out a pulse if input is 999,999
                comparator #(.N(20), .M(999998)) reset_if_gte666666 (.a(rand_num), .gte(gte_999998));

                //Syncronizer used to align the reset pulse with the clock
                sync compSync (.clk(clk),
                                                    .d(gte_999998),
                                                    .q(synced_gte_999998));


                //Reset pulse
                assign reset_OR_gte999998 = synced_gte_999998 | reset; //Based on the active HIGH reset that is in counter.sv


                assign rand_num_out = rand_num;

endmodule
```

A.5.1 counter

```
/*
Module Name: counter
Description: Counts up to a maximum of an N-bit value on every positive edge of the clock signal. Resets on the positive edge of the reset signal.
*/
module counter #(parameter N = 2)
                                            (input logic clk,
                                             input logic reset,
                                             output logic [N-1:0] q);

                always_ff@(posedge clk, posedge reset)
                        if (reset)
                                q <= 0;
                        else
                                q<= q + 1;
endmodule
```

A.5.2 comparator

```
/*
Module Name: comparator
Description: A simple comparator. Compares an N-bit input to the parameter M.
For this project, only GT and GTE as comparisons are used.
*/
module comparator #(parameter N = 10, M = 800)
```

```
                                                            (input logic [N-1:0] a,
                                                             output logic gt, gte);

                assign gt = (a > M);
                assign gte = (a >= M);
endmodule
```

A.5.3 sync

```
/*
Module Name: sync
Description: A simple syncronizer.
Assigns a 1-bit internal signal (n1) to a 1-bit input (d) on the positive edge of the clock signal.
Then, assigns the 1-bit output (z) to n1 on the next positive edge of the clock signal.
*/
module sync (input logic clk,
                                  input logic d,
                                  output logic q);

                logic n1;

                always_ff@(posedge clk)
                        begin
                                n1 <= d;
                                q <= n1;
                        end
endmodule
```

A.6 led_flasher

```
/*
Module Name: led_flasher
Description: Provides output that is used to light up LEDs when enabled (during top-level state 1).
This includes both the 7-segment displays and the LEDs above the switches.
*/
module led_flasher (input logic clk, reset, output logic [6:0] segments, output logic is_over, output logic [10:0] row_leds);

        //The 19th bit is used as the clock signal in the counter
        wire [18:0] dividedClock;

        //Stores the value from the counter (up to 6)
        // Used in the comparator to determine when to reset the counter.
        // Also used in the case statement below to create outputs for the 7-segment displays.
        wire [2:0] led_num;

        //Used to reset the counter
        wire gte_6;
        wire synced_gte_6;
        logic reset_OR_gte_6;

        //Used to keep track of the "sout" value in the shift register
        //      This value is assigned to the output "is_over" which is used to signal the end of state 1.
        // When state 1 ends, this module is kept constantly reset.
        logic flasher_status;

        //Used as a temporary variable to store output from the shift register.
        //
        logic [10:0] row_led_status;

        //Clock divider. Essentially the same as the clock divider from lab 5.
        counter #(.N(19)) clkdivider (.clk(clk), .reset(reset), .q(dividedClock));

        //Counter that counts up to 6. Uses the divided clock signal.
        counter #(.N(3)) led_count (.clk(dividedClock[18]), .reset(reset_OR_gte_6), .q(led_num));

        //Comparator. Used to send a signal when the value in led_count becomes greater than 6.
        comparator #(.N(3), .M(6)) reset_if_gte6 (.a(led_num), .gte(gte_6));

        //Syncronizer, used as in lab 5
        sync compSync (.clk(clk),
                                              .d(gte_6),
                                              .q(synced_gte_6));

        //For reset signal for led_count
        assign reset_OR_gte_6 = reset | synced_gte_6;

        //Case statement that outputs a specific combination of LEDs on each display
        //      This gives the appearance of "loading" when led_count counts as the LEDs on each display spin in a circle
        always_comb begin
                case (led_num)
                        0: segments = 7'b011_1111;
                        1: segments = 7'b101_1111;
                        2: segments = 7'b110_1111;
                        3: segments = 7'b111_0111;
                        4: segments = 7'b111_1011;
                        5: segments = 7'b111_1101;
                        6: segments = 7'b111_1110;
                        default: segments = 7'b111_1111;
                endcase
        end

