ECE 271 Final Project

Team Members: David Mora, Nathan Stageberg, Luke Goldsworthy, and Kathleen Xiong

December 2019

Table of Contents

1. Introduction

The purpose of this project is to mimic a robotic arm. This robotic arm is represented with three DC motors. The reason why these motors could represent a robotic arm is because the motors show the direction and speed that the robotic arm would have. Each motor will have a direction that it spins and speed of the spin; therefore, the design project can be split up into two parts, the implementation of the direction and speed. The DC motors will either spin clockwise or counter clockwise and will only do so when the enable is at a logic high. The enable allows the motor to start spinning in whatever direction is indicated. Then the motor speed is chosen for all of the motors.

The first part of this project describes how direction will be implemented. Whether they turn clockwise or counter clockwise is controlled with either the PS/2 Keyboard or the Nes Controller. The PS/2 keyboard and Nes Controller will control the three motors to turn whichever way that's indicated when specified buttons are pressed. However, it's designed to only register that either the PS/2 keyboard or the Nes controller is powering it, and never at the same time. It's designed this way so that the two inputs won't have any collisions when data is read from two different sources. The Nes Controller will turn motor 1 when buttons up and down are pressed. Then motor 2 will spin when left and right are pressed; and lastly, motor 3 will spin when a and b are pressed. Similarly, the PS/2 keyboard will spin a certain motor when certain buttons are pressed. The PS/2 keyboard will use 6 keys. Two keys will control the direction of motor rotation for each of the three axes of motion.

Then the second part of this project is determining the speed. The speed of the motors are controlled by an Analog Potentiometer. When the Analog potentiometer is dialed higher, it increases the speed of the motors which will speed up the spinning. In order to actually know how fast or slow the motors are running, the speed is displayed onto a seven segment display and a VGA output. As the speed increases by the Analog Potentiometer, the Seven segment display will output the value in hexadecimal digits, and the VGA output will display the speed going from red to blue (red being the lowest and blue being the highest).

2. High Level Description

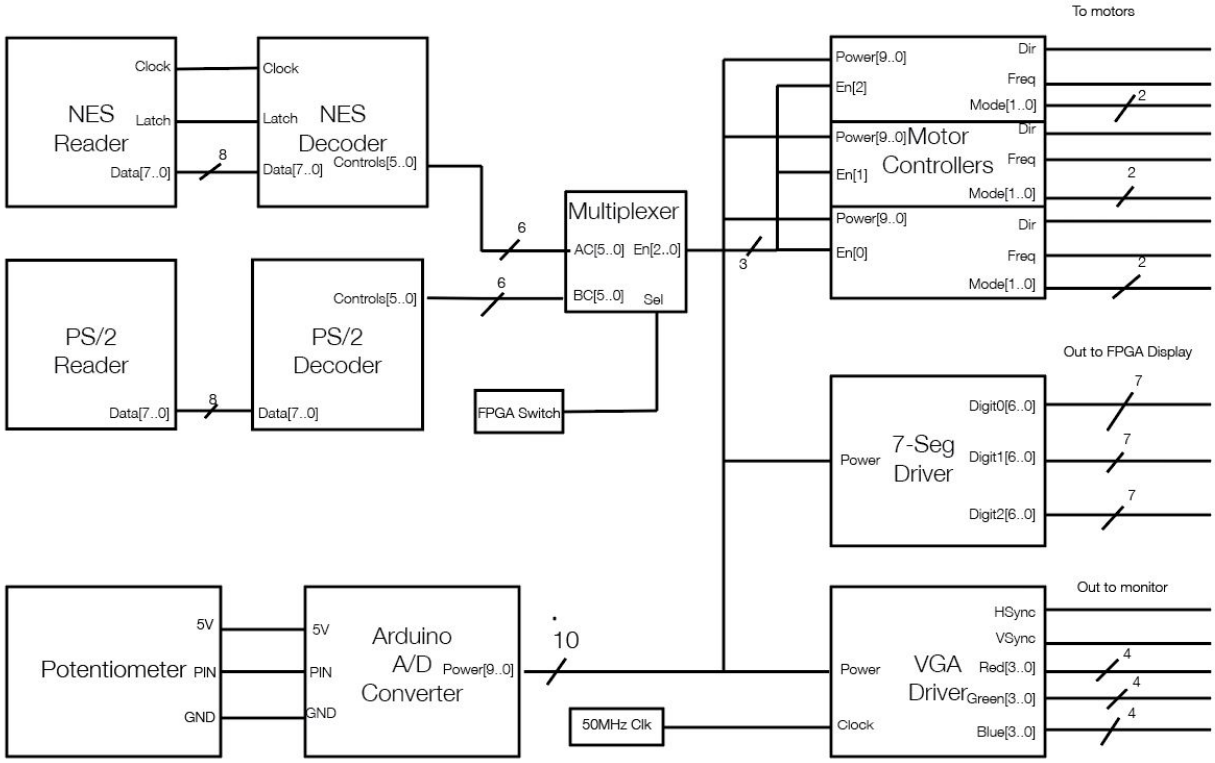


Figure 1: Top Level Design diagram

The figure above shows the high-level, functional view of the design. The motor control from two of our inputs are passed through a multiplexer, so that the user can either command the motors with the NES controller, or the keyboard, but not both simultaneously. Both input blocks decode the data received from the peripheral and produce signals that the motor control logic an understand. Our other input is a potentiometer, which controls the speed at which the motors rotate. To connect the potentiometer, we relied on an analog/digital converter. We tried using the internal one provided by the MAX10 FPGA, but it was problematic, so we used an Arduino Mega board instead. To keep the design simple, a single speed value is used by all three motors.

The value of the potentiometer is also displayed in hexadecimal base in the 7-segment displays of the DE-10 Lite board in addition to being represented as a solid color in a VGA display, shifting from red to blue as the potentiometer value increases. The NES and PS/2 controllers determine which motors are active and when.

3. Controller Descriptions

3.1 Inputs

The following three descriptions are descriptions of all inputs for this project. The project uses the NES controller, PS/2 Keyboard, and the Analog Potentiometer as inputs for the project as a whole.

3.1.1 NES Controller Description

The hardware of the NES controller can be broken down into two parts. The NES controller is a game controller with buttons such as a, b, start, and so on. These buttons are then programmed to give an input signal into our design using both a NES reader and a NES decoder.

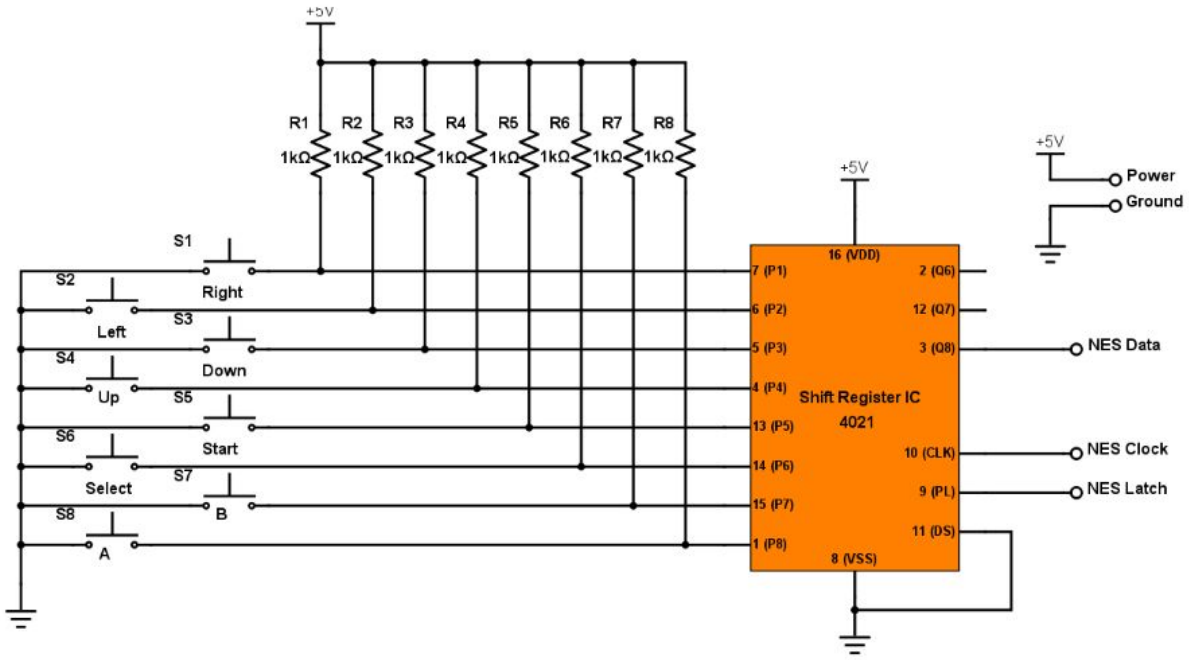


Figure 2: NES Controller schematic[1]

First the NES reader takes in the inputs from the actual hardware and then passes it onto whatever the inputs would control. The NES reader block's job is to allow other outputs to be able to read the input signals from the NES controller. In the figure above it demonstrates the function of the NES reader. The main idea is that when the NES controller's button is pressed, the signal is sent into the NES reader as the NES data. Then this Data follows the clock cycle and is pass through when the latch is pulled. This latch also is in sync with the clock signal and will generate a latch to whatever input is pressed and create a signal. In the figure, the latches are represented in far left. When these latches are connected it takes time for the them disconnect

[1] https://www.allaboutcircuits.com/projects/nes-controller-interface-with-an-arduino-uno/

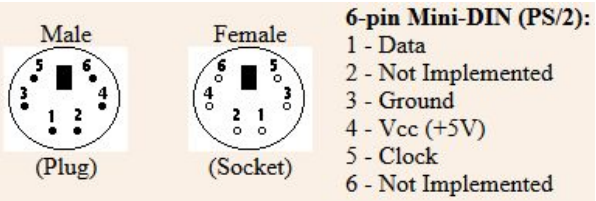
and connect again; therefore, in the NES reader will stores the data until the latch is in sync with the clock and it can register a new input for the data. This explains how the NES controller can press a single button and it will keep the motor running even when the button is not pressed anymore.This is the first part in demonstrating the inter-workings for the NES controller.

The second part of the NES controller, specific to this design is the NES decoder. This Decoder is split up into three decoders, one for every motor that needs to be controlled. For motor one, we can assign the down and up to control the direction of the motor. This is done by making sure that when the up or down button is pressed on the NES controller, it will output a logic 1 into the first motor. This logic block is done using if statements to make sure that when up is pressed then there is a logic one output, and when down is pressed then there is also a logic one outputted. Then the last statement is an else statement that will generate a logic 0, or low output for any other case. So, if another other is button is pressed, if no button was pressed, or if both buttons are pressed at the same time, it will give motor 1 a logic zero output. The same logic is used for the rest of the NES decoders for the other motors. The only difference is that each motor is assigned different buttons to control the direction and enable.

3.1.2 PS/2 Keyboard Description

The top level design of the PS/2 keyboard module involves two blocks, the 'keyboard' module (which will be referred to as the keyboard analyzer) which takes in the actual physical keyboard's clock and data signals via the PS/2 port, which then sends the 8-bit value that represents a key's make code (represented in hexadecimal) to the decoder. For our purposes, we are using only 6 keys of the full keyboard, aimed to control three different stepper motors by enabling them and determining direction. The decoder will enable the X-axis motor if either the A or D keys are pressed, the Y-axis if either the W or S keys are pressed, or the Z-axis if the left and right arrow keys are pressed. A, W, and Left-Arrow keys make their specific motor rotate counter clockwise, while the D, S, and Right-Arrow keys make the motors turn clockwise. If none of those buttons are pushed, there will be no rotation for any motor.

Pins of the PS/2 Plug	Where they connect to
1-Data	Keyboard Analyzer
2-N/A	Nothing
3-GND	Ground



4-Vcc	5v source of the FPGA
5-Clock	Keyboard Analyzer
6-N/A	Nothing

Figure 3: A list of what logic values connect to different pins of the PS/2 port.

Figure 4: Visual of the PS/2 Male and Female ports, and their connections[2]

Within the keyboard analyzer is an 11-bit register (that functions similarly to a shift register) that takes in the 11 bits of data over the clock input's rising edge, and then outputting the 8 data bits on the falling edge, which is taken to the decoder. The decoder takes this input in as a bus and, using SystemVerilog, directly compares their value to the hex values of the 6 keys we are using, and will change output values as mentioned above.

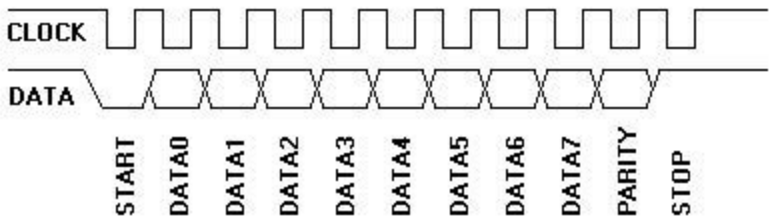


Figure 5: Waveforms of a PS/2 Keyboard transferring Data to a device (such as a PC, or the FPGA in our case)[3]

Eight of the eleven bits from the data line represent the make code of the key being pushed, being the 2nd through 9th data value taken. The first data bit is always the least significant bit, so in the diagram above, DATA0 is the LSB. The 10th bit is the Parity bit, which is used in detecting errors, but overall not critical to our implementation. The last bit is the stop bit, which is always 1, telling the device taking the keyboard input to stop.

[2] http://www.burtonsys.com/ps2_chapweske.htm
[3] http://www.burtonsys.com/ps2_chapweske.htm

3.1.3 Analog Potentiometer Description

The analog potentiometer is obviously analog, which generates special considerations for implementation. The output is generated by the middle pin, with the left pin serving as a ground and the right pin serving as VCC. As the potentiometer's knob is twisted, the middle pin's current source shifts from closer to one pin than the other, generating a value in between the two.

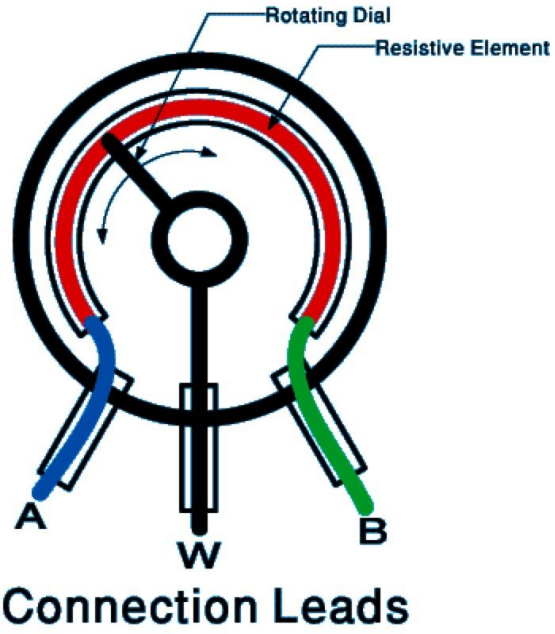


Figure 6: Analog Potentiometer pins and demonstration of voltage

For this project, all input was digital, so we converted the analog input to a digital signal with bit width 10. This process will be covered in the HDL Components section. The signal updates on time increments relative to the size of the value being transmitted. For example, when values close to 0 are being updated they update at a rate around 20 Hz, but values close to 1023 only update at around 0.55 Hz.

3.2 Outputs

The next three descriptions are brief explanations of all of the outputs used for the project. The project uses the 7-Segment display, VGA output, and the DC motors as the outputs.

3.2.1 DE10-Lite 7-Segment Display Description

Incoming data is accepted at a bit width of 4, and is output to a seven-segment display, representing a hexadecimal value. This input comes from within the DE-10 Lite board, and the seven-segment display is active low.

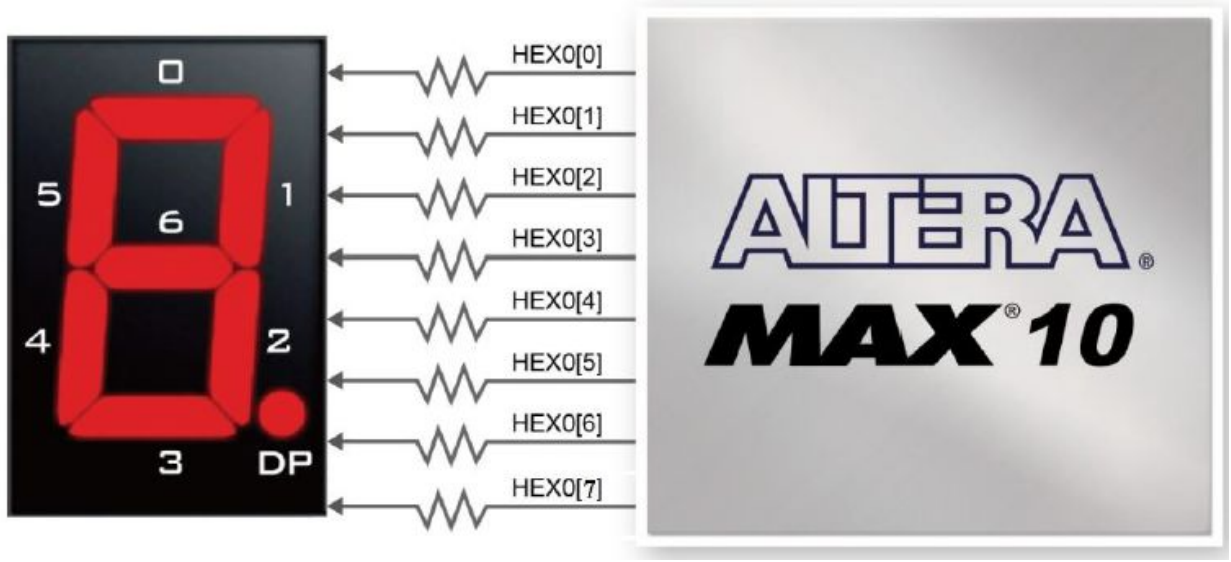


Figure 7: Details of DE-10 Lite output to display

3.2.2 VGA Output Description

The incoming data from the potentiometer is fed into the VGA logic block along with a 50 MHz clock to generate the 14 VGA output bits. Two of the outputs, VSync and HSync define the output style (640x480 at 60 Hz), while the other 12 provide the RGB values. The VSync and HSync outputs ensure that the RGB display values are only being output during the display interval in the diagram and tables below.

