ECE 271 Design Project

Group 14

Spring 2020

Samuel Barton, Carson Edmonds, Ben Jones, Jasper Morrison

June 5th, 2020

Contents

1. Project Description

On a general level, the focus of this project was to learn how to build logic blocks without guidance to do a specific task, and to be able to understand the logic implemented well enough to thoroughly test the modules created, a task that the common Electrical Engineer could use frequently in the future. Specifically to this project, the goal was to be able to interface an NES controller (acting as a shift register) with multiple blocks of sequential and combinational logic to translate the pressed buttons on the controller into a letter A-G and output on a seven segment display. With the 8 inputs on the NES controller, 7 out of the 8 inputs would represent each letter, displaying the letter on one of the seven-segment displays. There are three key features of the design, each coming from three different major logic blocks: the NES reader, encoder, and seven-segment display decoder.

Considering the circumstances of the current term, the testing and simulation portion of this project will be rather extensive. Despite the underwhelming outcome of not being able to see the finished product, this gives the developers the ability to understand, thoroughly, the subtle nuances of digital logic and simulation.

2. High Level Description

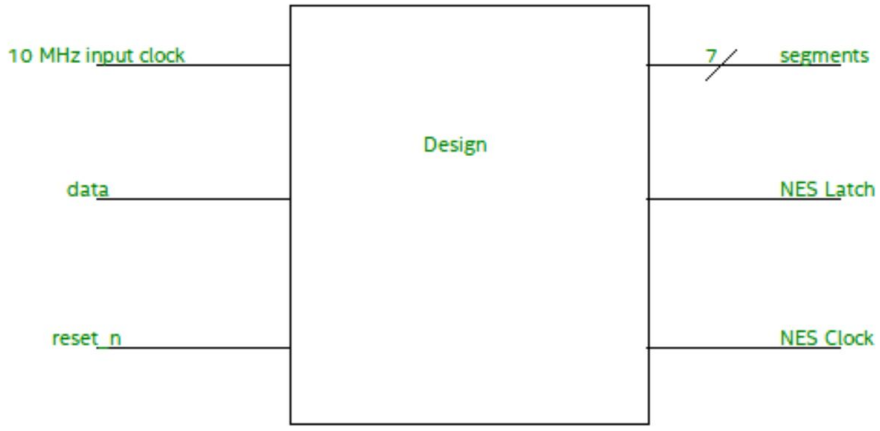

Figure 2.1: The basic Hardware diagram showing the inputs that interact with this design, as well as the three outputs that are dependent on the inputs shown.

The full design shown in Figure 2.1 shows that the entirety of the design is run off of 3 inputs (the 10 MHz input clock, data input from the NES controller, and an active-low reset coming from the FPGA. From the NES controller, 7 out of the 8 switches (A, B, Select, Start, Up, Down, Left) will all be representing a specific note in the musical scale. From there, the data from the controller will be pushed to the NES reader, where the singular bits of data that had been

pushed from the NES controller are re-established into an 8-bit bus and passed to the encoder. The encoder then takes the 8-bit signal and puts it in a form that is easy for the 4:7 Decoder that makes up the Seven-Segment Display Decoder to read and create a valid note output (A-G) on the display.

- Inputs
    - Input_clk: A 10MHz clock signal stemming from the FPGA. For testing reasons, the clock period will typically be much shorter than that of a 10MHz frequency for simplicity and shorter simulation times. The main purpose of this signal is to be slowed to approximately 5kHz, which is the frequency of the clock used in the NES reader.
    - Reset_n: An active low push button that would be connected to a push button on the FPGA. This signal resets all components in the above design on its falling edge.
    - Data_in: The data of the past buttons being pushed on the NES controller. Had this lab been done in-person, the NES controller would act as a parallel-in, serial-out shift register, outputting the most significant bit of the parallel data during a shift period.
- Outputs:
    - NES_latch: When in conjunction with a physical NES controller PISO shift-register, this latch signal is used to tell the controller to load (in parallel) the new inputs from the data bus.
    - NES_clock: Similar to the NES_latch, this clock signal would also be connected to the shift register acting as the NES controller, creating the “shift” signals that output the current most significant bit into the serial out signal that would be the data pin in the design above.

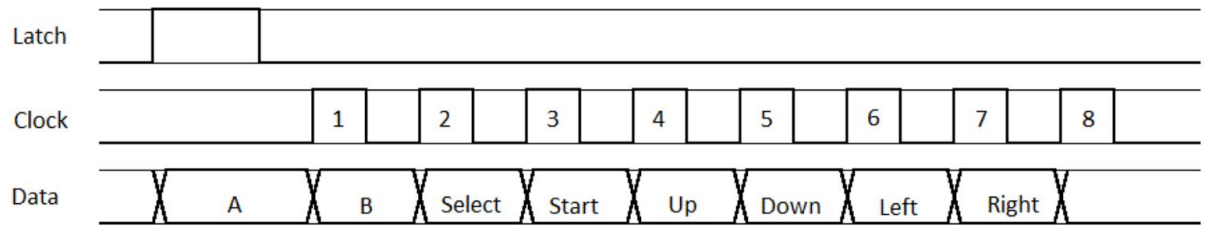

Figure 2.2: NES Latch/Clock timings, along with button inputs that correspond to the given bit, with A being in the place of the MSB. [1]

2.1 Top-Level Module

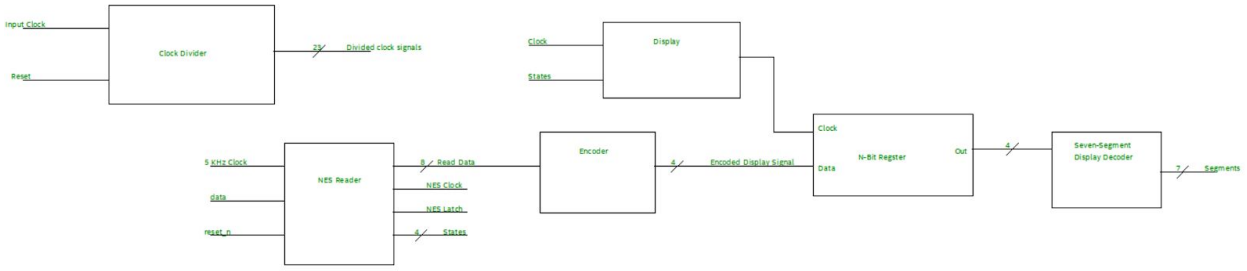

Figure 2.3: Top module showing the 6major modules used, as well as the inputs shown in Figure 2.1.

Regarding Figures 2.1 and 2.3 above, it is apparent where the inputs shown in the general block diagram in Figure 2.1 lead to, as well as a general flow of how the input data will be processed and outputted by the designed modules. Sections 3-7 will cover the specific design and functionality of each of the blocks shown in the figure above.

3. NES Reader Description and Functionality

3.1 Description

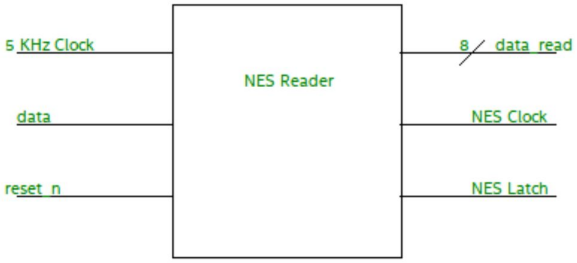

Figure 3.1: NES Reader block diagram showing the 3 inputs (Clock, data, and active-low reset), along with the 3 outputs (8-bit read data bus, NES Clock, and NES Latch).

