ECE 271, Design Project, Group 3

Hyunjae Kim, David Korotky, Elayne Trimble

August 13th, 2021

Contents

1 Project Description

1.1 Introduction

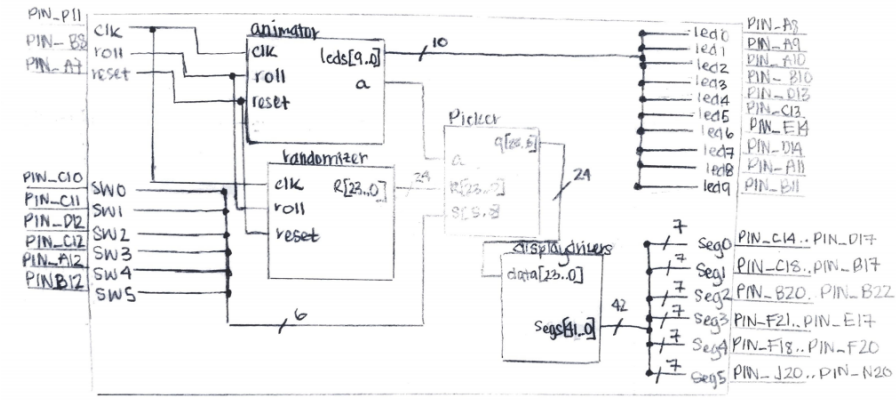

Figure 1: Top-level block diagram.

The purpose of this project was to create a dice machine which, when activated, would light up ten LEDs in sequential order on the FPGA, and only then would display a requested amount of random dice rolls on the FPGA seven segment display according to a series of six switches.

This machine takes in several inputs. A 10MHz clock is required for the sequential logic to run. A press of the roll button is also required to start the machine, as well as a reset button which would set all the random values back to 0. Finally, six switch inputs are needed to pass the amount of dice to be rolled.

As you can see in the diagram above, multiple outputs are also required for simulation. Each LED on the FPGA requires an output to tell it whether or not to light up, these are labeled leds[0] to leds[9]. 42 assignments are required to match each segment on the seven segment decoder to a logic value.

1.2 Implementation

The link to a demonstration of the device: https://youtu.be/JSjCpp2NogQ.

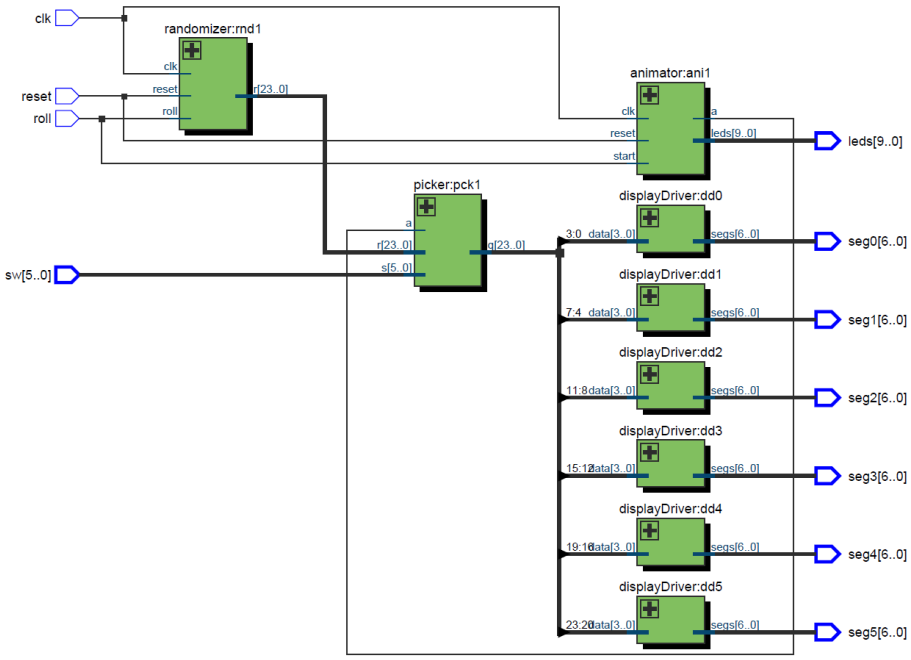

Figure 2: Final register-transfer level diagram generated by Quartus.

Flow Status	Successful - Fri Aug 13 23:02:29 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	diceMachine
Top-level Entity Name	diceMachine
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	1,965 / 49,760 (4 %)
Total registers	93
Total pins	61 / 360 (17 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 3: Compilation reported generated by Quartus.

2 High Level Description

2.1 Functional Unit: Top Level Dice Machine

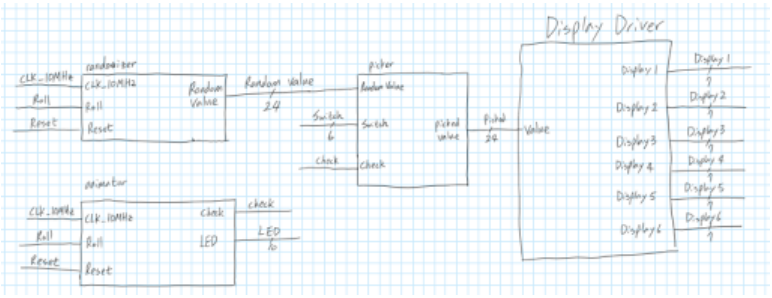

Figure 4: Above is the top level design for the dice machine. As described in the project description, we take in several inputs, including a 10MHz clock, roll, reset, and the six switches. The inputs are passed simultaneously to the randomizer and the animator. We assume that the animator takes many orders of magnitude longer to complete than the randomizer. When the animator finishes, it sends a signal enabling the random dice values to be shown. However, all six dice values were calculated almost instantaneously the moment the button was pressed. They are merely displayed at the end of the animation.