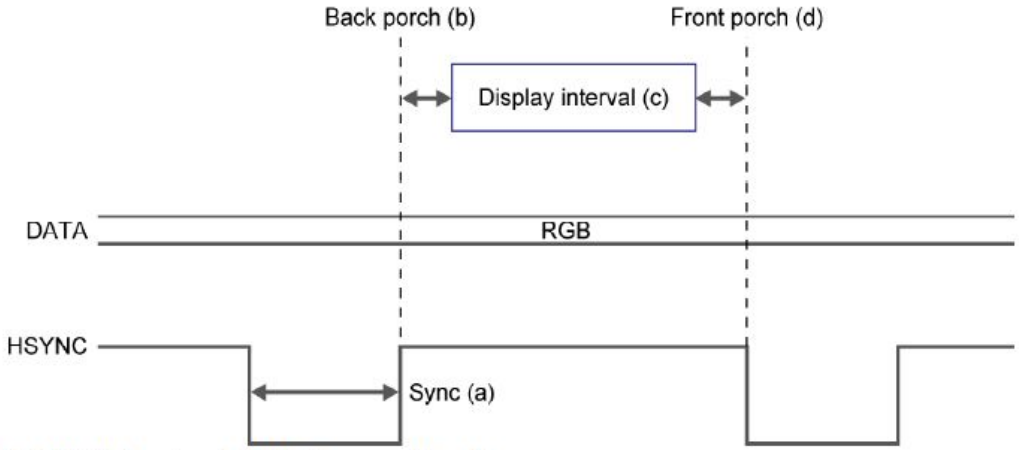


Figure 3-22 VGA horizontal timing specification

Table 3-9 VGA Horizontal Timing Specification

VGA mode		*Horizontal Timing Spec*				
Configuration	*Resolution(HxV)*	*a(pixel clock cycle)*	*b(pixel clock cycle)*	*c(pixel clock cycle)*	*d(pixel clock cycle)*	*Pixel clock(MHz)*
VGA(60Hz)	640x480	96	48	640	16	25

Table 3-10 VGA Vertical Timing Specification

VGA mode		*Vertical Timing Spec*				
Configuration	*Resolution(HxV)*	*a(lines)*	*b(lines)*	*c(lines)*	*d(lines)*	*Pixel clock(MHz)*
VGA(60Hz)	640x480	2	33	480	10	25

Figure 8: VGA timing specification data[4]

The RGB values are twelve bits total, meaning each color can be assigned one hex value. In this project, only red and blue will be active. The RGB values are exported from the DE-10 Lite board to the three corresponding pins on the VGA port, as shown in the figure below.

[4] ECE 272 Section 6

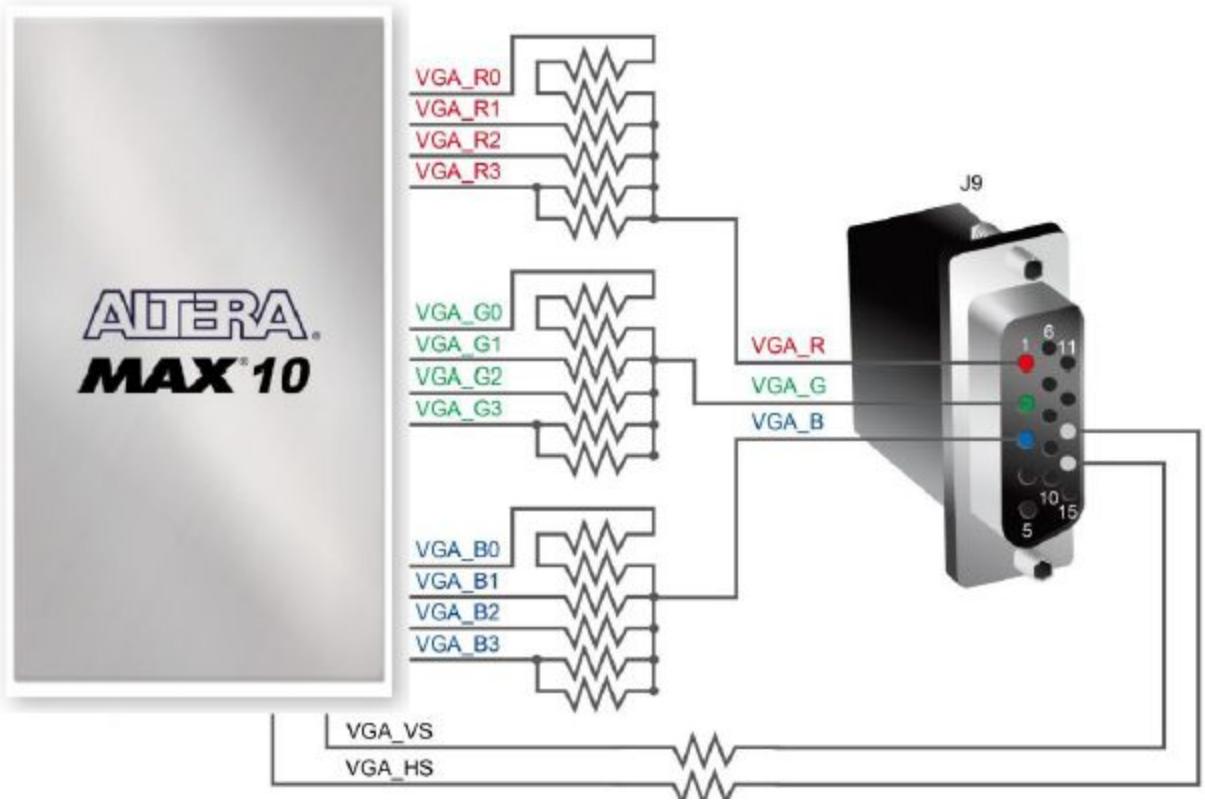


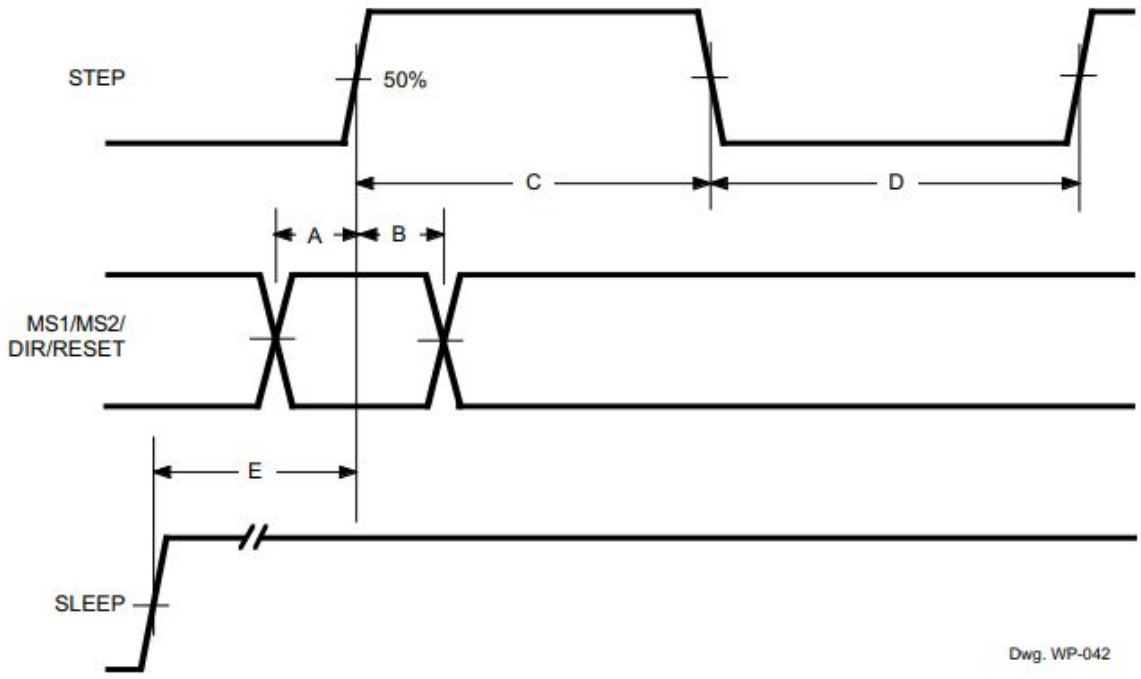
Figure 9: VGA output pins visual[5]

3.2.3 DC Motor (Basic Motion) Description

The incoming data from the keyboard or the NES controller along with the speed value are fed to the motor control components. The logic is made of three instances of a block called Motor Controller. The Motor Controller uses direction and speed information to produce signals that are, in turn, sent to a stepper motor driver based on the Allegro 3967 driver which greatly simplifies the operation of this kind of motors to two lines (direction and step) at a minimum. It can be configured to run the motor in full-step, half-step, quarter-step, and eighth-step. Through experimentation we found that quarted-step motion provides a good balance between speed and smoothness. For the step signal, a transition from low to high makes the driver run one step (or fraction of a step) so the speed of the rotation is determined by the frequency of the pulses applied to the step input of the driver.

[5] DE10-Lite User Manual

Timing Requirements
(T_A = +25°C, V_{CC} = 5 V, Logic Levels are V_{CC} and Ground)



A. Minimum Command Active Time Before Step Pulse (Data Set-Up Time) ..... 200 ns
B. Minimum Command Active Time After Step Pulse (Data Hold Time) ..... 200 ns
C. Minimum STEP Pulse Width ..... 1.0 µs
D. Minimum STEP Low Time ..... 1.0 µs
E. Maximum Wake-Up Time ..... 1.0 ms

Figure 10: Signals and timing requirements for DC Motor[6]

The Motor Controller module takes care of generating a square wave with a frequency that is proportional to the speed value (a 10-bit number provided by the potentiometer/ADC component). This wave is applied to the step input of the driver as long as the enable input of the Motor Controller is high. The direction input is applied directly to the direction pin of the driver taking into account the timing requirements of the driver as illustrated above.

[6] https://www.allegromicro.com/en/Products/Motor-Drivers/Brush-DC-Motor-Drivers/A3967

4. HDL Components

4.1 Top-Level Components

To make the system composition easier to understand, we decided to use schematic entry for the top-level. Although SystemVerilog is relatively easy to understand when uses in structural mode, the diagram below is much more descriptive than a series of module declarations in text would be.

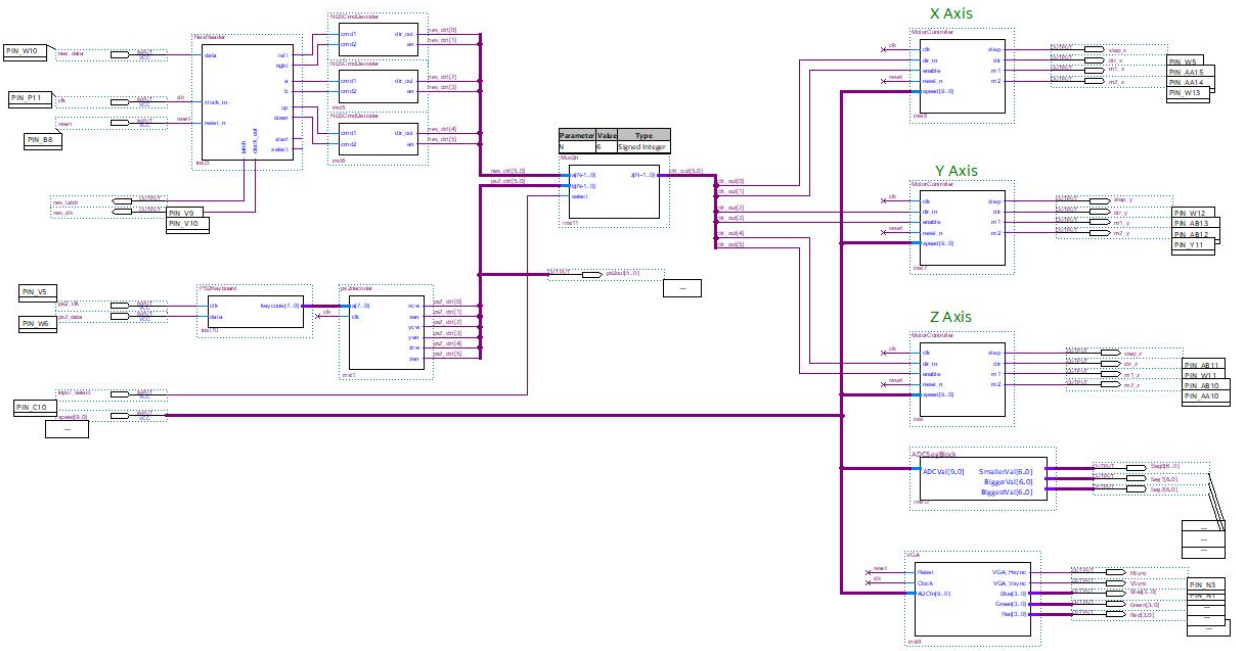


Figure 11: Top level design view.

Inputs: button states from the NES controller, key codes from PS/2 keyboard, input route selector, and digitized potentiometer voltage. The inputs are described in more detail in the individual block sections.

Outputs: control signals to NES controller, motor control signals, VGA representation of current motor speed, seven-segment representation of current motor speed.

Description: the system is very simple and has a clear demarcation between inputs and outputs. Using a switch as a selector, the operator can choose to use a keyboard or a game controller to send movement instructions to the circuit. These instructions are then routed to the relevant axis motor controller who then produces the correct signal to move a stepper motor. The speed is determined using a potentiometer/ACD component and this value is routed to all the motor controllers and to VGA and seven-segment display drivers.

4.1.1 Multiplexer

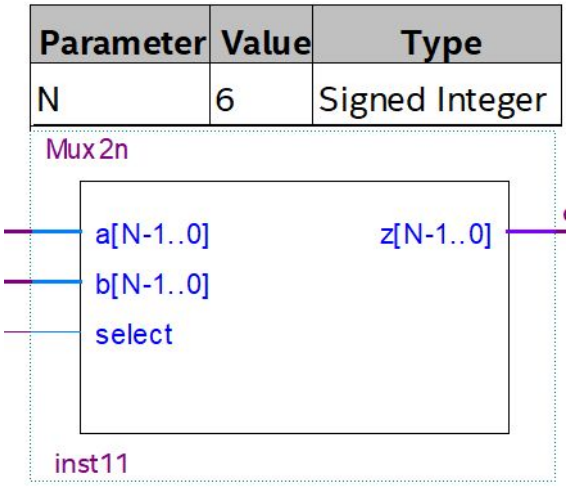


Figure 12: Multiplexer HDL Design

Inputs: two 6-bit busses collating the motion signals from the game controller and the keyboard, and a select line to configure which input bus is active.
Outputs: a 6-bit bus with the motion signals.
Description: this is a simple multiplexer with the only notable characteristic that the input and output lines can be configurable in width. This was useful earlier in the design.

4.2 NES Components

There are two main parts of the NES controller. The first major part is the NES reader which will read the data from the NES hardware when a button is pressed. Then the next part is the NES decoder that's specific to this project. Because the project is working with three DC Motors, the project is designed to split specific buttons to each motor; thus, there are three different NES decoders.