NES controllers act as parallel-in serial-out shift registers, essentially taking 8 bits of data and storing it between 8 different D flip-flops connected in series that will “push” the data out of one side of the string of flip-flops one at a time, at the rising clock edge. The NES Reader portion of the interface between controller and console takes the individually shifted bits from the controller and stores them back into an 8-bit signal that is then sent to the console in the standard use of the NES. Not only is the reader responsible for storing the bits of data that are shifted out, but the reader is also responsible for creating two very important waveforms that will determine how the controller functions. These two waveforms are called the NES latch and NES

clock signals. On the rising edge of the latch signal, the 8-bits of data stored coming from each of the 8 buttons on the controller are simultaneously loaded into the shift register, while the rising edge of the clock then acts as the clock edge that pushes the data out of flip-flop that holds the least significant bit.

Keeping in mind that the buttons must be pushed to become activated, the data coming out of the NES controller will be in an active-low form, meaning that the NES reader will have to store 0's indicating when a button on the NES has been pushed.

3.2 Modules

The NES reader module consists of a 4-bit counter, along with two comparators, a 1-bit counter, and a module that "decodes" the inputs from the NES controller and creates a signal on the 8-bit bus to be passed to the following modules.

3.2.1 4-bit Counter

Figure 3.2: Block diagram of the 4-bit counter showing the two inputs (the slowed 5 KHz input clock and reset) along with the 4-bit "counted" output bus.

The sole purpose of the 4-bit counter is to break down the 5 KHz clock into 16 different states, which is useful for the creation of both NES latch and clock as well making the creation of the read data bus much simpler. A 4-bit counter was used due to the property of a counter resetting once it hits its maximum sum that can be displayed by the number of output bits in a specific module. In the case of a 4-bit counter, the "count" resets after 15. This number is important, because NES latch and clock both rely on the state number (0-15) to determine whether they are high or low at a given instance, which makes the need to do exact timing of waveforms (a characteristic that is necessary in the conventional NES controller/reader/console relationship) irrelevant for the purpose of our design.

3.2.2 Comparator

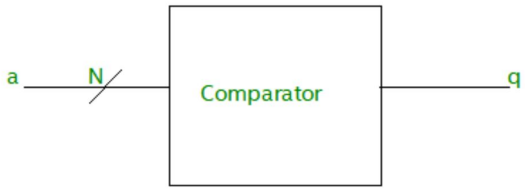

Figure 3.3: Block diagram of the modular N-bit comparator. The N-bits are characterized by the parameters provided in each module, and can vary depending on the designer's needs.

The function of the comparators used in the NES reader module was to check for equality compared to a set value given by parameters. Both comparators that were used in the NES Reader module checked for the instance when the counter in the above section had been reset to the state (or output value) of 0. The purpose of one of the comparators was to reset the input decoder (explained in section 3.2.4) to clear the bus storing the inputs being received, allowing for accurate accumulation of data that corresponded to the latch and clock signals also created by the NES reader. The second comparator was used to signify whether the state of the counter was in the “latch” state or not. If the value of the state was between 0 and 1, the latch of the NES should be high, allowing the data stored in the parallel inputs of the controller to be loaded into the shift register that makes up the controller. For the remaining 15 states, the latch should remain low, and the NES clock should be oscillating.

3.2.3 Single-bit Counter

Figure 3.4: The block diagram of the 1-bit counter used to create the NES Clock signal. This module is very similar to the 4-bit counter given in the sections above.

Similar to the 4-bit counter shown in section 3.2.1, the single-bit counter uses the 5 KHz clock used for the 4-bit counter, yet it slows the period to be half of that of the 5 KHz clock. Attached to the reset pin of the single-bit counter is the latch signal created by the comparator described in section 3.2.2. Essentially what this is doing is stopping this counter from incrementing and dividing the input clock signal any time that the latch is high, which (in reference to the waveform of NES latch and clock shown in figure 2.2), is accurate to the desired function of NES clock. Connecting the latch signal to this single-bit counter that creates the NES clock signal also ensures that the NES clock only has 8 rising edges to shift exactly the 8 bits that are present in the shift register out.

3.2.4 NES Data Decoder

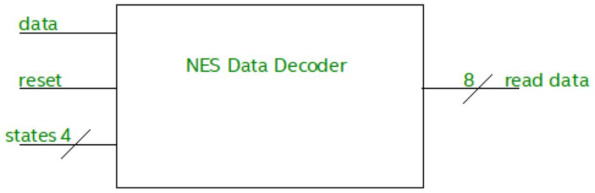

Figure 3.5: Block diagram that displays the inputs that are used for the Data decoder, as well as the 8-bit bus that outputs the collected data from the controller.

The NES Data Decoder takes in the serially-outputted data from the NES controller and re-configures the 8 bits that had been stored into a single 8-bit bus that can then be passed to the remaining portion of the design. The data pin will be connected to the Serial Out pin of the PISO shift register in the NES controller. The reset pin of the data decoder is connected to an OR gate that, when either the reset button on the FPGA has been pushed or the states have been reset by the 4-bit counter, the read data bus is then cleared to 0000 0000. This allows for the decoder to be “reading” accurate data from the shift register, and resetting the read values simultaneously to when the latch goes high. Depending on the state value, the data input that comes in from the NES controller is put in a specific bit in the 8-bit output bus. Table 3.1 shows the state of the counter with the corresponding spot in the output bus.

State	Bit
1	7
3	6
5	5
7	4
9	3
11	2
13	1
15	0

Table 3.1: State and output bit relationship inside the NES Data Decoder module.

Due to the property of the shift register “shifting” out the least significant bit of the collected data from the buttons being pressed, the order of most significant bit to least significant bit inside the shift register becomes the opposite inside the NES reader, and, specifically, the NES data decoder. For example, if the NES controller was holding the value

1100 1010 in the shift register, after going through each of the states, the 8 bit output will read 0101 0011, essentially flipping the bits of the extracted data in the NES controller.

3.3 Functionality

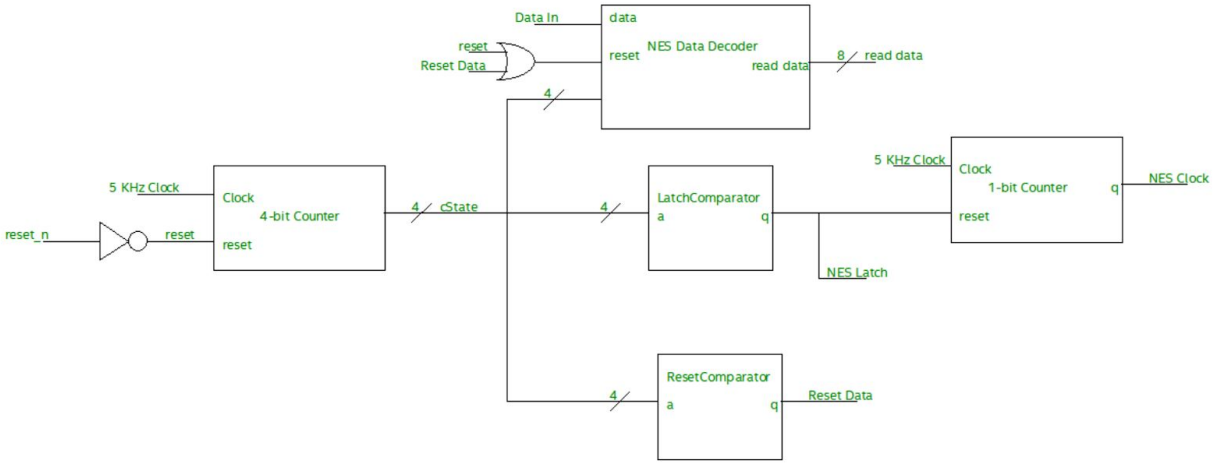

Figure 3.6: Block Diagram showing the top-level of the NES reader module.

In Figure 3.6, the wiring between each of the modules described in this section is shown. One element to notice is that the reset signal given from the use of the push button is active low. Keeping in mind that the modules that are resettable in this design are run on an active high reset signal, the reset_n signal from the push button is inverted and further referenced for the rest of the modules as “reset”. As roughly described in the previous sections, the majority of these modules rely on state information from the 4-bit counter which essentially connects the NES data decoder, the latch comparator, and the reset comparator in parallel to each other.