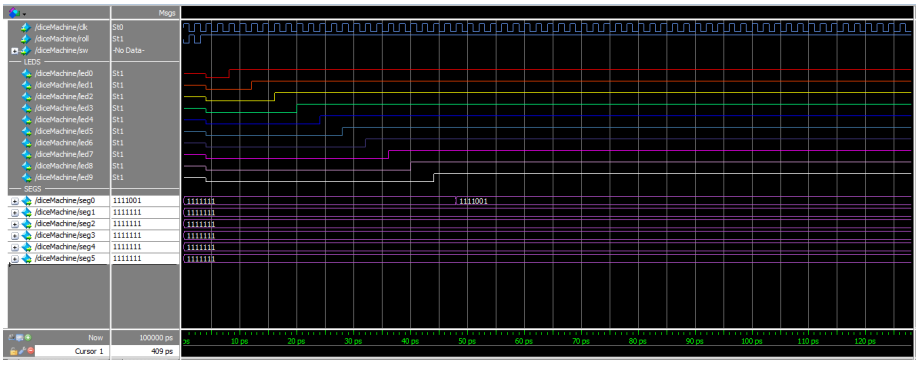

Figure 5: Top-level simulation.

2.1.1 Individual Block: Picker

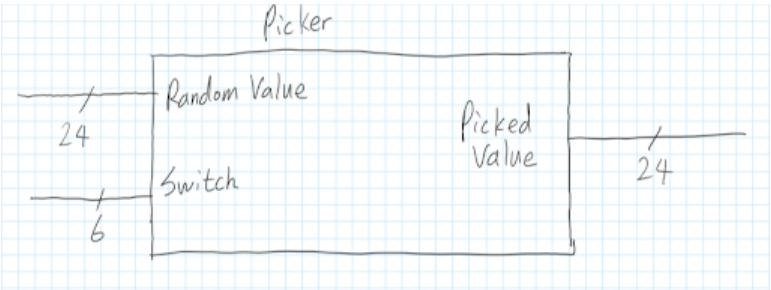

Figure 6: The picker selects 4-bit values among 24-bit input 'Random Value' depending on 6-bit input 'switch', and un-selected 4-bit values are encoded into 0000. After the selection and encoding process, the changed 24-bit value named 'Picked Value' is printed out.

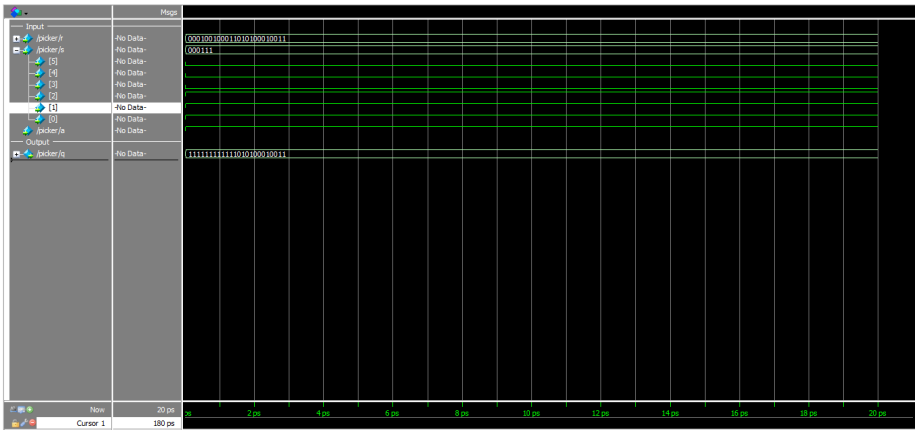

Figure 7: Picker simulation.

2.1.2 Individual Block: Display Driver

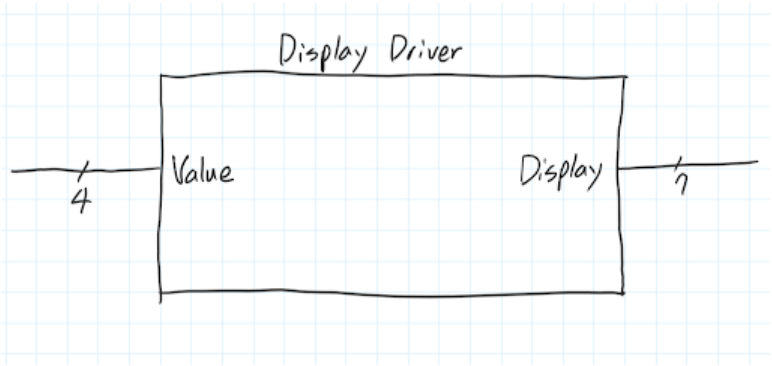

Figure 8: The Display Driver converts 4-bit input value to 7-bit display output to demonstrate the input value on a seven segment display. By using K-map and SystemVerilog, 7-bit output 'Display' prints out a 7-bit binary which enables seven segment displays to show 1-6.

Figure 9: Display driver simulation.

2.2 Functional Unit: Randomizer

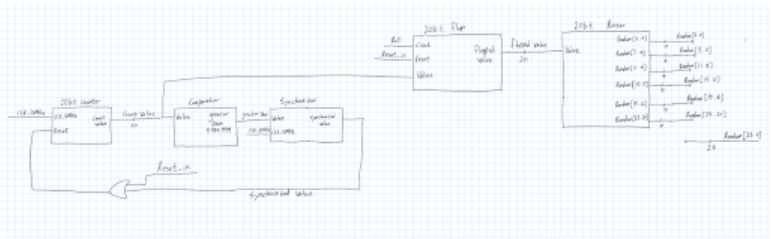

Figure 10: The randomizer is the block in charge of generating random values. It's input values are clock, roll, and reset. The source of the randomness comes from the clock itself, which begins to run as soon as the device is initialized. The clock cycles are counted up to 999999 which provides six decimal digits of information to work with. Because the count resets immediately after all digits 1-9 have been exhausted, the subsequently parsed result is *completely* random since the result depends only on when within the extremely fast clock cycle the button is pressed. There is no way for a human being to consistently press the button within 1000000 clock cycles. Afterwards, the result is passed back into our top level file to be parsed.