4.2.1 NES Reader

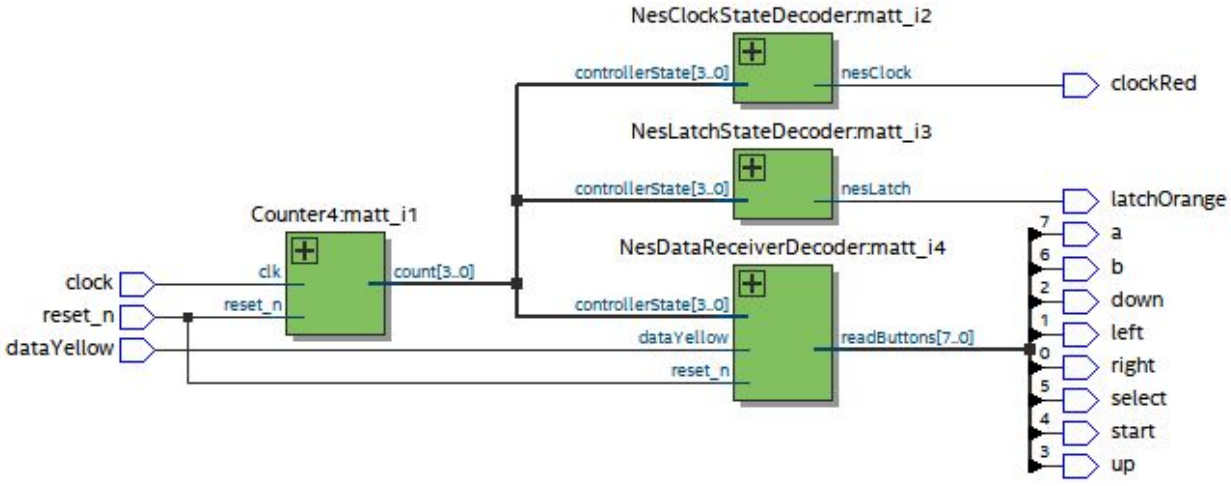


Figure 13: NES Reader Design

In the figure above, it demonstrates the basic logic blocks for the NES reader. As shown, there are three inputs: the clock, reset, and dataYellow. This dataYellow is the data of which button has been pressed. Then the clock input is just assigned to any clock signal, and the reset will reset the whole reader when a specific button is pressed. Then the outputs of this NES reader is the clock Red signal, the latch orange, and the buttons. The clock Red and latch orange are unassigned for this digital design. The start and select buttons are also unassigned outputs because they don't have a motor to control. Then finally the other buttons are assigned to the three motors; up and down to motor 1, left and right to motor two, and a and b to motor 3. The rest of the circuit shows how the logic blocks are connected in order to work the way it needs to. The NES Reader is designed to take in some data of button(s) being pushed then sending that signal over to something else. The NES clock state decoder takes the clock signal and then outputs it out, this logic block also keeps track of the state the NES is in like memory. Then the NES latch state decoder will output a latch, and it will also remember the latch in what state it is in before a new data input is sent in. Then finally the NES data receiver decoder will take the information from the data Yellow and output that button being pressed.

4.2.2 NES decoder for DC Motor 1

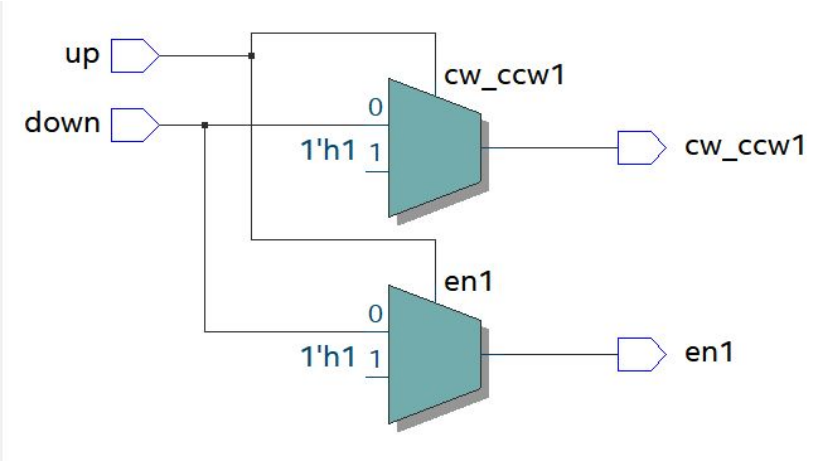


Figure 14: Design of NES decoder for DC Motor 1

This is a simple MUX which will choose to output a logic one when up or down are also one. This demonstrates the logic for the NES controller to move motor 1. The inputs are up and down. This means that when up or down is pressed it will send a logic high signal into the MUX. Then the MUX will take the two bits and output a one bit of logic high into the clockwise/counterclockwise and the enable. The outputs are the clockwise and counter clockwise direction and the enable for motor 1. The enable will allow the motor to start spinning only when there is a logic one fed into it. Therefore, when either up and down are pressed, it will send a signal to start spinning and enable the spin.

4.2.3 NES decoder for DC Motor 2

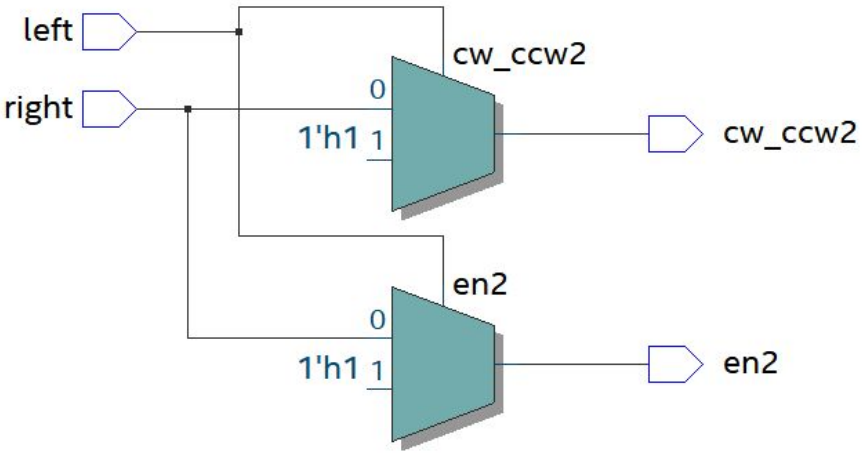


Figure 15: Design of NES decoder for DC Motor 2

The same logic is used here as the NES decoder for DC Motor 1. However, instead of using up and down it uses the buttons left and right. Please refer to NES decoder for DC Motor 1.

4.2.4 NES decoder for DC Motor 3

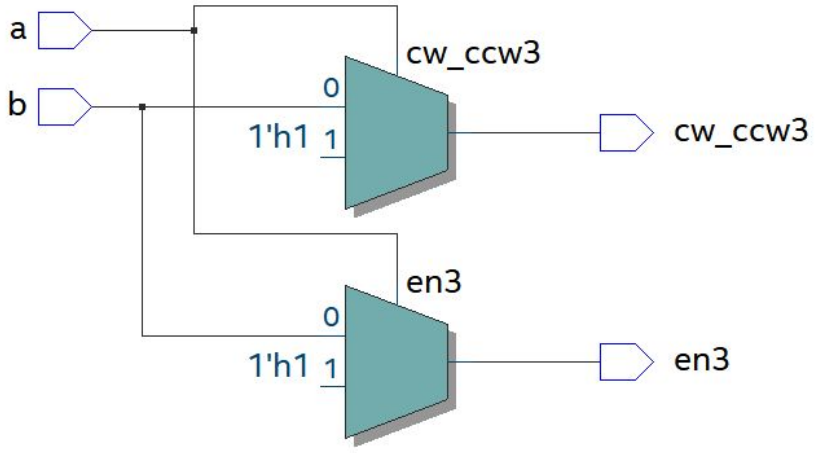


Figure 16: Design of NES decoder for DC Motor 3

The same logic is used here as the NES decoder for DC Motor 1. However, instead of using up and down it uses the buttons a and b. Please refer to NES decoder for DC Motor 1.

4.3 PS/2 Components

4.3.1 Keyboard

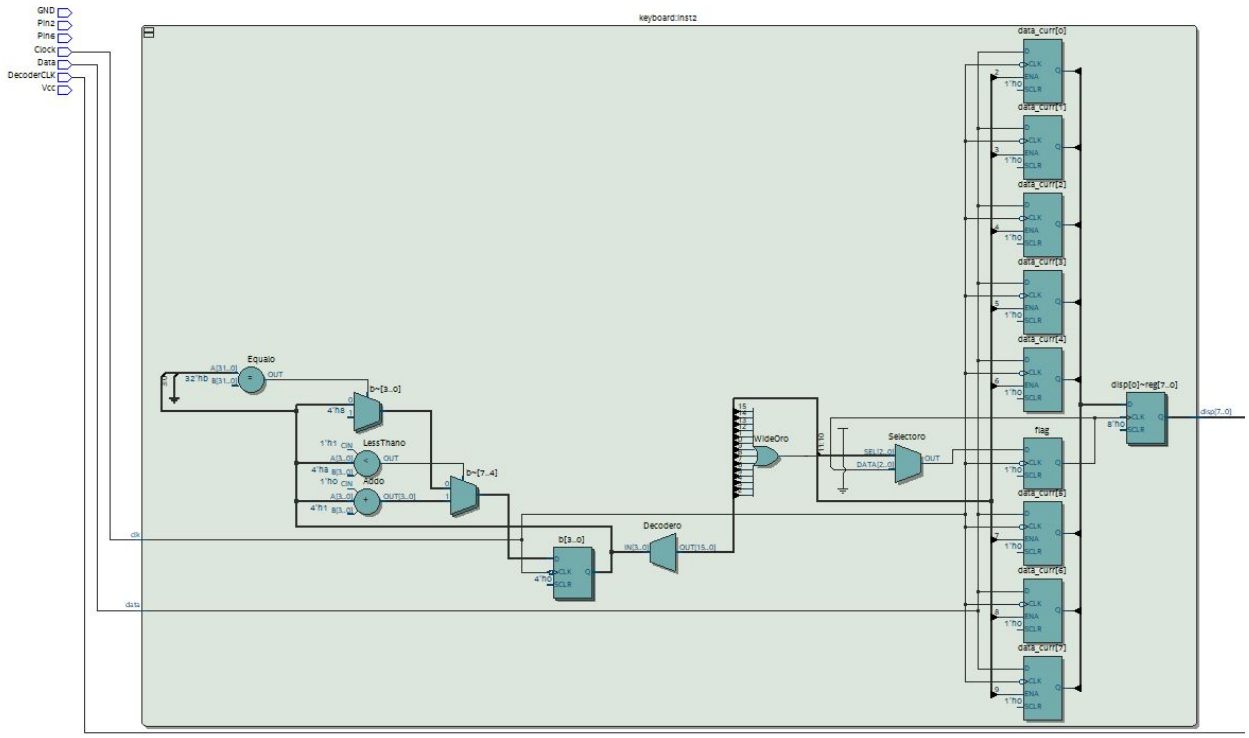


Figure 17: PS/2 Keyboard Design, synthesized with Quartus Lite

Inputs: The 6 pins on the PS/2 Port. The Data signal is pin 1, Clock signal is pin 5, GND is pin 3, Vcc is pin 4, and pins 2 and 6 are not implemented, so they go to nothing. All inputs are 1-bit.
Outputs: This module will output a single 8-bit signal that represents the make code of the key being pushed.
Description: This module is designed to analyze signals outputted by the PS/2 keyboard and decode what keys they represent, responding properly to the keyboard's clock, and output the value representing the key pressed to whatever device it is connected to, in our case staying on the FPGA.

4.3.2 Keyboard Decoder

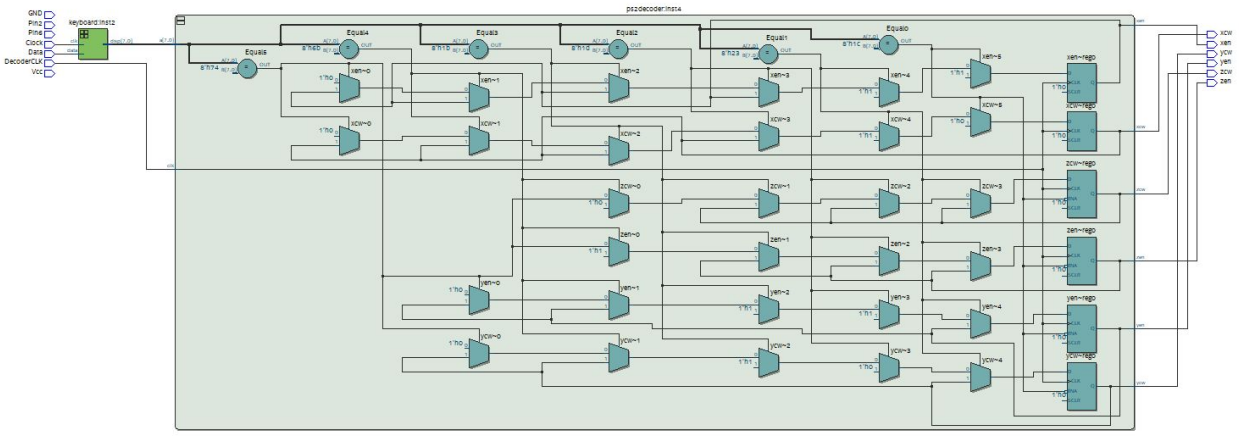


Figure 18: PS/2 Keyboard Decoder Design, synthesized with Quartus Lite

Inputs: The decoder takes in two inputs: One being an 8-bit bus, the value that the keyboard analyzer outputs. The other is a clock, but not the same clock from the keyboard, rather a separate (preferably faster) clock that is used to send signals to our outputs and make the SystemVerilog code work properly.
Outputs: There are 6 outputs, 2 for representing needed values in each dimensional axis. There is a clockwise output (_CW) and enable output (_EN) for the X, Y, and Z axes. All are 1-bit.
Description: Assuming a single key is pushed at a time, the keyboard analyzer will send the value pressed to this decoder, where six specific keys are used for our project, two for controlling each dimension. If the value that is taken in represents the A key, then the X-Axis motor will be enabled and the clockwise signal will be low, meaning that the motor should rotate counter-clockwise. If any key other than the 6 keys being used are pressed, all enable and rotation values will be low, so no rotation in the motors occur.

4.4 Potentiometer Components

4.4.1 Potentiometer

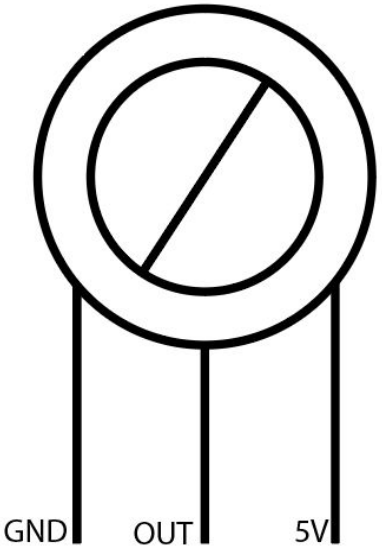


Figure 19: Potentiometer physical design

Inputs: The potentiometer takes two inputs, one from the VCC and one from ground. These two voltages are both put onto a wire inside the potentiometer, which allows us to define the output.
Outputs: The singular output of the potentiometer is a value between 0V and 5V, which is determined by turning the knob on the potentiometer.
Description: As the knob twists closer to the VCC, the voltage increases, and as the knob twists closer to the ground, the voltage decreases.

4.4.2 Analog/Digital Converter

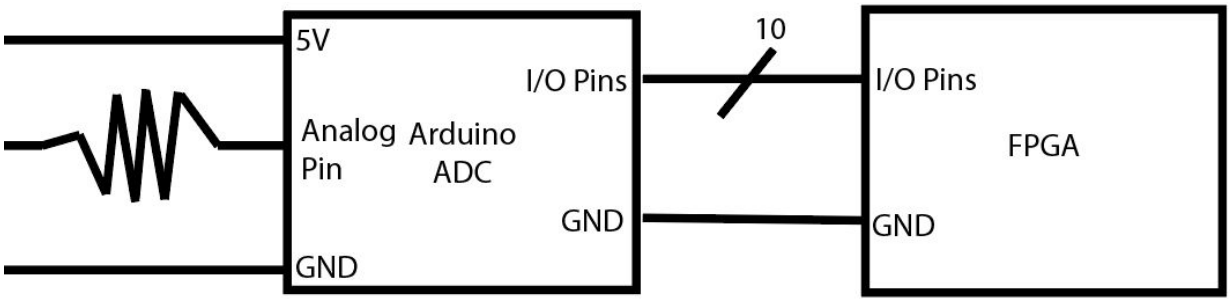


Figure 20: Analog/Digital Converter design

Inputs: The analog/digital converter only has one input pin, which holds an analog voltage between 0V and 5V.

Outputs: The output logic takes the input pin and converts the voltage into a ten-bit value based on its voltage relative to the 5V maximum. For example, a 5V input would output all 1s, and a 0V input would output all 0s, and a 2.5V input would generate the value 1000000000 (binary), halfway between 0000000000 and 1111111111. This ten-bit value is transferred to the FPGA via the output pins on the arduino to the input pins on the DE-10 Lite.
Description: The analog/digital converter is unique, as it is the only piece of logic not rendered by the FPGA. The analog/digital converter is on an Arduino Mega, which connects to the FPGA via the I/O pins on the respective boards.