The full description of the NES Reader (in steps) goes as the following:

1) The 4-bit counter must be reset using the active low push button. As the button is pressed, the signal (reset_n) is flipped (becoming reset) and used for the remaining modules.
2) The 4-bit counter counts the rising clock edges of the 5 KHz clock up to a value of 15.
3) The sum created by the counter represents a state of the system and is then passed to the latch comparator, data decoder, and reset comparator.
    a) The latch comparator creates a latch signal if the state is between 0 and 1, otherwise the NES clock begins alternating at half of the speed of the 5 KHz input clock.
    b) The reset comparator sends a reset signal if the state is equal to 0 to reset the data decoder.

c) On the odd numbered states, the data decoder fills the most significant bit starting when the state is equal to 1 to the least significant bit when the state is equal to 15.

4. Encoder Description and Functionality

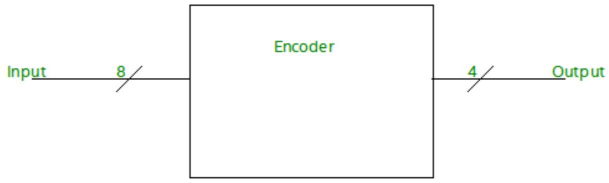

Figure 4.1: The simple block diagram that shows the single 8-bit input bus that leads into the encoder module as well as the 4-bit output bus.

The sole purpose of the encoder module is to take the data collected from the buttons of the NES controller, and organize them in a way that would be more readable and easier to handle for the seven segments to display. For the given design, the goal was to assign a single button with a single displayed output on the seven-segment display, implying that a module similar to that of a priority encoder. For example, if the button A and B were to be pushed at the same time, the button with the highest priority would be the value displayed on the seven-segment display. Bearing in mind that the NES reader essentially flips the data received in the NES controller, Table 4.1 shows the bit placement of each of the buttons, as well as their priority found in the encoder.

Button	Raw Data Bit (Controller)	Read Data Bit (Reader)	Priority
A	0	7	1
B	1	6	2
Select	2	5	3
Start	3	4	4
Up	4	3	5
Down	5	2	6
Left	6	1	7
Right	7	0	8

Table 4.1: Data bit conversions between the controller and the reader, as well as the priority encoding of each button (1 being the highest priority and 8 being the lowest). [2]

After priority had been given to the most significant bits in the read data bus, the next step was to then create a signal that was more comprehensible for the seven-segment display decoder to work with. Referencing the input values to the displayed outputs on the displays themselves (a table of the inputs to displays can be found in Section 5), the arbitrary choice to have the button A represent the note A, and Left represent the note G made the encoding fairly simple. A table of input buttons with the outputs and displayed notes is found in Table 4.2.

Button	Read Data (8 bits)	Priority	Output Value (4 bits)	Note
A	1XXX XXXX	1	0000	A
B	01XX XXXX	2	0001	B
Select	001X XXXX	3	0010	C
Start	0001 XXXX	4	0011	D
Up	0000 1XXX	5	0100	E
Down	0000 01XX	6	0101	F
Left	0000 001X	7	0110	G
Right	0000 0001	8	0111	Not Used

Table 4.2: Input buttons with their corresponding read data signals, encoded output signals, and corresponding notes that each button represents.

5. Seven Segment Display Description and Functionality

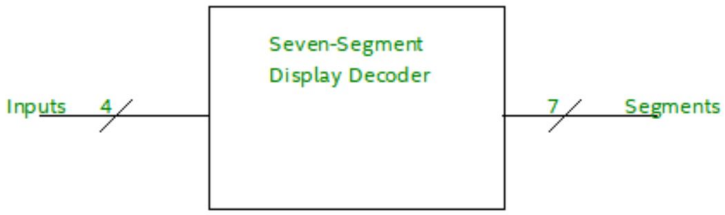

Figure 5.1: Block Diagram of the frequently used block diagram for 4:7 Decoder used to decode signals to be sent to the Seven-Segment displays.

The purpose of the seven segment display module is to display a 4-bit input which comes from the decoder and output letters onto the FPGA's seven segment display. The display takes in a 4-bit binary input and then outputs onto the seven segment display. For this project the outputs will be the letters A-G. In Table 5.1 the truth table shows the logic for the display. The seven segment display is logic LOW so on is 0 and off is 1. In Figure 5.1 the seven

segment display design is shown. The 4-bit data input passes through the logic gates and based in the truth table outputs one of the letters A-G. This design has been simulated and the expected outputs agree with the actual outputs upon testing in ModelSim.

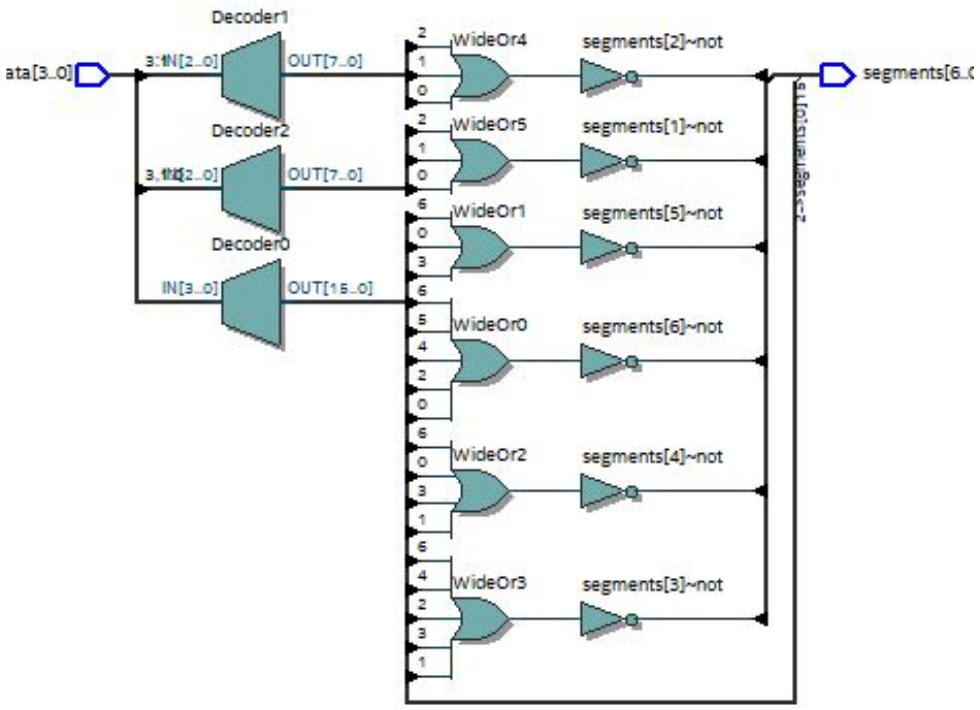

Figure 5.2: Synthesized design (provided by RTL Viewer in Quartus) of the Seven Segment Display module.

Binary input (4-bit)	Seg0	Seg1	Seg2	Seg3	Seg4	Seg5	Seg6	Displayed output
0000	0	0	0	1	0	0	0	A
0001	1	1	0	0	0	0	0	B
0010	0	1	1	0	0	0	1	C
0011	1	0	0	0	0	1	1	D
0100	0	1	1	0	0	0	0	E
0101	0	1	1	1	0	0	0	F
0110	0	0	0	0	1	0	0	G

Table 5.1:Seven Segment Display for desired outputs A-G.

6. Clock Divider Description and Functionality

Figure 6.1: Block diagram of the Clock Divider module used.

The purpose of this module is to reduce a 10 MHz clock signal from the FPGA to a usable 5 kHz signal that will be used by the NES reader. To achieve this, a counter is used. The counter contains a 23-bit output bus named “divided” which increments on each rising edge of the 10 MHz clock. The 10th bit is used in the design as the NES system clock, which is used for both clock signals and latch signals.