        //Shift register that keeps track of the status of LEDs above the switches
        // Shifts a 1 into itself whenever led_count resets
        // This, whenever a cycle is complete, one more LED lights up. This repeats 10 times to light up every LED.
        // When 11 cycles are complete, sout becomes 1. This then makes the is_over output 1, ending state 1 in the top-level document.
        shiftreg #(.N(11)) row_led_reg (.clk(synced_gte_6), .reset(reset), .q(row_led_status), .sout(flasher_status));


        assign is_over = flasher_status;
        assign row_leds = row_led_status;

endmodule
```

A.6.1 counter

See the subsections under counter_to_999999.

A.6.2 comparator

See the subsections under counter_to_999999.

A.6.3 sync

See the subsections under counter_to_999999.

A.6.4 shiftreg

```
/*
Module Name: shiftreg
Description: A slightly modified version of the shift register from the textbook.
The register stores a 1 to its most significant bit on every rising edge of the clock.
The least significant bit is output. When this value becomes 1, it triggers the change from state 1 to
      state 0 at the top-level.
*/
module shiftreg #(parameter N = 8) (input logic clk, reset, output logic [N-1:0] q, output logic sout);
          always_ff@(posedge clk, posedge reset)
          if (reset)
                              q <= 0;
                    else
                              q <= {1'b1, q[N-1:1]};
          assign sout = q[0];
endmodule
```

B Simulation Files (Do scripts)

B.1 dice_roller (top level)

```
add wave *
radix signal reg_ones -unsigned
radix signal reg_tens -unsigned
radix signal reg_hunds -unsigned
radix signal reg_thou -unsigned
radix signal reg_ten_thou -unsigned
radix signal reg_hund_thou -unsigned

force en_0 1
force en_1 1
force en_2 1
force en_3 0
force en_4 0
force en_5 0

force -freeze sim:/dice_roller/clk 1 0, 0 {5 ps} -r 10
force /dice_roller/inv_reset 0 , 1 @ 5
force /dice_roller/inv_sample_rand_num 1 , 0 @ 5298, 1 @ 5400

run 8000
```

B.2 register

```
add wave *
radix signal d -unsigned
radix signal q -unsigned

force clk 1 0, 0 1 -r 2
force reset 1 0, 0 1

force d 10#1 2
force d 10#64 4
force d 10#123456 6

run 8

noforce d
force reset 1 0, 0 1

run 4
```

B.3 en_sevenseg

```
add wave *

force data 0000 0
force en 1 1
force data 1000 2
force en 0 3
force data 0010 4
force en 1 5

force data 1111 6

run 7
```

B.4 mux2

```
add wave *

force a 0000000 0
force b 1111111 0
force s 0 0
force en 0 0

force s 1 1
force en 1 2
force a 0101010 3
force b 0000000 4
force s 0 5

run 6
```

B.5 big_parser

```
add wave *
radix -unsigned

force data 0 0
force data 2 1
force data 20 2
force data 200 3
force data 2000 4
force data 20000 5
force data 200000 6

run 10
```

B.6 counter_to_999999

```
add wave *
radix signal counter_to_999999/rand_num -unsigned
radix signal counter_to_999999/rand_num_out -unsigned

force clk 1 0, 0 1 -r 2
force reset 1 0, 0 1

run 25
```

B.6.1 counter

```
add wave *

force clk 1 0, 0 1 -r 2
force reset 1 0, 0 1

run 25
```

B.6.2 comparator

```
add wave *
radix signal a -unsigned

force a 10#0 0
force a 10#50 1
force a 10#799 2
force a 10#800 3
force a 10#801 4

run 5
```

B.6.3 sync

```
add wave *

force clk 1 0, 0 1 -r 2
force d 0 0, 1 1 -r 4

run 20
```

B.7 led_flasher

```
add wave *
radix signal dividedClock -unsigned

force clk 1 0, 0 1 -r 2
force reset 1 0, 0 1
run 20

force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
```

```
force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
force dividedClock 1111111111111111111
run 2
noforce dividedClock
run 2
force dividedClock 1111111111111111111
run 4
```

B.7.1 counter

See the subsections under counter_to_999999.

B.7.2 comparator

See the subsections under counter_to_999999.

B.7.3 sync

See the subsections under counter_to_999999.

B.7.4 shiftreg

```
add wave *

force clk 1 0, 0 1 -r 2
force reset 1 0, 0 1

run 20


force reset 1 0, 0 1
run 6
```