ECE271, Final Project

Ben Adams, Grant Haines, Benjiman Walsh

December 5, 2019

Contents

1 Introduction

The purpose of this project is to create a digital logic design that uses various input modules with various output modules.

2 High Level Descriptions

3 Controller Descriptions

This section is meant to provide a brief but thorough low-level view of the operations of our chosen input devices.

3.1 NES Controller

The Nintendo Entertainment System (NES) first became available in America in 1985 and revolutionized society as the first accessible home video game system. NES controllers work by receiving "clock" and "latch" signals from the NES console and transmitting a data signal to the console. NES controllers use a shift register to store all of the controller's button data when the console sends the latch signal (As in figure 1). Each successive clock signal shifts the controller register down and the controller's data wire outputs a value that represents the next button's signal (See figure 2).

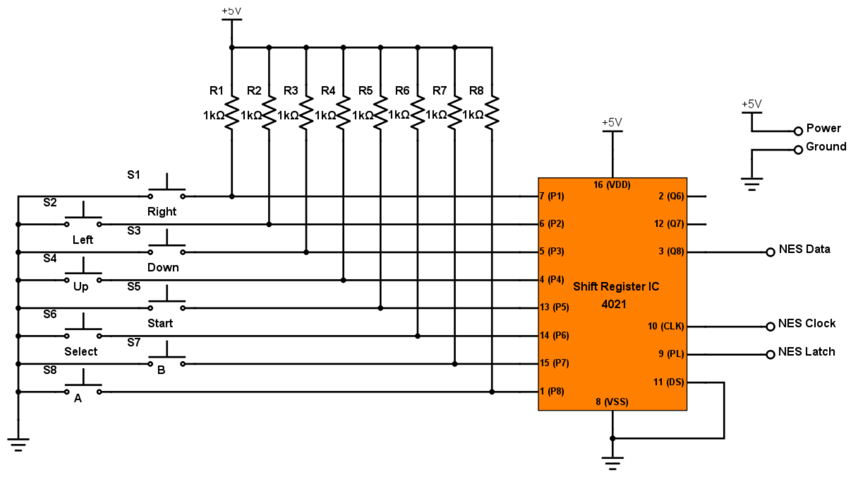

Figure 1: "Button Mashing" on the NES

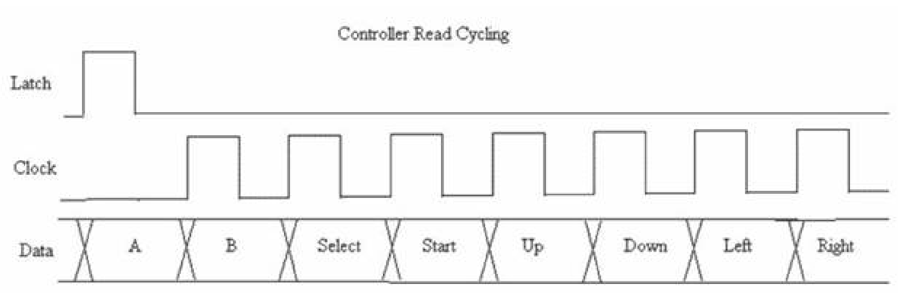

Figure 2: "Button Mashing" on the NES

The NES controller decoder SystemVerilog module was provided for us in the course materials.

4 HDL Components

4.1 Top Module

INSERT TOP MODULE DIAGRAM HERE
INSERT TOP MODULE DESCRIPTION HERE

4.2 VGA Output

INSERT VGA OUTPUT DIAGRAM HERE
Input: The VGA module takes a 50MHz clock signal, a reset signal, and three button signals for the red, green, and blue inputs.

Output: The VGA module outputs a vSync and an hSync signal, and three 4-bit values for the red, green, and blue display colors.

Description: This module is designed to output a 640x480 resolution VGA signal, allowing it to display an RGB color value on a screen. The hSync signal tells a computer monitor how quickly to update each column of the screen, while the vSync signal tells it how quickly to update each row. The color inputs work by having each button press increment a 4-bit counter that goes to the appropriate color output.