7. Display Module Description and Functionality

Figure 7.1: Display module block diagram.

The purpose of this module was to guarantee that the seven-segment display was only displaying the value of the data that had been collected through the full period of the latch, rather than updating the display any time a new bit of data had been read from the NES controller. To do so, the module was designed to update the seven-segment display half way through the 15th state inside the NES reader using the slowed 5 KHz clock, and the current state of the Reader module(the input node “Data” would include the data pertaining to the current state). Choosing the midpoint of the 15th state guaranteed that every bit of data stored

in the PISO shift register of the NES controller had been shifted out, but the data read in the NES reader hadn't been reset. The display module was designed to act as an updated clock signal for a 4-bit register, assuring the 4 bits of encoded data created was only being sent to the seven-segment displays at the appropriate time. The connection between the Display and N-bit (acting as a 4-bit) register module is shown in Figure 7.2.

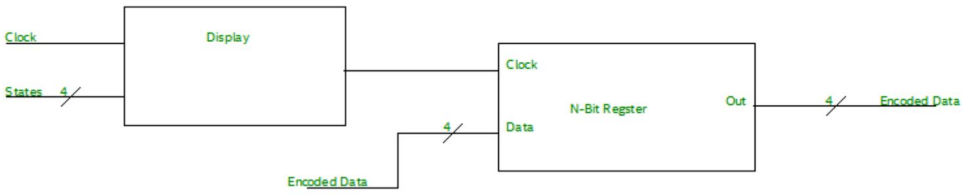

FIgure 7.2: Block diagram showing the connection between the Display and N-Bit Register module.

8. Synthesized Design

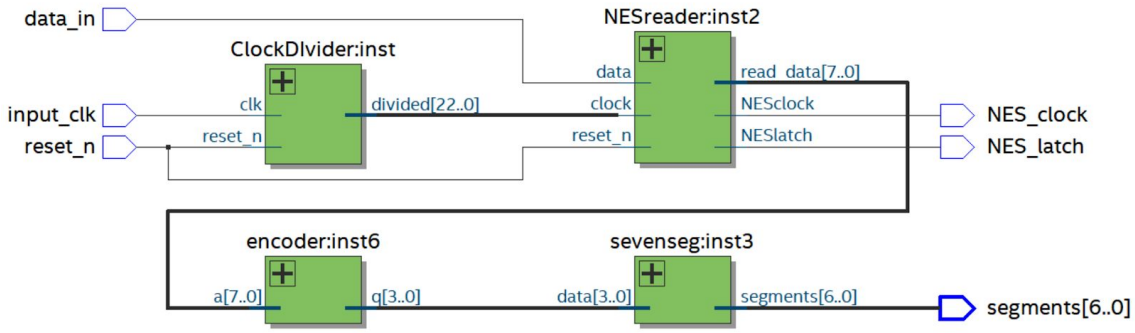

Figure 7.1: Synthesized block diagram of the top level module designed.

Resource Summary

Flow Status	✔	Successful - Thu Jun 04 21:38:17 2020
Logic Element Usage	✔	50 / 49,760 (< 1 %)
Memory Block Usage	✔	0 / 1,677,312 (0 %)
DSP Block Usage	✔	Not available
I/O Usage	✔	12 / 360 (3 %)

Figure 7.2: Resource summary of the top-level design after synthesis.

According to the resource summary shown in Figure 7.2, the amount of logic elements used in the top-level design of this project is much lower than the total amount of elements that could be used, and no memory is used for this design.

9. SystemVerilog & Verilog Files

This section shows all of the SystemVerilog files that were created during the design process of this project. The SystemVerilog files used in the NES Reader module can be found in section 7.1, encoder in section 7.2, seven-segment display decoder in 7.3, clock divider module in section 7.4, as well as the top-level SystemVerilog file in section 7.5.

9.1 NES Reader

9.1.1 Counter

```
1   module counter #(parameter N = 8)
2                   (input logic clk,
3                    input logic reset,
4                    output logic [N-1:0] q);
5
6         always_ff @(posedge clk, posedge reset)
7            if (reset)     q = 0;
8            else           q = q + 1;
9
10  endmodule
```

NESreader/counter.sv [3]

9.1.2 NES Data Decoder

```
1   module NESdatadecoder(input   logic data,
2                         input   logic reset,
3                         input   logic NES_clk,
4                         input   logic [3:0] cState,
5                         output  logic [7:0] read);
6
7           always_ff @(negedge NES_clk, posedge reset)
8              if(reset)   read     =  0;
9              else case (cState [3:0])
10                1:        read[7]  <= data;
11                3:        read[6]  <= data;
12                5:        read[5]  <= data;
13                7:        read[4]  <= data;
14                9:        read[3]  <= data;
15                11:       read[2]  <= data;
16                13:       read[1]  <= data;
17                15:       read[0]  <= data;
18                default: read      <= 0;
19             endcase
20
21  endmodule
```

NESreader/NESdatadecoder.sv [4]

9.1.4 Comparator

```
module comparator #(parameter N = 12, parameter A = 8)
                   (input logic [N-1:0] a,
                    output logic q);

      assign q = (a == A);

endmodule
```

NESreader/comparator.sv

9.2 Encoder

```
module encoder #(parameter N = 8)
                (input  logic [N-1:0] a,
                 output logic [3:0] q);

      always_comb
         if      (!a[7])  q <= 4'b0000; //a
         else if (!a[6])  q <= 4'b0001; //b
         else if (!a[5])  q <= 4'b0010; //c
         else if (!a[4])  q <= 4'b0011; //d
         else if (!a[3])  q <= 4'b0100; //e
         else if (!a[2])  q <= 4'b0101; //f
         else if (!a[1])  q <= 4'b0110; //g
         else if (!a[0])  q <= 4'b0111; //not used
         else             q <= 4'b1111; //not used

endmodule
```

DesignTop/encoder.sv

9.3 Seven Segment Display

```
module sevenseg(input logic [3:0] data,
                output logic [6:0] segment
always_comb
   case(data)
      // abcdefg
      0: segments = 7'b0001000;//a
      1: segments = 7'b1100000;//b
      2: segments = 7'b0110001;//c
      3: segments = 7'b1000010;//d
      4: segments = 7'b0110000;//e
      5: segments = 7'b0111000;//f
      6: segments = 7'b0000100;//g
      //7: segments = 7'b0001111;
      //8: segments = 7'b0000000;
      //9: segments = 7'b0001100;
      default: segments = 7'b1111110;
   endcase
endmodule
```

DesignTop/sevenseg.sv [3]

9.4 Clock Divider

```
1  module clockDIvider (input    logic clk,
2                       input    logic reset_n,
3                       output   logic [22:0] divided);
4
5  //input clock will be 10 MHz, a 5 kHz clock will be
6  //available at signal divided[10]
7
8        always_ff @(posedge clk, negedge reset_n)
9           if (reset_n == 0)    divided <= 0;
10          else                 divided = divided + 1;
11
12
13
14 endmodule
```

DesignTop/clockDIvider.sv

9.5 Display

```
1  module display #(parameter N = 4)
2                  (input  logic clk,
3                   input  logic [N-1:0] state,
4                   output logic q);
5
6        always_comb
7           if (!clk)
8              if (state == 15)  q = 1;
9              else              q = 0;
10          else                 q = 0;
11 endmodule
```

DesignTop/display.sv

9.6 Nregister

```
1  module Nregister #(parameter N = 4)
2                    (input   logic clk,
3                     input   logic [N-1:0] d,
4                     output  logic [N-1:0] q);
5
6        always_ff @ (posedge clk)
7           q <= d;
8
9  endmodule
```

DesignTop/Nregister.sv [3]

