ECE272 Final Design

Controller Input and VGA

Drew Ortega
Cole Swanson
Jonathan Alexander

November 30, 2018

Contents

1 Introduction **2**

2 Top Level Design **2**

3 Video Output **3**

4 Controllers **4**
- 4.1 NES/SNES Controller . . . 4
- 4.2 IR Controller . . . 5
  - 4.2.1 Self-Clocking Signal . . . 5
  - 4.2.2 IR Signal & RC 5 Code . . . 5
- 4.3 PS/2 Controller . . . 6

5 HDL Modules **7**
- 5.1 NES/SNES input decoder . . . 7
- 5.2 IR Input Top Module . . . 8
  - 5.2.1 IR Clocks . . . 9
  - 5.2.2 IR Simplifier . . . 11
  - 5.2.3 IR Parser . . . 12
- 5.3 PS/2 Input Decoder . . . 13
- 5.4 Input Decoder/Multiplexer . . . 14
- 5.5 Game Logic Controller . . . 15
- 5.6 Player Top Module . . . 16
- 5.7 Pixel Counter . . . 17
- 5.8 Renderer . . . 19

6 Putting it Together **20**

7 Extra Implementation **21**

8 References **21**

9 Appendix **21**
- 9.1 Source Code . . . 21
  - 9.1.1 NES/SNES Input Decoder . . . 21
  - 9.1.2 IR Input Top Module . . . 23
  - 9.1.3 IR Clocks . . . 24
  - 9.1.4 IR Simplifier . . . 25
  - 9.1.5 IR Parser . . . 25
  - 9.1.6 PS/2 Input Decoder . . . 28
  - 9.1.7 Input Decoder/Multiplexer . . . 29
  - 9.1.8 Game Logic Controller . . . 31
  - 9.1.9 Player Top Module . . . 31
  - 9.1.10 Pixel Counter . . . 33
  - 9.1.11 Renderer . . . 34
- 9.2 Do Files . . . 36
  - 9.2.1 NES/SNES Input Decoder . . . 36
  - 9.2.2 IR Input Top Module . . . 37
  - 9.2.3 IR Clocks . . . 38
  - 9.2.4 IR Simplifier . . . 39
  - 9.2.5 IR Parser . . . 39
  - 9.2.6 PS/2 Input Decoder . . . 41
  - 9.2.7 Input Decoder/Multiplexer . . . 43
  - 9.2.8 Game Logic Controller . . . 45
  - 9.2.9 Player Top Module . . . 46
  - 9.2.10 Pixel Counter . . . 47
  - 9.2.11 Renderer . . . 47

1 Introduction

This design project is meant to create logic capable of receiving data from three different controller inputs: SNES/NES, IR, and PS/2. Using these inputs, the design should implement some form of movement and collision, which in this design will be allowing for control of two separate player objects that will interact when occupying the same square in a grid. The type of input used will be determined by a two bit mode input. Communication with each controller will be handled by separate decoders, and the data from each controller will be fed into the top level module, then decoded into the desired inputs: up, down, left, and right. The inputs pressed will determine the output displayed through a VGA output, which will consist of a 2x2 grid with squares being empty or filled by a player object, signified by a change in color. The VGA counter and decoder are based on designs from the ECE272 lab course.

This design seeks to implement a simple game, two players(differentiated by color, one red and one blue) in a grid that can collide with each other, resulting in a black coloration. The inputs for each player are handled separately, and as a result the input for either player can differ from the other. Specific inputs from each type of controller are mapped to the up, down, left, and right movements, and pressing those buttons, or a combination of them, will result in movement of the player object. If both players are in the same square, the color of that square will be a combination of the two player's colors.

2 Top Level Design

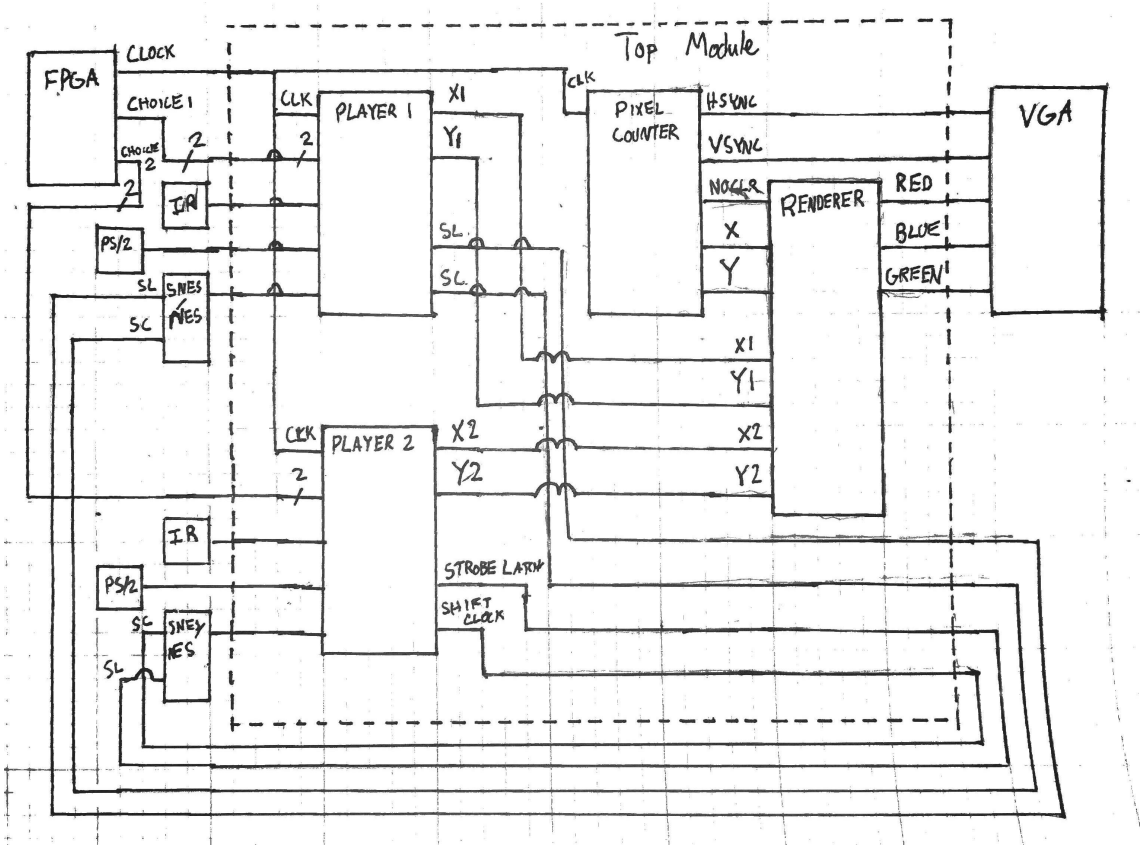

Figure 1: Top Level Design Block Diagram

In order to handle multiple inputs for each player, several layers of modules have to be created. Each player should handle their own inputs for a controller, with a 2-bit choice bus defining which controller input to use. The player module should handle this choice and make a decision on how

to read the data and what information to use. The player module should also take this input and alternate coordinates on a 2x2 grid. It should ensure that:

1. Ensure a player cannot leave the grid.
2. Ensure a player's position cannot be driven by multiple controllers at once.
3. Recognize that a NES and SNES controller may use the same port; Use Choice to determine which controller type it is.

The player module should output an X coordinate and a Y coordinate for the player's position, and the Strobe Latch and Shift Clock that is required for the NES and SNES controllers. Since there are two players, each player must send its coordinate/position data to the renderer. The renderer decides for the 2x2 grid, and what color to make each square. An empty square should be black, Player 1 Red, Player 2 Blue, and if they are in the same position, Purple. H_Sync and V_Sync are outputted to VGA, as well as the per-pixel RGB values. More about VGA is found in Section 3. Throughout all of the design, a 50 MHz base clock is assumed.

Inputs: Two sets of inputs for each player. Each set contains a 2-bit bus 'Choice', representing which controller to use. Each set also contains 3 different single bit serial inputs, representing the input from each controller or type, for SNES/NES, PS-2, and IR. A 50 MHz base Clock is also a required input.

Outputs: Two sets of a Strobe Latch and Shift Clock are outputted, which are specific to the possible SNES/NES controller input. The Design also outputs two single bit values H_Sync and V_Sync, representing timing specifications for VGA output. It also outputs 3 4-bit busses Red, Blue, and Green that are specific to VGA.

3 Video Output

The VGA port has 5 pins that actually carry data. There are HSync and VSync, which stand for Horizontal Sync and Vertical Sync. When these are pulsed, they mark the end of the current line of pixels or frame. The other 3 pins each represent one channel of color: red, green, and blue. These are, at the appropriate times, driven to an analogue signal representing the strength of the color at a certain position.

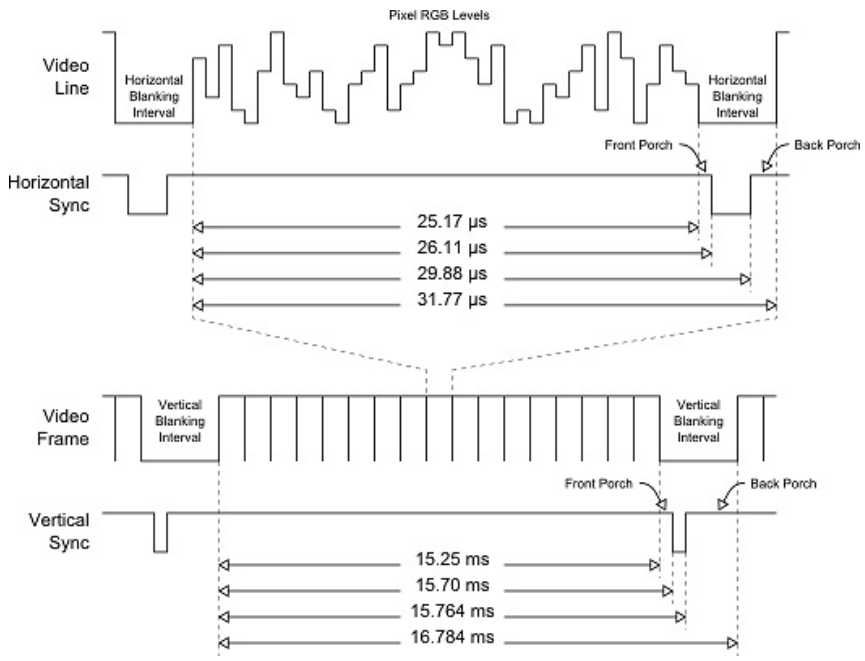

Figure 2: Diagram illustrating the timing of a line and how it fits into a frame [5]

VGA is all about timing out the HSync and VSync pulses. There are two counts to keep track of: the current pixel, and the current line. Every clock cycle the pixel count is incremented by one. The pixel counts until the horizontal state resets, at which point the line count is incremented by 1. Pixels are drawn from left to right, then top to bottom, like reading a book. There are four possible states the horizontal and vertical timings can be in: Display Time, Front Porch, Sync Pulse, and Back Porch. The Display Time is when you actually drive the color output to specific values. The pixel you're coloring can be found by consulting the current pixel and line, which serve as x and y coordinates. The two porches surround the sync pulse, and are periods where the colors are not being driven, but the sync is not being pulsed either. Then the Sync Pulse state is exactly as it seems, it's the time when the sync is pulsed to low. This base format is followed for every resolution and framerate you could want, the only thing that changes is the speed of the clock signal, and the specific times spent in each state. In our case, we are targeting 640 x 480 60Hz, so 640 horizontal pixels, 480 vertical pixels, at 60 frames per second.

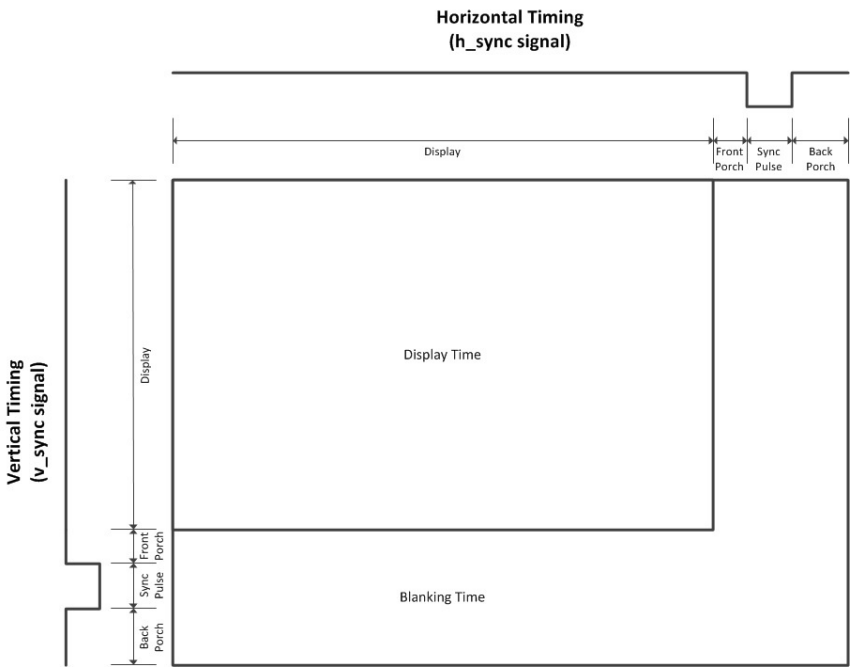

Figure 3: Diagram illustrating when color is output [4]

Whenever both the horizontal and vertical states are during Display Time, the color pins are being driven to color the current pixel, found by checking the current pixel and line count. Otherwise, the color pins are being driven to 0.