4.5 7-Segment Components

4.5.1 7-Segment High Level

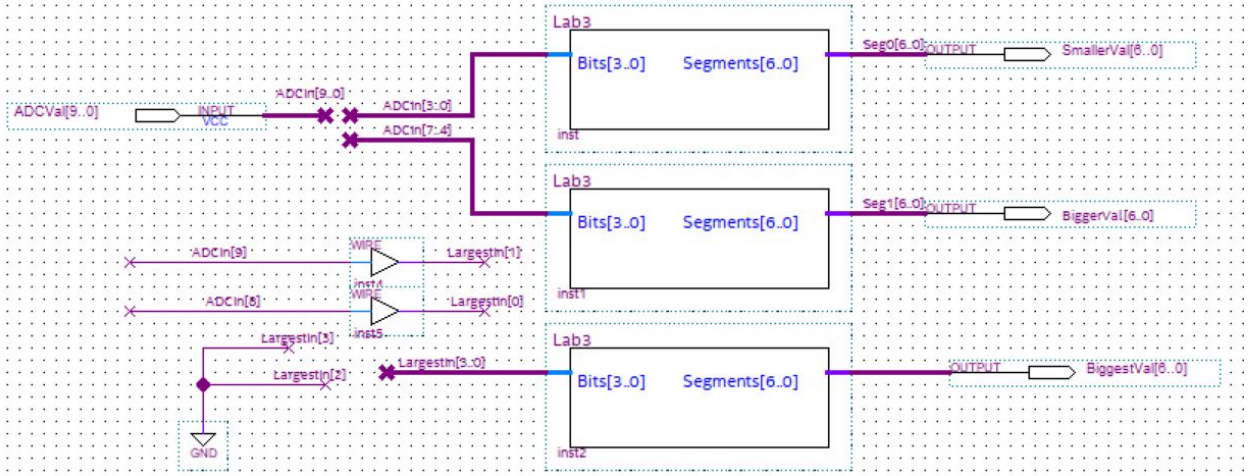


Figure 21: Seven-Segment display highest level design

Inputs: The Seven-Segment display's top level HDL file takes input from the potentiometer's analog/digital converter, and from ground. The ten bit input is broken into three separate wires, two with four bits and one with two.
Outputs: There are 21 output pins, creating three digits on the seven-segment display on our FPGA. The display will show a hexadecimal value between 000 and 3FF, indicating current level of voltage from the potentiometer, implicitly also indicating motor speed.
Description: The split of the ten-bit wire separates the two highest significance bits from the four mid-significance and four lowest-significance bits. The two highest significance bits are put into a bus of width 3 with two ground signals, forcing the bus to always be [00XX]. Next, these three four-bit width busses are put through three seven segment decoders to create the output.

4.5.2 7-Segment Decoder (Lab3)

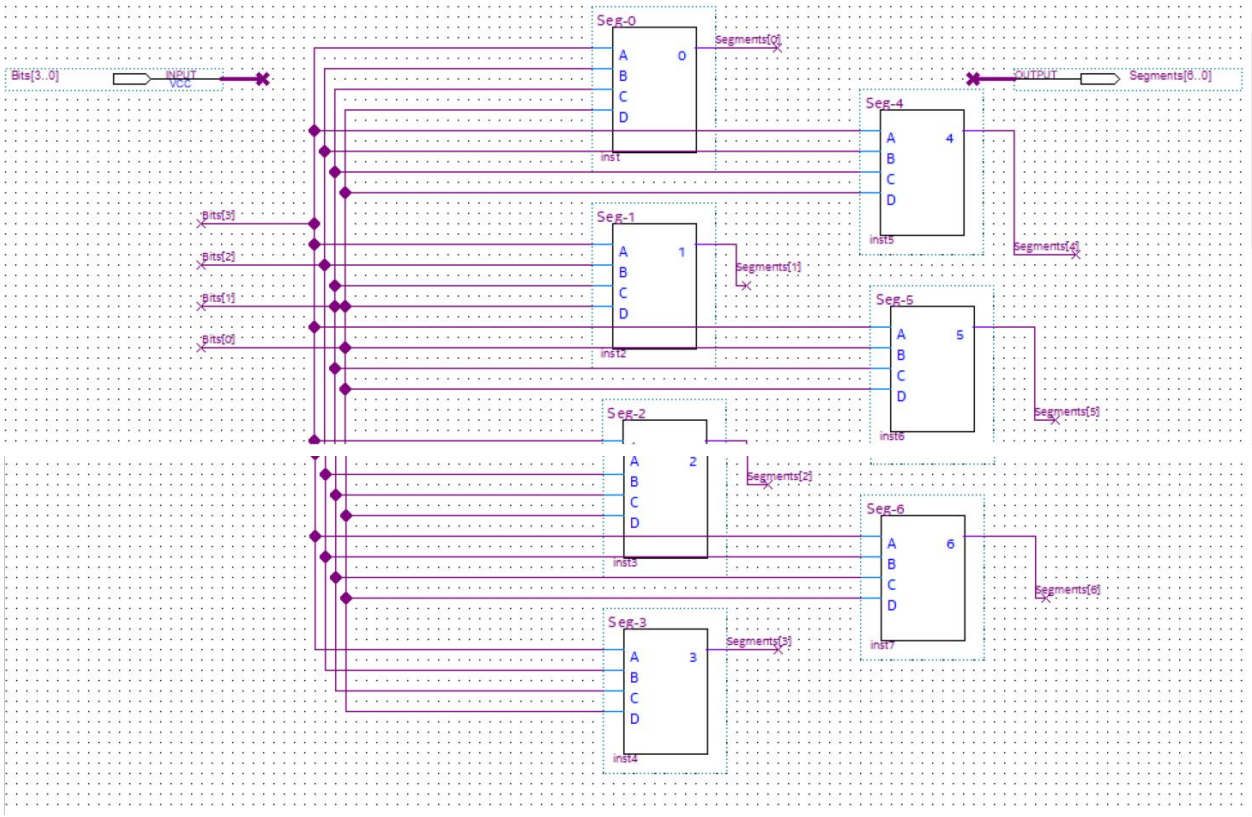


Figure 22: Seven-Segment decoder design

Inputs: The seven-segment decoder takes a four input bus representing a hex value from 0 to F.
Outputs: The seven-segment decoder generates seven bits of output, one for each segment on the display. It is important to note that these output signals are active low.
Description: In binary, this value is broken into its individual bits which are sent through individual segment drivers generated from karnaugh maps, creating our output. The output is seven bits that visually create a number in hexadecimal based on the four bit binary input value.

4.5.3 Individual Segment Decoders

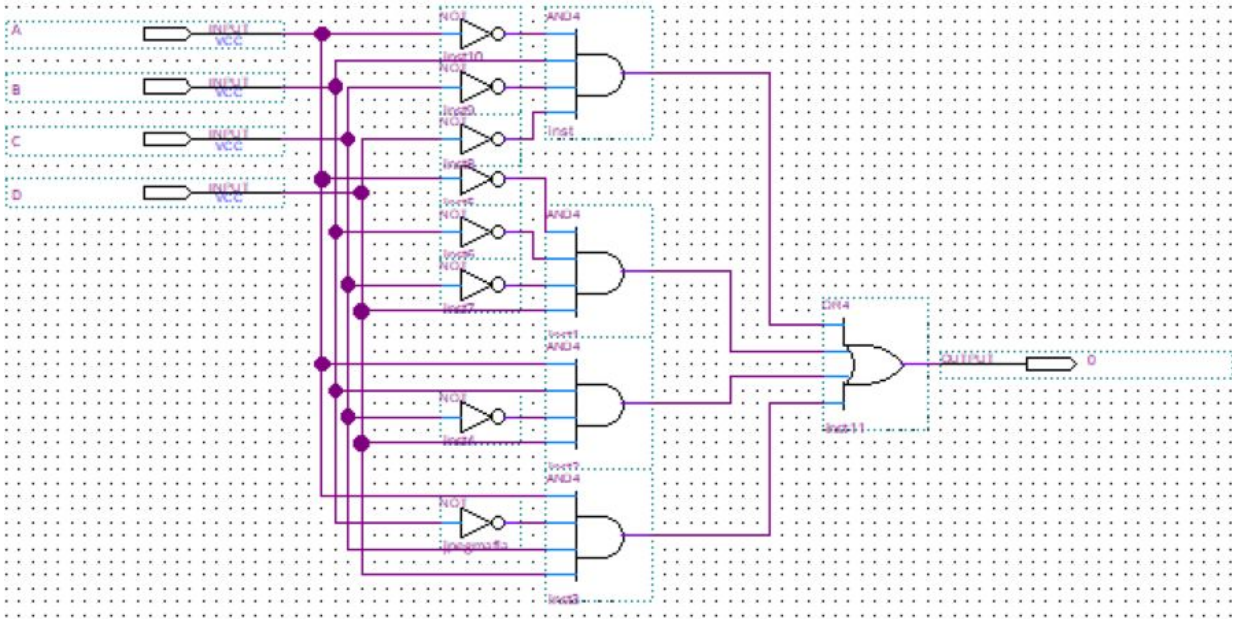


Figure 23: An example individual segment decoder. Logic derived from Karnaugh maps.

Inputs: All seven of the individual segment decoders take a four-bit input value as four discrete bits.
Outputs: Each individual segment decoder outputs either 1 or 0.
Description: The four bits go through some sum-of-products logic gates to determine a single boolean output bit. This bit powers a single chunk of a seven-segment display, under the assumption that the display is active low.

4.6 VGA Components

4.6.1 VGA Driver

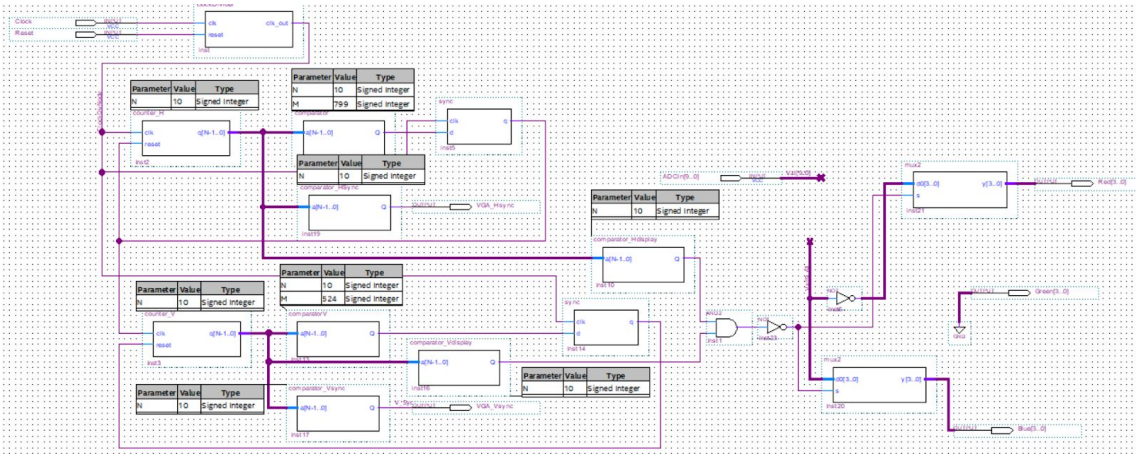


Figure 24: VGA driver at highest level (Less abstracted model)

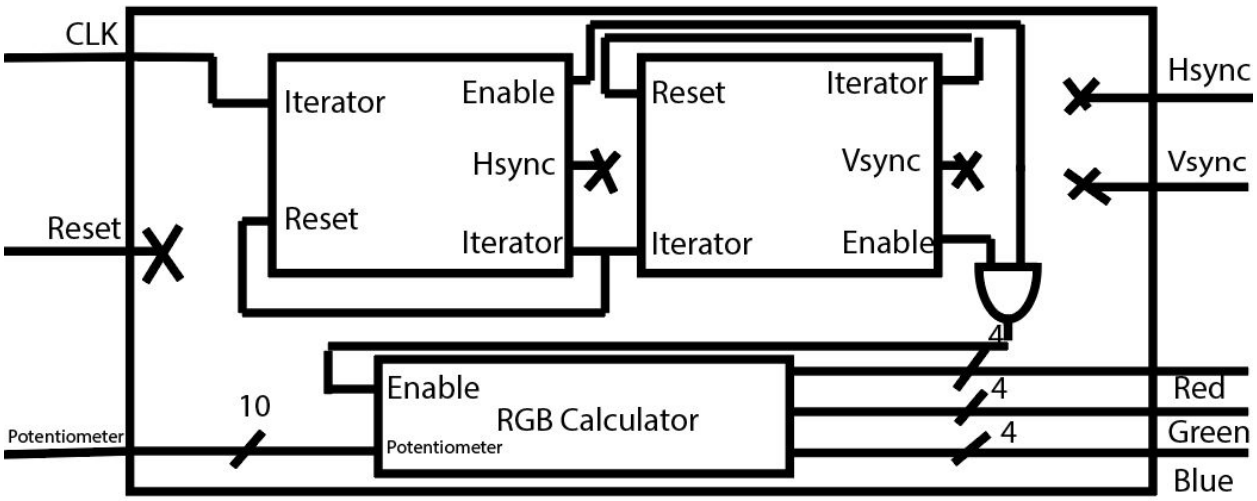


Figure 25: VGA driver at highest level (More abstracted model)

Inputs: The VGA Driver has 3 types of input, clock, reset, and the ten-bit signal from the analog/digital converter from the potentiometer. The clock signal accepts a 50MHz clock. The reset doesn't actually do anything. The ten bit signal from the arduino generates color values for the output.
Outputs: The driver outputs a color from red to blue, based on the four most significant bits of the potentiometer's value. The driver also outputs Hsync and Vsync signals, generated to create a VGA on 60Hz with resolution 640x480. When these outputs are all properly sent through a VGA port, it should generate a single solid color on a VGA monitor.
Description: The 50MHz clock signal is passed through internal logic that slows it down properly to drive a 60Hz monitor, and the slowed clock is then passed through the hsync and vsync logic to create a proper enable window for the RGB values. The RGB values are determined by the four most significant bits of the potentiometer's data. The four bit value becomes the blue output, and is inverted to become the red output. There is no green output. This generates the effect of transitioning from red through purple to blue as the value gets larger. The reset signal really doesn't do anything.

4.7 DC Motor Components

Inputs: The module accepts a clock signal, reset signal, direction indicator and motion enable signals, and a 10-bit speed value.
Outputs: a variable frequency step signal, direction indicator and two mode signals that configure the stepping rate (full, half, quarter, and eighth of a step). These are the signals require to operate an Allegro A3967 stepper motor controller.

Description:

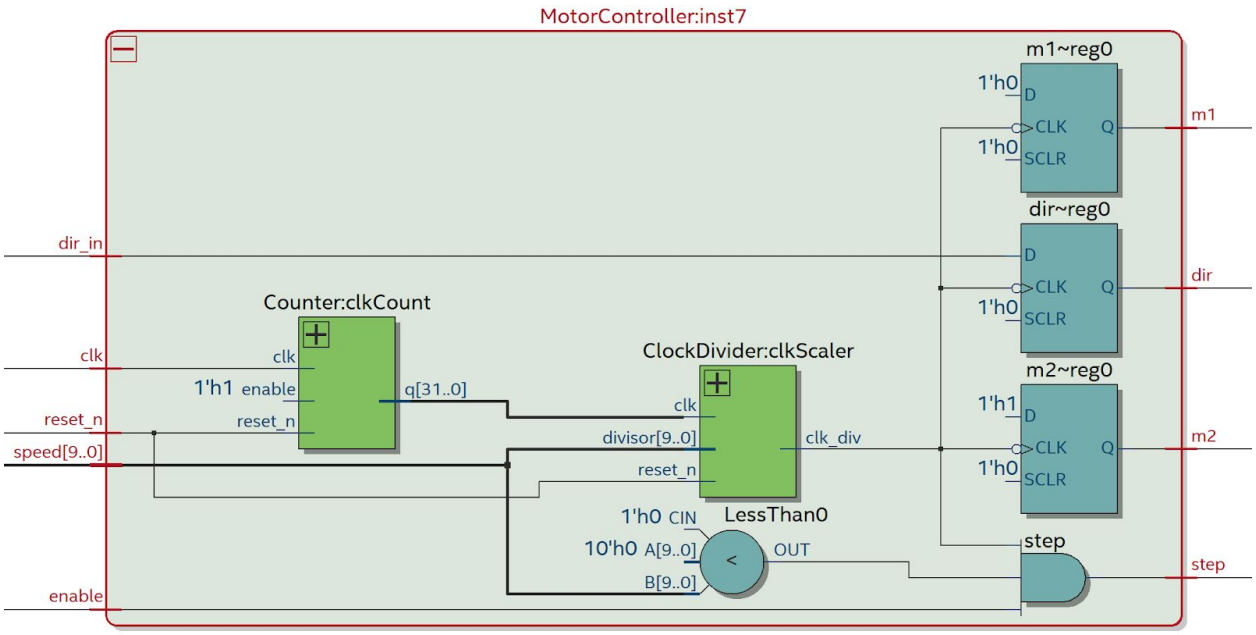


Figure 26: RTL view of the full MotorController module.

The Motor Controller component consists of a clock scaling stage that divides the FPGA's 50 MHz clock by 2^{11} bringing it down to 24.4 KHz. Although the Allegro diver specifies 1 µs as the minimum pulse width, we found that, in practice, with the particular motors we were using, the minimum pulse that would work was 40 µs, thus the maximum frequency the controller should produce is 12.5 KHz.