9.5 Design Top

```
19  module DesignTop(
20      reset_n,
21      data_in,
22      input_clk,
23      NES_latch,
24      NES_clock,
25      segments
26  );
27
28
29    input wire  reset_n;
30    input wire  data_in;
31    input wire  input_clk;
32    output wire NES_latch;
33    output wire NES_clock;
34    output wire [6:0] segments;
35
36    wire  [22:0] divided;
37    wire  [7:0] read_data;
38    wire  [3:0] SYNTHESIZED_WIRE_0;
39
40
41
42
43
44  ClockDIvider   b2v_inst(
45      .clk(input_clk),
46      .reset_n(reset_n),
47      .divided(divided));
48
49
50  NESreader   b2v_inst2(
51      .data(data_in),
52      .clock(divided[10]),
53      .reset_n(reset_n),
54      .NESlatch(NES_latch),
55      .NESclock(NES_clock),
56      .read_data(read_data));
57
58
59  sevenseg b2v_inst3(
60      .data(SYNTHESIZED_WIRE_0),
61      .segments(segments));
62
63
64  encoder  b2v_inst6(
65      .a(read_data),
66      .q(SYNTHESIZED_WIRE_0));
67      defparam b2v_inst6.N = 8;
68
69
70    endmodule
71
```

DesignTop.v

10. Simulation Files

10.1 NES Reader

10.1.1 Counter

```
vsim counter

add wave *

force clk 0 @ 0, 1 @ 10 -r 20
force reset 0 @ 0, 1 @ 1, 0 @ 2

run 200
```

Figure 9.1.1(a): Counter test do file.

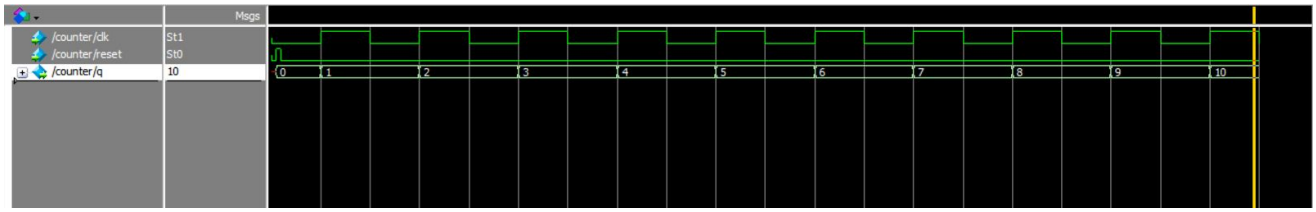

Figure 9.1.1(b): Waveforms from counter simulation.

For testing the counter, the do file creates 10 instances of a rising edge of a clock signal, and the output that displays the running total of outputs. As there were two different instances of the same counter with different parameters, the counter that was simulated held 8 bits of data compared to the much smaller 4 or single bits that were used in this project.

10.1.2 Comparator

```
vsim comparator

add wave *

force a 16#0000 @ 0, 16#0001 @ 10, 16#0002 @ 20, 16#0003 @ 30, 16#0004 @ 40, 16#0005 @ 50, 16#0006 @ 60
force a 16#0007 @ 70, 16#0008 @ 80, 16#0009 @ 90, 16#000A @ 100, 16#000B @ 110, 16#000C @ 120, 16#000D @ 130
force a 16#000E @ 140, 16#000F @ 150

run 160
```

Figure 10.1.2(a): Comparator test do file.

Figure 10.1.2(b): Waveform corresponding to the simulation of the comparator testing module.

Similar to the modularity induced in the counter module, the designed comparator module is also modular with controls being given to the parameters in the Quartus design. The standard SystemVerilog code, however, checks when the current state reaches 8, explaining the reason behind why the output signal is going high at that state.

10.1.3 NES Reader

```
vsim NESreader

add wave *

force reset_n 1 @ 0, 0 @ 10, 1 @ 20
force clock 1 @ 0, 0 @ 100 -r 200
force data 0 @ 0, 1 @ 5

run 20000
```

Figure 10.1.3(a): NES Reader do file

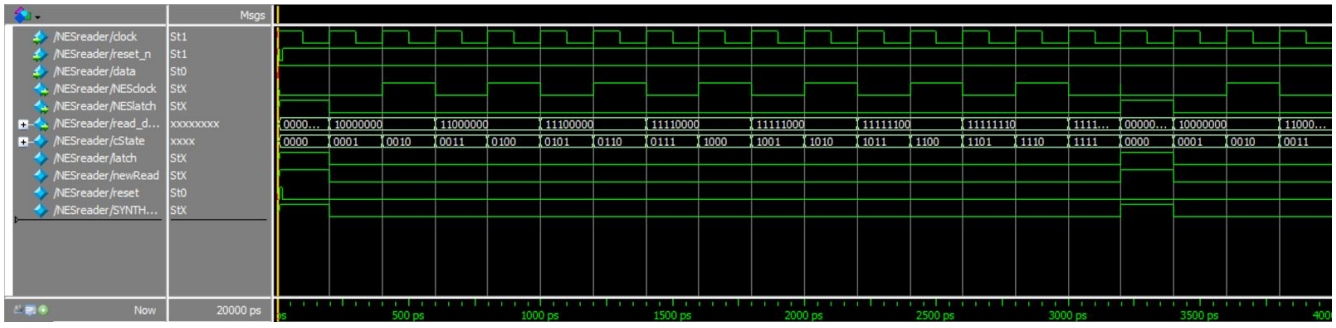

Figure 10.1.3(b): Waveforms produced from running the corresponding do file.

This simulation was to test whether the NES reader as a whole (counters, comparators, and decoders) were working as expected. The main goal of this module was to place the input data in the 8-bit bus, “filling” the bus from MSB to LSB, which was checked by leaving the input data high to see the individual bits filling up. This module also checked whether the NES latch and clock signals were being created at the appropriate times and states, while the data was being reset synchronously with the rising edge of the latch.

10.2 Encoder

```
vsim encoder

add wave *

force a 11111111 @ 0, 01111111 @ 10, 00111111 @ 20, 10111111 @ 30, 10011111 @ 40
force a 11011111 @ 50, 11001111 @ 60, 11101111 @ 70, 11100111 @ 80, 11110111 @ 90
force a 11110011 @ 100, 11111011 @ 110, 11111001 @ 120, 11111101 @ 130
force a 11111100 @ 140, 11111110 @ 150, 11111111 @ 160

run 170
```

Figure 10.2.1: Encoder Test Do file

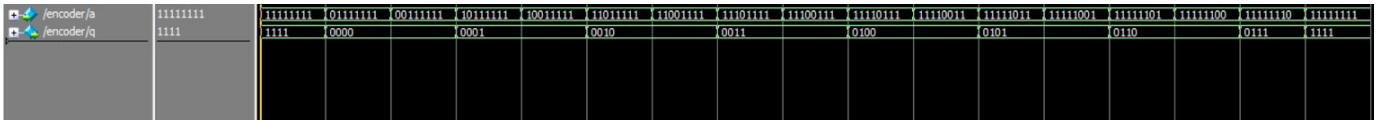

Figure 10.2.3: Encoder simulation waveform

Testing the encoder by switching a single bit in two different instances allowed for a more simple examination of the encoder module. The gist of what the do file is doing to test the outputs is checking to see if the register prioritizes the active low inputs in the correct order, and output the correct encoded system based on the first low bit the module receives.

10.3 Seven Segment Display Decoder

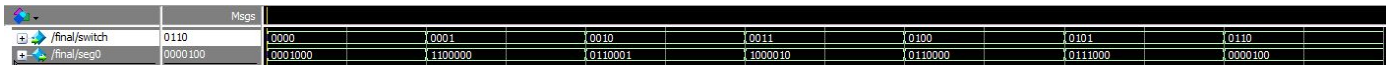

Figure 10.3.1: ModelSim waveform of the seven segment display. It shows the possible outputs for the different inputs.