4 Controllers

4.1 NES/SNES Controller

The NES and SNES controllers utilize three lines to transfer data, along with a ground and 5V power supply. Communication with the controllers uses a data latch line, clock line and data line. The clock and data latch lines are supplied to the controller, while the data line is generated. When the latch line is driven high, the controllers log the pressed buttons within an active low shift register. The first bit of this register is automatically output on the data line. Whenever the clock input is cycled, the shift register moves along by one bit. The NES controller has eight buttons, and an eight bit shift register, while the SNES has sixteen buttons and a sixteen bit shift register. Within the NES controller, the zeroth bit of the shift register corresponds to the right input, the first to the left input, the second to the down input, and the fourth to the up input[7]. The final four inputs were ignored. Within the SNES controller, the up input corresponds to the fifth bit in the shift register, the down input to the sixth bit, the left input to the seventh bit, and the right input to the eight bit[6].

To communicate with the NES or SNES controllers, the host must do the following:

1. Drive the data latch input high to populate the shift register
2. The first data bit can now be read
3. The clock input should be driven high, then low
4. The next data bit can now be read
5. Steps 3-4 should be repeated 7 times for the NES, 15 times for the SNES(as the first bit of the register is already present, one less clock cycle is needed)[6]

4.2 IR Controller

4.2.1 Self-Clocking Signal

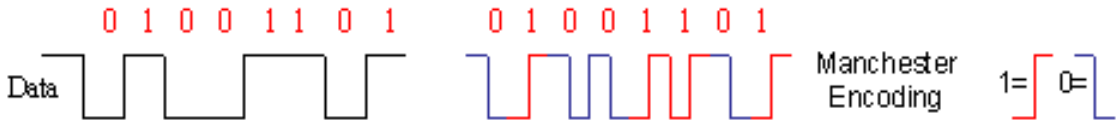

Figure 4: Manchester Code/Bi-Phase Coding: from Data to Output[1]

The IR Controller uses a self-clocking signal to send data to the receiver. For a signal to be self-clocking, the source must have a clock of a specified speed and a serial data output. The source combines the clock and serial output in to a single serial output of a designated encoding. For the IR Controller, the IEEE Manchester Code specification, also known as "Bi-Phase Coding" (BPC), is used. For BPC, the source checks the data it wishes to output on every rising edge of its clock. If the data to be transmitted is a 1, than the serial output is driven to mimic the clock for that cycle. Essentially, it rises on a rising edge of a clock, and falls on the falling edge of the clock. If the data to be transmitted is a 0, than the serial output begins to be the inverted signal of the clock. It falls on the rising edge of the clock, and rises on the falling edge of the clock.

What a Bi-Phase Coding might look like can be found in Figure 4. The left side of the figure represents what a serial data stream might look like normally, and the right shows the Bi-Phase version of the same stream.

4.2.2 IR Signal & RC 5 Code

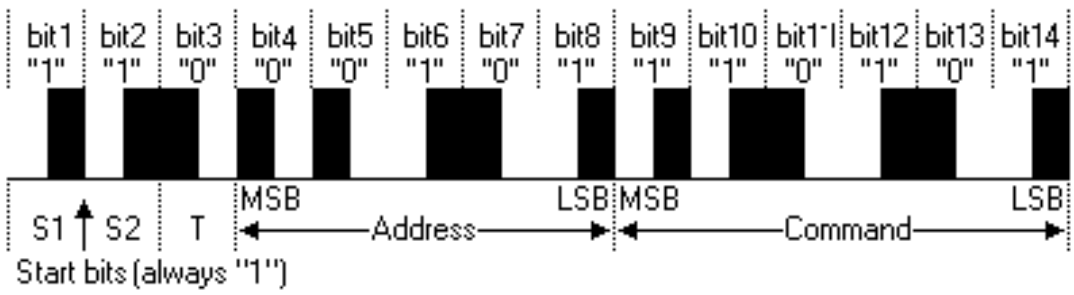

Figure 5: RC 5 Code - Chunk and delay [2]

This implementation of IR reception assumes that the IR signal received will be in the form of Manchester Code, as explained in 4.2.1. The IR Decoder uses the RC-5 standard. RC-5 sends 32

bursts at a frequency of 36 kHz for each time the Manchester Data should be 1, and does not send any data at any other time. This means that the period of the actual Manchester code is 1.125 kHz/1125 Hz. The actual sent Bi-Phase/Manchester code has a certain specification as well. The first two bits sent will always be 1, which helps the receiver gauge timing. Each section of the expected input is as follows:

- Section 1 - Start Bits (2 bits): Help orient the reciever to understand source clock timing.
- Section 2 - Toggle Bit (1 bit): Defines whether or not this chunk is a new key press or a held key already transmitted.
- Section 3 - Address (5 bits): Defines the address of the receiver. If this does not match the expected address, the data is ignored.
- Section 4 - Data (6 bits): Defines the actual data of the chunk, up to 6 bits. A single combination represents an individual button.

After the full chunk has been transmitted, the serial data signal is driven to 0 for 50 clock cycles before data is transmitted again. After the 50 clock cycles, if a button on the controller is still being pressed, the transmitter on the controller will send the same data chunk again, except the toggle bit will be 0 instead of 1. If after 50 cycles no data is read, there is no button being pressed or the receiver is not receiving data.[3] The structure and timing of RC 5 Code can be found in Figure 5.

4.3 PS/2 Controller

The standard for PS/2 has multiple possible states for the peripheral device. It can be in its idle state, a communication halt state, and a host-to-device state. For this project's purposes, we will only implement the idle state, as it is the default state, where the device sends key press and unpress updates to the host. The idle state's standard functions almost exactly like a Serial Peripheral Interface. The negative edge of the clock signal driven by the device indicates the data value for each bit. There is a standard for what each bit in the 11 bit sequence represents. The sequence is as follows:

- 1 start bit - This is always 0
- 8 data bits - Least significant bit first
- 1 parity bit - Odd parity
- 1 stop bit - This is always 1

The 8 data bits are a unique number that represents a specific key being pressed, or unpressed. The parity bit is a bit representing how many 1s there are in the data bit. It functions as a quick and simple error check, if the parity bit does not match the data then you know the data was corrupted somehow.

5 HDL Modules

5.1 NES/SNES input decoder

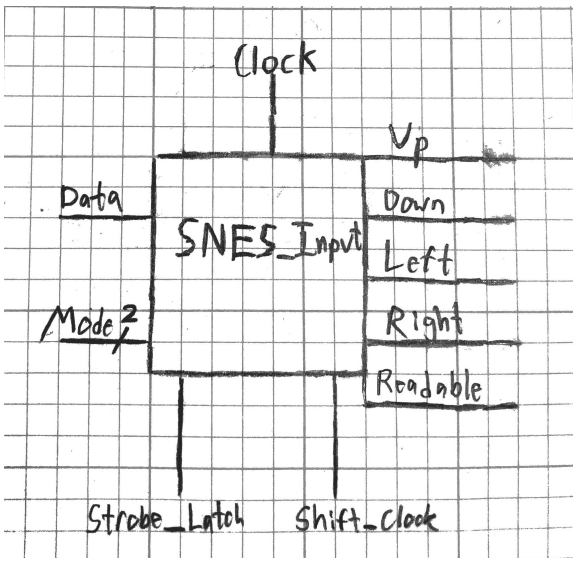

Figure 6: Block Diagram of the NES/SNES input decoder

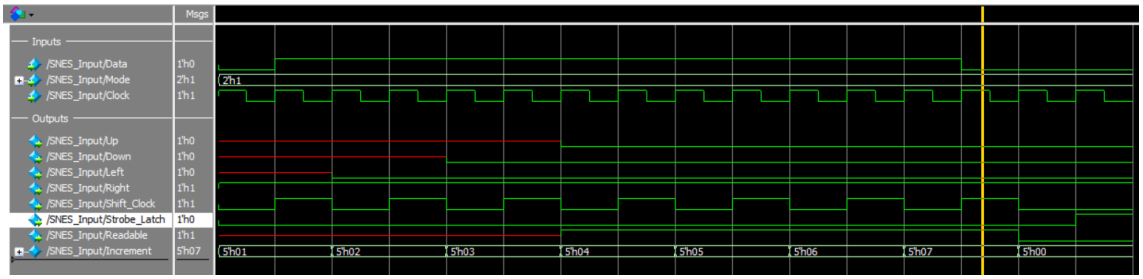

Figure 7: ModelSim of the NES/SNES controller decoder in NES mode.

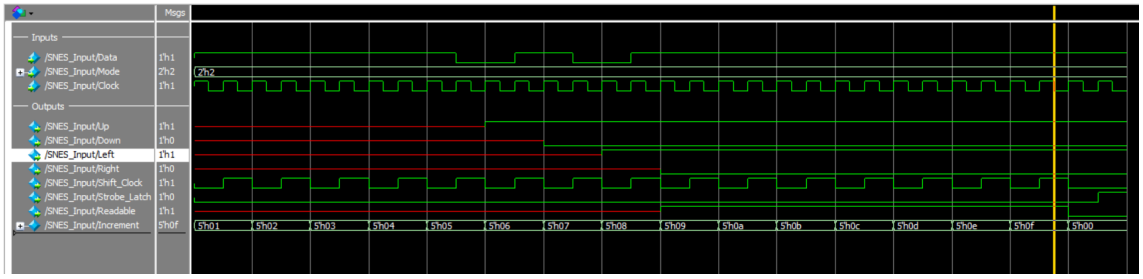

Figure 8: ModelSim of the NES/SNES controller decoder in SNES mode

Inputs: One bit data representing the current bit from the controller shift register, one bit clock, and two bit mode, of which the least significant bit is used to determine which controller is in use(NES or SNES)

Outputs: One bit up, down, left, and right outputs representing the decoded values from the controller, and one bit readable output which is driven high once the current set of values have

been decoded. Also present are the one bit strobe latch output, which functions as the data latch for the controller, which is driven high then low to log the inputs to the controller, and a one bit shift clock, which is used to shift the values in the controller shift register.

Simulation: The simulation for this module was broken up into two files, one for each controller type.

NES: For this simulation, the data bit for right was driven low(representing an input of 'right'). The data_latch is initially driven high. As expected, the decoder logs the right input as pressed, and continues through the inputs by providing a shift_clock to the controller. Once the data has been cycled three times, all values of interest are accounted for, and the readable output is driven high. After seven cycles of the output, the data_latch is again driven high and readable is driven low; new inputs are ready to be read. The value of increment represents the current bit of the shift register.

SNES: For this simulation, the data bits for up and left were driven low(representing inputs of 'left' and 'up'). The data_latch is initially driven high. As expected, the decoder logs the up and left inputs as pressed, and continues through the inputs by providing a shift_clock to the controller. Once the data has been cycled eight times, all values of interest are accounted for, and the readable output is driven high. After fifteen cycles of the output, the data_latch is again driven high and readable is driven low; new inputs are ready to be read. The value of increment represents the current bit of the shift register.

Description: This module will use the system clock to drive its logic. On the first clock cycle, the data latch output will be driven high to log the controller inputs. The data latch will then be driven low, and the shift clock will be cycled seven times for the NES controller, or fifteen times for the SNES controller, with the bit corresponding to the inputs of interest recorded. Once all inputs of interest have been recorded, the readable output will be driven high, signaling all inputs are accounted for. Pressed buttons on the controllers will be signaled by a high value on the corresponding output from the decoder(up, down, left, or right).