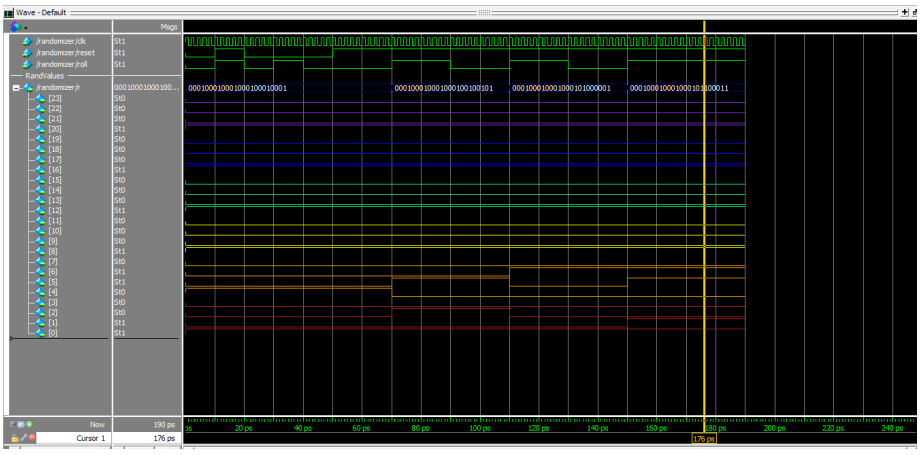

Figure 11: Randomizer simulation.

2.2.1 Individual Block: Counter

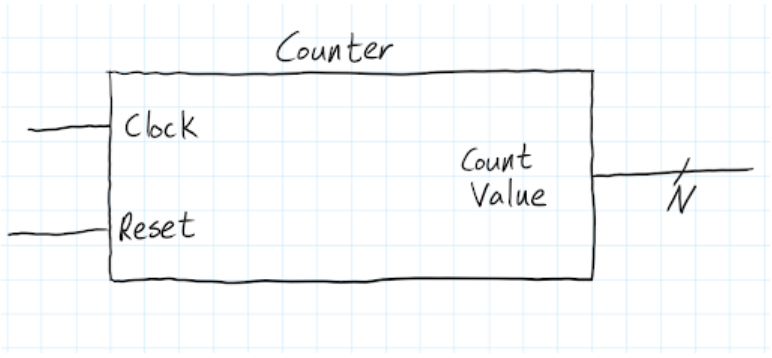

Figure 12: When the clock is rising, the counter adds 1 to the output 'Count Value'. Unless the reset is turned on, the counter will add 1 to the output 'Count Value' until it reaches $2^N - 1$. When the reset is turned on, the counter resets the output q to be 0.

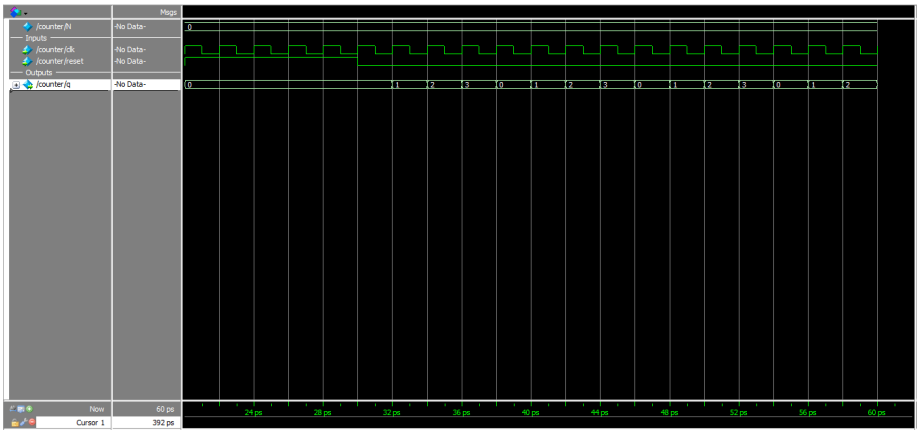

Figure 13: Counter simulation.

2.2.2 Individual Block: Flopr

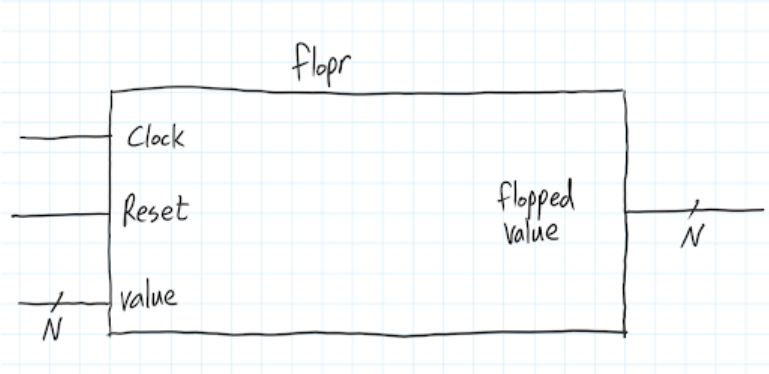

Figure 14: Flopr is N-bit re-settable asynchronous D Flip-Flop. When the clock input is rising edge, N-bit input 'value' passes through to N-bit output 'flopped value', otherwise, 'flopped value' holds previous value. When the Reset is turn on (is 1), the N-bit value 'flopped value' is reset-ted.