4.2.1 VGA hCounterComp

INSERT DIAGRAM HERE

Inputs: A 25MHz clock signal, a 50MHz clock signal, and a reset signal. Outputs: The horizontal sync rate for the VGA output and a signal indicating that it is in the display area for the screen.

Description: The hCounterComp creates the hSync signal for the VGA driver, as well as indicating that the signal is within the display area so that the RGB values can be communicated to the screen. The hSync signal travels through the horizontal row of pixels on the screen, displaying color when appropriate.

4.2.2 VGA vCounterComp

INSERT DIAGRAM HERE

Inputs: The hSync signal as a clock, a 50MHz clock signal, and a reset signal. Outputs: The vertical sync rate for the VGA output and a signal indicating that it is in the display area for the screen.

Description: The vCounterComp module generates the vSync signal for the VGA driver, which tells it when to move on to the next row of pixels, as well as telling it when it is within the displayable area.

4.2.3 VGA counter

INSERT DIAGRAM HERE

Inputs: A clock signal and a reset signal. Outputs: An N-bit binary value.

Description: This counter adds one to its output value each rising clock edge.

4.2.4 VGA displayMux

INSERT DIAGRAM HERE

Inputs: A select signal and red, green, and blue binary input values. Outputs: Either the inputted RGB values or 0 values.

Description: The VGA displayMux decides whether to send the inputted RGB values to the screen or not to, depending on whether or not we are within the displayable area, as given by the hSync and vSync modules.

4.2.5 VGA comparator

INSERT DIAGRAM HERE

Inputs: A select signal and red, green, and blue binary input values. Outputs: Either the inputted RGB values or 0 values.

Description: The VGA displayMux decides whether to send the inputted RGB values to the screen or not to, depending on whether or not we are within the displayable area, as given by the hSync and vSync modules.

4.2.6 VGA synchronizer

INSERT DIAGRAM HERE

Inputs: A clock signal and a 1-bit data value. Outputs: A 1-bit data value.

Description: The synchronizer takes asynchronous inputs and syncs them to the clock edge.

4.2.7 clockDivBy2

INSERT DIAGRAM HERE

Inputs: A clock signal and a reset signal. Outputs: A clock signal half as fast as the input clock signal.

Description: This module takes in a clock input and divides its frequency by 2. For example, a 50MHz clock input would become a 25MHz clock input.

5 Appendix

5.1 Source Code

5.1.1 NES Controller Reader

```
module NesReader(
  input logic dataYellow,
  input logic clock,
  input logic reset_n,
  output logic latchOrange,
  output logic clockRed,
  output logic up,
  output logic down,
  output logic left,
  output logic right,
  output logic start,
  output logic select,
  output logic a,
  output logic b
  );
  logic [3:0] count;

  Counter4 matt_i1(
    .clk              (clock),
    .reset_n          (reset_n),
    .count            (count)
  );

  NesClockStateDecoder matt_i2(
    .controllerState  (count),
    .nesClock         (clockRed)
  );

  NesLatchStateDecoder matt_i3 (
    .controllerState  (count),
    .nesLatch         (latchOrange)
  );

  NesDataReceiverDecoder matt_i4 (
    .dataYellow       (dataYellow),
    .reset_n           (reset_n),
    .controllerState  (count),
    .readButtons      ({a, b, select, start, up, down, left, right})
  );
endmodule


module Counter4(
  input logic clk, reset_n,
  output logic [3:0] count);

  always_ff @ (posedge clk, negedge reset_n)
    if(!reset_n) count <= 4'b0;
    else count <= count + 1;
endmodule


module NesLatchStateDecoder(
  input logic [3:0] controllerState,
  output logic nesLatch);

  always_comb
    case(controllerState)
      4'h0: nesLatch = 1;
      default: nesLatch = 0;
    endcase
endmodule


module NesClockStateDecoder(
  input logic [3:0] controllerState,
  output logic nesClock);

  always_comb
    case (controllerState)
      4'h2: nesClock = 1;
      4'h4: nesClock = 1;
      4'h6: nesClock = 1;
      4'h8: nesClock = 1;
      4'ha: nesClock = 1;
      4'hC: nesClock = 1;
      4'hE: nesClock = 1;
      default: nesClock = 0;
    endcase
endmodule


module NesDataReceiverDecoder(
  input logic dataYellow,
  input logic reset_n,
  input logic [3:0] controllerState,
  output logic [7:0] readButtons);

  always_ff @ (posedge controllerState[0], negedge reset_n)
    if(!reset_n) readButtons <= 8'b0;
    else case(controllerState[3:0])
      4'h1: readButtons[7] <= dataYellow;        //a button
      4'h3: readButtons[6] <= dataYellow;        //b button
      4'h5: readButtons[5] <= dataYellow;        //select button
      4'h7: readButtons[4] <= dataYellow;        //start button
      4'h9: readButtons[3] <= dataYellow;        //up button
      4'hB: readButtons[2] <= dataYellow;        //down button
      4'hD: readButtons[1] <= dataYellow;        //left button
      4'hF: readButtons[0] <= dataYellow;        //right button
      default: readButtons <= readButtons;
    endcase
endmodule
```