5.2 IR Input Top Module

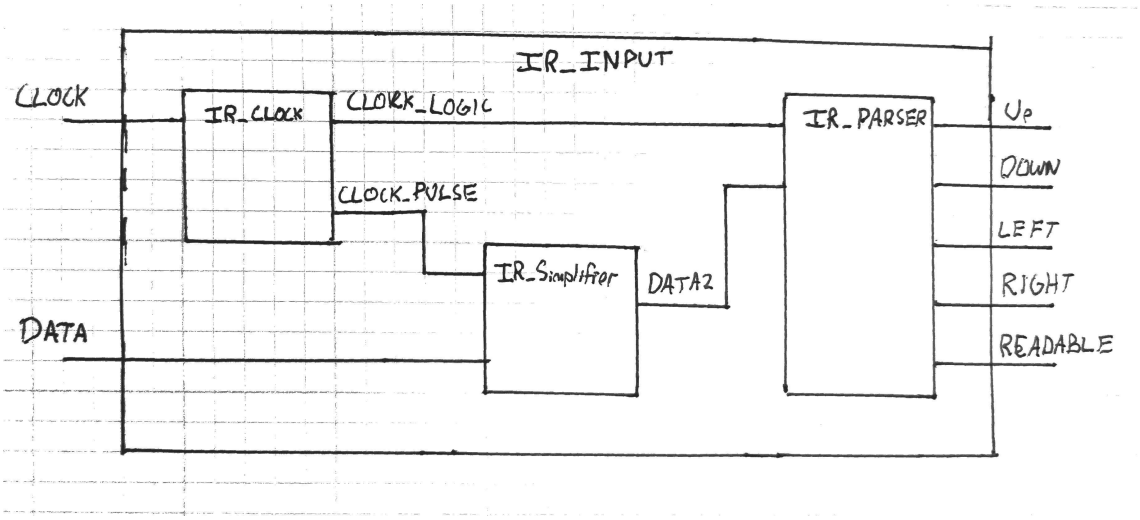

Figure 9: IR Top Module Diagram

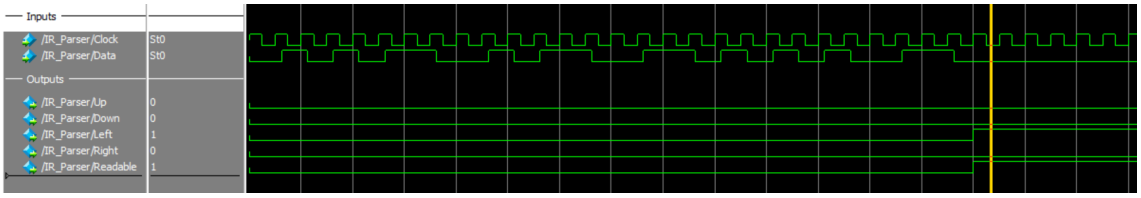

Figure 10: IR Top Module Simulation

Input: One Serial Data stream 'Data', Clock signal.

Output: One output for each cardinal direction (Up, Down, Left, Right), and signal describing whether or not the directions are valid, named 'Readable'.

Simulation: For this simulation, The same bi-phase input that was used for the parser simulation is used. The data is sent as a 11-1-01101-000010. The correct address is received (01101), so it parses the data. A "Left" press corresponds to 000010, so the Left output is raised to HIGH and Readable becomes True since ot now has valid output.

Description: For the IR Decoder, it was noted that its output must match the output of any other input decoder, such as the SNES/NES and PS/2. Because of this, the Up, Down, Left, Right, and Readable streams were required to be used for this particular module. This particular module assumes that Up is 0, Down is 1, left is 2, and Right is 3. See Sections 5.2.1, 5.2.2, and 5.2.3 for more detailed implementation on particular parts of IR.

5.2.1 IR Clocks

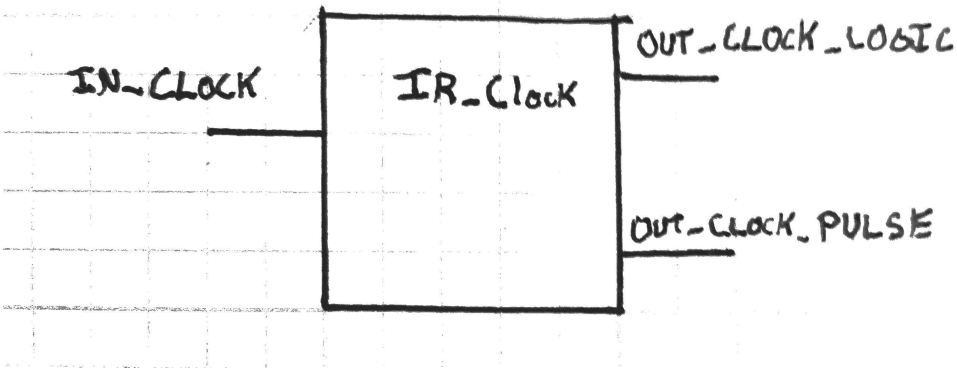

Figure 11: IR Clock Slowing Block Diagram

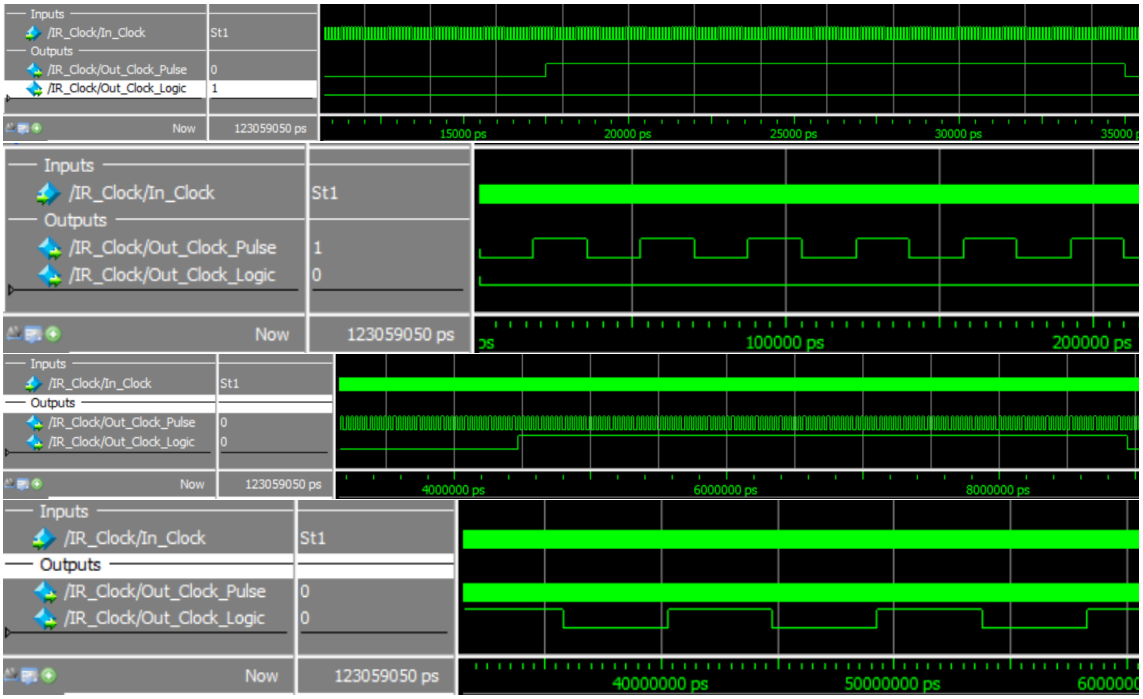

Figure 12: IR Clock Sim

Inputs: Takes in a single 50 MHz Clock.

Outputs: Outputs a 144 kHz clock for evaluating pulse simplification, and a 2.25 kHz Clock for evaluating manchester code values.

Simulation: In this simulation, a 50 MHz clock was inputted. If one were to count the amount of ticks that were waited, there 174 ticks between the original clock and the Pulse clock. There are 64 ticks between the logic clock and the pulse clock. This is the expected amount.

Description: The IR Input receiver takes in the BPC serial data stream and a 50MHz Clock signal. A slowed clock is generated to slow it down to approximately 72 kHz. This is achieved by dividing the 50 MHz clock by 694. However, if the Clock was flipped every 694 50 MHz cycles, it would have a clock cycle of 36 kHz. To account for this, we must double the resulting clock, by dividing the original clock by 347. With the 72 kHz clock, for each rising edge of this clock the IR signal will have either a rising edge or a falling edge. Since the IR signal is asynchronous, meaning its clock will not line up with our clock, the 72 kHz clock can check on each of its rising edges whether or not the Data input is a 0 or a 1. An example of what the async data vs our internal clock can be found in Figure 12.

5.2.2 IR Simplifier

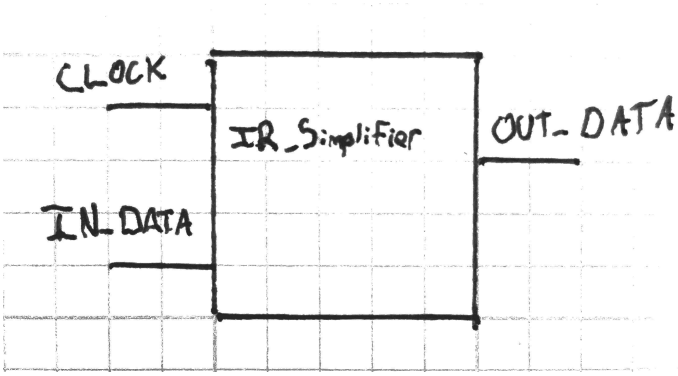

Figure 13: IR Simplifier Block Diagram

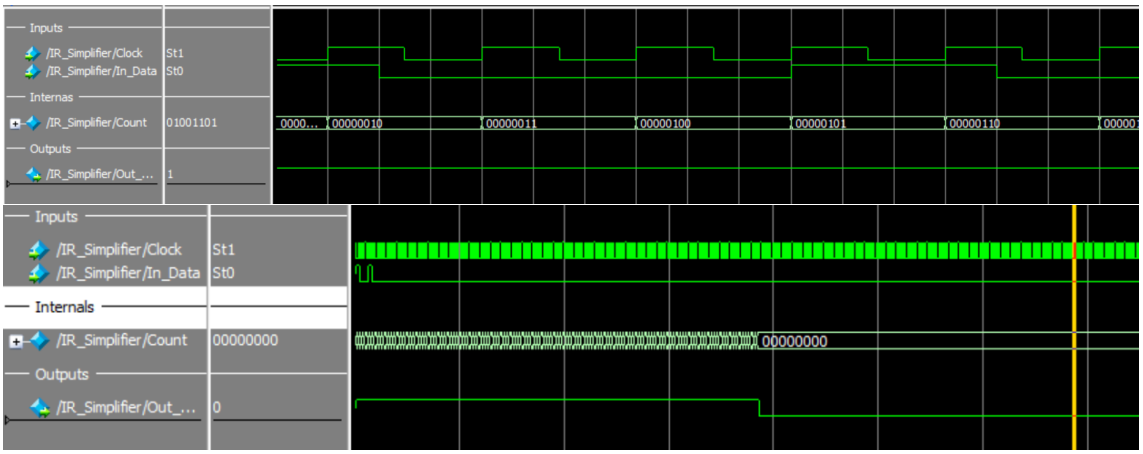

Figure 14: IR Simplifier Simulation

Inputs: The IR Simplifier takes in the 144 kHz Pulse evaluation clock and the raw data pulses from an IR sensor.

Outputs: A simplified data stream is outputted, as if the raw data did not pulse on HIGH.

Simulation: For this simulation, Only two raw data clock pulses were used. While not pulsing, the data was held at LOW, although realistically it would remain at X since no physical data is coming in. However, if it were at X, it would still work since it is only checked if the value is HIGH. No other pulses were used since any further pulses would have been ignored. As can be seen from the results of this simulation, Receiving the HIGH pulse made the simplified data output held HIGH for 32 ticks at 32 kHz, or 128 ticks at our faster 144 kHz.

Description: The rising edge of a clock occurs frequently enough that an average of two rising edges occurs on each pulse. As soon as a HIGH value is read, the rest of the input values are ignored and 32 ticks at 32 kHz are waited for. During this time, the simplified output is held at HIGH under the assumption that the IR data is still high. The parser should account for this as it is possible to be corrupted.

5.2.3 IR Parser

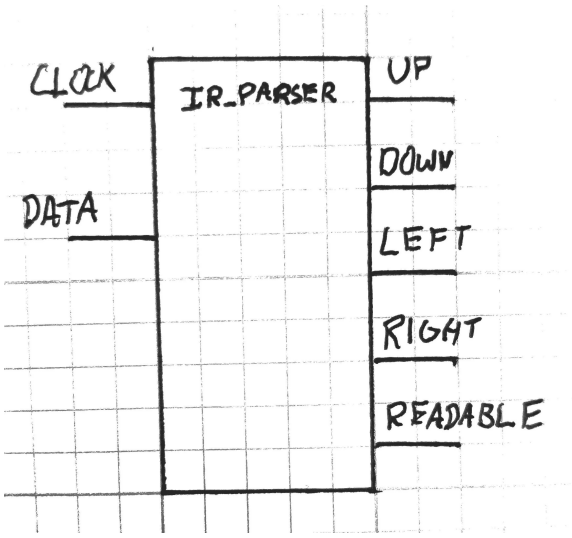

Figure 15: IR Parser Block Diagram

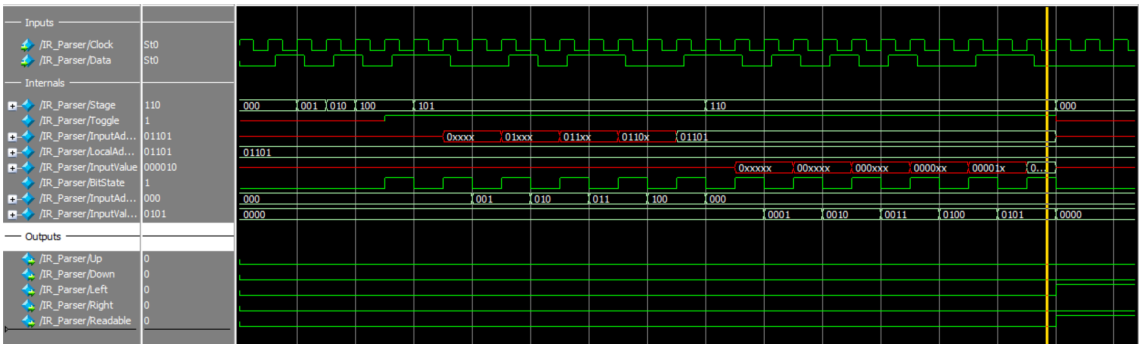

Figure 16: IR Decoder Simulation

Inputs: The IR parser takes in a 2.25 kHz clock and a simplified Data stream in manchester encoding, following RC-5 specification.