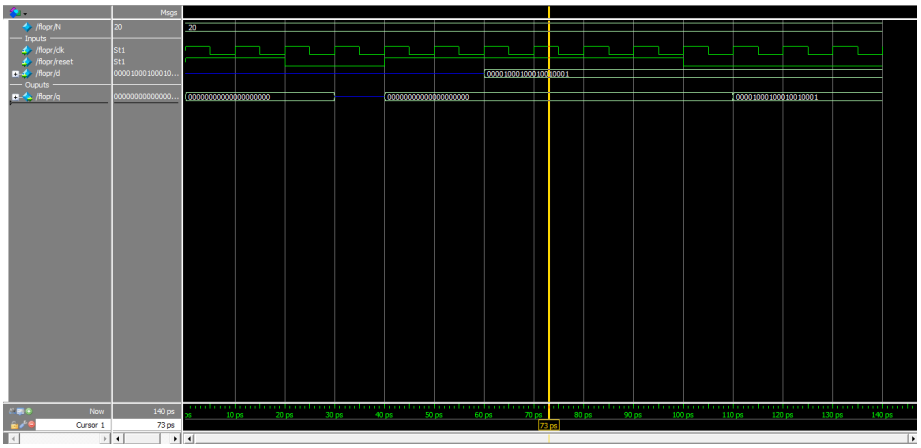

Figure 15: Register simulation.

2.2.3 Individual Block: Parser

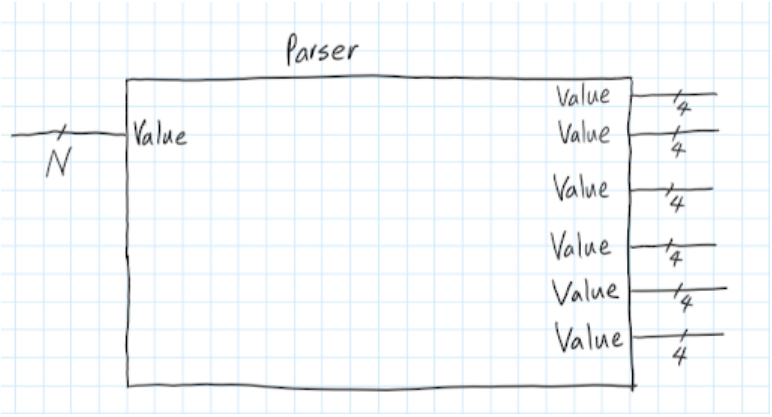

Figure 16: Parser splits N-bit input values to six of 4-bit output values. Each of 4-bit outputs represents each digit of the N-bit input value in decimals. After splitting N-bit input values, Parser converts each split digit to value which is between 1 and 6 to demonstrate the dice.

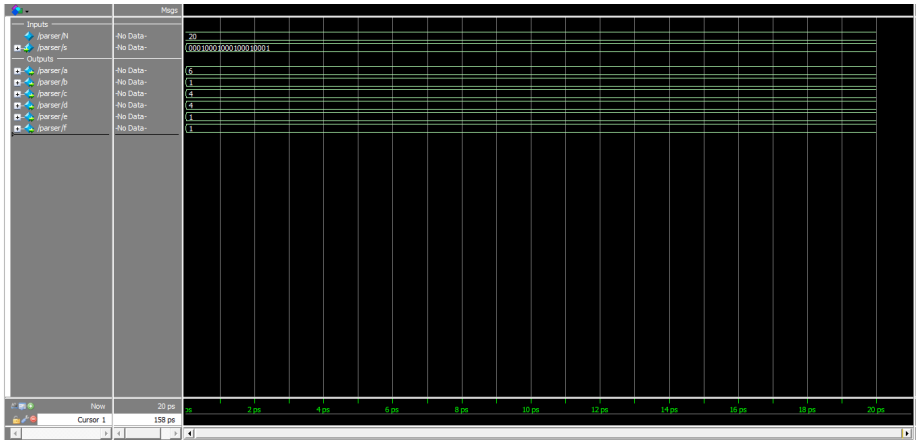

Figure 17: Parser simulation.