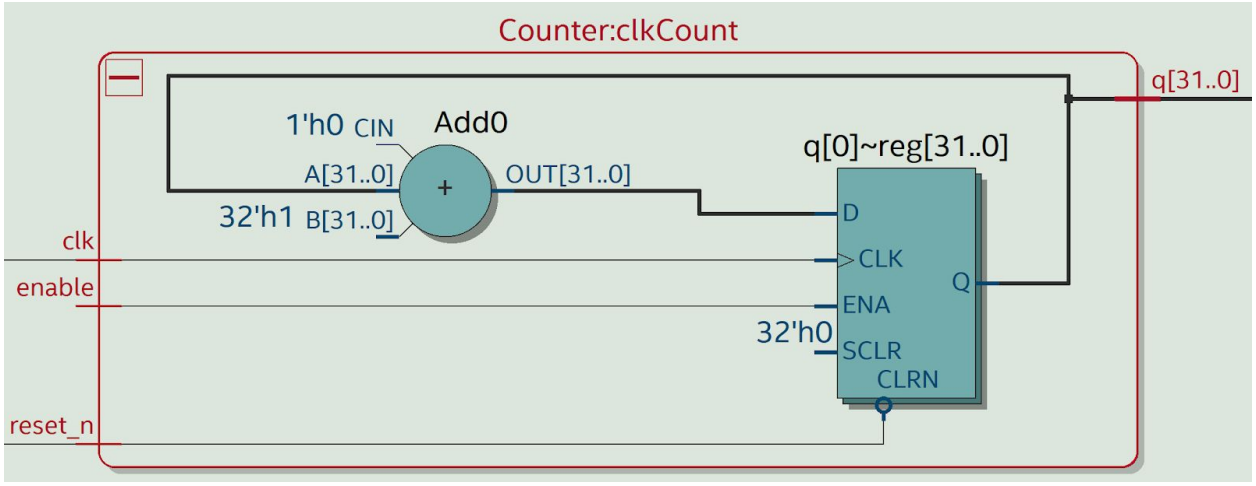


Figure 27: RTL view of the 32-bit binary counter.

This signal is then used to drive a counter that will count up to the value provided by the speed input and then reset. Every time the counter reached the speed value, its output changes from high to low or low to high, thus the frequency of the output is proportional to the speed value, this signal is called the motor pulse.

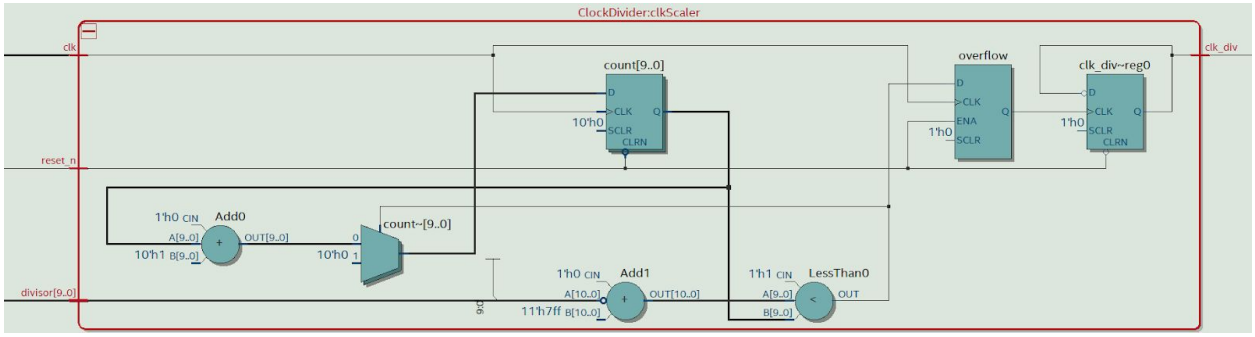


Figure 28: RTL view of the variable counter.

The A3967 specifies 200 ns as the setup and hold times for the configuration signals (direction, m1, m2), so they are passed through registers that are triggered by the falling edge of the motor pulse. Since this pulse is never smaller than 20 us, signals are compliant with the timings. Finally, the motor pulse is only active (present on the step output) when the speed is non zero and the enable input is high.

The m1 and m2 signals were hardwired to the values m1:0 and m2:1 which sets the controller to issue quarter-step increments which provided a good compromise between smooth operation and maximum speed. A more sophisticated algorithm could vary these values depending on the current speed input.

5. Appendix

5.1 Verilog and SystemVerilog Source Code

5.1.1 Top Level

We created the top level using schematic entry. For simulation purposes, we used Quartus' Verilog generation function. This is the code generated:

```
// Copyright (C) 2018  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
```

```
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details.

// PROGRAM        "Quartus Prime"
// VERSION        "Version 18.1.0 Build 625 09/12/2018 SJ Lite Edition"
// CREATED        "Fri Dec 06 19:10:26 2019"

module FinalProject(
      ps2_clk,
      ps2_data,
      nes_data,
      reset,
      clk,
      input_select,
      speed,
      step_z,
      step_y,
      dir_y,
      dir_z,
      step_x,
      dir_x,
      nes_latch,
      nes_clk,
      m1_x,
      m2_x,
      m1_y,
      m2_y,
      m1_z,
      m2_z,
      VSync,
      HSync,
      Blue,
      Green,
      nes_ctrl,
      ps2out,
```

```
      Red,
      Seg0,
      Seg1,
      Seg2
);


input wire  ps2_clk;
input wire  ps2_data;
input wire  nes_data;
input wire  reset;
input wire  clk;
input wire  input_select;
input wire  [9:0] speed;
output wire step_z;
output wire step_y;
output wire dir_y;
output wire dir_z;
output wire step_x;
output wire dir_x;
output wire nes_latch;
output wire nes_clk;
output wire m1_x;
output wire m2_x;
output wire m1_y;
output wire m2_y;
output wire m1_z;
output wire m2_z;
output wire VSync;
output wire HSync;
output wire [3:0] Blue;
output wire [3:0] Green;
output wire [5:0] nes_ctrl;
output wire [5:0] ps2out;
output wire [3:0] Red;
output wire [6:0] Seg0;
output wire [6:0] Seg1;
output wire [6:0] Seg2;
```

```
wire  [5:0] ctrl_out;
wire  [5:0] nes_ctrl_ALTERA_SYNTHESIZED;
wire  [5:0] ps2_ctrl;
wire  [7:0] SYNTHESIZED_WIRE_0;
wire  SYNTHESIZED_WIRE_1;
wire  SYNTHESIZED_WIRE_2;
wire  SYNTHESIZED_WIRE_3;
wire  SYNTHESIZED_WIRE_4;
wire  SYNTHESIZED_WIRE_5;
wire  SYNTHESIZED_WIRE_6;




MotorController   b2v_inst(
      .clk(clk),
      .dir_in(ctrl_out[4]),
      .enable(ctrl_out[5]),
      .reset_n(reset),
      .speed(speed),
      .step(step_z),
      .dir(dir_z),
      .m1(m1_z),
      .m2(m2_z));


PS2Decoder  b2v_inst1(
      .clk(clk),
      .a(SYNTHESIZED_WIRE_0),
      .xcw(ps2_ctrl[0]),
      .xen(ps2_ctrl[1]),
      .ycw(ps2_ctrl[2]),
      .yen(ps2_ctrl[3]),
      .zcw(ps2_ctrl[4]),
      .zen(ps2_ctrl[5]));


PS2Keyboard b2v_inst10(
```

```
	.clk(ps2_clk),
	.data(ps2_data),
	.keycode(SYNTHESIZED_WIRE_0));


Mux2n b2v_inst11(
	.select(input_select),
	.a(nes_ctrl_ALTERA_SYNTHESIZED),
	.b(ps2_ctrl),
	.z(ctrl_out));
	defparam	b2v_inst11.N = 6;


ADCSegBlock b2v_inst12(
	.ADCVal(speed),
	.BiggerVal(Seg1),
	.BiggestVal(Seg2),
	.SmallerVal(Seg0));


NesReader	b2v_inst3(
	.data(nes_data),
	.clock_in(clk),
	.reset_n(reset),
	.latch(nes_latch),
	.clock_out(nes_clk),
	.up(SYNTHESIZED_WIRE_5),
	.down(SYNTHESIZED_WIRE_6),
	.left(SYNTHESIZED_WIRE_1),
	.right(SYNTHESIZED_WIRE_2),


	.a(SYNTHESIZED_WIRE_3),
	.b(SYNTHESIZED_WIRE_4));


NESCmdDecoder	b2v_inst4(
	.cmd1(SYNTHESIZED_WIRE_1),
	.cmd2(SYNTHESIZED_WIRE_2),
```

```
	.dir_out(nes_ctrl_ALTERA_SYNTHESIZED[0]),
	.en(nes_ctrl_ALTERA_SYNTHESIZED[1]));


NESCmdDecoder	b2v_inst5(
	.cmd1(SYNTHESIZED_WIRE_3),
	.cmd2(SYNTHESIZED_WIRE_4),
	.dir_out(nes_ctrl_ALTERA_SYNTHESIZED[2]),
	.en(nes_ctrl_ALTERA_SYNTHESIZED[3]));


NESCmdDecoder	b2v_inst6(
	.cmd1(SYNTHESIZED_WIRE_5),
	.cmd2(SYNTHESIZED_WIRE_6),
	.dir_out(nes_ctrl_ALTERA_SYNTHESIZED[4]),
	.en(nes_ctrl_ALTERA_SYNTHESIZED[5]));


MotorController	b2v_inst7(
	.clk(clk),
	.dir_in(ctrl_out[2]),
	.enable(ctrl_out[3]),
	.reset_n(reset),
	.speed(speed),
	.step(step_y),
	.dir(dir_y),
	.m1(m1_y),
	.m2(m2_y));


MotorController	b2v_inst8(
	.clk(clk),
	.dir_in(ctrl_out[0]),
	.enable(ctrl_out[1]),
	.reset_n(reset),
	.speed(speed),
	.step(step_x),
	.dir(dir_x),
	.m1(m1_x),
```

```
        .m2(m2_x));


    VGA   b2v_inst9(
          .Reset(reset),
          .Clock(clk),
          .ADCIn(speed),
          .VGA_Hsync(HSync),
          .VGA_Vsync(VSync),
          .Blue(Blue),
          .Green(Green),
          .Red(Red));


    assign      nes_ctrl = nes_ctrl_ALTERA_SYNTHESIZED;
    assign      ps2out = ps2_ctrl;

    endmodule
```

5.1.2 NES Controller

Since the Ness controller's digital logic blocks can be described into separate logic blocks, the following are SystemVerilog code for all the different blocks.

5.1.2.1 NES 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
```

5.1.2.2 NES decoder for DC Motor 1

```
module cw_ccw1 (
      input logic up,
      input logic down,
      output logic cw_ccw1,
      output logic en1);

      always @(up or down or cw_ccw1 or en1)
      begin
            if (up == 1)
                  begin
                        cw_ccw1 = 1;
                        en1 = 1;
                  end
            else if (down ==1)
                  begin
                        cw_ccw1 = 1;
                        en1 = 1;
                  end
            else
                  begin
                        cw_ccw1 = 0;
                        en1 = 0;
                  end
      end

endmodule
```

5.1.2.3 NES decoder for DC Motor 2

```
module cw_ccw2 (
      input logic left,
      input logic right,
```

```
	output logic cw_ccw2,
	output logic en2);

	always @(left or right or cw_ccw2 or en2)
	begin
		if (left == 1)
			begin
				cw_ccw2 = 1;
				en2 = 1;
			end
		else if (right ==1)
			begin
				cw_ccw2 = 1;
				en2 = 1;
			end
		else
			begin
				cw_ccw2 = 0;
				en2 = 0;
			end
	end
endmodule
```

5.1.2.4 NES decoder for DC Motor 3

```
module cw_ccw3 (
	input logic a,
	input logic b,
	output logic cw_ccw3,
	output logic en3);

	always @(a or b or cw_ccw3 or en3)
	begin
		if (a == 1)
			begin
				cw_ccw3 = 1;
				en3 = 1;
			end
		else if (b ==1)
			begin
```

```
                        cw_ccw3 = 1;
                        en3 = 1;
                  end
            else
                  begin
                        cw_ccw3 = 0;
                        en3 = 0;
                  end
      end
endmodule
```

5.1.3 PS/2 Keyboard

5.1.3.1 keyboard (Keyboard Analyzer)

```
//CODE CITATION: “PS2 Keyboard.” Students' Gymkhana, Indian Institute of
Technology Kanpur,
//http://students.iitk.ac.in/eclub/assets/tutorials/keyboard.pdf.

module keyboard(input wire clk,
                          input wire data,
                          output reg [7:0] disp);
    reg [7:0]data_curr;
    reg [3:0]b;
    reg flag;

    initial    //sets the initial state of module when powered on
    begin
       b<=4'h1;
       flag<=4'h0;
       data_curr<=8'hf0;
       disp<=8'hf0;
    end

    always @(negedge clk)begin    //data is taken in at the negative edge
       case(b)
       1:; //nothing on first bit
       2:data_curr[0]<=data;
       3:data_curr[1]<=data;
       4:data_curr[2]<=data;
       5:data_curr[3]<=data;
```

```
        6:data_curr[4]<=data;
        7:data_curr[5]<=data;
        8:data_curr[6]<=data;
        9:data_curr[7]<=data;
        10:flag<=1'b1;    //Parity bit
        11:flag<=1'b0;    //end bit

        endcase
              if(b<=10)
                    b<=b+1;

              else if(b==11)
                    b<=1;

        end

    always @(posedge flag)begin    //outputs data to driver
        //if(data_curr==8'hf0)
        //else
              disp<=data_curr;
        end
endmodule
```

5.1.3.2 PS/2 Decoder

```
module ps2decoder(input logic [7:0]a,
                              input logic clk,
                              output logic xcw, xen,
                              ycw, yen,
                              zcw, zen);
    //AD will control the X-axis
    //WS will control the y-axis
    //Up and Down arrow will control Z-axis
    //output of 1 implies clockwise, 0 implies counterclockwise
      initial    //sets the initial state
    begin
              xen <= 0;
              yen <= 0;
              zen <= 0;
              xcw <= 0;
```

```
            ycw <= 0;
            zcw <= 0;
    end
    always @ (posedge clk)
       if(a == 8'h1c)begin    //a key
            xen <= 1;
            xcw <= 0;
       end
       else if(a == 8'h23)begin    //d key
            xen <= 1;
            xcw <= 1;
       end
       else if(a == 8'h1d)begin    //w key
            yen <= 1;
            ycw <= 0;
       end
       else if(a == 8'h1b)begin    //s key
            yen <= 1;
            ycw <= 1;
       end
       else if(a == 8'h6b)begin    //left arrow key
            zen <= 1;
            zcw <= 0;
       end
       else if(a == 8'h74)begin    //right arrow key
            zen <= 1;
            zcw <= 1;
       end
       else    begin
            xen <= 0;
            yen <= 0;
            zen <= 0;
            xcw <= 0;
            ycw <= 0;
            zcw <= 0;
       end
endmodule
```

5.1.3.3 PS2 Keyboard

```
// PROGRAM         "Quartus Prime"
// VERSION         "Version 18.0.0 Build 614 04/24/2018 SJ Lite Edition"
// CREATED         "Mon Dec 02 20:14:18 2019"

module PS2Keyboard(
    Data,
    GND,
    Pin2,
    Vcc,
    Clock,
    Pin6,
    DecoderCLK,
    xcw,
    xen,
    ycw,
    yen,
    zcw,
    zen
);


input wire    Data;
input wire    GND;
input wire    Pin2;
input wire    Vcc;
input wire    Clock;
input wire    Pin6;
input wire    DecoderCLK;
output wire    xcw;
output wire    xen;
output wire    ycw;
output wire    yen;
output wire    zcw;
output wire    zen;

wire    [7:0] SYNTHESIZED_WIRE_0;

keyboard    b2v_inst2(
```

```
    .clk(Clock),
    .data(Data),
    .disp(SYNTHESIZED_WIRE_0));


ps2decoder    b2v_inst4(
    .clk(DecoderCLK),
    .a(SYNTHESIZED_WIRE_0),
    .xcw(xcw),
    .xen(xen),
    .ycw(ycw),
    .yen(yen),
    .zcw(zcw),
    .zen(zen));

endmodule
```

5.1.4 Analog Potentiometer

```
NOTE: This code is in C written for the Arduino Mega
const int analogPin = A0;//the analog input pin attach to
const int ledPin =13;//the led attach to
int inputValue = 0;//variable to store the value coming from sensor
/*****************************************/
void setup()
{
  pinMode(ledPin,OUTPUT);
  Serial.begin(9600);

  DDRL = B11111111;
  DDRG = DDRG | B00000011;
}
/*****************************************/
void loop()
{
  inputValue = analogRead(analogPin);//read the value from the sensor

  int lo = inputValue & 255;
  int hi = (PORTG & B11111100) | (inputValue >> 8);
```

```
  //int newVal = inputValue / 1024 * 255;

  //PORTL = newVal;
  PORTL =lo;
  PORTG = hi;

  Serial.println(hi * 256 + lo);

  digitalWrite(ledPin,HIGH);
  delay(inputValue);
  digitalWrite(ledPin,LOW);
  delay(inputValue);
}
/*****************************************/
```

5.1.5 DE10-Lite 7 Segment Display

There is no verilog code for the seven-segment display

5.1.6 VGA Output