Outputs: The parser outputs Up, Down, Left, Right, to correspond which button was being pressed on this key press. It also outptus readable, specifying whether or not the output data is currently valid.

Simulation: For the IR parser simulation, The button "Left" was simulated. the manchester stream read in 11-1-01101-000010. The given address matched so it continued on. The value given, 000010, was hard coded to represent a Left key press. As can be seen from the simulation, the output is not changed until this keypress was completed.

Description: The IR parser's job is to have a rising edge for each falling and rising edge of the RC-5 code. Because of this, the IR data can come in at an asynchronous pace not ligning up with its own clock, and it will still read data correctly. The start bits help the parser decide when to actually start parsing and when it is at rest. It then reads in the toggle bit which is not used for this design project. It then reads the address, which is hard coded for this project to be 01101. If the read address matches the internal address, it considers the following 6 data bits to be valid. Only 4 possibilities of the 6 data bits are actually used. Values 000000-000010 represent all directions, and any other values are considered other button presses and do not affect the output.

5.3 PS/2 Input Decoder

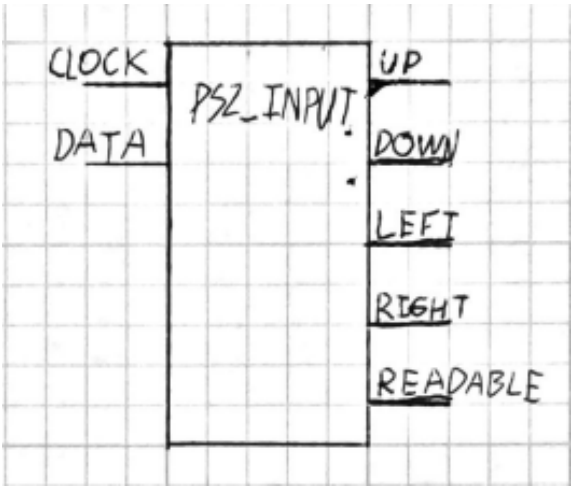

Figure 17: PS/2 Input Block

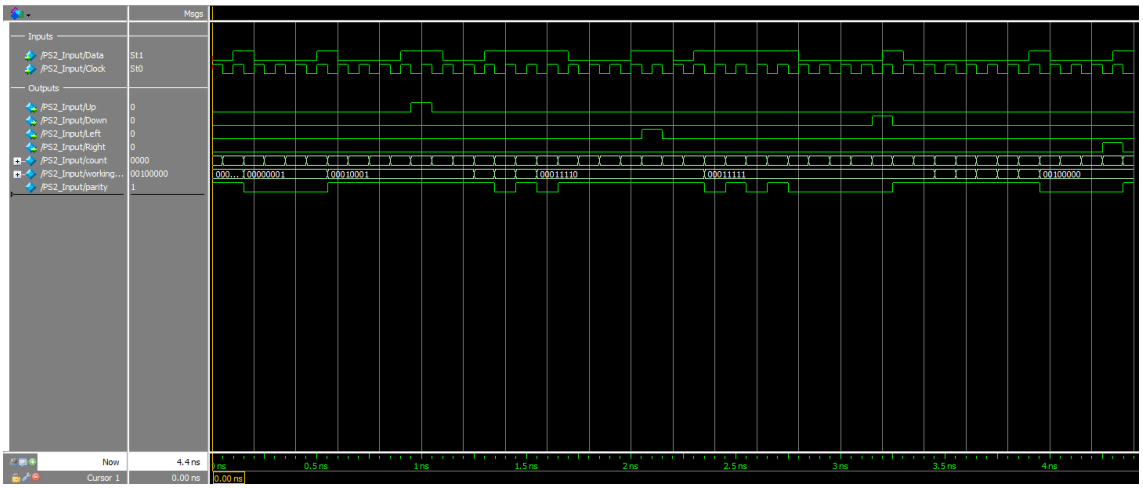

Figure 18: PS/2 Input Simulation

Inputs: There are two inputs for this module, the CLK and Data signal from the PS/2 Port.

Outputs: One output for each cardinal direction (Up, Down, Left, Right), and signal describing whether or not the directions are valid or not, 'Readable'.

Simulation: This simulation feeds valid PS/2 inputs that represent the keys W, A, S, and D being pressed down, which should result in the output of Up, Left, Down, then Right.

Description: This module functions as a shift register, with extra logic to handle the PS/2 standard. It waits for the 2nd bit, and reads each bit from 2 through 9 into a data register, starting with the least significant bit and ending with the most significant bit. In addition to storing the data, every time a 1 is received it updates an internal parity bit. The 10th bit is the keyboard's parity bit, and is compared to the internally tracked parity bit. If the two bits match, it updates the outputs to indicate which key was pressed, depending on which data was transferred. If the parity bits don't match, nothing occurs. When the 11th bit is transferred, everything is reset to its initial state.

5.4 Input Decoder/Multiplexer

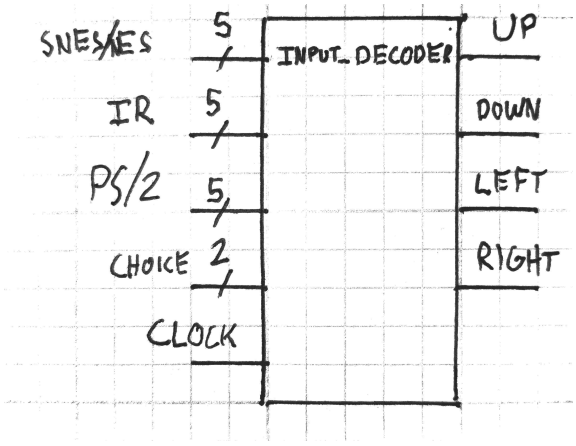

Figure 19: Input Decoder Block Diagram

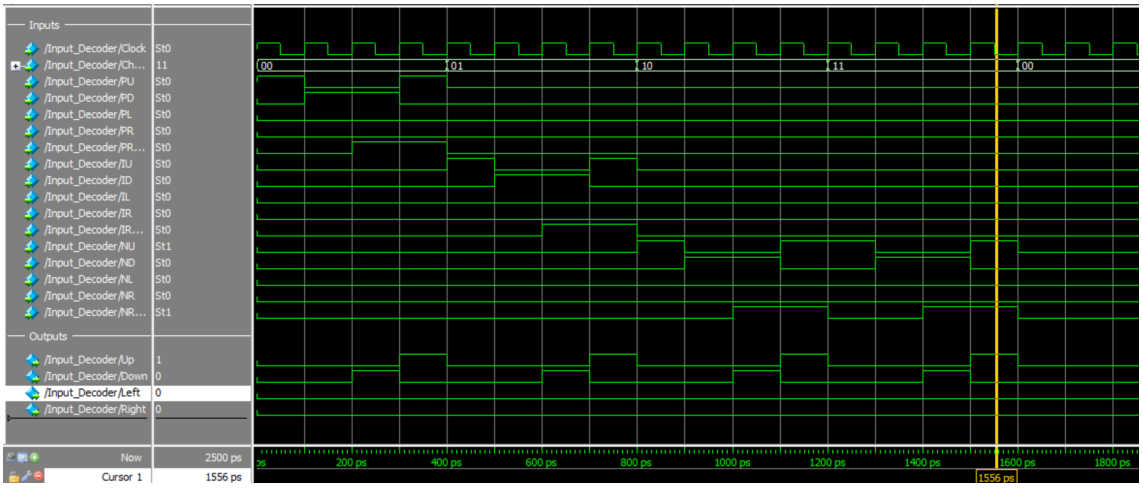

Figure 20: Input Decoder Simulation

Inputs: There are 17 inputs for this module. 5 come from each controller, representing their Up, Down, Left, Right and Readable. One input is the Choice, deciding which controller to use. Finally, a clock is inputted.

Outputs: The Input Decoder outputs a Up, Down, Left, and Right, which are the value passed through from the controller themselves.

Simulation: For this simulation, each of the controllers are tested. PS/2 is tested first, where it sends up and down while not readable, and up and down while readable. The same outputs are tried for both IR and SNES/NES, where all inputs are only readable half of the time. The simulation also shows that some values are driven to 1 from another controller, but are ignored since only one controller can be chosen at once.

Description: The input decoder's job is to use the "Choice" bus as well as the "Readable" values for each controller to decide which values to give to the game logic. A 0 corresponds to PS/2, a 1 corresponds to IR, and a 2 or 3 correspond to SNES and NES, respectively. If the chosen controller is Readable, their values are passed through. if not, the values are driven to 0.

5.5 Game Logic Controller

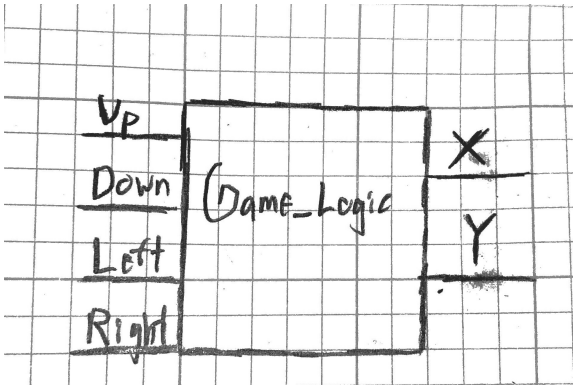

Figure 21: Block Diagram of the Game Logic Controller

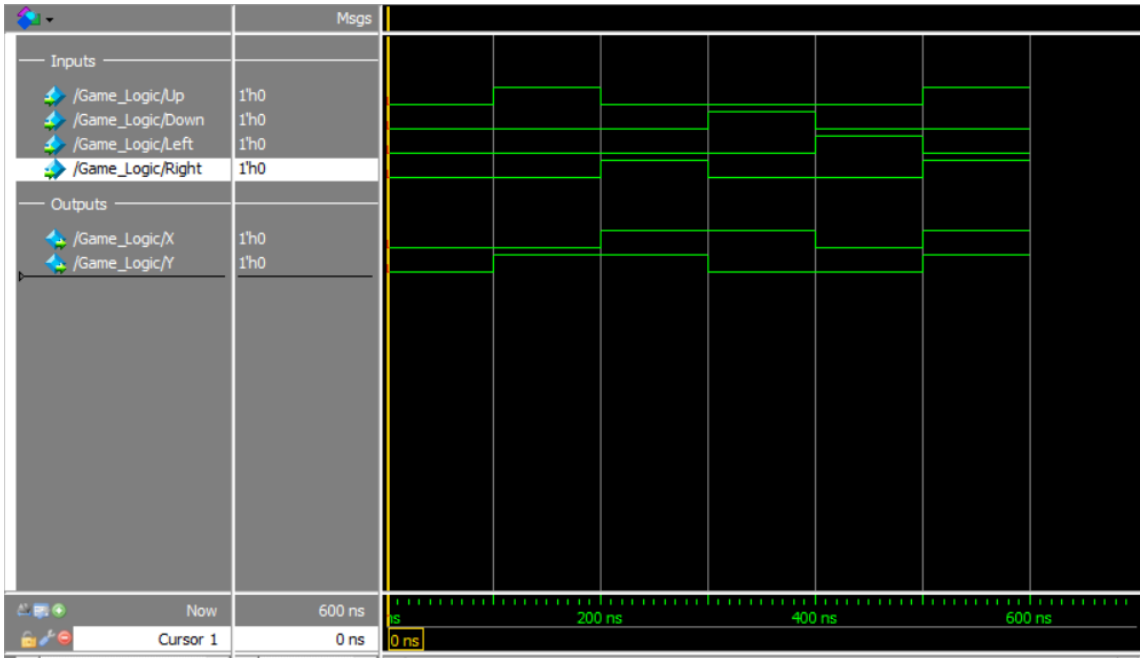

Figure 22: ModelSim of the Game Logic Controller.

Inputs: One bit inputs for up, down, left, and right

Outputs: One bit data inputs for X and Y, representing the position of the player object within the ftwo by two grid where 0,0 is the lower left corner.

Simulation: Within this simulation, the initial position of the player, 0,0. The up input is driven high, and as expected the coordinate output for Y also changes to high. Next, a right input is provided, and again the coordinates change such that X is high. After this, down is driven high and then left is driven high, resulting in the coordinates shifting to 0,1 then 0,0. Finally, both up and right are driven high, resulting in a coordinate of 1,1.

Description: This module will use combinational logic to determine the coordinate state of a player. The current position is stored within the module, and from the provided values of up, down, left, and right the next position is determined. An input that would result in the player

going outside of the coordinate bounds(0,0 to 1,1) is ignored. Multiple inputs can be accepted at once, allowing for diagonal movement.