2.3 Functional Unit: Animator

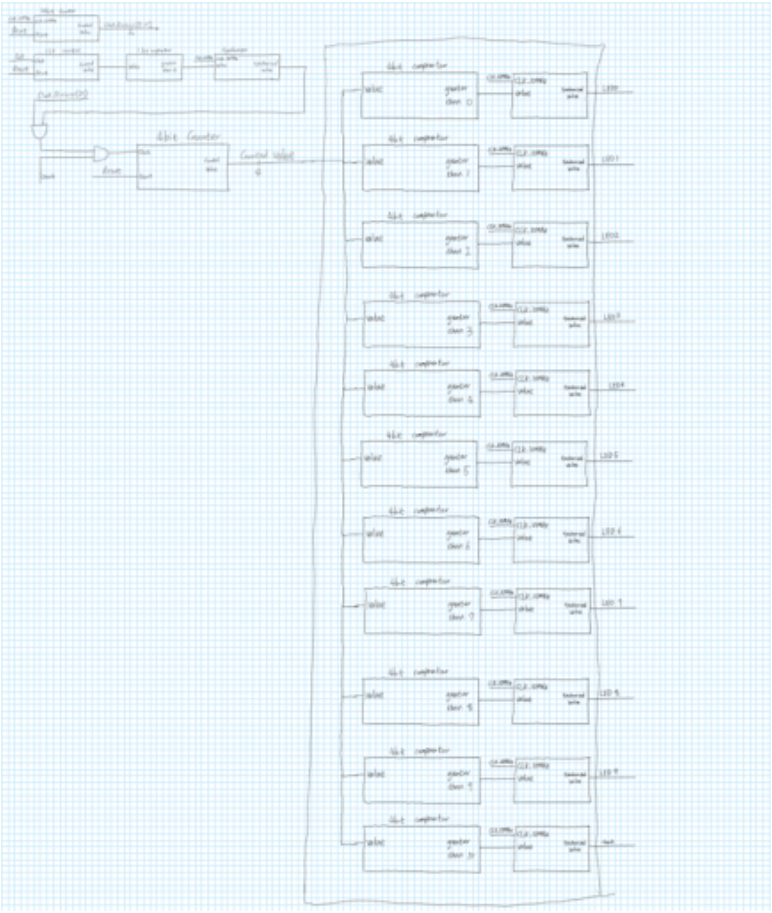

Figure 18: Animator lights up 10 LEDs in the FPGA board as the time flows, and checks whether it is a condition to display the random numbers on seven segment displays or not. The 10MHz clock is divided into 22-bits. The 1-bit counter receives the signal from the start and reset button, and the output of the combination of 1-bit comparator and synchronizer acts as an element of clock input of the 4-bit counter with the most significant bit of 10MHz clock, ClockDivision[21]. The 4-bit counter outputs a 4-bit value which lights up 10 LEDs, and receives the same reset value with a 22-bit counter. 4-bit output is connected to each input of 11 comparator, and synchronizer outputs the values which are connected to 10 LEDs and checks the output which is element of clock input of 4-bit counter depending on if the input of synchronizer is greater than the setup value.

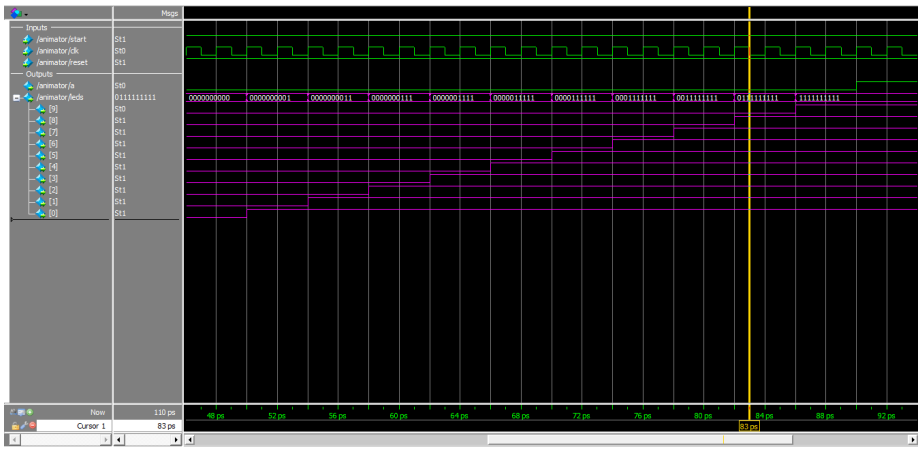

Figure 19: Animator simulation.

2.3.1 Individual Block: Comparator

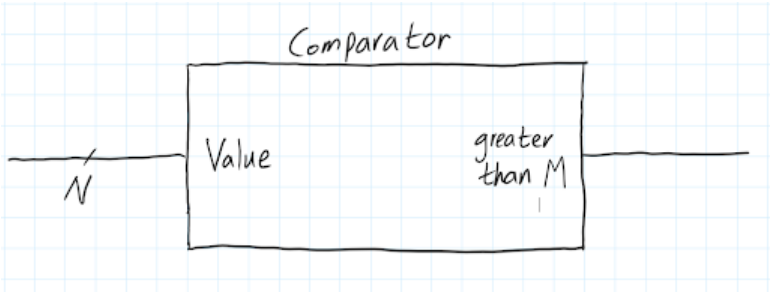

Figure 20: The comparator compares the input value and the parameter M. The output logic is 1 when the N-bit input value is greater than the parameter M, and if the N-bit input value is equal or less than the parameter M, then the output logic is 0.

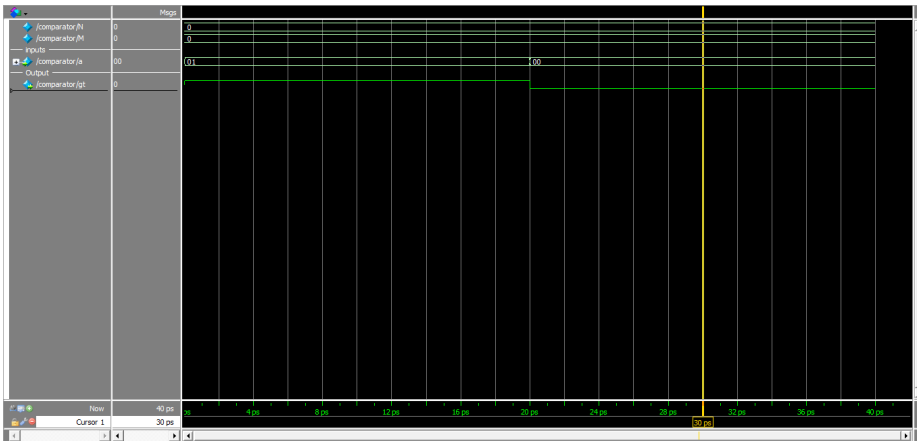

Figure 21: Comparator simulation.

2.3.2 Individual Block: Synchronizer

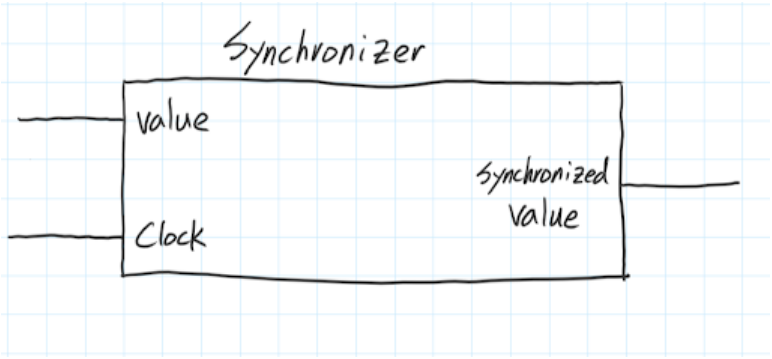

Figure 22: The synchronizer synchronizes the input value to the input clock. To be specific, the input of the synchronizer acts as clock input. Moreover, if the input has irregular matching points with the clock input, then it is complicated to perform the expected result. The synchronizer synchronizes the input which acts as the clock to the clock input in order to perform the expected result.