5.1.2 Square Wave Generator

```
module periodTime(input logic clk,
                  input logic [2:0] data,
                  output logic q);

int compareNumber;
int count;
```

```
always_comb
        case(data)
            0: compareNumber = 6400;    // just mod input clock until audio spectrum periods
            1: compareNumber = 3200;    // one octave
            2: compareNumber = 1600;
            3: compareNumber = 8000;

            4: compareNumber = 4000;
            5: compareNumber = 2000;
            6: compareNumber = 1000;
            7: compareNumber = 500;        // consider adding default case
        endcase

always_ff @(posedge clk)
    begin
        if (count >= compareNumber)    // could modify to not restart notes when changed
            count <= 0;
        else
            count <= count +1;
    end

always_comb
    begin
            if( count < compareNumber)
                q = (count > compareNumber/2);     //assigns output with initial low
            else
                q = 0;
    end

endmodule
```

5.1.3 VGA Output

```
module vgaOutput
                (input clock50MHz,
                 input inReset,
                 input inRed,
                 input inGreen,
                 input inBlue,
                 output hSync,
                 output vSync,
                 output [3:0] outRed, outGreen, outBlue);

        vga_counter #(.N(4)) redCounter (
                .clk(inRed),
                .reset(inReset),
                .q(redCount)
        );

        vga_counter #(.N(4)) greenCounter (
                .clk(inGreen),
                .reset(inReset),
                .q(greenCount)
        );

        vga_counter #(.N(4)) blueCounter (
                .clk(inBlue),
                .reset(inReset),
                .q(blueCount)
        );

        clockDivBy2 clockDivider(
                .clock50MHz(clock50MHz),
                .inReset(~inReset),
                .outClock(clock25MHz)
        );

        vga_hCounterComp #(.a(10'd96), .b(10'd48), .c(10'd640), .d(10'd16)) hSyncCounter (
                .inClock(clock25MHz),
                .clock50MHz(clock50MHz),
                .inReset(~inReset),
                .signal(hSync),
                .displaySignal(hSignal)
        );

        clockDivBy2 syncDivider(
                .clock50MHz(hSync),
                .inReset(~inReset),
                .outClock(hClock)
        );

        vga_vCounterComp #(.a(10'd2), .b(10'd33), .c(10'd480), .d(10'd10)) vSyncCounter (
                .inClock(hClock),
                .clock50MHz(clock50MHz),
                .inReset(~inReset),
                .signal(vSync),
                .displaySignal(vSignal)
        );

        vga_displayMux display (
                .select(hSignal & vSignal),
                .inRed(redCount),
                .inGreen(greenCount),
                .inBlue(blueCount),
                .outRed(outRed),
                .outGreen(outGreen),
                .outBlue(outBlue)
        );

endmodule
```