5.6 Player Top Module

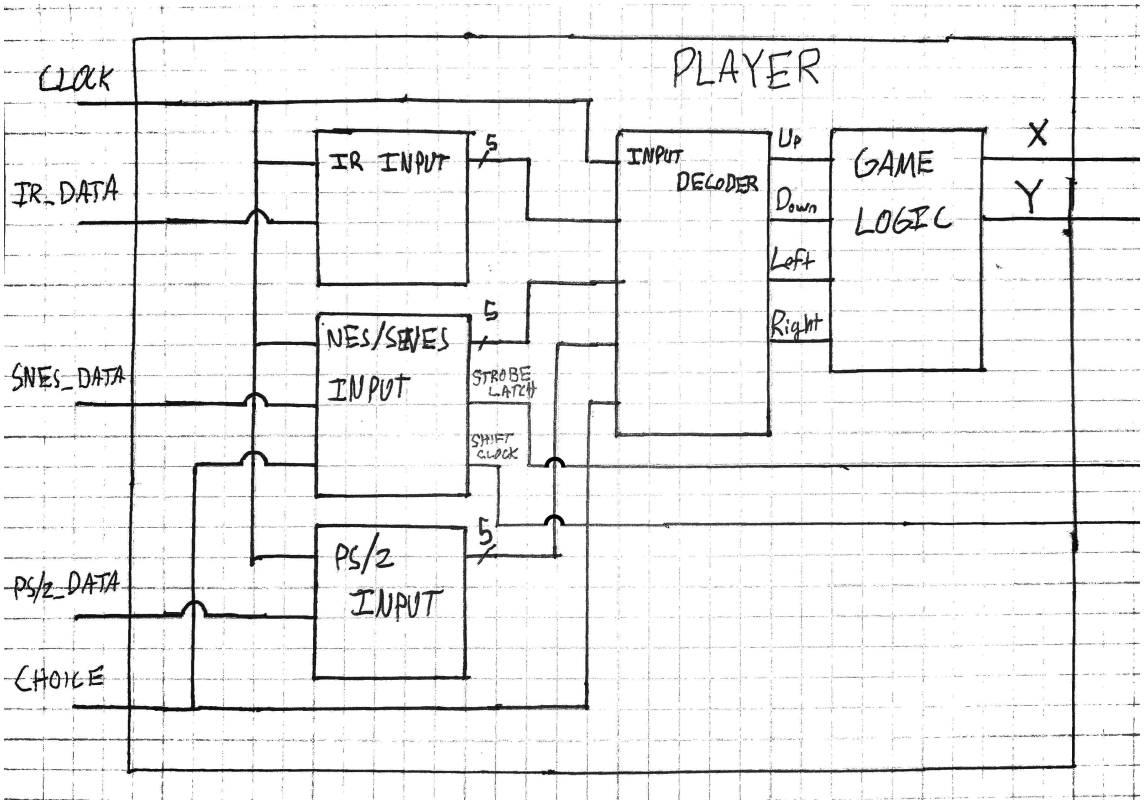

Figure 23: Player Block Diagram

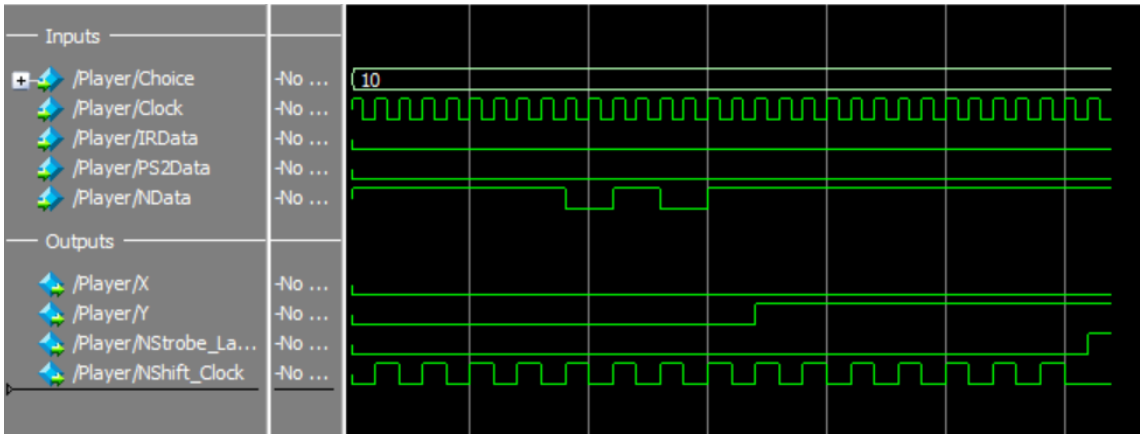

Figure 24: Player Simulation

Inputs: The player module takes in 3 1-bit values representing the serial input for all three controllers, a 2-bit Choice of which controller, and a 50 MHz Clock signal.

Outputs: The Player module outputs an X and a Y bit representing its position in the 2x2 grid.

Simulation: The simulation of the Player module takes in an example data stream for the SNES. after reading in several clock cycles of values, it is decided to move the Y coordinate up to 1 since the key press was Up.

Description: The Player's job is to take in all inputs from all possible controllers, and pick the one to be used. It then handles any game logic for its position, such as running of screen based on controller input of Up, Down, Left, and Right. It then outputs its dynamic coordinate data.

5.7 Pixel Counter

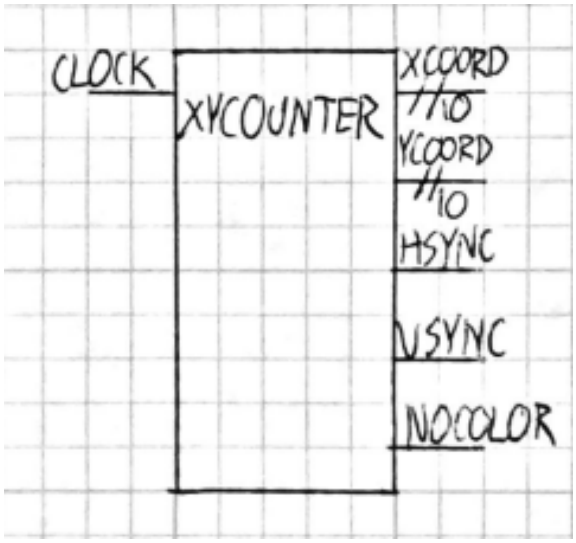

Figure 25: Pixel Counter Block

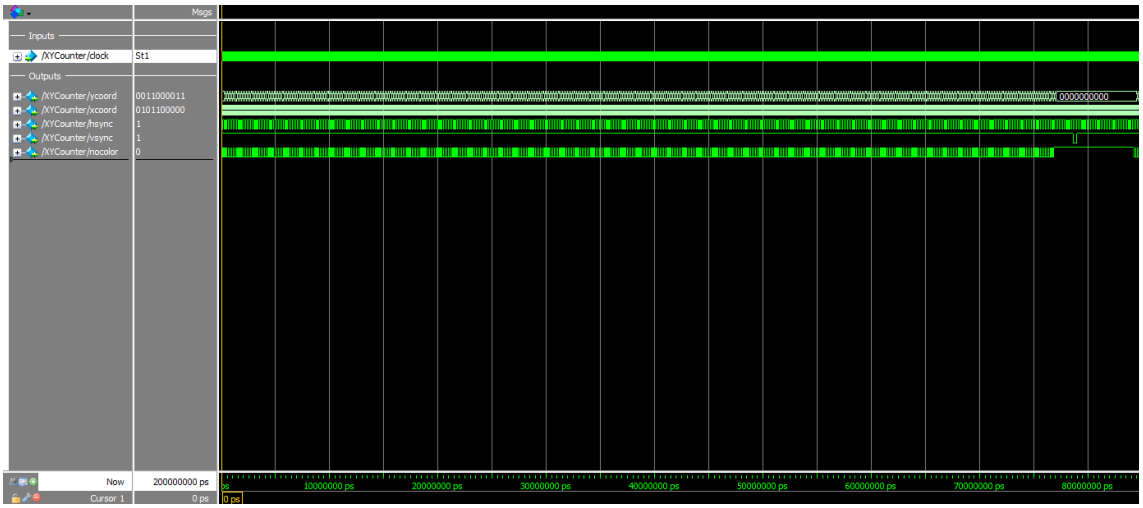

Figure 26: Pixel Counter Simulation Frame-Scale

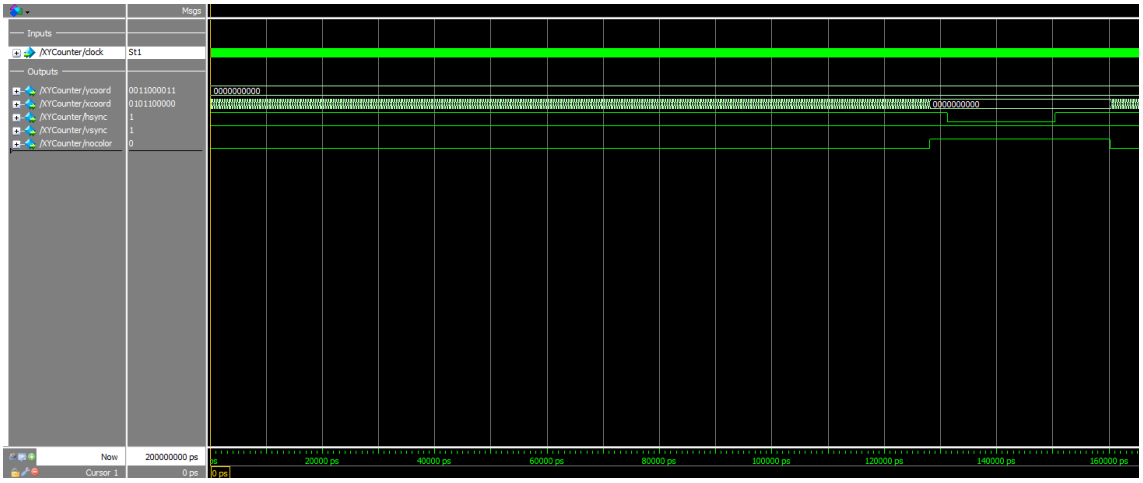

Figure 27: Pixel Counter Simulation Line-Scale

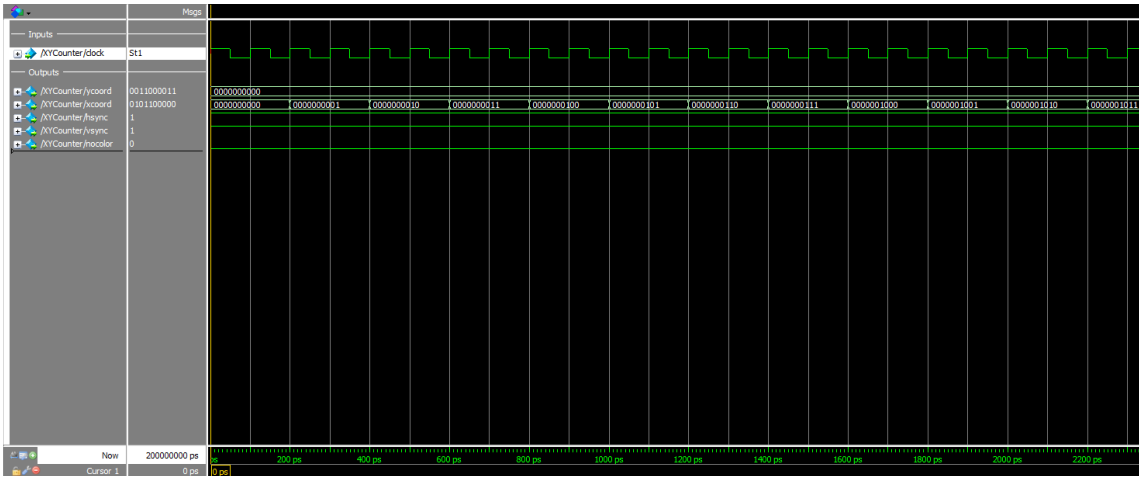

Figure 28: Pixel Counter Simulation Pixel-Scale

Inputs: A 50MHz clock signal.

Outputs: X and Y coordinates indicating the current pixel being drawn on the monitor, an HSync and VSync signal that is fed directly to the VGA output, and a nocolor signal indicating when the color outputs on the VGA should be brought to 0.

Simulation: In this simulation, the only input provided is that of a 50MHz clock signal. The output is sampled for an entire second, and then the waves can be viewed to verify that the timing of all the outputs is correct.

Description: Every clock pulse, a clock enable bit is flipped, and the logic is only updated when the clock enable is 1. There are two internal registers, xstate and ystate, which keep track of the current state in the current row and column. It will drive nocolor to 1 when there should be no color output, and pulse HSync and VSync whenever they need to be pulsed, according to the timings for a 640 x 480 60Hz signal (described in Section 3).