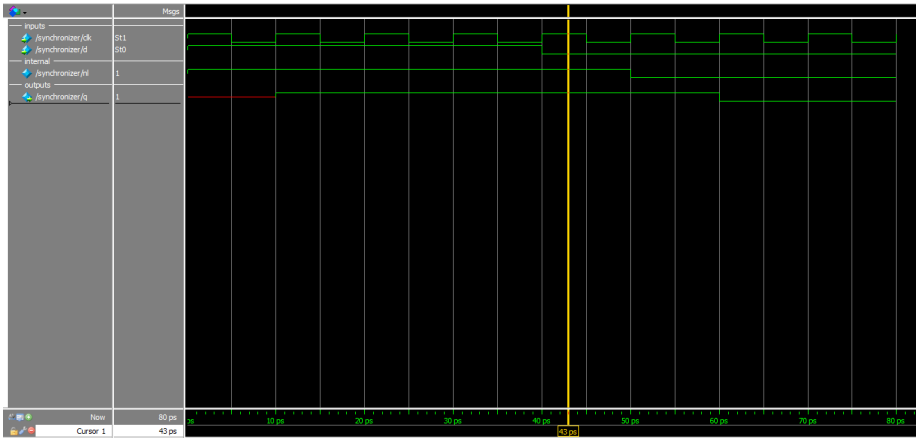

Figure 23: Synchronizer simulation.

A SystemVerilog Files

Listing 1: counter.sv

```
1  // adapted from HDL Example 5.5 (p. 260)
2  module counter#( parameter N=0 )
3    ( input  logic        clk
4    , input  logic        reset
5    , output logic[N-1:0] q )
6    ;
7
8    always_ff@( posedge clk, posedge reset ) begin
9      if ( reset ) q <= 0;
10     else         q <= q+1;
11   end
12
13 endmodule
```

Listing 2: comparator.sv

```
1  // adapted from HDL Example 5.3 (p. 248)
2  module comparator#( parameter N=0, M=0 )
3    ( input  logic[N-1:0] a
4    , output logic        gt )
5    ;
6
7    assign gt = a > M;
8
9  endmodule
```

Listing 3: synchronizer

```
1  // adapted from HDL Example 4.20 (p. 197)
2  module synchronizer
3    ( input  logic d
```

```
4    ,              clk
5    , output logic q )
6    ;
7
8    logic nl;
9
10   always_ff@( posedge clk ) begin
11     nl <= d;
12     q  <= nl;
13   end
14
15 endmodule
```

Listing 4: flopr.sv

```
1  // adapted from HDL Example 4.18 (p. 195)
2  module flopr#( parameter N=20 )
3    ( input  logic        clk
4    ,                     reset
5    , input  logic[N-1:0] d
6    , output logic[N-1:0] q )
7    ;
8
9    always_ff@( posedge clk, posedge reset ) begin
10     if ( reset ) q <= 0;
11     else         q <= d;
12   end
13
14 endmodule
```

Listing 5: parser.sv

```
1  module parser#( parameter N=20 )
2    ( input  logic[N-1:0] s // input to be parsed into 6 numbers
3    , output logic[3:0]   a // six numbers constrained to the range
4    ,                     b // range [1,6], inclusive
5    ,                     c
6    ,                     d
7    ,                     e
8    ,                     f )
9    ;
10
11   assign a = ((s/1)%6)+1;
12   assign b = ((s/10)%6)+1;
13   assign c = ((s/100)%6)+1;
14   assign d = ((s/1000)%6)+1;
15   assign e = ((s/10000)%6)+1;
16   assign f = ((s/100000)%6)+1;
17
18 endmodule
```

Listing 6: picker.sv

```
1  module picker
2    ( input  logic[23:0] r   // random numbers
3    , input  logic[5:0]  s   // switches
4    , input  logic       a   // animation completion signal
```

```
5    , output logic[23:0] q ) // output to display drivers
6    ;
7
8    always_comb begin
9      if ( a ) begin // if animation is done, transmit signal
10       if ( s[0] ) q[3:0]   = r[3:0]; // if switch is high then transmit
11       else        q[3:0]   = 4'b1111; // else transmit blank signal
12       if ( s[1] ) q[7:4]   = r[7:4];
13       else        q[7:4]   = 4'b1111;
14       if ( s[2] ) q[11:8]  = r[11:8];
15       else        q[11:8]  = 4'b1111;
16       if ( s[3] ) q[15:12] = r[15:12];
17       else        q[15:12] = 4'b1111;
18       if ( s[4] ) q[19:16] = r[19:16];
19       else        q[19:16] = 4'b1111;
20       if ( s[5] ) q[23:20] = r[23:20];
21       else        q[23:20] = 4'b1111;
22     end
23     else begin // else ensure displays are blank
24       q = 24'b111111111111111111111111;
25     end
26   end
27
28 endmodule
```

Listing 7: displayDriver.sv

```
1  // adapted from HDL Example 4.24 (p. 201)
2  module displayDriver
3    ( input  logic[3:0] data
4    , output logic[6:0] segs )
5    ;
6
7    always_comb begin
8      case ( data )
9        0:       segs = 7'b1000000;
10       1:       segs = 7'b1111001;
11       2:       segs = 7'b0100100;
12       3:       segs = 7'b0110000;
13       4:       segs = 7'b0011001;
14       5:       segs = 7'b0010010;
15       6:       segs = 7'b0000010;
16       default: segs = 7'b1111111;
17     endcase
18   end
19
20 endmodule
```