10.4 Clock Divider

```
vsim ClockDIvider

add wave clk reset_n divided

force clk 1 @ 0, 0 @ 5 -r 10
force reset_n 1 @ 0, 0 @ 1, 1 @ 2
run 1000000
```

Figure 10.4.1: Clock Divider Test Do File

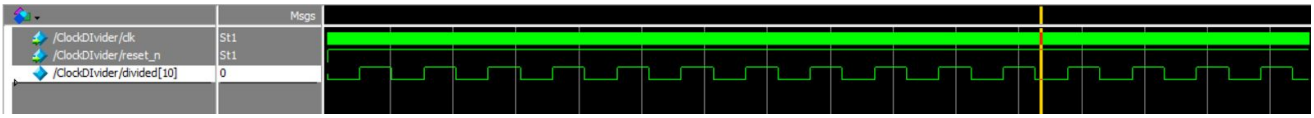

Figure 10.4.2: Clock divider simulation waveform

The divided frequency was confirmed to be relatively accurate by measuring the distance the period of the clk signal with the divided[10] signal, even with the arbitrary clock periods given in the do file above which were much shorter than the true period of a 10 MHz signal.

$$Period\ of\ clk:\ 10\ ps\ \rightarrow Frequency\ =\ 1x10^{11}\ Hz$$

$$Period\ of\ divided[10]:\ \Delta t\ =\ 26798080\ ps - 26777600\ ps\ = 20480\ ps$$

$$\rightarrow Frequency\ =\ \frac{1}{20480x10^{-12}}\ =\ 48828125\ Hz$$

$$\frac{1x10^{11}Hz}{48828125\ Hz}\ =\ \frac{10^{7}Hz}{x} \rightarrow x\ =\ 4882.8\ Hz\ \approx 5kHz$$

10.5 Display

```
vsim display

add wave *

force clk 0 @ 0, 1 @ 5 -r 10
force state 16#0 @ 0, 16#1 @ 10, 16#2 @ 20, 16#3 @ 30, 16#4 @ 40, 16#5 @ 50
force state 16#6 @ 60, 16#7 @ 70, 16#8 @ 80, 16#9 @ 90, 16#A @ 100, 16#B @ 110
force state 16#C @ 120, 16#D @ 130, 16#E @ 140, 16#F @ 150

run 160
```

Figure 10.5.1: Do file used to test the Display module.

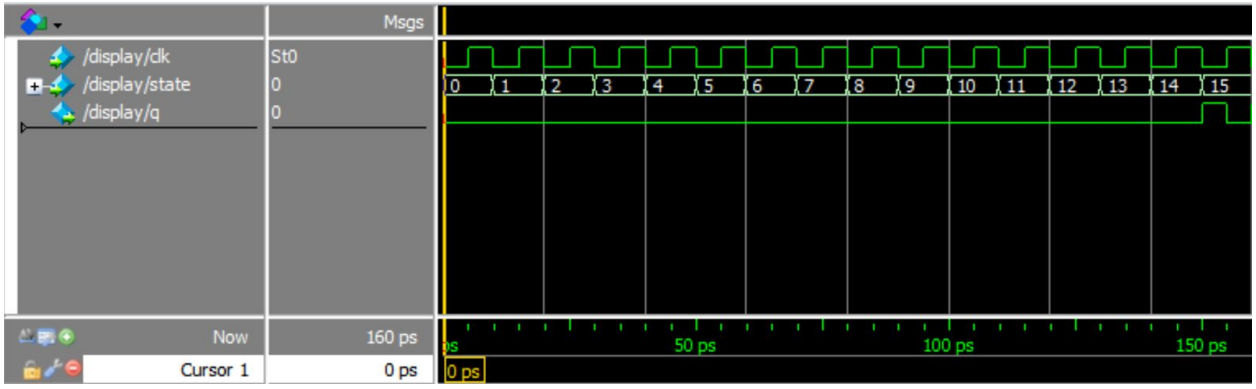

Figure 10.5.2: Corresponding waveform from the do file executed.

10.6 Nregister

```
vsim Nregister

add wave *

force clk 0 @ 0, 1 @ 5 -r 10
force d 16#0 @ 0, 16#1 @ 10, 16#2 @ 20, 16#3 @ 30, 16#4 @ 40, 16#5 @ 50
force d 16#6 @ 60, 16#7 @ 70, 16#8 @ 80, 16#9 @ 90, 16#A @ 100, 16#B @ 110
force d 16#C @ 120, 16#D @ 130, 16#E @ 140, 16#F @ 150

run 160
```

Figure 10.6.1: Do file simulating the function of the N-bit register. The default parameter of this module was N = 4 which explains why only 4 bits are being manipulated.

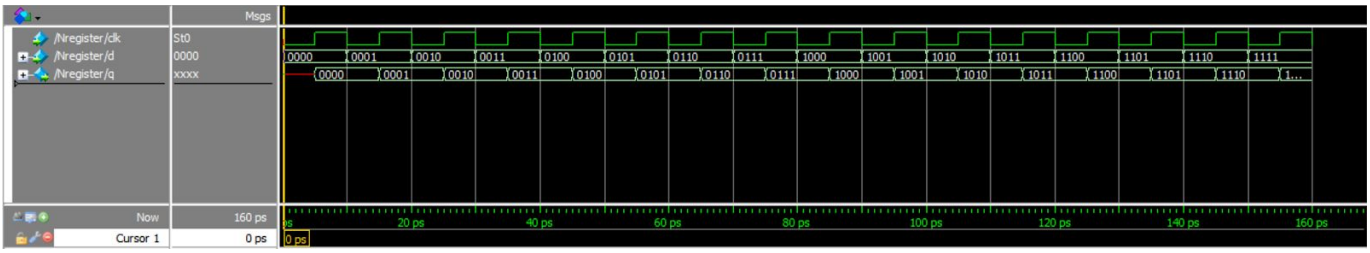

Figure 10.6.2: Derived waveforms from executing the do file above.

10.7 Top Level Design

10.7.1 NES-Encoder Test

```
vsim NES_enc

add wave data_in clock reset_n read_data NESlatch NESclock notes state

force clock 0 @ 0, 1 @ 20 -r 40
force reset_n 1 @ 0, 0 @ 5, 1 @ 10
force data_in 0 @ 0
force data_in 1 @ 620, 0 @ 700
force data_in 1 @ 1260, 0 @ 1420
force data_in 1 @ 1900, 0 @ 2140
force data_in 1 @ 2540, 0 @ 2860
force data_in 1 @ 3180, 0 @ 3580
force data_in 1 @ 3820, 0 @ 4300
force data_in 1 @ 4460, 0 @ 5020
force data_in 1 @ 5100, 0 @ 5700
force data_in 1 @ 5740

run 7660
```

Figure 10.7.1(a): Do file for checking if the NES reader interfaced with the encoder correctly

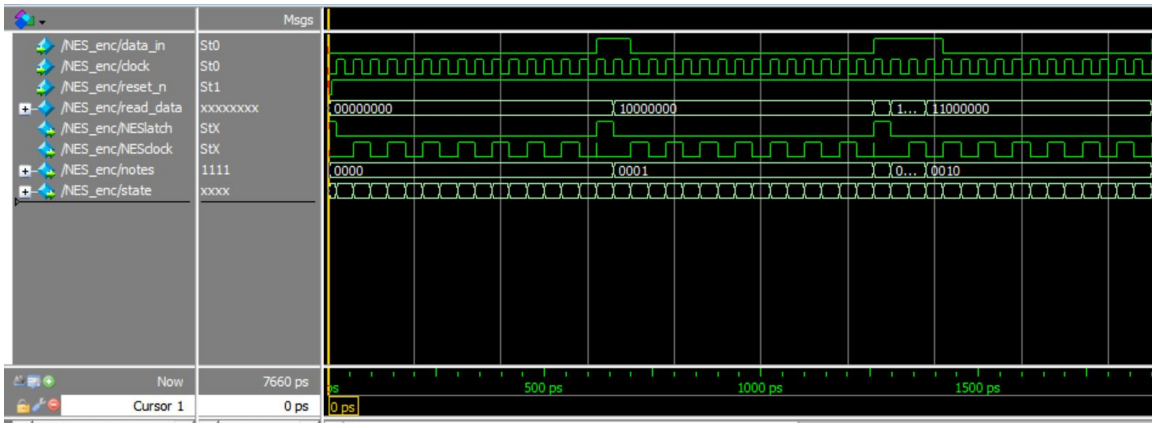

Figure 10.7.1(b): NES_encoder simulation which tests the interface between the data read by the NES reader and the decoder to put in the correct signals for the seven segment display