5.8 Renderer

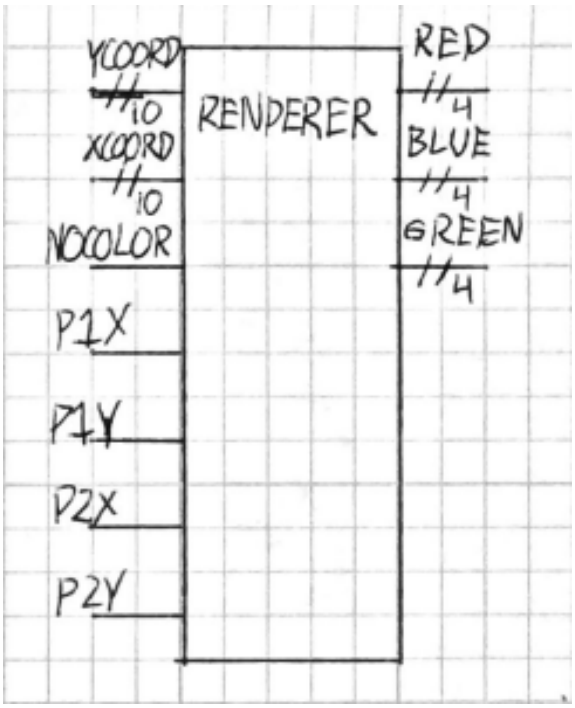

Figure 29: Renderer Block

Figure 30: Renderer Simulation

Inputs: 4 bits that indicate the x and y coordinate of both players, a nocolor signal indicating when the Renderer needs to output low for all colors as part of VGA timings, and the x and y coordinate of the current pixel being drawn by VGA.

Outputs: The Red, Green, and Blue values for the current pixel being rendered, represented as three four bit values.

Simulation: In this simulation, it drives the x and y inputs to 4 coordinates, one in each quadrant. For each coordinate, it drives the player locations of each player such that it tests the four possible color states for each coordinate: no players in quadrant, player 1 in quadrant, player 2 in

quadrant, both players in quadrant. Finally, it drives nocolor to 1, to test that color output is set to 0 when nocolor is 1.

Description: If the nocolor input is 1, all color outputs are driven to 0. Next, it will immediately drive green to 0 as it's not used in our display. Next, it will check what quadrant of the screen the current pixel being drawn is in, and what players are located in that quadrant. If player 1 is in the quadrant, red is driven to full color, otherwise it's driven to 0. Same for player 2 and blue.

6 Putting it Together

Creating the top module consisted of putting all of the modules specified in Section 5 together. The top-most modules that the top module interfaces directly with are the Player, Pixel Counter, and Renderer. All that is needed is 3 single bit inputs for each player for each controller, a choice of controller for each player, and a 50 MHz clock. The resulting simulation can be found in Figure 31.

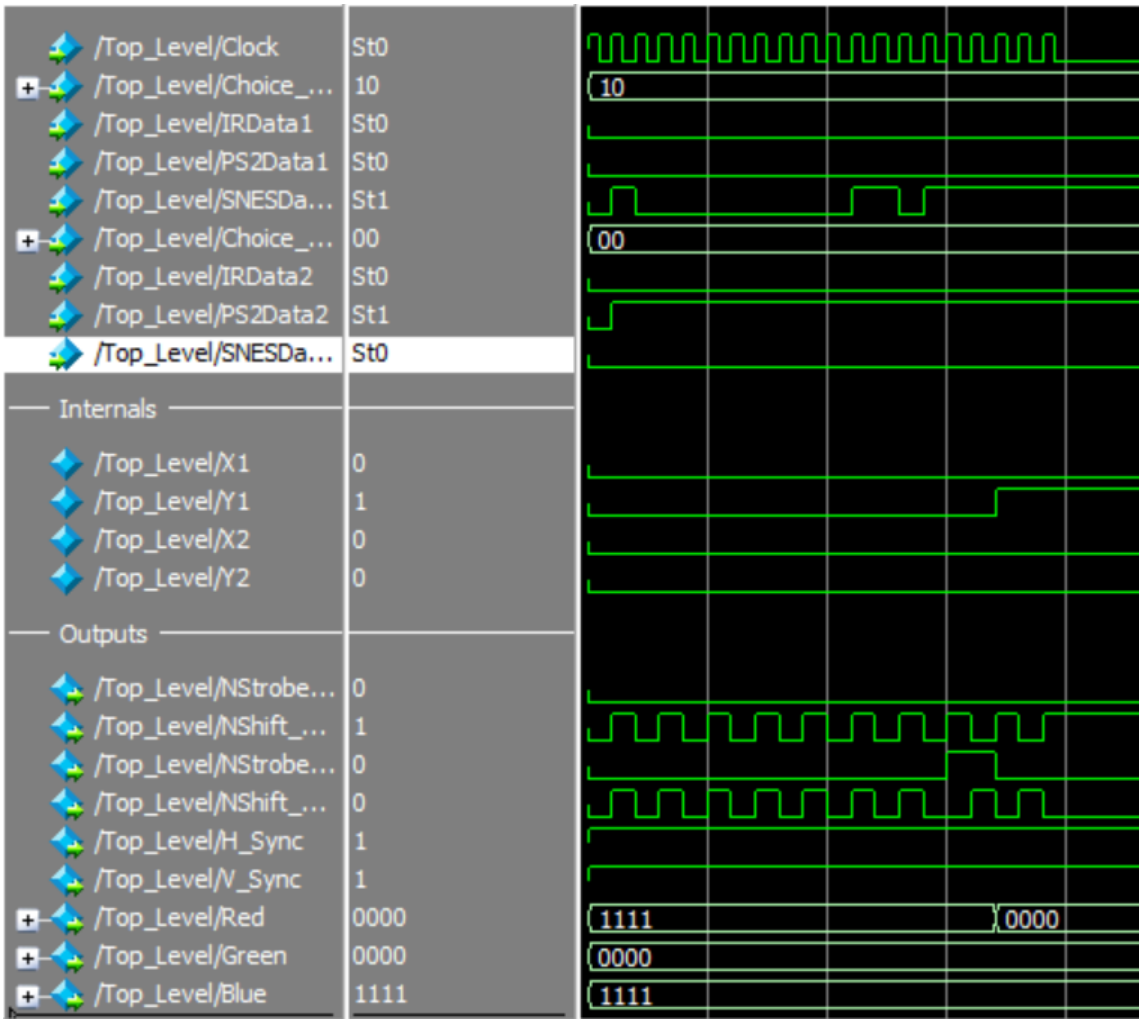

Figure 31: Top Level Simulation

The resulting image shows the position and pixel colors changing based on the clock. Since there are many inputs and outputs, a small section of time is show in the figure. However, an extended diagram would show the continuation of a full frame and a re render of the bottom left location, changing its color from purple. An issue with this simulation is that controller inputs happen to fast- an actual player could not press buttons at any rate specified by our given clocks.

7 Extra Implementation

Our group decided to implement functionality for two player as our bonus feature. Each player has a separate controller, and the players are represented on the screen by blocks of different colors, one red and one blue. The players are both confined to the 2x2 grid, but if both player happen to occupy the same square, they will collide, producing a block colored as a mix between the two players, in this case, black. Players are free to continue moving after collision.

8 References

[1] Davis, Larry. Encoding Dictionary; Manchester Encoding.
`http://www.interfacebus.com/manchester-encoding.html`

[2] Bergmans, Sam. Philips RC-5 Protocol.
`https://www.sbprojects.net/knowledge/ir/rc5.php`

[3] Vishay. Data Formats for IR Remote Control.
`https://www.vishay.com/docs/80071/dataform.pdf`

[4] Image. Eric Eastwood. StackOverflow. VGA Decoding - Dealing with tolerances
`https://electronics.stackexchange.com/questions/92900/vga-decoding-dealing-with-tolerances`

[5] Image. Thomas Crosley. Stack Overflow. How exactly does a VGA cable work?
`https://electronics.stackexchange.com/questions/166681/how-exactly-does-a-vga-cable-work`

[6] Christy, Jim. Super Nintendo Entertainment System: pinouts and protocol
`https://gamefaqs.gamespot.com/snes/916396-super-nintendo/faqs/5395`

[7] Booth, Taylor. Hacking into an old NES controller
`http://www.taylorbooth.com/NESController.htm`

9 Appendix

Source code of this pdf, as well as well as helpful scripts and original figures used in this document can be found in its repository, available at
`https://github.com/drewsortega/DigitalLogicVGA`

9.1 Source Code

9.1.1 NES/SNES Input Decoder

```
//The NES/SNES also require a 5V and GND input, but I think those could be implemented elsewhere
module SNES_Input(
    input logic Data, //Data is the controller output
    input logic[1:0] Mode, //Mode determines if the controller is NES or SNES
    input logic Clock, //System clock
    output logic Strobe_Latch, //Sent to controller
    output logic Shift_Clock,  //Sent to controller
    output logic Up,
    output logic Down,
    output logic Left,
    output logic Right,
    output logic Readable //Signals if data should be read(i.e. all inputs are accounted for)
    );

    logic[4:0] Increment = 0;
    logic Alternate = 0;
```

```

    always_ff @ (posedge Clock)
        begin
            if (Mode[1] == 0) //NES Mode
                begin
                    if (Increment == 0 && Strobe_Latch == 0)
                        Strobe_Latch = 1;   //Strobe_Latch causes the controller to store the pressed buttons in a shift register
                    else
                        begin
                            if(Alternate == 0)
                                begin
                                    Shift_Clock = 0;
                                    case(Increment) //Data is active low
                                        4'b0000 :
                                            begin
                                                Strobe_Latch = 0;
                                                Right = ~Data;
                                            end
                                        4'b0001: Left = ~Data;
                                        4'b0010: Down = ~Data;
                                        4'b0011: Up = ~Data;
                                    endcase
                                    if(Increment >= 3)
                                        Readable = 1; //Readable signals that all (of interest) inputs have been read and the data is ready
                                    Increment = Increment + 1;
                                    if(Increment == 8) //The shift register for NES is 8 bits long, but starts already outputing the first data value,
                                                                //so we reset after 7 increments
                                        begin
                                            Increment = 0;
                                            Readable = 0;
                                        end
                                    Alternate = 1;
                                end
                            else
                                begin
                                    Alternate = 0; //Alternate determines whether inputs should be read, or a clock signal should be sent to the controller
                                    Shift_Clock = 1; //Shift_Clock moves the controller shift register along
                                end
                        end
                end
            else //SNES Mode
```

```
                    begin
                        if (Increment == 0 && Strobe_Latch == 0)
                            Strobe_Latch = 1;
                        else
                            begin
                                if(Alternate == 0)
                                    begin
                                        Shift_Clock = 0;
                                        case(Increment)
                                            4'b0000: Strobe_Latch = 0;
                                            4'b0101: Up = ~Data;     //SNES has the outputs in a different order, for some reason
                                            4'b0110: Down = ~Data;
                                            4'b0111: Left = ~Data;
                                            4'b1000: Right = ~Data;
                                        endcase
                                        if(Increment >= 8)
                                            Readable = 1;
                                        Increment = Increment + 1;
                                        if(Increment == 16) //SNES shift register is 16 bits long, so we increment 15 times
                                            begin
                                                Increment = 0;
                                                Readable = 0;
                                            end
                                        Alternate = 1;
                                    end
                                else
                                    begin
                                        Alternate = 0;
                                        Shift_Clock = 1;
                                    end
                            end
                    end
        end
endmodule
```

9.1.2 IR Input Top Module

```
module IR_Input (
    input logic Clock,
    input logic Data,

    output logic Up,
    output logic Down,
    output logic Left,
    output logic Right,
    output logic Readable
);
    logic Clock_Pulse = 1'b0;
    logic Clock_Logic = 1'b0;
    logic Simplified_Data;

    IR_Clock newClock(.In_Clock(Clock), .Out_Clock_Pulse(Clock_Pulse), .Out_Clock_Logic(Clock_Logic));
    IR_Simplifier simplify(.Clock(Clock_Pulse), .In_Data(Data), .
```

```
        Out_Data(Simplified_Data));
    IR_Parser irparser(.Clock(Clock_Logic), .Data(Simplified_Data), .Up
        (Up), .Down(Down), .Left(Left), .Right(Right), .Readable(
        Readable));
endmodule
```

9.1.3 IR Clocks

```
module IR_Clock (
    input logic In_Clock,
    output logic Out_Clock_Pulse,
    output logic Out_Clock_Logic
);

logic[10:0] Count = 11'b00000000000;
logic[7:0] Count2 = 8'b00000000;
logic Initialized = 1'b0;

always_ff @(posedge In_Clock) begin
    // Base Clock - 50 MHz = 50000 KHz.
    // IR Clock - 36 KHz.

    // 50000/(36*8) = ~174.
    // Therefore 4 rising edges for every 36 kHz pulse. This guarentees
        that
    // 0-173 = 174 states. f=~144 kHz

    // Wait for 64 ticks on the 144 kHz, should have rising edge for
        every rising and falling edge of data.
    if(Initialized == 1'b0) begin
        Out_Clock_Pulse <= 0;
        Out_Clock_Logic <= 0;
        Initialized <= 1;
    end
    else begin
        if(Count < 173) begin
            if(^Out_Clock_Pulse == 1'bx) begin
                Out_Clock_Pulse <= 1'b0;
            end
            if(^Out_Clock_Logic == 1'bx) begin
                Out_Clock_Logic <= 1'b0;
            end
            Count <= Count + 1;
        end
        else begin // accounts for 173
            if(Out_Clock_Pulse == 0) begin
                Out_Clock_Pulse <= 1;
                if(Count2 < 63) begin //acounts for 0-62
                    Count2 <= Count2 + 1;
                end
                else begin //accounts for 63
                    if(Out_Clock_Logic == 0) begin
                        Out_Clock_Logic <= 1;
                    end
                    else if(Out_Clock_Logic == 1) begin
                        Out_Clock_Logic <= 0;
                    end
                    Count2 <= 0;
                end
```

```
            end
            else if(Out_Clock_Pulse == 1) begin
                Out_Clock_Pulse <= 0;
            end
            Count <= 0;
        end
    end
end

endmodule
```