B Simulation Files

Listing 8: leds.do—A Do script for testing the LED animation. Additional troubleshooting was required in order to simulate the animation properly due to initially floating signals on button presses.

```
1  vsim -gui work.diceMachine
2
```

```
restart -force -nowave

add wave clk
add wave roll
add wave reset
add wave led0
add wave led1
add wave led2
add wave led3
add wave led4
add wave led5
add wave led6
add wave led7
add wave led8
add wave led9

force clk 1 0ps, 0 {1ps} -r 2ps
force roll 0 0ps, 1 1ps, 0 2ps, 1 3ps
force roll 0 1ns, 1 2ns
force reset 0 0ps, 1 1ps, 0 2ps, 1 3ps

run 100ns
```

C Pin Assignments

Listing 9: assignment.qsf

```
set_location_assignment PIN_P11 -to clk
set_location_assignment PIN_B8  -to roll
set_location_assignment PIN_A7  -to reset

set_location_assignment PIN_C10 -to sw[0]
set_location_assignment PIN_C11 -to sw[1]
set_location_assignment PIN_D12 -to sw[2]
set_location_assignment PIN_C12 -to sw[3]
set_location_assignment PIN_A12 -to sw[4]
set_location_assignment PIN_B12 -to sw[5]

set_location_assignment PIN_A8  -to leds[0]
set_location_assignment PIN_A9  -to leds[1]
set_location_assignment PIN_A10 -to leds[2]
set_location_assignment PIN_B10 -to leds[3]
set_location_assignment PIN_D13 -to leds[4]
set_location_assignment PIN_C13 -to leds[5]
set_location_assignment PIN_E14 -to leds[6]
set_location_assignment PIN_D14 -to leds[7]
set_location_assignment PIN_A11 -to leds[8]
set_location_assignment PIN_B11 -to leds[9]

set_location_assignment PIN_C14 -to seg0[0]
set_location_assignment PIN_E15 -to seg0[1]
set_location_assignment PIN_C15 -to seg0[2]
set_location_assignment PIN_C16 -to seg0[3]
set_location_assignment PIN_E16 -to seg0[4]
set_location_assignment PIN_D17 -to seg0[5]
set_location_assignment PIN_C17 -to seg0[6]
```

```
30
31  set_location_assignment PIN_C18 -to seg1[0]
32  set_location_assignment PIN_D18 -to seg1[1]
33  set_location_assignment PIN_E18 -to seg1[2]
34  set_location_assignment PIN_B16 -to seg1[3]
35  set_location_assignment PIN_A17 -to seg1[4]
36  set_location_assignment PIN_A18 -to seg1[5]
37  set_location_assignment PIN_B17 -to seg1[6]
38
39  set_location_assignment PIN_B20 -to seg2[0]
40  set_location_assignment PIN_A20 -to seg2[1]
41  set_location_assignment PIN_B19 -to seg2[2]
42  set_location_assignment PIN_A21 -to seg2[3]
43  set_location_assignment PIN_B21 -to seg2[4]
44  set_location_assignment PIN_C22 -to seg2[5]
45  set_location_assignment PIN_B22 -to seg2[6]
46
47  set_location_assignment PIN_F21 -to seg3[0]
48  set_location_assignment PIN_E22 -to seg3[1]
49  set_location_assignment PIN_E21 -to seg3[2]
50  set_location_assignment PIN_C19 -to seg3[3]
51  set_location_assignment PIN_C20 -to seg3[4]
52  set_location_assignment PIN_D19 -to seg3[5]
53  set_location_assignment PIN_E17 -to seg3[6]
54
55  set_location_assignment PIN_F18 -to seg4[0]
56  set_location_assignment PIN_E20 -to seg4[1]
57  set_location_assignment PIN_E19 -to seg4[2]
58  set_location_assignment PIN_J18 -to seg4[3]
59  set_location_assignment PIN_H19 -to seg4[4]
60  set_location_assignment PIN_F19 -to seg4[5]
61  set_location_assignment PIN_F20 -to seg4[6]
62
63  set_location_assignment PIN_J20 -to seg5[0]
64  set_location_assignment PIN_K20 -to seg5[1]
65  set_location_assignment PIN_L18 -to seg5[2]
66  set_location_assignment PIN_N18 -to seg5[3]
67  set_location_assignment PIN_M20 -to seg5[4]
68  set_location_assignment PIN_N19 -to seg5[5]
69  set_location_assignment PIN_N20 -to seg5[6]
```

D Synthesized Logic

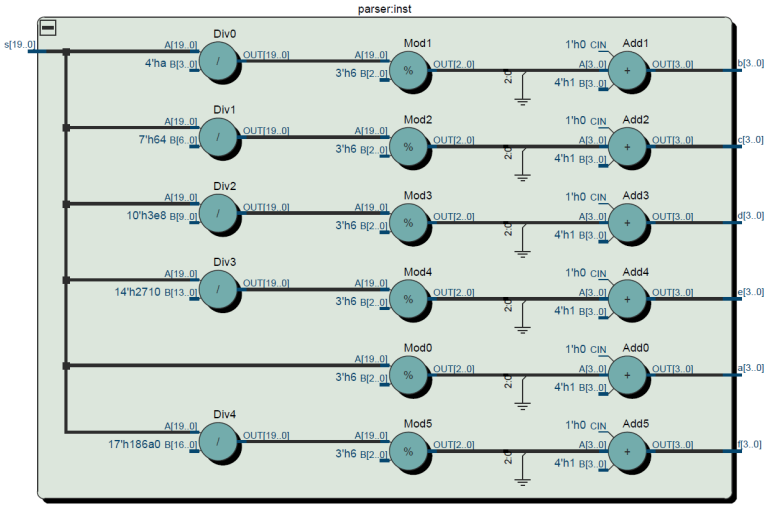

Figure 24: Synthesized logic for the parser.