10.7.2 Full Top-Level Simulation

```
vsim DesignTop

add wave reset_n data_in divided read_data segments NES_clock NES_latch state disp disp_data

force divided 00000000000000000000000 @ 0, 00000000000010000000000 @ 20 -r 40
force reset_n 1 @ 0, 0 @ 5, 1 @ 10
force data_in 0 @ 0
force data_in 1 @ 620, 0 @ 700
force data_in 1 @ 1260, 0 @ 1420
force data_in 1 @ 1900, 0 @ 2140
force data_in 1 @ 2540, 0 @ 2860
force data_in 1 @ 3180, 0 @ 3580
force data_in 1 @ 3820, 0 @ 4300
force data_in 1 @ 4460, 0 @ 5020
force data_in 1 @ 5100, 0 @ 5700
force data_in 1 @ 5740

run 7660
```

Figure 10.7.2(a): Do file doing a test of the full top-level design.

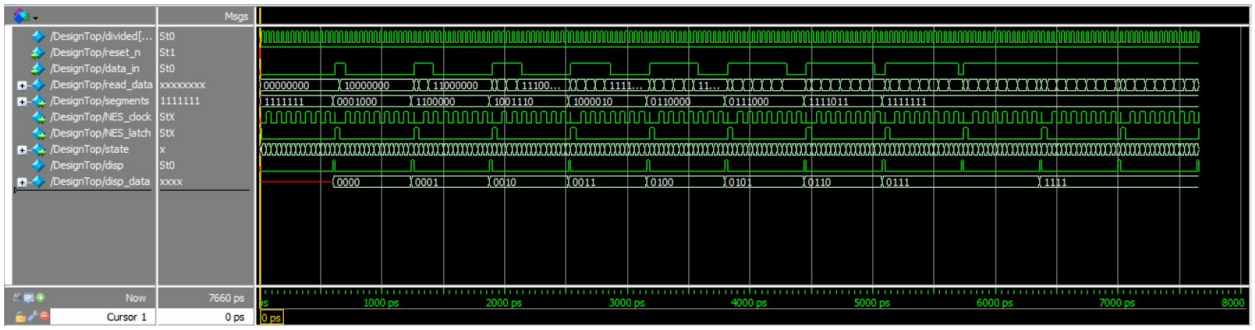

Figure 10.7.2(b): Waveforms corresponding to the simulated do file above.

This simulation was to show the full interaction between input data, the functionality of the NES reader, the encoder changing the read data into a simpler form, and the seven segment display showing the note corresponding to the note that had just been pressed. As testing the functionality of each module was the goal in mind, the data inputs that were forced

were chosen to mimic the user activating all buttons on the board, then all but the A button, B button, and so on.

11. Physical Implementation

Figure 12.1: Seven segment output ‘A’

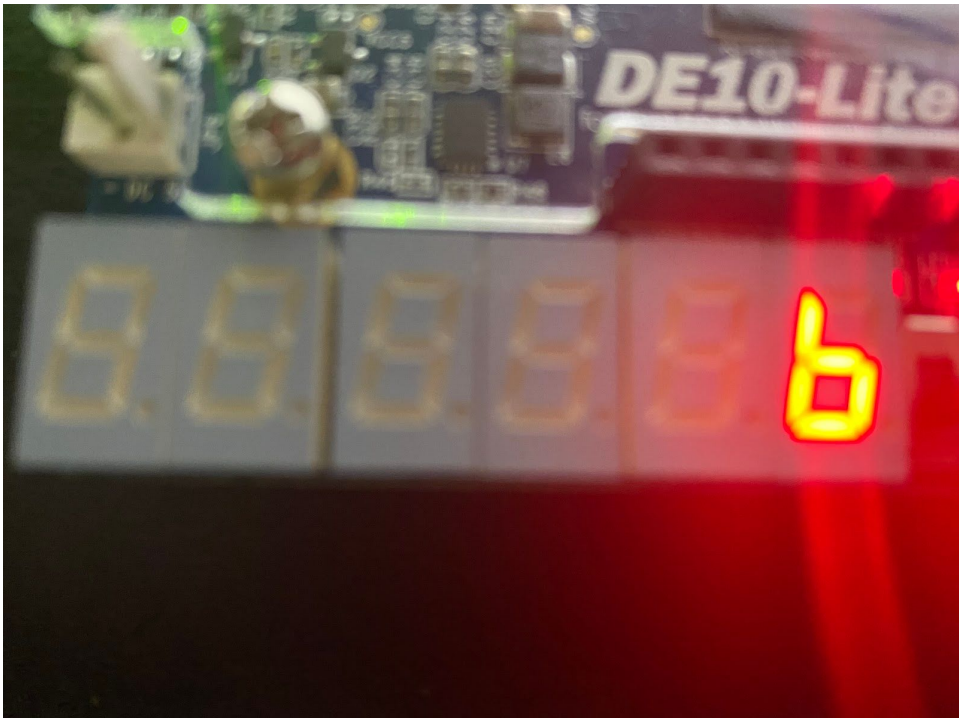

Figure 12.1: Seven segment output ‘B’

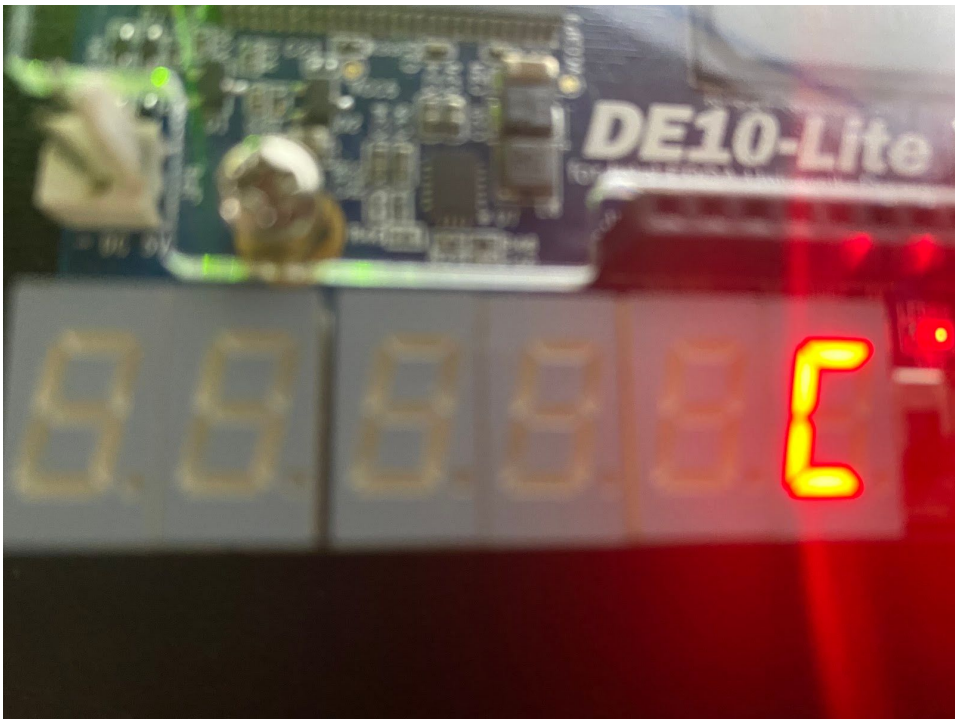

Figure 12.1: Seven segment output ‘C’