5.1.4 VGA hCounterComp

```
module vga_hCounterComp #(parameter a = 10, b = 10, c = 10, d = 10)
                (input inClock,
                 input clock50MHz,
                 input inReset,
                 output signal,
                 output displaySignal);

        logic [9:0] currentCount;

        vga_counter #(.N(10)) count1 (
                .clk(inClock),
                .reset(cntReset | inReset),
                .q(currentCount)
        );
        vga_comparator #(.N(10)) aTob (
                .a(currentCount),
                .b(a),
                .gte(signal)
        );

        vga_comparator #(.N(10)) bToc (
                .a(currentCount),
                .b(a + b),
                .gte(disp1)
        );

        vga_comparator #(.N(10)) cToD (
                .a(currentCount),
                .b(a + b + c),
                .lt(disp2)
        );

        vga_comparator #(.N(10)) reset (
                .a(currentCount),
                .b(a + b + c + d),
                .eq(compSignal)
        );

        vga_synchronizer sync1 (
                .clk(clock50MHz),
                .d(compSignal),
                .q(cntReset)
        );

        assign displaySignal = disp1 & disp2;

endmodule
```

5.1.5 VGA vCounterComp

```
module vga_vCounterComp #(parameter a = 10, b = 10, c = 10, d = 10)
                (input inClock,
                 input clock50MHz,
                 input inReset,
                 output signal,
                 output displaySignal);

        logic [9:0] currentCount;

        vga_counter #(.N(10)) count1 (
                .clk(inClock),
                .reset(cntReset | inReset),
                .q(currentCount)
        );

        vga_comparator #(.N(10)) aTob (
                .a(currentCount),
                .b(a),
                .gte(signal)
        );

        vga_comparator #(.N(10)) bToc (
                .a(currentCount),
                .b(a + b),
                .gte(disp1)
        );

        vga_comparator #(.N(10)) cToD (
                .a(currentCount),
                .b(a + b + c),
                .lt(disp2)
        );

        vga_comparator #(.N(10)) reset (
                .a(currentCount),
                .b(a + b + c + d),
                .eq(compSignal)
        );

        vga_synchronizer sync1 (
                .clk(clock50MHz),
                .d(compSignal),
                .q(cntReset)
        );

        assign displaySignal = disp1 & disp2;

endmodule
```

5.1.6 VGA counter

```
module vga_counter #(parameter N = 4)
                (input logic clk,
                 input logic reset,
                 output logic [N-1:0] q);

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

endmodule
```

5.1.7 VGA displayMux

```
module vga_counter #(parameter N = 4)
                (input logic clk,
                 input logic reset,
                 output logic [N-1:0] q);

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

endmodule
```

5.1.8 VGA comparator

```
module vga_comparator #(parameter N = 1)
                (input logic [N-1:0] a, b,
                 output logic eq, neq, lt, lte, gt, gte);
```

```
        assign eq        = (a == b);
        assign neq        = (a != b);
        assign lt        = (a < b);
        assign lte        = (a <= b);
        assign gt        = (a > b);
        assign gte        = (a >= b);

endmodule
```

5.1.9 VGA synchronizer

```
module vga_synchronizer
                (input logic clk,
                 input logic d,
                 output logic q);

        logic n1;

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

endmodule
```

5.1.10 clockDivBy2

```
module clockDivBy2
        (input clock50MHz,
         input inReset,
         output reg outClock);

        always @(posedge clock50MHz) begin
                if (inReset)
                          outClock <= 1'b0;
                else
                          outClock <= ~outClock;
        end

endmodule
```