```
module clockDivider (input logic clk,
                     input logic reset,
                     output logic clk_out);
      always_ff @ (posedge clk, negedge reset)
            begin
                  if(!reset)
                        clk_out <= 0;
                  else
                        clk_out <= ~clk_out;
            end
endmodule

module ColorDecoder(input logic C1,
                    input logic C2,
                    output logic [3:0] color);
logic [1:0] data;
assign data = {C1,C2};
      always_comb
            case(data)
            0:                              color = 4'b0000;
```

```
            1:                          color = 4'b0101;
            2:                          color = 4'b1010;
            3:                          color = 4'b1111;
            default:                color = 4'b0000;
      endcase
endmodule

module comparator #(parameter N = 10, M = 799)
            (input logic [N-1:0] a,
            output logic Q);
      assign Q = (a == M);
endmodule

module comparator_Hdisplay #(parameter N = 10)
            (input logic [N-1:0] a,
            output logic Q);
      assign Q = (a > 144 && a < 784);
endmodule

module comparator_HSync #(parameter N = 10)
            (input logic [N-1:0] a,
            output logic Q);
      assign Q = (a > 96);
endmodule

module comparator_Vdisplay #(parameter N = 10)
            (input logic [N-1:0] a,
            output logic Q);
      assign Q = (a > 35 && a < 515);
endmodule

module comparator_Vsync #(parameter N = 10)
            (input logic [N-1:0] a,
            output logic Q);
      assign Q = (a < 2);
endmodule

module comparator2 #(parameter N = 26)
            (input logic [N-1:0] Clk,
```

```
            output logic NewClk);
      assign NewClk = (Clk == 25000000);
endmodule

module comparatorV #(parameter N = 10, M = 524)
            (input logic [N-1:0] a,
            output logic Q);
      assign Q = (a == M);
endmodule

module counter #(parameter N = 8)
                        (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

module counter_H #(parameter N = 10)
                        (input logic clk,
                            input logic reset,
                            output logic [N-1:0]q);
      always_ff @(posedge clk, posedge reset)
      begin
            if (reset) q <= 0;
            else if (q < 800)
            begin
                  q <= q + 1;
            end
            else
            begin
                  q <= 0;
            end
  end
endmodule

module counter_modified #(parameter N = 8)
                        (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 + 10;
endmodule

module counter_V #(parameter N = 10)
                          (input logic clk,
                              input logic reset,
                              output logic [N-1:0]q);
      always_ff @(posedge clk, posedge reset)
      begin
            if (reset) q <= 0;
            else if (q < 525)
            begin
                  q <= 1;
                  q <= q + 1;
            end
            else
            begin
                  q <= 0;
            end
  end
endmodule

module mux2 (input logic [3:0] d0,
                          input s,
                          output logic [3:0] y);
      assign y = s ? 0 : d0;
endmodule

module sync(input logic clk,
                          input logic d,
                          output logic q);
      logic n1;
      always_ff @(posedge clk)
            begin
            n1 <= d ; //nonblocking
```

```
            q <= n1; //nonblocking
            end
endmodule
```

5.1.7 DC motor (Basic Motion)

```
module MotorController (input logic clk,
                        input  logic dir_in,
                        input  logic enable,
                        input  logic reset_n,
                        input  logic [9:0] speed,
                        output logic step,
                        output logic dir,
                        output logic m1,
                        output logic m2);

logic step_out, motor_pulse;
logic [31:0] count;

Counter #(32) clkCount(clk, reset_n, 1, count);

ClockDivider clkScaler(count[9], reset_n, speed, motor_pulse);

always_ff @(negedge motor_pulse)
  begin
      dir <= dir_in;
      m1 <=0;
      m2 <=1;
  end

always_comb
  begin
    step_out = enable && speed > 0;
  end

assign step = step_out && motor_pulse;

endmodule
```

```
module Counter #(parameter N = 8)
                        (clk, reset_n, enable, q);

      input  logic clk;
      input  logic reset_n;
      input  logic enable;
      output logic [(N-1):0] q;

      always_ff @(posedge clk, negedge reset_n)
        begin
            if(!reset_n)
                  q <= 0;
            else if (enable)
                  q <= q + 1;
         end
endmodule

module ClockDivider(input  logic clk,
                    input  logic reset_n,
                    input  logic [9:0] divisor,
                    output logic clk_div);

  logic [9:0] count, threshold, max_value;
  logic overflow;
  assign max_value = '1;

  always_ff @(posedge clk, negedge reset_n)
    begin
        if (!reset_n)
          count <= 0;
        else if (count >= threshold)
          begin
          count <= 0;
          overflow <= 1;
          end
        else
          begin
          overflow <= 0;
          count = 1 + count;
```

```
        end
    end

  always_ff @(posedge overflow, negedge reset_n)
    begin
        if (!reset_n)
          clk_div <= 0;
        else
          clk_div <= !clk_div;
    end

    always_comb
    begin
        threshold = max_value - divisor;
    end

endmodule
```

5.2 Simulation Results

5.2.1 Top Level

Because of the number of inputs and outputs and the diverse functions of the system, it is difficult to simulate the entire system and still get a coherent picture. Another problem is the difference in time scales between the different functional blocks. Some work at scales of microseconds, others at hundreds of microseconds, others at milliseconds and so on. Additionally, the inputs depend on dynamic signals that are not easy to reproduce. To make the process more manageable, we decided to simulate it using the NESEmulator module, and capture some of the most interesting and representative sections of the waveforms.

Do file:

```
vsim -gui work.FinalProject
add wave -divider "Global signals"
add wave -position end  sim:/FinalProject/clk
add wave -position end  sim:/FinalProject/reset
add wave -position end  sim:/FinalProject/input_select
add wave -divider "Keyboard"
add wave -position end  sim:/FinalProject/ps2_clk
add wave -position end  sim:/FinalProject/ps2_data
add wave -divider "Global signals"
add wave -position end  sim:/FinalProject/nes_clk
add wave -position end  sim:/FinalProject/nes_latch
```

```
add wave -position end  sim:/FinalProject/nes_data
add wave -divider "Game Controller"
add wave -position end  sim:/FinalProject/speed
add wave -divider "X Axis"
add wave -position end  sim:/FinalProject/dir_x
add wave -position end  sim:/FinalProject/step_x
add wave -position end  sim:/FinalProject/m1_x
add wave -position end  sim:/FinalProject/m2_x
add wave -divider "Y Axis"
add wave -position end  sim:/FinalProject/dir_y
add wave -position end  sim:/FinalProject/step_y
add wave -position end  sim:/FinalProject/m1_y
add wave -position end  sim:/FinalProject/m2_y
add wave -divider "Z Axis"
add wave -position end  sim:/FinalProject/dir_z
add wave -position end  sim:/FinalProject/step_z
add wave -position end  sim:/FinalProject/m1_z
add wave -position end  sim:/FinalProject/m2_z
add wave -divider "VGA"
add wave -position end  sim:/FinalProject/VSync
add wave -position end  sim:/FinalProject/HSync
add wave -position end  sim:/FinalProject/Blue
add wave -position end  sim:/FinalProject/Green
add wave -position end  sim:/FinalProject/Red
add wave -divider "7 Segment Display"
add wave -position end  sim:/FinalProject/Seg0
add wave -position end  sim:/FinalProject/Seg1
add wave -position end  sim:/FinalProject/Seg2
force clk 0 0, 1 10 ns -repeat 20 ns
force reset 0 0, 1 20 ns
force input_select 0 0
force speed 10'h3fd
force b2v_inst100/preload 7'b0111111
run 300 us
```

Then for every iteration:

```
force b2v_inst100/preload 7'b(new pattern)
run 300 us
```

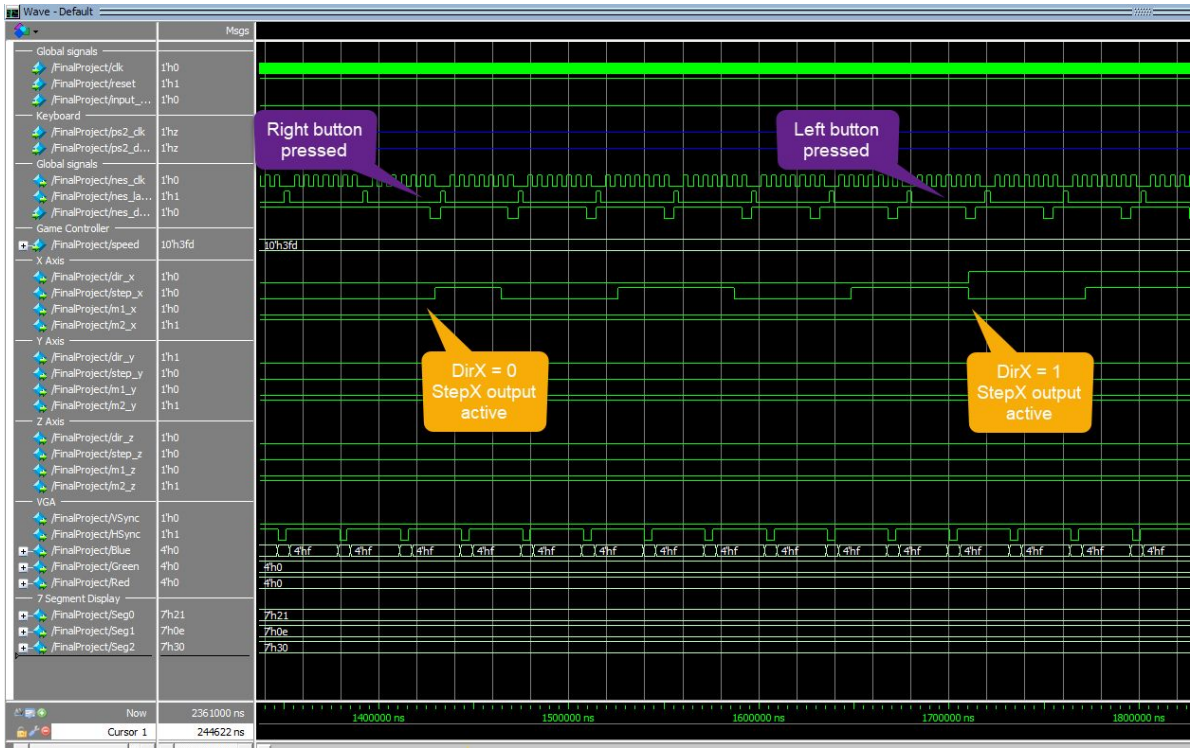


Figure 29: Right and left actions - X axis motor

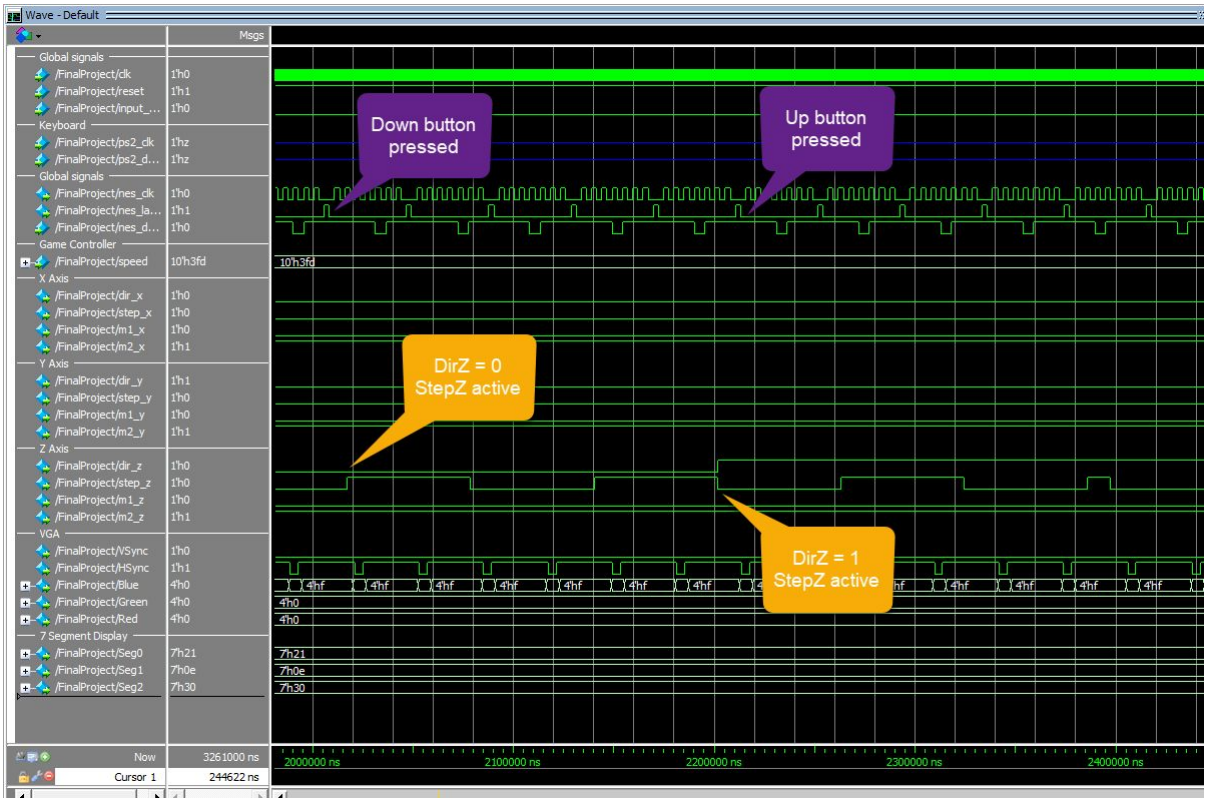


Figure 30: Up and down actions - Z axis motor

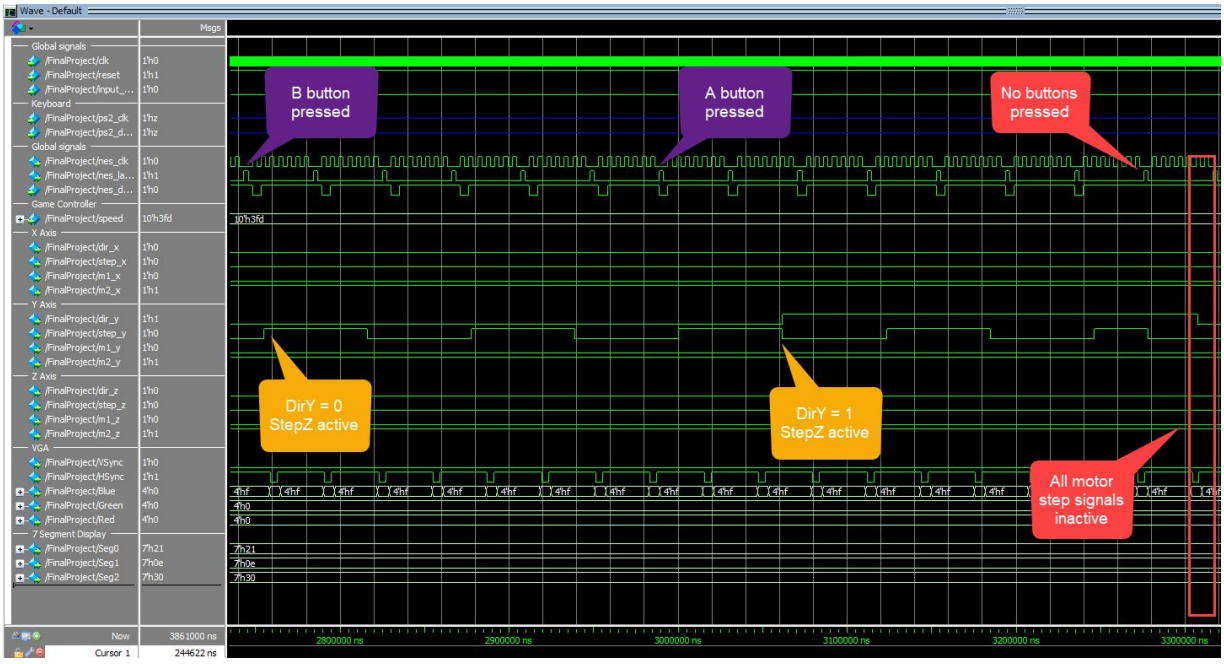


Figure 31: A and b actions - Y axis motor. Also, no action, not motor activity.

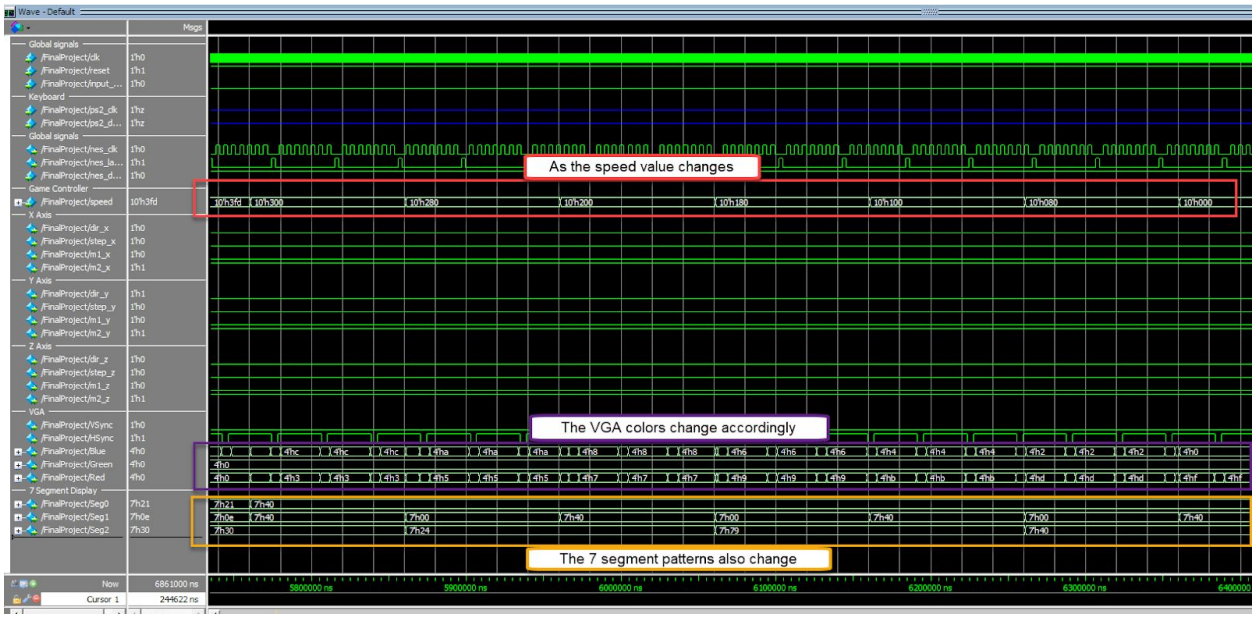


Figure 32: VGA and 7-segment displays reacting to speed changes.

5.2.2 NES Controller

Since the NES controller is an external device driven by the NESReceived module, we wrote a test module that emulates its behavior:

```
module NESEmulator (input  logic latch,
                    input  logic clock,
                    output logic data);

logic [7:0] register;
logic [7:0] count = 8'h00;

always_ff @(posedge latch)
begin
    register <= count;
    count <= count + 1;
end

always_ff @(posedge clock)
begin
    register <= {register[7:0], 1'b0};
end

assign data = register[7];

endmodule
```

What the emulator does is shift out the value of a counter that is incremented with each latch signal. In this way, the simulation can test all the possible combinations of buttons that the NES controller can produce. To do the simulations, we built a testbench circuit that ties up the emulator to the receiver and used it to test both receiver as a unit, and the whole receiver/decoded subsystem as a functional unit.

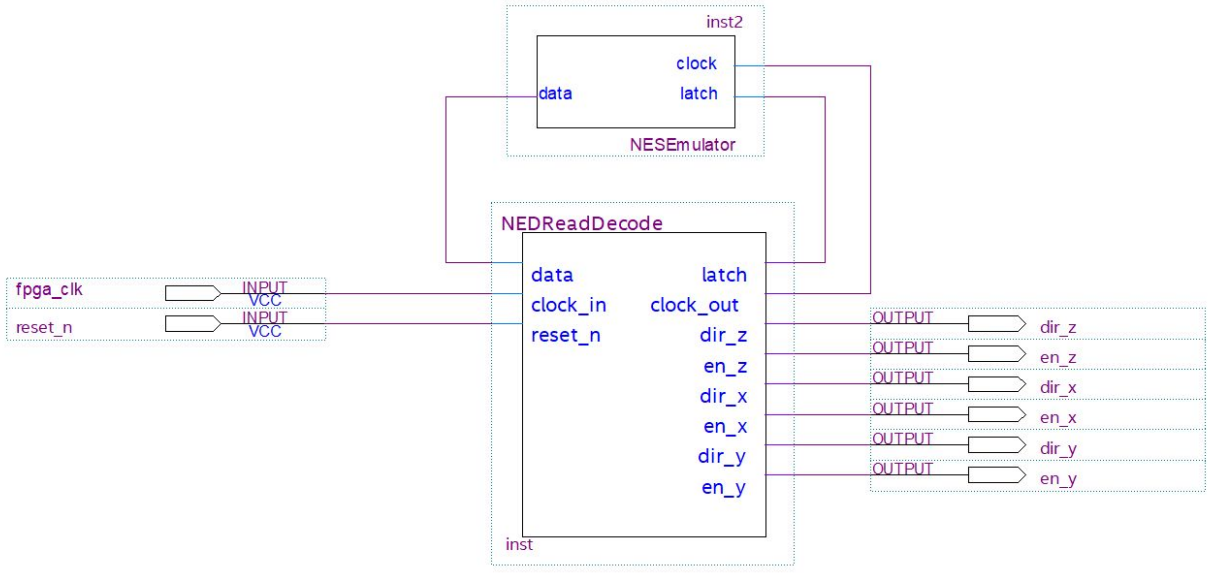


Figure 33: Circuit used to test the game controller reader/decoder.

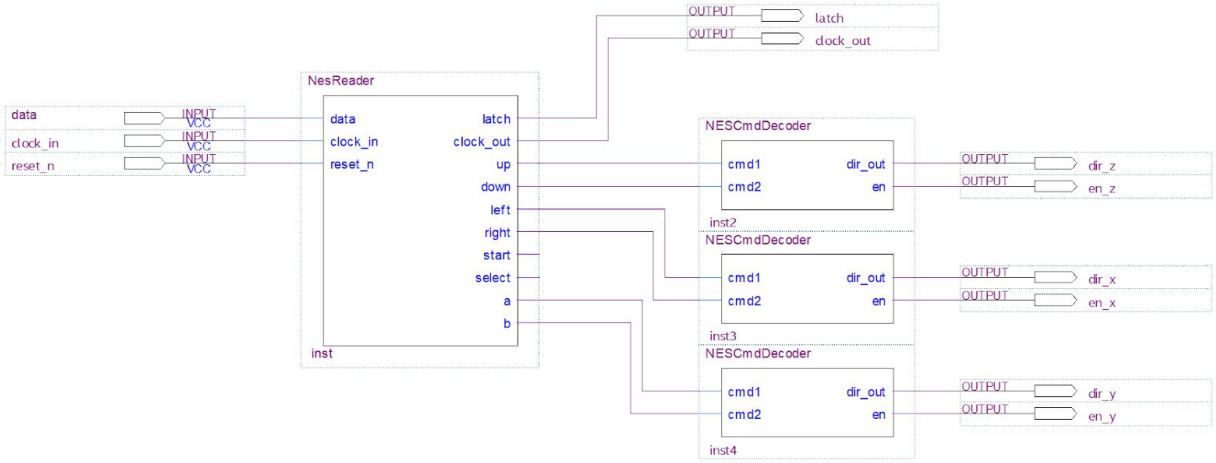


Figure 34: Detailed view of the components being tested.

Using this setup, the functional simulations was run with the following do file:

```
vsim -gui work.nesFunctional
add wave -position end  sim:/nesFunctional/b2v_inst/clock_out
add wave -position end  sim:/nesFunctional/b2v_inst/latch
add wave -position end  sim:/nesFunctional/b2v_inst/data
add wave -position end  sim:/nesFunctional/b2v_inst/b2v_inst/left
add wave -position end  sim:/nesFunctional/b2v_inst/b2v_inst/right
add wave -position end  sim:/nesFunctional/b2v_inst/b2v_inst/up
add wave -position end  sim:/nesFunctional/b2v_inst/b2v_inst/down
```

```
add wave -position end  sim:/nesFunctional/b2v_inst/b2v_inst/a
add wave -position end  sim:/nesFunctional/b2v_inst/b2v_inst/b
add wave -position end  sim:/nesFunctional/dir_z
add wave -position end  sim:/nesFunctional/en_z
add wave -position end  sim:/nesFunctional/dir_x
add wave -position end  sim:/nesFunctional/en_x
add wave -position end  sim:/nesFunctional/dir_y
add wave -position end  sim:/nesFunctional/en_y
force fpga_clk 0 0, 1 10 ns -repeat 20 ns
force reset_n 0 0, 1 20 ns
run 20 ms
```

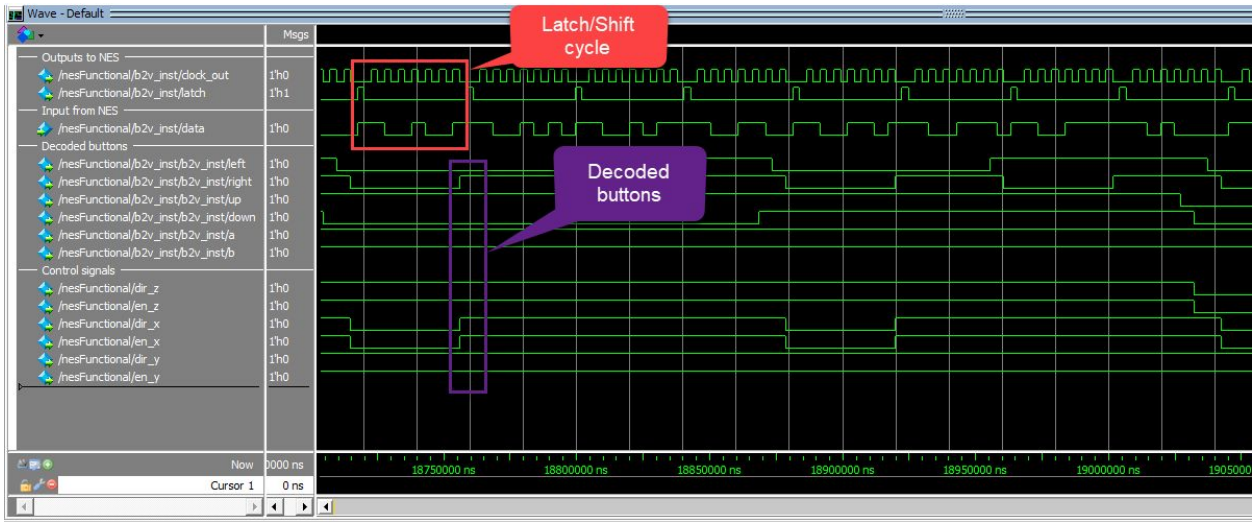


Figure 35: NES receiver reconstructing the button statuses.

NES Decoder

Commands:

```
force cmd1 0 0, 1 100 ns, 0 200 ns, 1 500 ns
force cmd2 0 0, 1 300 ns, 0 400 ns, 1 500 ns
run 600
```

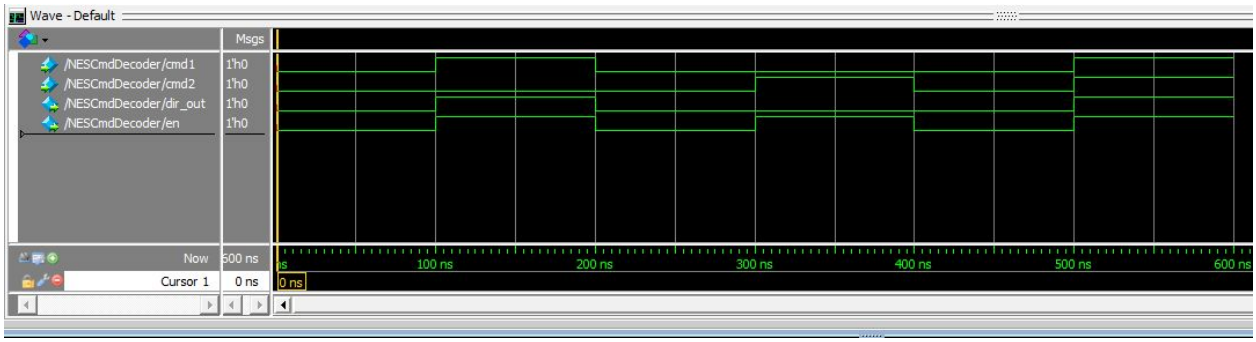


Figure 36: NES Decoder function

5.2.3 PS/2 Keyboard

To simulate the D key being pressed, the following commands were run:

```
force Clock 0 @ 0, 1 @ 1, 0 @ 2
force Clock 1 @ 5, 0 @ 6, 1 @ 10, 0 @ 11, 1 @ 15, 0 @ 16, 1 @ 20, 0 @ 21, 1 @ 25, 0 @ 26, 1 @ 30, 0 @ 31, 1 @ 35, 0 @ 36, 1 @ 40, 0 @ 41, 1 @ 45, 0 @ 46, 1 @ 50, 0 @ 51, 1 @ 55, 0 @ 56
force -freeze sim:/PS2Keyboard/DecoderCLK 1 0, 0 {1 ps} -r 2
force Data 1 @ 5, 0 @ 10, 1 @ 15, 1 @ 20, 1 @ 25, 0 @ 30, 0 @ 35, 0 @ 40
```

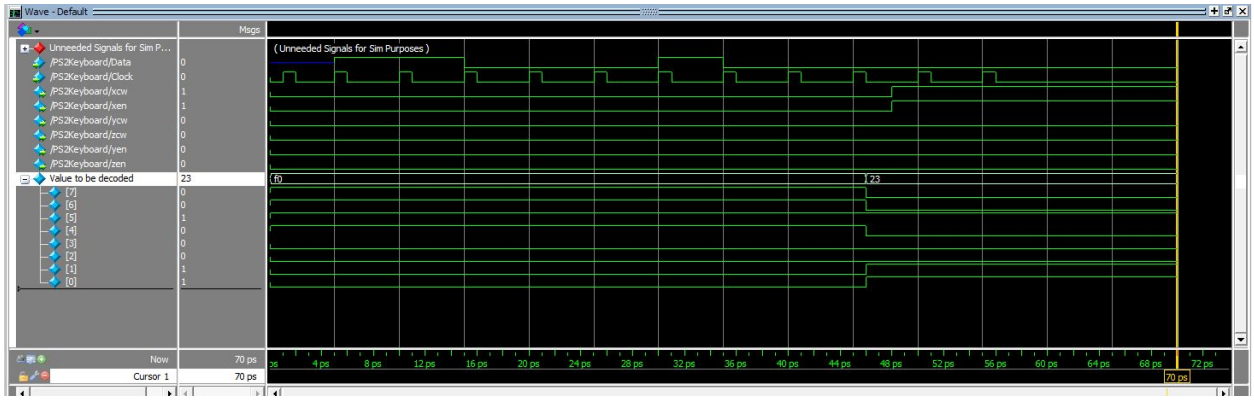


Figure 37: PS2 Decoder transforming the D key code into motor instructions

To simulate the S key being pressed, the following commands were run:

```
force Clock 0 @ 0, 1 @ 1, 0 @ 2
force Clock 1 @ 5, 0 @ 6, 1 @ 10, 0 @ 11, 1 @ 15, 0 @ 16, 1 @ 20, 0 @ 21, 1 @ 25, 0 @ 26, 1 @ 30, 0 @ 31, 1 @ 35, 0 @ 36, 1 @ 40, 0 @ 41, 1 @ 45, 0 @ 46, 1 @ 50, 0 @ 51, 1 @ 55, 0 @ 56
force -freeze sim:/PS2Keyboard/DecoderCLK 1 0, 0 {1 ps} -r 2
force Data 1 @ 5, 1 @ 10, 0 @ 15, 1 @ 20, 1 @ 25, 0 @ 30, 0 @ 35, 0 @ 40
```

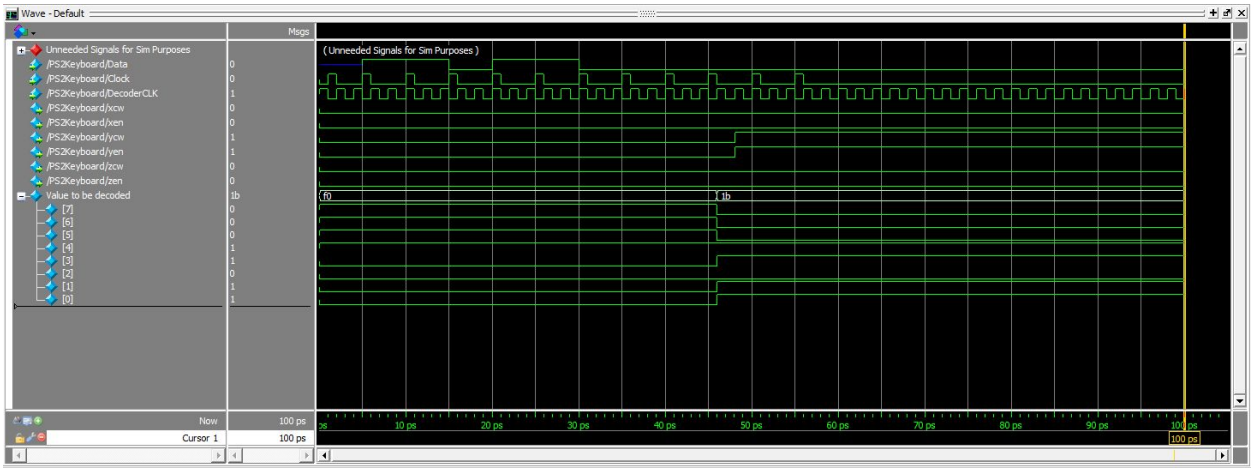


Figure 38: PS2 Decoder transforming the S key code into motor instructions

To simulate the left arrow key being pressed, the following commands were run:

```
force Clock 0 @ 0, 1 @ 1, 0 @ 2
force Clock 1 @ 5, 0 @ 6, 1 @ 10, 0 @ 11, 1 @ 15, 0 @ 16, 1 @ 20, 0 @ 21, 1 @ 25, 0 @ 26, 1 @ 30, 0 @ 31, 1 @ 35, 0 @ 36, 1 @ 40, 0 @ 41, 1 @ 45, 0 @ 46, 1 @ 50, 0 @ 51, 1 @ 55, 0 @ 56
force -freeze sim:/PS2Keyboard/DecoderCLK 1 0, 0 {1 ps} -r 2
force Data 1 @ 5, 1 @ 10, 0 @ 15, 1 @ 20, 0 @ 25, 1 @ 30, 1 @ 35, 0 @ 40
```

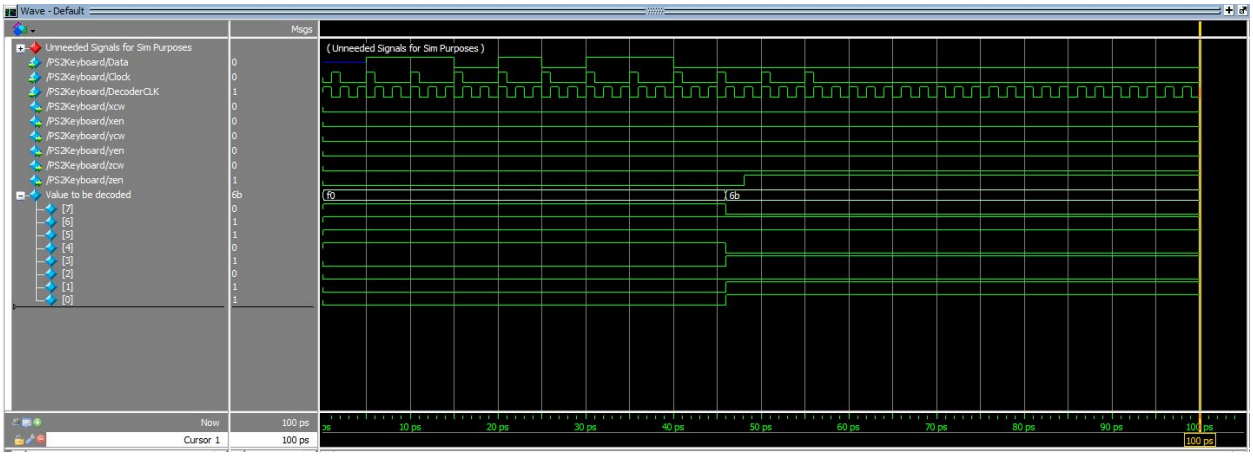


Figure 39: PS2 Decoder transforming the ← key code into motor instructions

5.2.4 Analog Potentiometer

The potentiometer/ADC are external to the FPGA, so it cannot be simulated. We tested that the subsystem was working correctly by leveraging the seven-segment display.

5.2.5 DE10-Lite 7-Segment Display

The seven segment display logic was used several times in lab projects during the term so it was tried and true. The screenshot below shows the expected behavior of the display with the display patterns changing as the ADC value changes;

Do file:

```
vsim -gui work.ADCSegBlock
add wave -position end  sim:/ADCSegBlock/ADCVal
add wave -position end  sim:/ADCSegBlock/BiggerVal
add wave -position end  sim:/ADCSegBlock/BiggestVal
add wave -position end  sim:/ADCSegBlock/SmallerVal
force ADCVal 10'h000
run
force ADCVal 10'h00F
run
force ADCVal 10'h0F0
run
force ADCVal 10'hF00
run
force ADCVal 10'hF0F
run
force ADCVal 10'h0FF
run
force ADCVal 10'h3FF
run
force ADCVal 10'hAAA
run
```

Figure 40: 7 segment display splitting 10-bit value into the separate digits.

5.2.6 VGA Output

The following screenshots depict different areas of the same simulation results, showing interesting parts of a full video frame. The simulation do file is:

```
vsim -gui work.VGA
add wave -position end  sim:/VGA/Clock
add wave -position end  sim:/VGA/Reset
add wave -position end  sim:/VGA/ADCIn
add wave -position end  sim:/VGA/H_Sync
add wave -position end  sim:/VGA/V_Sync
add wave -position end  sim:/VGA/Blue
add wave -position end  sim:/VGA/Green
add wave -position end  sim:/VGA/Red
force Clock 0 0, 1 10 ns -repeat 20 ns
force Reset 0 0, 1 20 ns
force ADCIn 10'h0
run 30 ms
```

Initialization

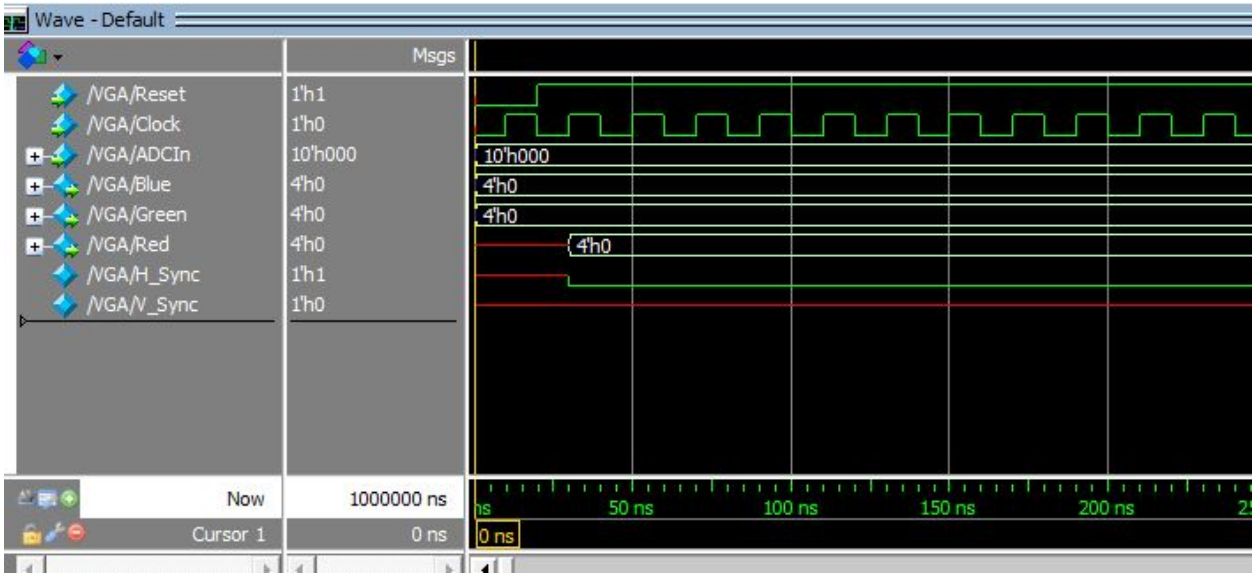


Figure 41: VGA startup

Start of sync signals

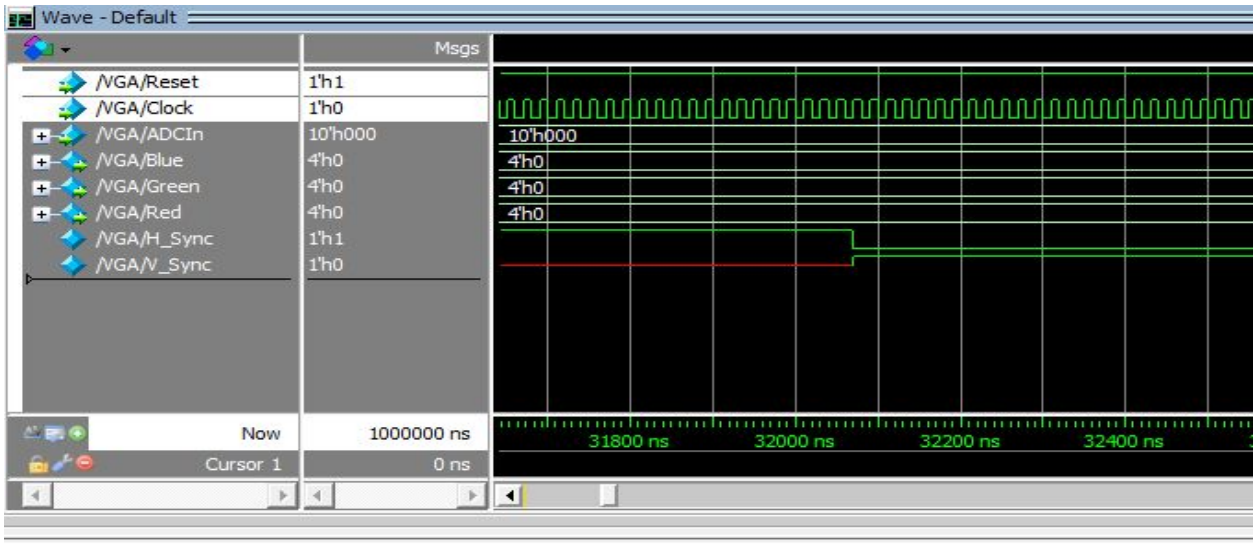


Figure 42: HSync and VSync startup.

Representative VGA waveforms (clock and reset removed for clarity).

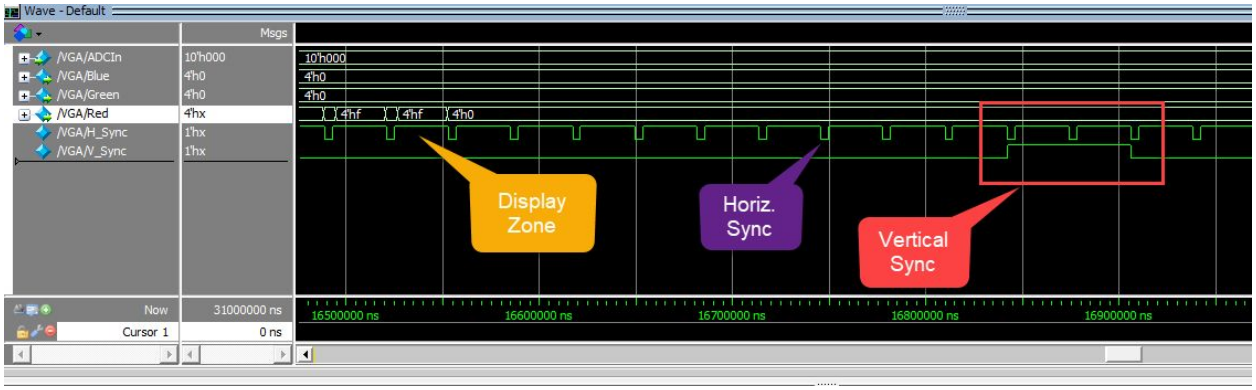


Figure 43: Pixel and line signals along with display color data.

5.2.7 DC Motor

For the motor simulation we used the following do file:

```
vsim -gui work.MotorController
add wave -position end  sim:/MotorController/clk
add wave -position end  sim:/MotorController/enable
add wave -position end  sim:/MotorController/dir_in
add wave -position end  sim:/MotorController/reset_n
add wave -position end  sim:/MotorController/speed
add wave -position end  sim:/MotorController/step
add wave -position end  sim:/MotorController/dir
```

```
add wave -position end  sim:/MotorController/m1
add wave -position end  sim:/MotorController/m2
add wave -position end  sim:/MotorController/step_out
add wave -position end  sim:/MotorController/motor_pulse
force clk 0 0, 1 10 ns -repeat 20 ns
force reset_n 0 0, 1 20 ns
force dir_in 0 0
force enable 0 0, 1 20 us
force speed 10'h3e0
run 20 ms
force speed 10'h2ff
run 20 ms
force dir_in 1 0
force speed 10'h200
run 20 ms
```

The motor is simple in terms of inputs and outputs. The simulation shows the change frequency for the step signal as the speed changes, the higher the speed the higher the frequency.

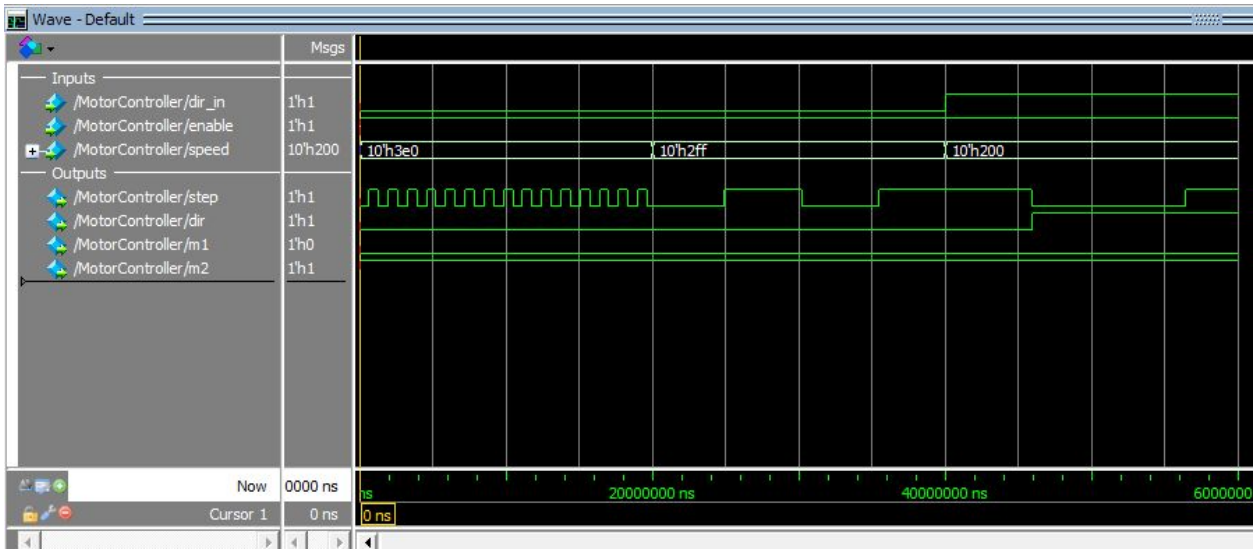


Figure 44: Frequency response too speed input.

The screenshot below highlights the synchronization mechanism that insures that control signals comply with the setup and hold times specified by the motor driver IC.

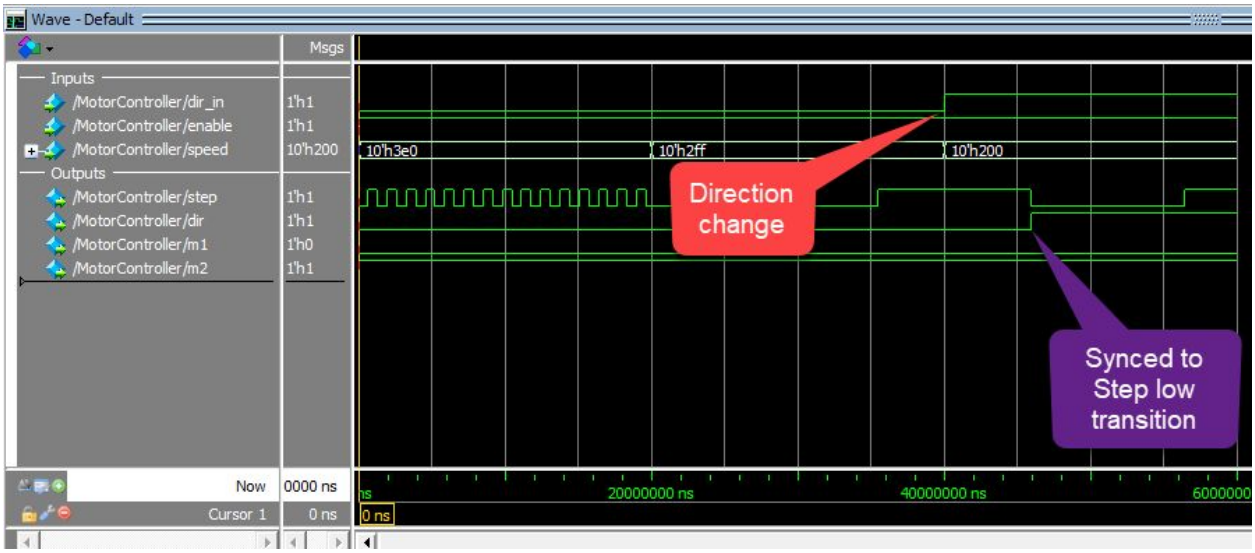


Figure 45: Signal timings.

6. Physical Implementation

We implemented our design on the actual FPGA board. Here is a link to a YouTube video showing it in action.

7. References

[1] All About Circuits, “NES Controller Interface with an Arduino UNO”
https://www.allaboutcircuits.com/projects/nes-controller-interface-with-an-arduino-uno/

[2] Adam Chapweske, “PS/2 Mouse/Keyboard Protocol”
http://www.burtonsys.com/ps2_chapweske.htm

[3] Terasic Inc. “ DE10-Lite User Manual” http://eecs.oregonstate.edu/tekbots/courses/ece272

[4] ECE 272. “Section 6: Video Graphics Array (VGA)”
http://eecs.oregonstate.edu/tekbots/courses/ece272/section6

[5] Allegro MicroSystems. “A3967: Microstepping Driver with Translator”
https://www.allegromicro.com/en/Products/Motor-Drivers/Brush-DC-Motor-Drivers/A3\67