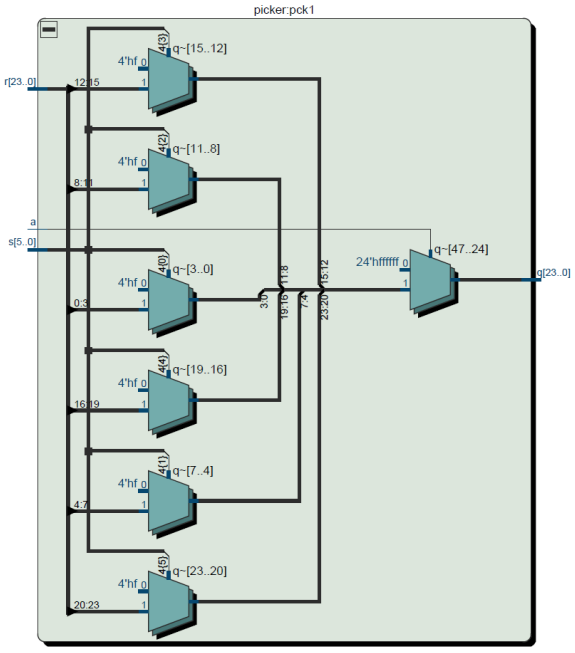

Figure 25: Synthesized logic for the picker.

E Quartus Files

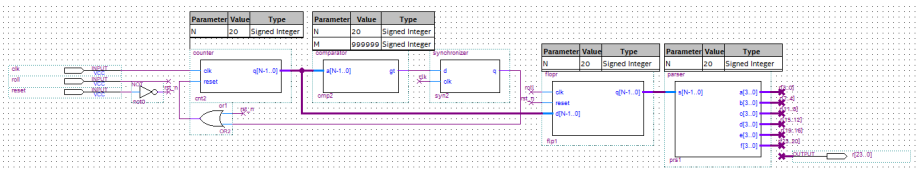

Figure 26: randomizer.bdf

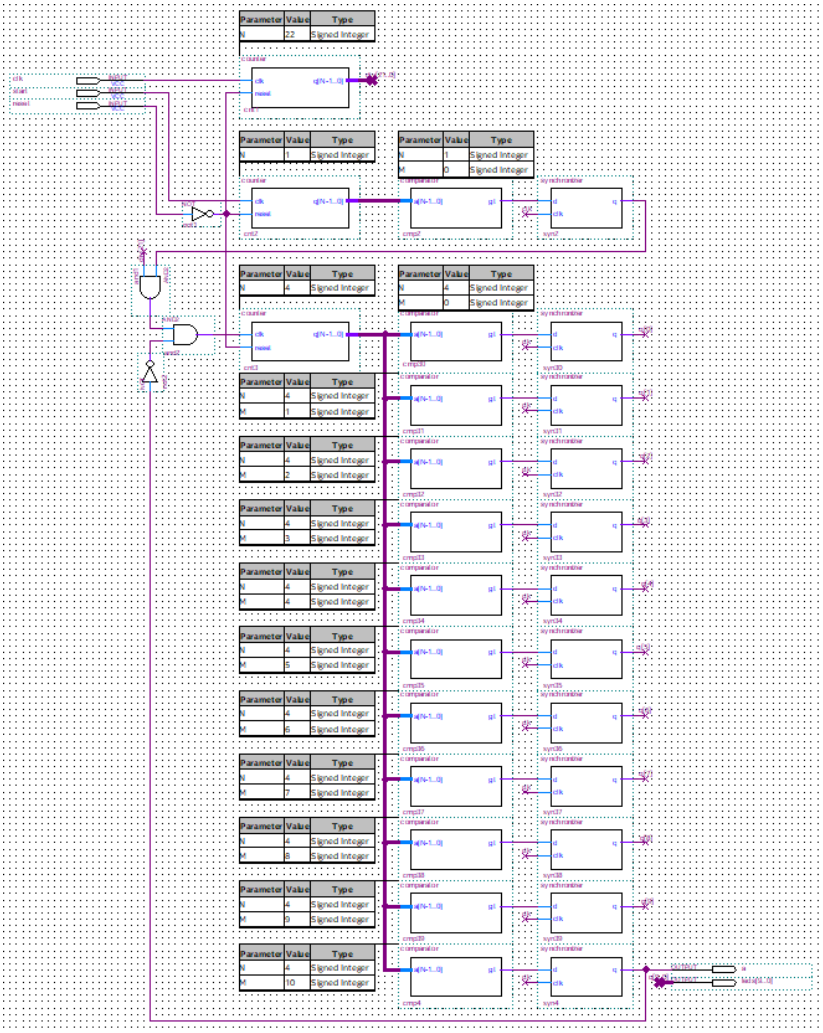

Figure 27: animator.bdf

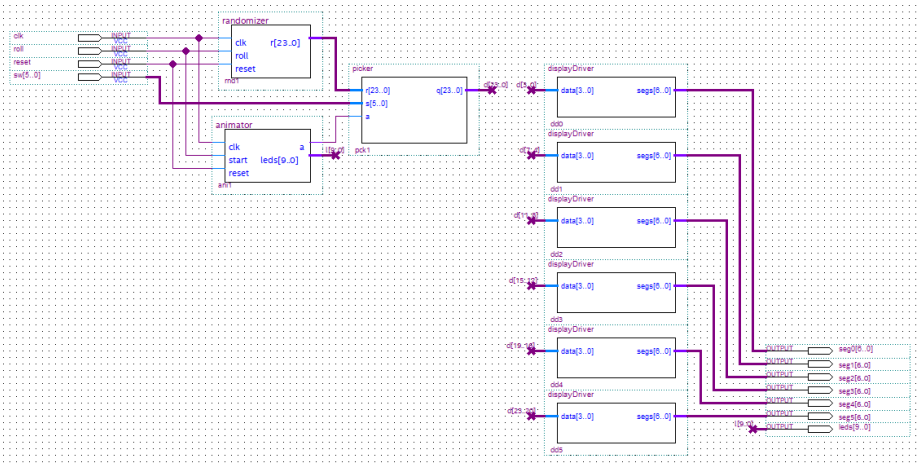

Figure 28: diceMachine.bdf (top-level)