5.2 Simulation Results

5.2.1 NES Controller Reader

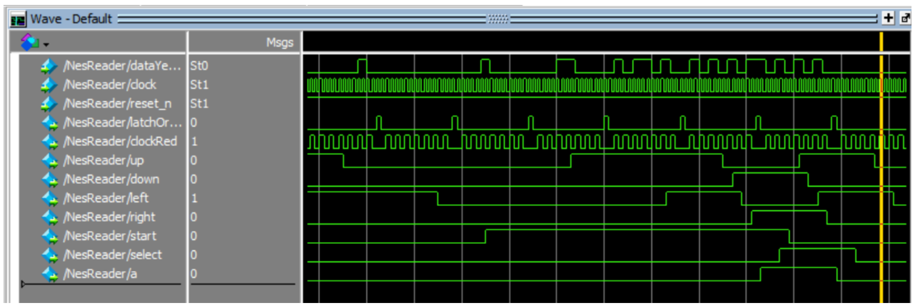

Figure 3: "Button Mashing" on the NES

At first, I wanted to test the NES controller reader by just simulating a bunch of random inputs as seen in Figure 3. I remembered the NES game CONTRA had a cheat code that involved most of the controller's buttons (all but SELECT). The "Contra Code" was then simulated I'm gonna rewrite this.

Figure 4: CONTRA screenshot

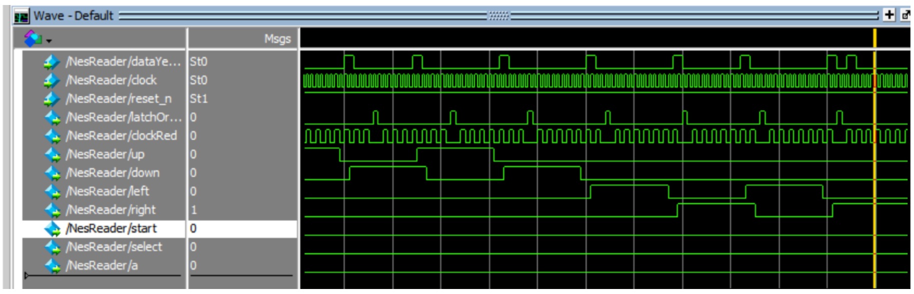

Figure 5: Simulating the "Contra Code"

```
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    1 {80 ps} , 0 {100 ps} , 0 {120 ps} , 0 {140 ps} #up
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 1 {100 ps} , 0 {120 ps} , 0 {140 ps} #down
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    1 {80 ps} , 0 {100 ps} , 0 {120 ps} , 0 {140 ps} #up
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 1 {100 ps} , 0 {120 ps} , 0 {140 ps} #down
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 1 {120 ps} , 0 {140 ps} #left
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 0 {120 ps} , 1 {140 ps} #right
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 1 {120 ps} , 0 {140 ps} #left
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 0 {120 ps} , 1 {140 ps} #right
force -freeze sim:/NesReader/dataYellow 0 0, 1 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 0 {120 ps} , 0 {140 ps} #b
force -freeze sim:/NesReader/dataYellow 1 0, 0 {20 ps} , 0 {40 ps} , 0 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 0 {120 ps} , 0 {140 ps} #a
```

```
force -freeze sim:/NesReader/dataYellow 0 0, 0 {20 ps} , 0 {40 ps} , 1 {60 ps} ,
    0 {80 ps} , 0 {100 ps} , 0 {120 ps} , 0 {140 ps} start
```

5.2.2 Square Wave Generator

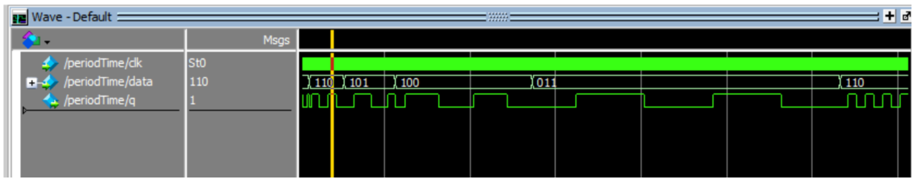

Figure 6: Simulating button inputs to control the square wave oscillator