9.1.4 IR Simplifier

```
module IR_Simplifier (
    input logic Clock,
    input logic In_Data,
    output logic Out_Data
);
logic[7:0] Count = 8'b00000000;

//looks for a single HIGH and holds it for 32 cycles at 36 kHz

always_ff @(posedge Clock) begin
    if(Count == 0) begin
        if(In_Data == 1) begin
            Count = Count + 1;
            Out_Data = 1;
        end
        else begin
            Out_Data = 0;
        end
    end
    else begin
        if(Count == 128) begin
            Count = 0;
        end
        else begin
            Count = Count + 1;
        end
        Out_Data = 1;
    end
end

endmodule
```

9.1.5 IR Parser

```
module IR_Parser(
    input logic Clock,
    input logic Data,

    output logic Up,
    output logic Down,
    output logic Left,
    output logic Right,
    output logic Readable
);


//Stage 0XX - Start Bit
```

```
//Stage 000 - no input OR first lower clock of input
//Stage 001 - on HIGH input
//Stage 010 - on second LOW
//Stage 011 - on Final HIGH of start bit

//Stage 100 - toggle bit
//Stage 101 - address
//Stage 110 - data

//local intermitent variables
logic[2:0] Stage = 3'b000;
logic[2:0] InputAddressCounter = 3'b000;
logic[3:0] InputValueCounter = 4'b0000;
logic BitState = 1'b0;

//read inputs
logic Toggle = 1'bx;
logic[4:0] InputAddress = 5'bxxxxx;
logic[5:0] InputValue = 6'bxxxxxx;

logic Initialized = 1'b0; //only used at beginning

logic[4:0] LocalAddress = 5'b01101; //constant which the local address
    needs to stay


always_ff @(posedge Clock)
begin
    if(Stage == 3'b000) begin
        if(Initialized == 1'b0) begin
            Up = 0;
            Down = 0;
            Left = 0;
            Right = 0;
            Readable = 0;
            Initialized = 1;
        end
        if(Data==1'b1) begin //input is starting on first HIGH

            //SyncMode 0 - at beginning of each data bit,
            Stage <= 3'b001;
        end
        if(!(Readable == 1'b0 || Readable == 1'b1)) begin
            Readable = 1'b0;
        end
    end
    // Data should be 0 when here
    else if(Stage == 3'b001) begin
        if(Data==1'b0) begin
            Stage <= 3'b010;
        end
        else begin
            //ERRORRRR!!!!, do something
        end
    end

    // Data shound be 1 when here
    else if(Stage == 3'b010) begin
```

```
        if(Data==1'b1) begin
            Stage <= 3'b100;
        end
        else begin
            //ERROR, do someting
        end
    end

        //every other data bit per clock cycle from now on represents
            each bit.

        //Toggle bit
    else if (Stage == 3'b100) begin
        if(BitState == 1'b0) begin
            if(Data == 1'b1) begin
                Toggle <= 1'b0;
            end
            else if(Data == 1'b0) begin
                Toggle <= 1'b1;
            end
            BitState <= 1'b1;
        end
        else if(BitState == 1'b1) begin
            BitState <= 1'b0;
            Stage <= 3'b101;
        end
    end

        //Address
    else if(Stage == 3'b101) begin
        if(BitState == 1'b0) begin
            if(Data == 1'b1) begin
                InputAddress[4-InputAddressCounter] <= 1'b0;
            end
            else if(Data == 1'b0) begin
                InputAddress[4-InputAddressCounter] <= 1'b1;
            end
            BitState <= 1'b1;
        end
        else if(BitState == 1'b1) begin
            if(InputAddressCounter < 4) begin
                InputAddressCounter <= InputAddressCounter + 1;
            end
            else begin
                InputAddressCounter <= 0;
                Stage<=3'b110;
            end
            BitState <= 1'b0;
        end
    end
    else if(Stage == 3'b110) begin
        if(BitState == 1'b0) begin
            if(Data == 1'b1) begin
                InputValue[5-InputValueCounter] <= 1'b0;
            end
            else if(Data == 1'b0) begin
                InputValue[5-InputValueCounter] <= 1'b1;
            end
```

```
            BitState <= 1'b1;
        end
        else if(BitState == 1'b1) begin
            if(InputValueCounter < 5) begin
                InputValueCounter <= InputValueCounter + 1;
            end
            else begin
                if(InputAddress == LocalAddress) begin
                    if(InputValue == 0) begin
                        Up = 1;
                        Down = 0;
                        Left = 0;
                        Right = 0;
                    end
                    else if(InputValue == 1) begin
                        Up = 0;
                        Down = 1;
                        Left = 0;
                        Right = 0;
                    end
                    else if(InputValue == 2) begin
                        Up = 0;
                        Down = 0;
                        Left = 1;
                        Right = 0;
                    end
                    else if(InputValue == 3) begin
                        Up = 0;
                        Down = 0;
                        Left = 0;
                        Right = 1;
                    end
                    else begin
                        Up = 0;
                        Down = 0;
                        Left = 0;
                        Right = 0;
                    end
                end
                InputValueCounter <= 0;
                Stage<=3'b000;
                Readable<=1'b1;
                Toggle = 1'bx;
                InputAddress = 5'bxxxxx;
                InputValue = 6'bxxxxxx;
            end
            BitState <= 1'b0;
        end
    end
end

endmodule
```

9.1.6 PS/2 Input Decoder

```
module PS2_Input(
    input logic Clock,
    input logic Data,

```

```
    output logic Up = 0,
    output logic Down = 0,
    output logic Left = 0,
    output logic Right = 0,
    output logic Readable = 1
);

    logic [3:0] count = 0;
    logic [7:0] workingdata = 0;
    logic parity = 1;

    always_ff @(negedge Clock) begin
        if (count == 10) begin
            count <= 0;
            parity <= 1;
            Up <= 0;
            Left <= 0;
            Down <= 0;
            Right <= 0;
        end else begin
            count <= count + 1;
        end

        if (count > 0 && count < 9) begin
            workingdata[count - 1] <= Data;
            parity <= parity ^ Data;
        end

        if (count == 9) begin
            if (parity == Data) begin
                case (workingdata)
                    'h11: Up <= 1;
                    'h1E: Left <= 1;
                    'h1F: Down <= 1;
                    'h20: Right <= 1;
                endcase
            end
        end
    end

endmodule
```

9.1.7 Input Decoder/Multiplexer

```
module Input_Decoder (
    input logic Clock,

    //controller choice
    input logic[1:0] Choice,

    //PS2 Inputs
    input logic PU,
    input logic PD,
    input logic PL,
    input logic PR,
    input logic PReadable,

    //IR Inputs
    input logic IU,
```

```
    input logic ID,
    input logic IL,
    input logic IR,
    input logic IReadable,

    //SNES/NES Inputs
    input logic NU,
    input logic ND,
    input logic NL,
    input logic NR,
    input logic NReadable,

    //Outputs
    output logic Up,
    output logic Down,
    output logic Left,
    output logic Right
);
always_ff @(posedge Clock)
begin
    //PS/2
    if(Choice == 0) begin
        if(PReadable==1) begin
            Up=PU;
            Down=PD;
            Left=PL;
            Right=PR;
        end
        else begin
            Up=0;
            Down=0;
            Left=0;
            Right=0;
        end
    end

    // IR
    else if(Choice == 1) begin
        if(IReadable==1) begin
            Up=IU;
            Down=ID;
            Left=IL;
            Right=IR;
        end
        else begin
            Up=0;
            Down=0;
            Left=0;
            Right=0;
        end
    end

    // SNES and NES
    else begin
        if(NReadable==1) begin
            Up=NU;
            Down=ND;
            Left=NL;
```

```
            Right=NR;
        end
        else begin
            Up=0;
            Down=0;
            Left=0;
            Right=0;
        end
    end
end
endmodule
```

9.1.8 Game Logic Controller

```
module Game_Logic(
    input logic Up,
    input logic Down,
    input logic Right,
    input logic Left,
    output logic X,
    output logic Y
    );

    logic[1:0] Current = 2'b00;

always_comb
    begin
        if(Down == 1)
            begin
                if(Current[0] == 1) //Move the player down if they are in an upper square
                    Current[0] = 0;
            end
        if(Up == 1)
            begin
                if(Current[0] == 0) //Move the player up if they are in a lower square
                    Current[0] = 1;
            end
        if(Right == 1)
            begin
                if(Current[1] == 0) //Move the player right if they are in a leftmost square
                    Current[1] = 1;
            end
        if(Left == 1)
            begin
                if(Current[1] == 1) //Move the player left in they arre in a rightmost square
                    Current[1] = 0;
            end
        Y = Current[0]; //Set output
        X = Current[1];
    end
endmodule
```

9.1.9 Player Top Module

```
module Player (
    //Clock from top level
```

```
    input logic Clock,

    //choice of controller
    input logic[1:0] Choice,

    //input data from all of the controllers
    input logic IRData,
    input logic NData,
    input logic PS2Data,

    // SNES/NES specific outputs to the controller
    output logic NStrobe_Latch,
    output logic NShift_Clock,

    //Outputs to renderer
    output logic X,
    output logic Y
);
    logic NReadable, IReadable, PReadable, NU, ND, NL, NR, IU, ID, IL,
        IR, PU, PD, PL, PR, UpDecoded, DownDecoded, LeftDecoded,
        RightDecoded;
    SNES_Input nController(
        //Required Values already defined
        .Data(NData),
        .Mode(Choice),
        .Clock(Clock),

        //SNES/NES Specific IO
        .Strobe_Latch(NStrobe_Latch),
        .Shift_Clock(NShift_Clock),

        //universal IO
        .Up(NU),
        .Down(ND),
        .Left(NL),
        .Right(NR),
        .Readable(NReadable)
    );

    IR_Input irController(
        //required values already defined
        .Clock(Clock),
        .Data(IRData),

        //universal IO
        .Up(IU),
        .Down(ID),
        .Left(IL),
        .Right(IR),
        .Readable(IReadable)
    );

    PS2_Input ps2Controller(
        .Clock(Clock),
        .Data(PS2Data),

        .Up(PU),
        .Down(PD),
```

```
        .Left(PL),
        .Right(PR),
        .Readable(PReadable)
    );

    Input_Decoder inputDecoder(
        .Clock(Clock),
        .Choice(Choice),

        .PU(PU),
        .PD(PD),
        .PR(PR),
        .PL(PL),
        .PReadable(PReadable),

        .IU(IU),
        .ID(ID),
        .IL(IL),
        .IR(IR),
        .IReadable(IReadable),

        .NU(NU),
        .ND(ND),
        .NL(NL),
        .NR(NR),
        .NReadable(NReadable),

        .Up(UpDecoded),
        .Down(DownDecoded),
        .Left(LeftDecoded),
        .Right(RightDecoded)
    );

    Game_Logic gameLogic(
        .Up(UpDecoded),
        .Down(DownDecoded),
        .Left(LeftDecoded),
        .Right(RightDecoded),

        .X(X),
        .Y(Y)
    );

endmodule
```

9.1.10 Pixel Counter

```
module XYCounter(
    input logic [0:0] clock,
    output logic [9:0] ycoord = 0,
    output logic [9:0] xcoord = 0,
    output logic [0:0] hsync = 1,
    output logic [0:0] vsync = 1,
    output logic [0:0] nocolor = 0
    );

    logic [0:0] clock_enable = 1;
    logic [9:0] xstate = 0;
    logic [9:0] ystate = 0;
```

```

    always_ff @(posedge clock)
    begin
        clock_enable <= !clock_enable;
        if (clock_enable) begin

            if (xstate < 800) begin
                xstate <= xstate + 1;
            end else begin
                xstate <= 0;
            end

            if (xstate < 640) begin
                xcoord <= xstate;
            end else if (xstate == 640) begin
                xcoord <= 0;
            end else if (xstate == 656) begin
                hsync <= 0;
            end else if (xstate == 752) begin
                hsync <= 1;
            end else if (xstate == 800) begin
                if (ystate < 525) begin
                    ystate <= ystate + 1;
                end else begin
                    ystate <= 0;
                end

                if (ystate < 480) begin
                    ycoord <= ystate;
                end else if (ystate == 480) begin
                    ycoord <= 0;
                end else if (ystate == 490) begin
                    vsync <= 0;
                end else if (ystate == 492) begin
                    vsync <= 1;
                end
            end

            if (xstate >= 640 || ystate >= 480) begin
                nocolor <= 1;
            end else begin
                nocolor <= 0;
            end
        end
    end


endmodule
```