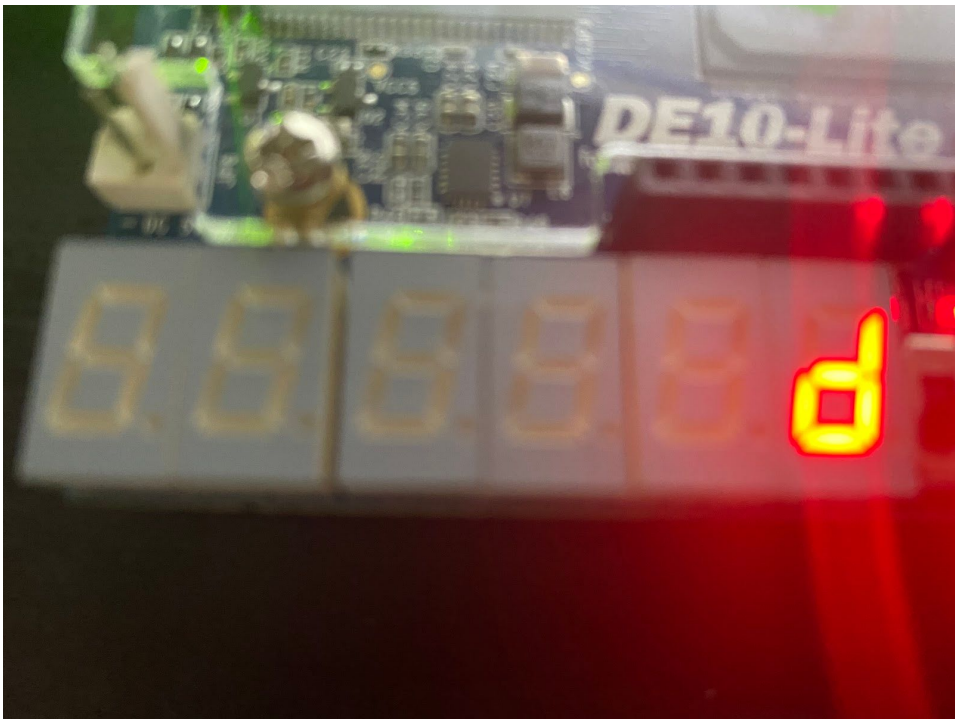

Figure 12.1: Seven segment output ‘D’

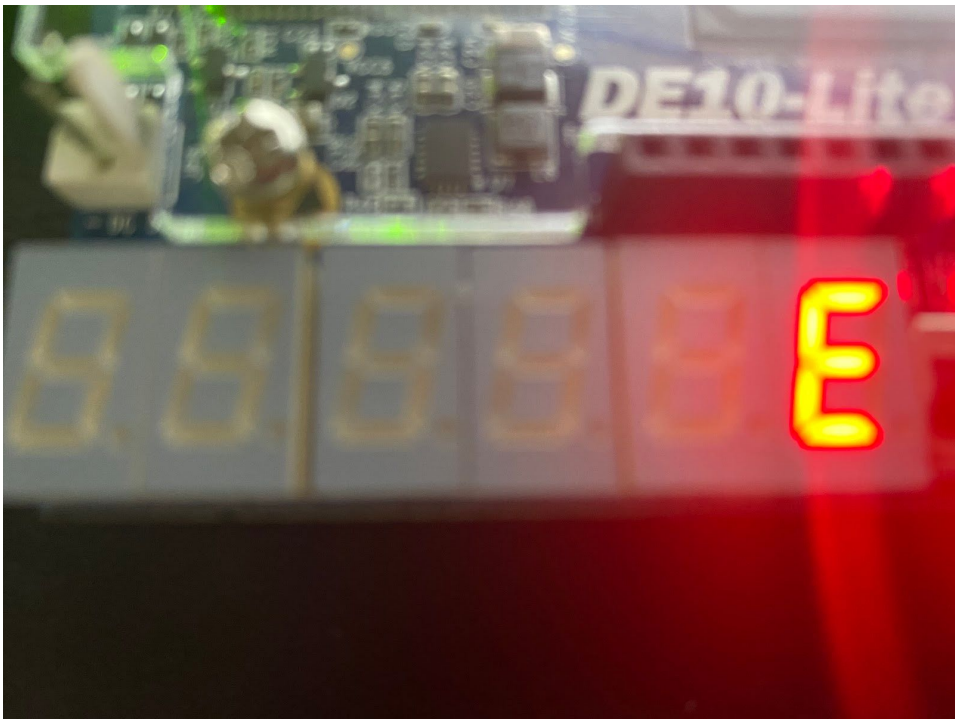

Figure 12.1: Seven segment output ‘E’

Figure 12.1: Seven segment output ‘F’

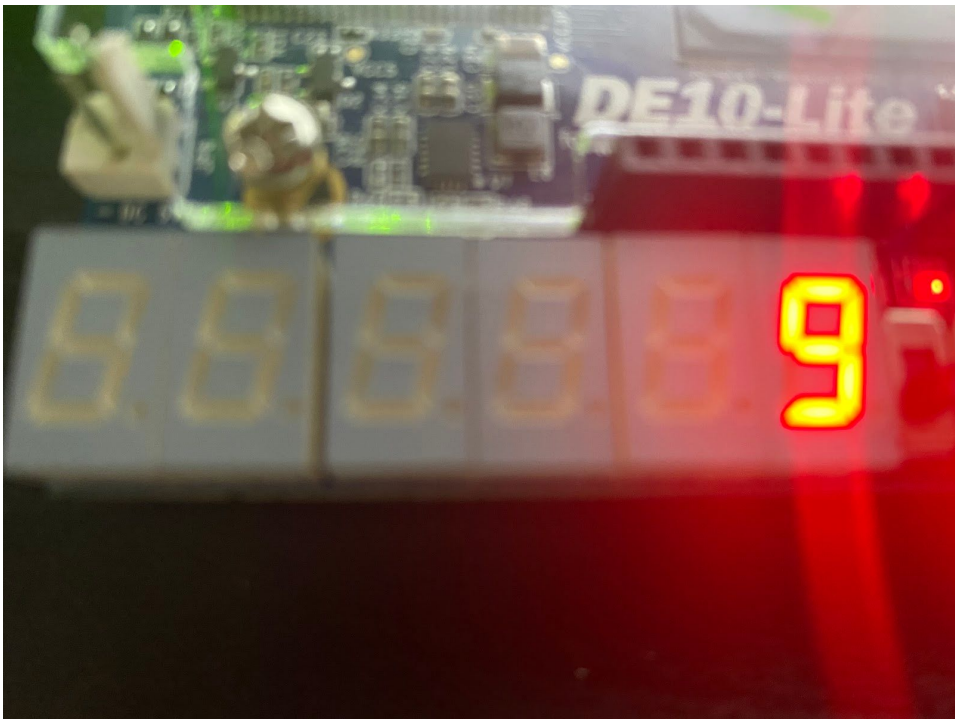

Figure 12.1: Seven segment output 'G'

12. Resources

[1] B. Chase , B. Excel, & J. Morgan, "Goblin Ranger Game." https://spaces.usu.edu/plugins/viewsource/viewpagesrc.action?pageId=53511846, N.D.
[2] J. Corleto, "NES Controller Interface with an Arduino UNO." https://www.allaboutcircuits.com/projects/nes-controller-interface-with-an-arduino-uno/, 2016.
[3] D. Harris & S. Harris, "Digital Design and Computer Architecture." https://ebookcentral.proquest.com/lib/osu/detail.action?docID=980017, 2013.
[4] M. Shuman, Sample NES Reader SystemVerilog code, 2020.
- Note: example code provided on canvas for the NES reader was referenced for the creation of the NES reader module. Though some of the elements of the given module had been changed, the NES data decoder was taken from the example code.