9.1.11 Renderer

```
module Renderer(
    input logic [9:0] ycoord,
    input logic [9:0] xcoord,
    input logic [0:0] nocolor,
    input logic [0:0] p1x,
    input logic [0:0] p1y,
    input logic [0:0] p2x,
    input logic [0:0] p2y,
```

```
    output logic [3:0] red,
    output logic [3:0] blue,
    output logic [3:0] green
    );

    always_comb
    begin
        if (nocolor == 0) begin
            green = 0;
            if (xcoord <= 320 && ycoord <= 240) begin
                if (p1x == 0 && p1y == 0) begin
                    red = 15;
                end else begin
                    red = 0;
                end
                if (p2x == 0 && p2y == 0) begin
                    blue = 15;
                end else begin
                    blue = 0;
                end
            end else if (xcoord > 320 && ycoord <= 240) begin
                if (p1x == 1 && p1y == 0) begin
                    red = 15;
                end else begin
                    red = 0;
                end
                if (p2x == 1 && p2y == 0) begin
                    blue = 15;
                end else begin
                    blue = 0;
                end
            end else if (xcoord <= 320 && ycoord > 240) begin
                if (p1x == 0 && p1y == 1) begin
                    red = 15;
                end else begin
                    red = 0;
                end
                if (p2x == 0 && p2y == 1) begin
                    blue = 15;
                end else begin
                    blue = 0;
                end
            end else if (xcoord > 320 && ycoord > 240) begin
                if (p1x == 1 && p1y == 1) begin
                    red = 15;
                end else begin
                    red = 0;
                end
                if (p2x == 1 && p2y == 1) begin
                    blue = 15;
                end else begin
                    blue = 0;
                end
            end
        end else begin
            red = 0;
            blue = 0;
            green = 0;
```

```
        end
    end


endmodule
```

9.2 Do Files

9.2.1 NES/SNES Input Decoder

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim SNES_Input

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Data
add wave -noupdate Mode
add wave -noupdate Clock
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate Up
add wave -noupdate Down
add wave -noupdate Left
add wave -noupdate Right
add wave -noupdate Shift_Clock
add wave -noupdate Strobe_Latch
add wave -noupdate Readable
add wave -noupdate Increment


# 5) Set some test patterns, the format is force Input value time(in ns
    )

force Clock 1 0, 0 {50 ps} -r 100

# Clock 1
force Mode 2 0
force Data 1 0

# Clock 2
force Data 1 100

#Clock 10
force Data 0 900

# Clock 12
force Data 1 1100

# Clock 14
force Data 0 1300

# Clock 16
```

```
force Data 1 1500

# 6) Run the simulation for 3200 ns
run 3200
```

9.2.2 IR Input Top Module

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim IR_Parser

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Clock
add wave -noupdate Data
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate Up
add wave -noupdate Down
add wave -noupdate Left
add wave -noupdate Right
add wave -noupdate Readable


# 5) Set some test patterns, the format is force Input value time(in ns
    )

force Clock 1 0, 0 {50 ps} -r 100

# Input 1 (Left)
# Clock 1
force Data 0 0

# Clock 2
force Data 1 125

# Clock 3
force Data 0 225

# Clock 4
force Data 1 325

# Clock 5
force Data 0 425

# Clock 6
force Data 1 525

#Clock 8
force Data 0 725

# Clock 10
force Data 1 925
```

```

# Clock 11
force Data 0 1025

# Clock 12
force Data 1 1125

# Clock 14
force Data 0 1325

# Clock 16
force Data 1 1525

#Clock 18
force Data 0 1725

#Clock 19
force Data 1 1825

# Clock 20
force Data 0 1925

# Clock 21
force Data 1 2025

# Clock 22
force Data 0 2125

# Clock 23
force Data 1 2225

# Clock 24
force Data 0 2325

# Clock 26
force Data 1 2525

# Clock 28
force Data 0 2725

# 6) Run the simulation for 4400 ns
run 4400
```

9.2.3 IR Clocks

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate -divider Inputs
add wave -noupdate /IR_Clock/In_Clock
add wave -noupdate -divider Outputs
add wave -noupdate /IR_Clock/Out_Clock_Pulse
add wave -noupdate /IR_Clock/Out_Clock_Logic
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {60470000 ps} 0}
quietly wave cursor active 1
configure wave -namecolwidth 150
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
```

```
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {0 ps} {131072 ns}
```

9.2.4 IR Simplifier

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim IR_Simplifier

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Clock
add wave -noupdate In_Data
add wave -noupdate -divide -height 32 Internas
add wave -noupdate Count
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate Out_Data


# 5) Set some test patterns, the format is force Input value time(in ns
    )

force Clock 1 0, 0 {75 ps} -r 150

# CLock 1, Data 1
force In_Data 1 0

force In_Data 0 200

#Cycle 2
force In_Data 1 600

force In_Data 0 800

#Ignore rest

force Clock 0 1

# 6) Run the simulation for 104925 ns
run 104925
```

9.2.5 IR Parser

```
# 1) Create a library for working in
```

```
vlib work

# 2) Load file for simulation
vsim IR_Parser

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Clock
add wave -noupdate Data
add wave -noupdate -divider -height 32 Internals
add wave -noupdate Stage
add wave -noupdate Toggle
add wave -noupdate InputAddress
add wave -noupdate LocalAddress
add wave -noupdate InputValue
add wave -noupdate BitState
add wave -noupdate InputAddressCounter
add wave -noupdate InputValueCounter
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate Up
add wave -noupdate Down
add wave -noupdate Left
add wave -noupdate Right
add wave -noupdate Readable


# 5) Set some test patterns, the format is force Input value time(in ns
    )

force Clock 1 0, 0 {50 ps} -r 100

# Input 1 (Left)
# Clock 1
force Data 0 0

# Clock 2
force Data 1 125

# Clock 3
force Data 0 225

# Clock 4
force Data 1 325

# Clock 5
force Data 0 425

# Clock 6
force Data 1 525

#Clock 8
force Data 0 725

```

```
# Clock 10
force Data 1 925

# Clock 11
force Data 0 1025

# Clock 12
force Data 1 1125

# Clock 14
force Data 0 1325

# Clock 16
force Data 1 1525

#Clock 18
force Data 0 1725

#Clock 19
force Data 1 1825

# Clock 20
force Data 0 1925

# Clock 21
force Data 1 2025

# Clock 22
force Data 0 2125

# Clock 23
force Data 1 2225

# Clock 24
force Data 0 2325

# Clock 26
force Data 1 2525

# Clock 28
force Data 0 2725

# 6) Run the simulation for 4400 ns
run 4400
```

9.2.6 PS/2 Input Decoder

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim PS2_Input

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
```

```
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Data
add wave -noupdate Clock
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate Up
add wave -noupdate Down
add wave -noupdate Left
add wave -noupdate Right
add wave -noupdate count
add wave -noupdate workingdata
add wave -noupdate parity


# 5) Set some test patterns, the format is force Input value time(in ns
    )

force Clock 1 0, 0 {50 ps} -r 100

# Input 1 (W key, Up)
# Clock 1
force Data 0 0

# Clock 2
force Data 1 100

# Clock 3
force Data 0 200

# Clock 6
force Data 1 500

#Clock 7
force Data 0 600

# Clock 10
force Data 1 900

# Input 2 (A key, Left)
# Clock 12
force Data 0 1100

# Clock 14
force Data 1 1300

# Clock 18
force Data 0 1700

# Clock 21
force Data 1 2000

# Input 3 (S key, Down)
# Clock 23
force Data 0 2200

# Clock 24
force Data 1 2300

#Clock 29
```

```
force Data 0 2800

# Clock 33
force Data 1 3200

# Input 4 (D key, Right)
# Clock 34
force Data 0 3300

#Clock 40
force Data 1 3900

#Clock 41
force Data 0 4000

# Clock 44
force Data 1 4300

# 6) Run the simulation for 4400 ns
run 4400
```

9.2.7 Input Decoder/Multiplexer

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim Input_Decoder

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Clock
add wave -noupdate Choice
add wave -noupdate PU
add wave -noupdate PD
add wave -noupdate PL
add wave -noupdate PR
add wave -noupdate PReadable

add wave -noupdate IU
add wave -noupdate ID
add wave -noupdate IL
add wave -noupdate IR
add wave -noupdate IReadable

add wave -noupdate NU
add wave -noupdate ND
add wave -noupdate NL
add wave -noupdate NR
add wave -noupdate NReadable
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate Up
add wave -noupdate Down
add wave -noupdate Left
```

```
add wave -noupdate Right


# 5) Set some test patterns the format is force Input value time(in ns)

force clock 1 0, 0 {50 ps} -r 100

#Clock 1
force Choice 0 0
force PU 1 0
force PD 0 0
force PL 0 0
force PR 0 0
force PReadable 0 0
force IU 0 0
force ID 0 0
force IR 0 0
force IL 0 0
force IReadable 0 0
force NU 0 0
force ND 0 0
force NR 0 0
force NL 0 0
force NReadable 0 0

#Clock 2
force PU 0 100
force PD 1 100

#Clock 3
force PReadable 1 200

#Clock 4
force PU 1 300
force PD 0 300

#Clock 5
force Choice 1 400
force PU 0 400
force PReadable 0 400
force IU 1 400

#Clock 6
force IU 0 500
force ID 1 500

#Clock 7
force IReadable 1 600

#Clock 8
force IU 1 700
force ID 0 700

#Clock 9
force Choice 10 800
force IU 0 800
force IReadable 0 800
force NU 1 800
```

```

#Clock 10
force NU 0 900
force ND 1 900

#Clock 11
force NReadable 1 1000

#Clock 12
force NU 1 1100
force ND 0 1100

#Clock 13
force Choice 11 1200
force NReadable 0 1200

#Clock 14
force NU 0 1300
force ND 1 1300

#Clock 15
force NReadable 1 1400

#Clock 16
force NU 1 1500
force ND 0 1500

#Clock 17
force Choice 0 1600
force NU 0 1600
force NReadable 0 1600


run 2500
```

9.2.8 Game Logic Controller

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim Game_Logic

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Up
add wave -noupdate Down
add wave -noupdate Left
add wave -noupdate Right
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate X
add wave -noupdate Y


```

```
# 5) Set some test patterns, the format is force Input value time(in ns)
force Up 0 0
force Down 0 0
force Left 0 0
force Right 0 0

# Up = 1, at 100 ns
force Up 1 100

# Up = 0, Right = 1 at 200 ns
force Up 0 200
force Right 1 200

# Right = 0, Down = 1 at 300 ns
force Right 0 300
force Down 1 300

# Left = 1, Down = 0 at 400 ns
force Left 1 400
force Down 0 400

# Left = 0, Up = 1, Right = 1 at 500 ns
force Left 0 500
force Up 1 500
force Right 1 500

# 6) Run the simulation for 600 ns
run 600
```

9.2.9 Player Top Module

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim Player

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate Choice
add wave -noupdate Clock
add wave -noupdate IRData
add wave -noupdate PS2Data
add wave -noupdate NData
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate X
add wave -noupdate Y
add wave -noupdate NStrobe_Latch
add wave -noupdate NShift_Clock


# 5) Set some test patterns, the format is force Input value time(in ns)
```

```

force Clock 1 0, 0 {50 ps} -r 100

# Clock 1
force Choice 10 0
force IRData 0 0
force PS2Data 0 0
force NData 1 0

# Clock 2
force NData 1 100

# Clock 10
force NData 0 900

# Clock 12
force NData 1 1100

# Clock 14
force NData 0 1300

# Clock 16
force NData 1 1500

# 6) Run the simulation for 3200 ns
run 3200
```

9.2.10 Pixel Counter

```
# 1) Create a library for working in
vlib work

# 2) Load file for simulation
vsim XYCounter

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate clock
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate ycoord
add wave -noupdate xcoord
add wave -noupdate hsync
add wave -noupdate vsync
add wave -noupdate nocolor

# 5) Set some test patterns, the format is force Input value time(in ns
    )
force clock 1 0, 0 {50 ps} -r 100

# 6) Run the simulation for 20000000 ns
run 200000000
```

9.2.11 Renderer

```
# 1) Create a library for working in
```

```
vlib work

# 2) Load file for simulation
vsim Renderer

# 3) Open some selected windows for viewing
view structure
view signals
view wave

# 4) Show some of the signals in the wave window
add wave -noupdate -divider -height 32 Inputs
add wave -noupdate ycoord
add wave -noupdate xcoord
add wave -noupdate nocolor
add wave -noupdate p1x
add wave -noupdate p1y
add wave -noupdate p2x
add wave -noupdate p2y
add wave -noupdate -divider -height 32 Outputs
add wave -noupdate red
add wave -noupdate blue
add wave -noupdate green

# 5) Set some test patterns, the format is force Input value time(in ns
    )
force ycoord 0000000101 0
force xcoord 0000000101 0
force nocolor 0 0
force p1x 1 0
force p1y 0 0
force p2x 1 0
force p2y 0 0

force p1x 0 50

force p1x 1 100
force p2x 0 100

force p1x 0 150

force xcoord 0110010000 200

force p1x 1 250

force p1x 0 300
force p2x 1 300

force p1x 1 350

force ycoord 0110010000 400

force p1y 1 450

force p1y 0 500
force p2y 1 500

force p1y 1 550
```

```

force xcoord 0 600

force p1x 0 650

force p1x 1 700
force p2x 0 700

force p1x 0 750

force nocolor 1 800

force nocolor 0 850

# 6) Run the simulation for 600 ns
run 1000
```